D1.2 Second report on a coordination language for robust, adaptive systems

Project acronym: ADMORPH
Project full title: Towards Adaptively Morphing Embedded Systems
Grant agreement no.: 871259

Due Date:	Month 22
Delivery:	Month 23
Lead Partner:	UvA
Editor:	Clemens Grelck, UvA
Dissemination Level:	Public (P)
Status:	Final
Approved:	xxx
Version:	2.0

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871259 (ADMORPH project).

DOCUMENT INFO – Revision History

Date and version	Author	Comments
28/06/2021 0.1	M. Maggio	Formal guarantees
08/10/2021 0.2	C. Grelck	Deliverable structure
11/10/2021 0.3	C. Grelck	TeamPlay language evolution
14/10/2021 0.4	C. Grelck	TeamPlay compiler and runtime
15/10/2021 0.5	F. Haas	ADMORPH eXchange Format
18/10/2021 0.6	L. Miedema	UPPAAL modeling
20/10/2021 0.7	L. Miedema	Compilation to ADMORPH eXchange Format
21/10/2021 0.8	C. Grelck	Finalised for internal review
05/11/2021 1.0	C. Grelck	Incorporated internal feedback
07/11/2021 1.1	F. Haas	Motivation ADMORPH eXchange Format
09/11/2021 1.2	C. Grelck	YASMIN real-time runtime environment
10/11/2021 1.3	C. Grelck	Introduction and conclusion rewritten
11/11/2021 1.4	C. Grelck	Finishing touches across document
13/11/2021 2.0	C. Grelck	Spell-checking and finalisation

List of Contributors

Date and version	Beneficiary	Comments
28/06/2021 0.1	ULUND	Formal guarantees
08/10/2021 0.2	UvA	Deliverable structure
11/10/2021 0.3	UvA	TeamPlay language evolution
14/10/2021 0.4	UvA	TeamPlay compiler and runtime
15/10/2021 0.5	UAU	ADMORPH eXchange Format
18/10/2021 0.6	UvA	UPPAAL modeling
20/10/2021 0.7	UvA	Compilation to ADMORPH eXchange Format
21/10/2021 0.8	UvA	Finalised for internal review
05/11/2021 1.0	UvA	Incorporated internal feedback
07/11/2021 1.1	UAU	Motivation ADMORPH eXchange Format
09/11/2021 1.2	UvA	YASMIN real-time runtime environment
10/11/2021 1.3	UvA	Introduction and conclusion rewritten
11/11/2021 1.4	UvA	Finishing touches across document
13/11/2021 2.0	UvA	Spell-checking and finalisation

Contents

Executive summary

Deliverable D1.2 is the second deliverable of work package 1: *Specification of Adaptive Systems*. It contains the second report on a coordination language for robust, adaptive systems. At the time of reporting three tasks of work package 1 are active, namely:

- Task 1.1: Coordination language design, led by UvA;
- Task 1.2: Validation of the coordination language, led by UvA;
- Task 1.3: Specification of formal guarantees for the adaptation layer, led by ULUND;

Work package 1 targets the specification of adaptive systems including their functional and non-functional behaviour, possible fault and attack models, and formal guarantees of the adaptation layer. Central to this work package is a (domain-specific) coordination language TeamPlay that allows us to specify software components, their properties and their orderly interplay at a very high level of abstraction. On top of the obvious aim of functional correctness, the coordination language is particularly concerned with non-functional properties of code execution including reliability, time and energy.

1 Introduction

We (partially) build our work work in this area on previous and on-going work on the TeamPlay coordination language [30]. Work on the underlying coordination model and the core language have been developed in the context of the Horizon-2020 project TeamPlay[1], but continue to be subjects of on-going research in the ADMORPH project.

In the context of the TeamPlay project our focus has been on the non-functional properties energy, time and security. In particular guarantees on worst case execution time play a vital role in the ADMORPH project as well. Energy and security are likewise relevant to ADMORPH, but possibly less prominently. Instead we add two new strands to the development of the TeamPlay language: robustness against partial hardware failure and robustness against cyber attack, both potentially leading to transient or permanent unavailability of computing resources.

The organisation of this deliverable deviates from the task structure of the work package. The chosen structure in our view reflects the actual work done in a more balanced way. The achievements described in Section 2 up until Section 7 can be attributed to Task 1.1 while the work described in Section 8 belongs to Task 1.3. Our work on Task 1.2 has only started with initial talks to the use case providing partners. Following our initial hiring delays experienced by UvA, we decided to focus on Task 1.1 for the time being. We do not expect any long-term or tangible impact on the project due to this decision.

We commence our journey with reporting on the advances of the TeamPlay coordination language: we carefully revised the syntax of the language for better readability and comprehensibility. Furthermore, we added so-called *modes* to the language. These modes permit the coordinated application to reflect on changing characteristics of its execution environment and thus to react, for instance, on faulting hardware (permanent or transient) or on cyber attacks. We report on this work in Section 2.

In the reporting period we have also spent considerable effort into our coordination compiler, named CECILE. We extended the compiler front-end to accommodate the various language extensions developed and adapted the various intermediate compiler analyses accordingly. Furthermore, we added three new code generators targeting our three novel runtime environments developed during the current reporting period. We describe our work on the CECILE coordination compiler in more detail in Section 3 and the three runtime environments thereafter.

In order to support the various TeamPlay language extensions specifically geared at fault-tolerance that we already described in Deliverable D1.1 we designed and implemented a corresponding fault-tolerant runtime environment. The result is a proof-of-concept runtime environment that dynamically reconfigures running applications upon detection of hardware failures. This runtime environment targets both permanent (or crash) faults as well as transient faults, for instance detected via n-modular redundancy. The runtime environment comes with a fault injection facility for demonstration purposes. However, our fault-tolerant runtime proof-of-concept has no real-time capabilities. We describe our work on the fault-tolerant runtime system in Section 4.

As a synergy with the above mentioned Horizon-2020 project TeamPlay, we developed the real-time runtime environment YASMIN (*Yet Another Scheduling MIddleware for exploratioN*). YASMIN schedules end-user applications with real-time requirements in user space and on be-

[1]European Union Horizon-2020 research and innovation programme grant agreement No. 779882 (TeamPlay), 2018–2020

half of the operating system all with an easy-to-use programming interface and portability across a wide range of hardware platforms. YASMIN treats heterogeneity on COTS heterogeneous embedded platforms as a first-class citizen supporting multiple functionally equivalent task implementations (versions) with distinct extra-functional behaviour, tailor-made for the TeamPlay coordination language. At the time of writing YASMIN supports the whole range of TeamPlay features, but not (yet) the fault-tolerance aspects of the language. We describe YASMIN in more detail in Section 5.

The third compilation target of the coordination compiler is not a runtime environment per sé, but the ADMORPH eXchange Format (AXF) meant to facilitate the interplay of tools contributed by various ADMORPH stakeholders. We describe the design of the ADMORPH eXchange Format (AXF), compilation of TeamPlay coordination programs to the AXF as well as another runtime environment implementing the AXF in Section 6.

The effects of transient faults and the timing impact of their mitigation can only be analysed by looking at long-running application behaviour. A wall-time simulator would take too much time to gather meaningful data over the behaviour of an application in the presence of transient faults. Therefore, we decided to analyse long-running application behaviour using simulated time. To this end, we have enlisted the help of the existing model checking tool *UPPAAL* and added code generation for UPPAAL as yet another compilation target to the TeamPlay coordination compiler CECILE. We summarise this thread of research in Section 7.

At last we come to Task 1.3 on the *specification of formal guarantees for the adaptation layer*. In this task, we are trying to quantify the natural resilience of control systems to faults and attacks. For this, we focus on a failure model that manifests itself in the controller missing deadlines (although this may come from never receiving a sensor data or experiencing delays), which are quantified according to the weakly-hard task model [7]. We analyse the robustness of control systems using two different dimensions: stability and performance. The result of the analysis is a set of constraints that, when satisfied, formally guarantee that control tasks do not misbehave. If the system is under attack, this means that the adaptation layer must guarantee a reaction time and the restoration of nominal condition within the threshold specified by the analysis conducted in this task. We provide a detailed account of our work on Task T1.3 in Section 8.

With Section 10 we end this Deliverable with some intermediate conclusions and an outlook on the coming steps to take.

2 Evolution of TeamPlay coordination language

In Deliverable D1.1 we discussed the underlying concepts of our TeamPlay DSL (domain-specific language) for adaptive multi-core coordination. We presented both the language as it was prior to the start of the ADMORPH project as well as the extensions we developed specifically for the purpose of ADMORPH. In the following we describe a careful evolution of the syntax of the core language in Section 2.1. Furthermore, we introduce TeamPlay *modes*, a recent extension to reflect on potential property changes in the runtime environment of the running coordination program, in Section 2.2.

The work described in this section is a synergy between the ADMORPH project and the Horizon-2020 project TeamPlay *Time, Energy and Security Analysis for Multi/Many-core Heterogeneous Platforms* (2018–2021, grant agreement no 779882).

2.1 Syntax evolution

Figure 1 shows the revised syntax of the core language in Extended Backus-Naur Form (EBNF). For conciseness we leave out the various additions described in Deliverable D1.1, both for ease of programming and abstraction as well as for fault-tolerance; they all remain unchanged. In the following we focus on the differences while generally speaking the commonalities prevail. As before, a coordination application starts with the keyword `app` followed by the name of the application and its actual specification enclosed in curly brackets.

Following global specifications of deadline and period of the application as a whole, the bulk of a coordination program is made up of two further sections headed by the keywords `components` and `channels`. In these sections we specify the properties of a DAG (directed acyclic graph). In graph terminology our components are *vertices* and our channels are *edges*. In real-time embedded systems terminology our components are usually referred to as *tasks* while our channels are called *dependencies*. Throughout this deliverable report we will use the terminology interchangeably.

Comparing old and new syntax in detail, we observe that the former keyword `edges` has been substituted by the new keyword `channels`. The new keyword emphasises the operational behaviour of a coordinated application, where graph edges do not only specify abstract dependencies between components, but rather concrete data transfers from source components to sink components.

A tangible difference between initial and revised coordination language design is that we decided to remove the `datatypes` section entirely. Originally, this section mapped abstract type identifiers to concrete C language types. We removed the datatype implementations because we found that having concrete C types in a coordination program would not fit the high-level character of coordination programs well. The corresponding information can now be provided in one of two possible ways: via the config file or by simply providing a matching C type definition with the component implementation code base. The former allows us to still provide auxiliary information about types in line with the initial syntax of the TeamPlay coordination language.

In all three categories, inports, outports and state ports we replaced the initial notion of a `connector` by the new *port lists*. Port lists essentially mimic the syntax of C structs, instead of using a tailor-made (but hence also exotic) syntax characteristic for connectors. Optional

```
CoordApp    ⇒  app Id { AppBody }
AppBody     ⇒  [ period FrequencyConst ]
               [ deadline FrequencyConst ]
               Components
               Channels
Components  ⇒  components { [ Component ]+ }
Component   ⇒  Id {
               [ inports PortList ]
               [ outports PortList ]
               [ state PortList ]
               [ Settings ]
               [ Version ]*
               }
Settings    ⇒  { Setting [ ; Setting ]* [ ; ] }

Setting     ⇒  period FrequencyConst
            |  deadline FrequencyConst
            |  arch StringConst
            |  security IntConst
            |  cname StringConst
Version     ⇒  version Id [Settings ]
PortList    ⇒  { Port [ ; Port ]* [ ; ] }
Port        ⇒  Type Id [ [ IntConst ] ]
Channels    ⇒  channels { [ Channel ]* }
Channel     ⇒  OneToOne | Broadcast
OneToOne    ⇒  OutPort -> InPort
Broadcast   ⇒  OutPort -> InPort [ & InPort ]+
InPort      ⇒  Id [ . Id ]
OutPort     ⇒  Id [ . Id ]
```

Figure 1: Revised grammar of coordination language in EBNF

multiplicities, i.e. the opportunity to send or receive multiple data tokens at once, now makes use of C array syntax.

A component may have multiple versions that are functionally equivalent but typically expose different non-functional behaviour. In the new syntax we have made the provision of

attributes between single-version and multi-version components uniform. All *settings* can be attributed to the component as well as to any of its versions (a.k.a. implementations or variants). In the absence of version specifications, hence a single-version component, all attributes are directly attached to the component specification. In the presence of version specifications, typically multi-version components, component-level attributes act as defaults for the various versions and can be overwritten with version-specific values.

Looking at the supported attributes we see little change. The keyword `targetArch` is simplified to `arch`. We also added the attribute and corresponding keyword `cname` that allows explicit specification of the name of the C function implementing the component or version thereof. By default, the name is automatically derived from component name and where applicable version name, but the ability to override the default has proven convenient in practice.

Other than renaming the introductory keyword from `edges` to `channels` the corresponding section shows little change. This aspect of our coordination DSL was already simplified drastically on the way towards Deliverable D1.1 and thus differs from external publications such as [30].

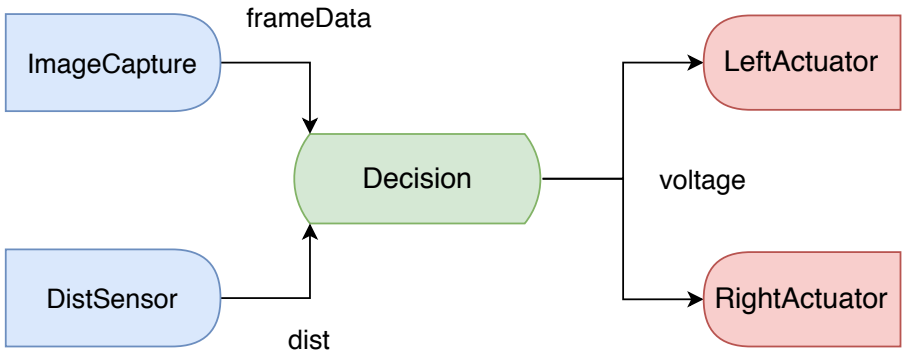


Figure 2: Example for TeamPlay component coordination (reproduced from Deliverable D1.1)

We illustrate the evolution of the TeamPlay language syntax by means of the same example we already used in Deliverable D1.1 and that is illustrated in Figure 2. Our example is an imaginary subsystem of a car with two sensors feeding messages to a decision controller. This decision controller synchronises the messages pair-wise and sends commands to two subsequent actuators. Figure 3 shows the corresponding TeamPlay coordination code using the revised syntax. For comparison with the old syntax we refer the interested reader to Deliverable D1.1.

2.2 Modes

One major theme of our work on the TeamPlay coordination language is dynamic adaptation to changing properties of the execution environment. For this purpose we extend the TeamPlay coordination language in multiple aspects. In Figure 4 we show the various extensions to the grammar of the core language as provided in Figure 1.

The first extension are so-called *modes*, that represent possibly changing relevant properties of the execution environment in the context of the coordination program. Following the new keyword *modes* we identify a sequence of mode specifications consisting of an identifier for the mode itself and its possible values as comma-separated list of identifiers enclosed in curly brackets. The notation is vaguely inspired by C enumeration types. All mode values must be

```
app car {

  components {

    DistSensor {
      outports { num dist }
    }

    ImageCapture {
      outports { frame frameData }
    }

    Decision {
      inports { num dist;
                frame frameData }
      outports { num voltage }
    }

    LeftActuator {
      inports { num voltage }
    }

    RightActuator {
      inports { num voltage }
    }
  }

  channels {
    DistSensor.dist -> Decision.dist
    ImageCapture.frameData -> Decision.frameData
    Decision.voltage -> LeftActuator.voltage & RightActuator.voltage
  }
}
```

Figure 3: TeamPlay coordination code for example of Figure 2 making use of the revised syntax

unique in order to let the coordination compiler identify the mode given any potential mode value. Mode values have no further interpretation than their names, but the textual order of definition induces a total ascending order that can later be used to compare any two values of a mode using the usual relational operators.

Mode specifications define a Cartesian product of possible states of the coordination environment, where each mode individually defines one independent axis or dimension. In practice, not all potential combinations of mode values may be useful or even permitted, but such restrictions are application-specific.

Modes can affect the coordination program in various ways, all introduced by the new keyword `if` followed by a *mode expression*. Mode expressions more or less follow the standard definition of Boolean expressions with operators inspired by the C language for negation, conjunction and disjunction. The usual precedences apply, and parentheses can be used to

```
CoordApp    ⇒  app Id { AppBody }
AppBody     ⇒  [ period FrequencyConst ]
               [ deadline FrequencyConst ]
               [ Modes ]
               [ Templates ]
               Components
               Channels
Modes       ⇒  modes { [ Mode ]+ }
Mode        ⇒  Id { Id [ , Id ]* }
Version     ⇒  version Id [ if ModeExpr ] Settings
Channel     ⇒  OneToOne | Broadcast | Select
OneToOne    ⇒  OutPort -> InPort [ if ModeExpr ]
Broadcast   ⇒  OutPort -> InPort [ if ModeExpr ]
               [ & InPort [ if ModeExpr ] ]+
Select      ⇒  OutPort -> InPort [ if ModeExpr ]
               [ | InPort [ if ModeExpr ] ]+
ModeExpr    ⇒  ( ModeExpr )
             |  ModeValueId
             |  ModeId RelOp ModeValueId
             |  ! ModeExpr
             |  ModeExpr || ModeExpr
             |  ModeExpr && ModeExpr
RelOp       ⇒  == | != | < | <= | > | >=
```

Figure 4: Grammar of coordination language extension for dynamic adaptation to coordination environment in EBNF form; grammar rules that are identical with those in Figure 1 are left out for conciseness

structure complex mode expressions. In practice, however, we expect mode expressions to be of rather simple nature. We support the usual six relational operators to relate a mode (identifier) as left operand to a mode value (identifier) as right operand. Using just a mode value (identifier) by itself is a short notation for checking the corresponding mode for equality with the given mode value.

The first application of modes can be seen in the context of multi-version components where modes can be used to restrict the choice of available versions. All further applications of modes affect the channels between components. A simple one-to-one channel can be conditioned on a mode expression. Likewise, individual right hand sides of a broadcast channel can be activated or deactivated via modes. A new kind of channel introduced in this context is the *select* channel

that routes a token to exactly one destination depending on the mode expressions. Unlike the broadcast channel that sends a token to any destination whose mode expression is met, the select channel evaluates mode expressions left to right and selects the first matching destination.

3 TeamPlay compiler extensions for fault-tolerance

This chapter details the extensions of the Teamplay language compiler, named CECILE, to implement the language extensions introduced in Deliverable D1.1. We commence by reviewing the existing TeamPlay compiler in Section 3.1. We then move on to describe the most relevant extensions of compiler front-end and compiler back-end in Section 3.2 and in Section 3.3, respectively.

3.1 Existing TeamPlay compiler

The structure of the TeamPlay compiler is illustrated in Figure 5. The compiler uses an XML-based configuration file to enable or disable specific passes in the compiler and allows the user to provide options specific to these passes. The compilation process can be divided into three phases: the compiler front-end, schedule generation and code generation. The compiler front-end consists of the syntactic and semantic analysis of the coordination file which is transformed into an intermediate representation for usage in subsequent phases. The scheduling policy generator uses the *Non-functional Properties File (NFP)* to obtain architecture-specific time and energy information per component. The last phase is code generation. The generated C code is compiled by a back-end compiler, and the resulting binary is linked with the component implementation object files and the TeamPlay runtime environment.

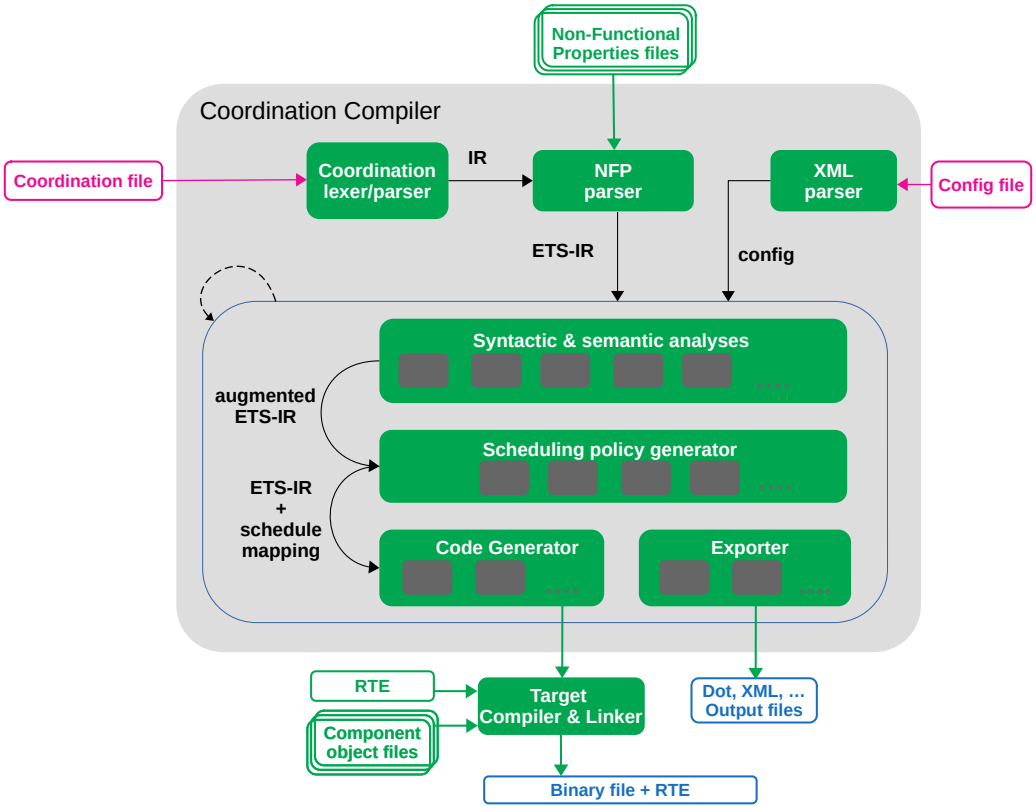


Figure 5: Overview of the TeamPlay compiler

3.2 Compiler front-end extensions

The compiler uses Xtext, a framework for developing domain-specific languages to generate the grammar for the coordination language [11]. The benefit of using Xtext is that it automatically generates syntax highlighting and auto-complete and bundles it into an eclipse plugin. The grammar is exported into a format that can be used with the ANTLR (ANother Tool for Language Recognition) runtime [1]. This runtime can be imported into various general-purpose programming languages to parse coordination files with the exported grammar. In our case, we use the ANTLR4 runtime with C++ to parse our language.

We extend the existing parser of the language to accommodate our language extensions for fault-tolerance as described in Deliverable D1.1 as well as the further language updates and extensions as described in Section 2. Implementing the specification of fault-tolerance methods requires extending the Xtext grammar and the parser. We add a profile class on each component which handles adding the options and the merging of profiles originating from different specification places such as the sub-network profile field, sub-network-inline, component profile field and component inline. Figure 6 shows an overview of cascading and inheritance of profiles.

First, the options from the sub-network are added (red box in the figure). These represent the weakest options and are overwritten by the specific, strong options. The profiles of the sub-network are inherited by the components inside. As mentioned in Deliverable D1.1, the profiles defined first in `profiles` are overwritten by those defined later. These options are overwritten by the inline options of the sub-network. We see a similar structure in the yellow box in which the component profiles overwrite and merge any settings defined by the sub-network. Finally, the inline component options overwrite the previously defined options. Options with the `vital` keyword cannot be overwritten. Fault-tolerance options with the `remove` keyword are removed in between merges/overwrites of options. This makes it possible to remove an option in the specification of the inline options of the sub-network and add it again in the inline specification of a component.

Sub-networks are implemented by dissolving them in the front-end phase of the compiler. Figure 7 illustrates this process using the example from Section 2. The result of this dissolving pass is Figure 2. First, we take a look at the `Sensors` sub-network. All edges that go to the sub-network ports are replaced with edges that go directly to the component the port is attached to. In this case, this means that the edge leaving the `ImageCapture` component is attached directly to the `Decision` component, the same goes for `DistanceSensor`. In the `Actuators` sub-network we essentially move the duplicate or broadcast edge from inside the sub-network to the outside, taking the components along. This algorithm becomes more complex when taking into account that it is possible to have broadcast edges both inside and outside the sub-network. This requires merging the broadcast edges into one.

Our implementations of TeamPlay modes is still on-going. The design of modes is carefully chosen to permit the compiler to statically compute projections of the described DAG, one per element in the Cartesian product of modes. This way modes can be integrated rather seamlessly into the compiler as all static analyses can still be performed for each combination of mode values.

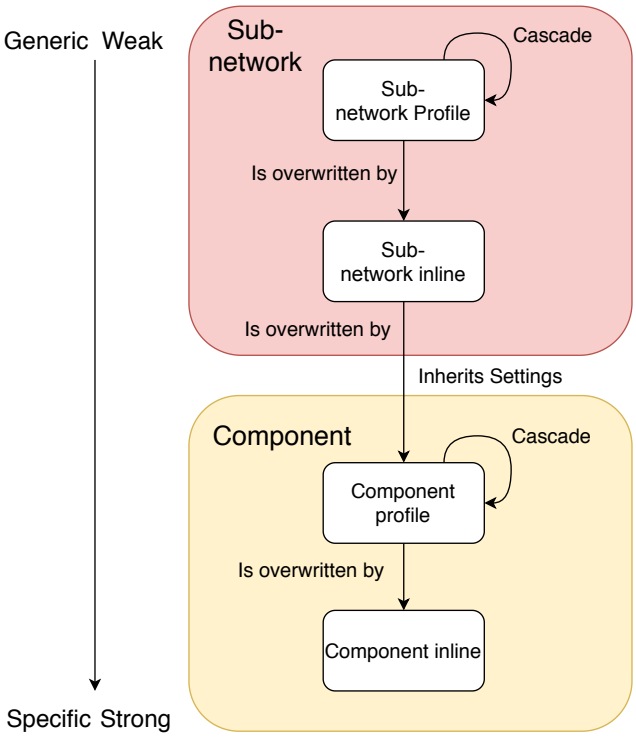


Figure 6: This figure shows the hierarchy of specifying options in different places. At the top of the picture, the specification is more general but also weaker since these options can always be overwritten by specification levels below it, unless the options is `vital`.

3.3 Compiler back-end extensions

The compiler back-end generates code targeting one of multiple runtime environments. These alternative runtime environments technically come as libraries. Hence, the compiler back-end's code generator combines the application-specific glue code with calls to the runtime environment library of choice. While our initial coordination compiler targeted a single rather generic Posix thread based runtime environment as an early proof-of-concept we have meanwhile added code generation support for three additional targets:

- a fault-tolerant runtime environment that implements the various fault-tolerance oriented extensions of the TeamPlay coordination language introduced in Deliverable D1.1;
- a real-time runtime environment running on top of commodity-off-the-shelf hardware and operating systems;
- the ADMORPH eXchange Format (AXF) serving the role of lingua franca among the various project partners and their tooling.

Each of the above runtime environments warrants its individual code generator. Typical areas which require application-specific glue code to be generated by our compiler are:

1. a representation of the coordination task graph of the application in question, as we need the flow of the data, dependencies, and (fault-tolerance) settings;

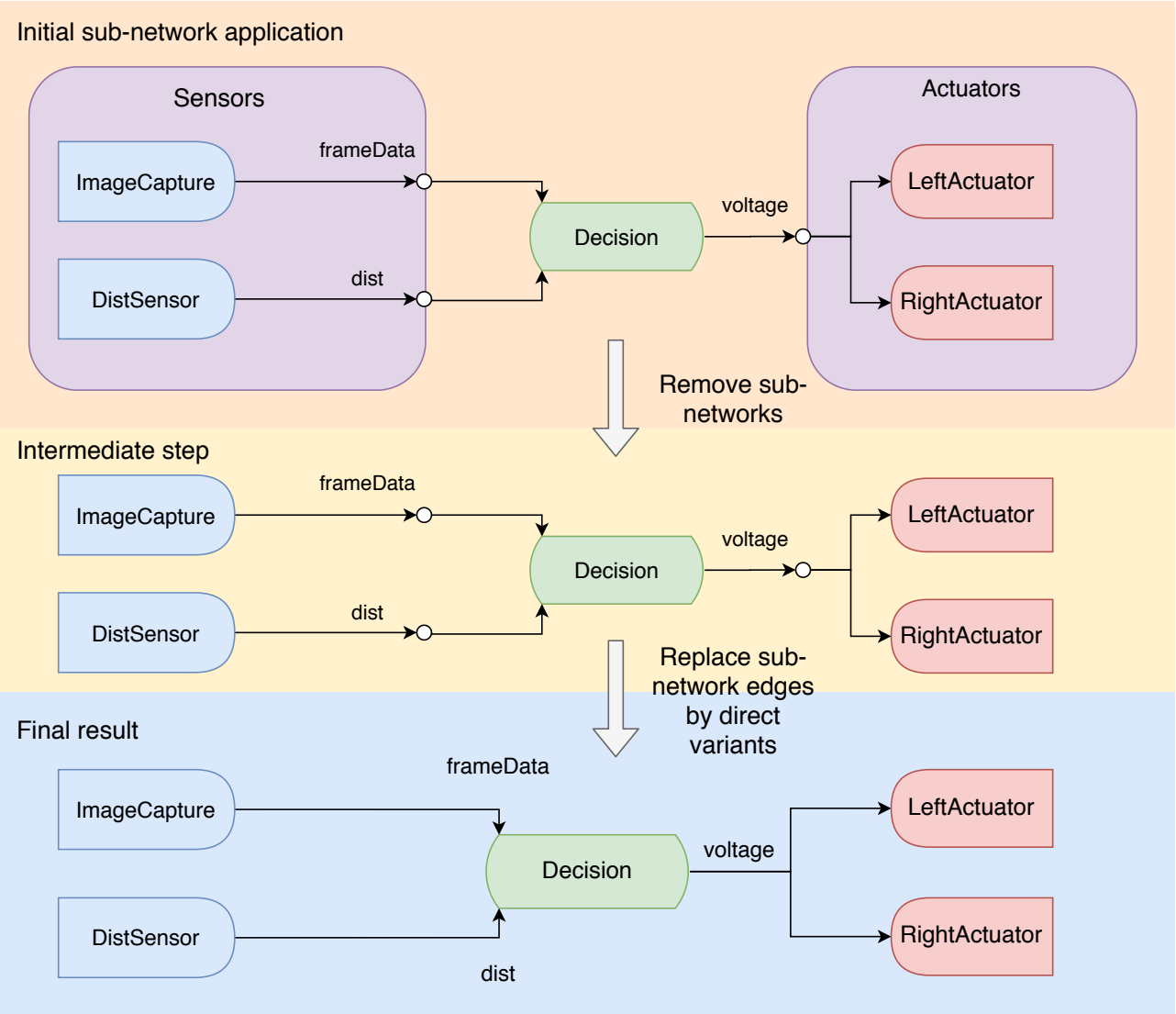


Figure 7: Steps to dissolve a sub-network. First the sub-networks are removed. Then the indirect edges that go the sub-network edge are replaced with edges that go directly to the attached component.

2. store the data into these buffers saved in the task graph;
3. take the contents of the buffers for when a component can execute;
4. generation of data structures to save the contents of the buffers for possible transport in the application;
5. call copy functions specific to the types of data tokens for each outport;
6. code to call the component-implementing C function associated with the component with the appropriate parameters taken from the buffers.

The choice of runtime environment and, hence, compilation target and code generator is made via the TeamPlay configuration file, see Figure 5.

4 Fault-tolerant runtime environment for TeamPlay

In order to support the various TeamPlay language extensions specifically geared at fault-tolerance that we already described in Deliverable D1.1 we designed and implemented a corresponding fault-tolerant runtime environment. Our proof-of-concept runtime environment dynamically reconfigures running applications upon detection of hardware failures. This runtime environment targets both permanent (or crash) faults as well as transient faults, for instance detected via n-modular redundancy. The runtime environment comes with a fault injection facility for demonstration purposes.

We commence with desribing the design of the base runtime environment prior to adding fault-tolerance capabilities and its various configuration parameters. This is followed by a discussion of fault detection facilities and the concept of error tokens that aims at preventing deadlocks in the execution of TeamPlay coordinated applications. At last, we go through the various fault-tolerance techniques, as pointed out in Deliverable D1.1 and sketch out their implementations on top of our base runtime environment.

4.1 Base system

The coordination approach we take requires us to make as few assumptions as possible about the underlying target hardware configuration as our coordination approach aims to be hardware architecture agnostic. We assume that the main property of CPS(oS) holds: CPS(oS) are distributed (possibly heterogeneous) systems which are not necessarily in the same physical location [5]. This means we deal with nodes of hardware components. We simulate these distributed systems using a thread for each node with POSIX threads or *pthreads* [6].

One of the first decisions we need to make is: in what way do threads in the simulator correspond to the real world? A straightforward idea is that each coordination component corresponds with a thread. This has the advantage that it is not necessary to manage the threads separately. Since the coordination task graph is static, each component knows which thread to communicate their output data to. In other exogenous streaming coordination systems like S-Net [16] (which targets HPC systems), having components correspond with threads is feasible. But when constructing a simulator for a CPS(oS) it is not realistic to assume that there are always sufficient hardware components to accommodate each coordination component separately. Hence we choose for an architecture in which the number of computation threads is static, related to the number of hardware components in the CPS(oS) but unrelated to the number of coordination components. This requires us to work with task queues, as each thread can execute multiple components.

We choose for a design in which there are two types of threads, a main or control thread and multiple computation threads. This design is illustrated in Figure 8. The control thread checks whether components are ready and puts the tasks into their appropriate queues. It is activated as soon as a component has finished computing and matches a centralised hardware component in a real world architecture. The choice for a centralised system is motivated by security concerns as it makes it more difficult to disrupt the entire system by taking control of a single computation node. In reality, this centralised system will have to be hardened against security faults. In order to deal with faults in this management hardware component, fault-tolerance methods such as primary-backup or n-modular redundancy can be applied. The computation

threads mirror the hardware components running the actual coordination component code from the real world.

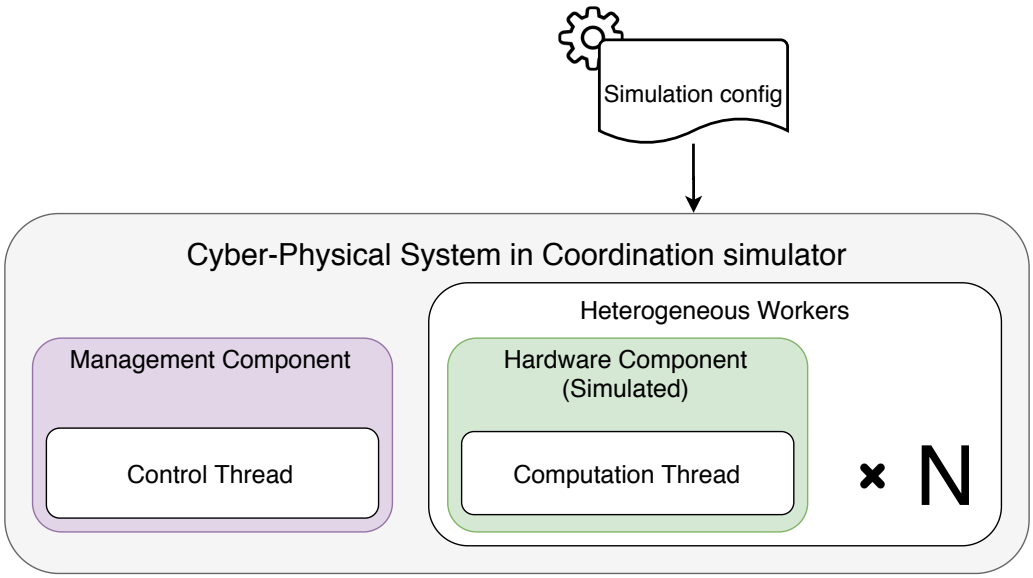


Figure 8: Illustration of the base architecture of the simulator. The management component is simulated with the use of a control thread. The heterogeneous worker elements are simulated by n computation threads.

4.2 Thread interaction

The interaction between the control thread and computation threads is displayed in Figure 9. The blue block on the left and green on the right indicate whether the action takes place in the worker threads or in the control thread. In the figure we show only one worker thread to highlight the interaction with the control thread. The figures in the middle show the shared data structures between the control and computation threads. The top data structure is the task queue associated with the thread. The middle figure is the coordination structure containing the coordination task graph. The bottom structure is the finished list which is used to communicate that a thread is finished and tokens are added to the buffers. The tables in the figure show the first four iterations on the simple coordination structure in the middle.

First we explain how the interaction works in an abstract manner, then we go over the example listed using tables in the figure. When the control thread launches (top right node), it goes through the source nodes and adds them to the task queues of the assigned threads. Each of these threads has a counting semaphore which corresponds to the number of items in their queue. When the semaphore reaches zero, the threads wait until new items appear in the task queue.

When a thread is alerted that new items appeared in the task queue (top data structure), it pops a component from the task queue (task queues are FIFO) to execute. After execution, it stores the output data in the graph data-structure and appends the id of the executed component into the finished list. The control thread is alerted that components are finished, so it can check whether new items can be added to the task queues. The computing thread

will then wait for the task queue semaphore. If the semaphore's value higher than zero it can continue popping another item from the task queue to start computing again.

After items are added to the task queues of the threads and the threads are alerted, the management thread will wait until items appear in the finished list. This is indicated in the figure by the bottom data structure with the dotted line facing right. This mechanism is implemented with a condition variable as one cannot reset a counting semaphore when the finished list is emptied. When items appear in the finished list, we need to check which components can fire again. First, we need to check whether the predecessors of the finished component can fire since, by firing, it can have opened a spot in the (bounded) FIFO buffers of the predecessors. Then, we check whether the successors of this component can fire since it has produced a token on its outports which may trigger the firing rule of the successor. Finally, we check whether the component itself can fire again. This way of checking ensures we only have to traverse the parts of the graph that have been changed. The components that can fire are added to the task queues belonging to the threads and the threads are alerted so they can continue computing. The components that are ready are added to the task queue. This marks the completion of a cycle.

Now we will explain the example found in the figure. First, both threads will launch. The worker thread sees that there are no items in the task queue (i.e., semaphore value is zero) so it will wait. In the first cycle, the control thread adds the `Source` component to the task queue. As the source component does not have any dependencies, it can fire as long as the buffers can hold the data and it is not already present in any task queue. The task is put in the task queue associated with the computing thread to which `Source` is assigned. The control thread will increment the semaphore. This leads to the awakening of the worker thread, which will pop `Source` from the FIFO queue. The worker thread will then execute the code associated with `Source` component. After computation, the output token of `Source` will be added to the buffer on the edge leading to the next component, `A`. The computing thread puts the id of the `Source` component into the finished list and sends a signal to the condition variable on which the control thread is waiting. The computing thread loops back to the first item (after initialisation) and will wait until the control thread has added new items to the task queue owned by the worker thread.

When the control thread receives the signal for the condition variable, it will loop trough the finished list and check the task graph for components which are ready. This is done by looping trough the predecessors, successors and the component itself, to see if they can fire. `Sink` has no predecessors but it does have one successor, `A`, which can fire since `Source` just fired. `Source` can also fire again. The components which can fire again are put into the task queue. Next, component `A` is fired, the result is again stored in the buffer after the fired component, this time leading to `Sink`. Then the control thread is again alerted that the worker thread has finished a computation. The control thread notices that `Source` can be fired since it has no dependencies, but it is already in a task queue, so it cannot be added again. Following the execution of `A`, `Sink` can be fired, but `A` has insufficient tokens from `Source` to fire again. Now, `Source` is taken from the task queue and executed, as it was added the previous cycle. The component checking process of this cycle is identical to the first cycle, as `Source` and `A` are added again. Then for the last round explained in this example, `Sink` will be popped from the task queue and executed. In the control thread, `A` cannot be added to the task queue again since it was already added when `Source` finished. `Sink` cannot fire again since the buffer on the edge coming from `A` does

not have sufficient tokens.

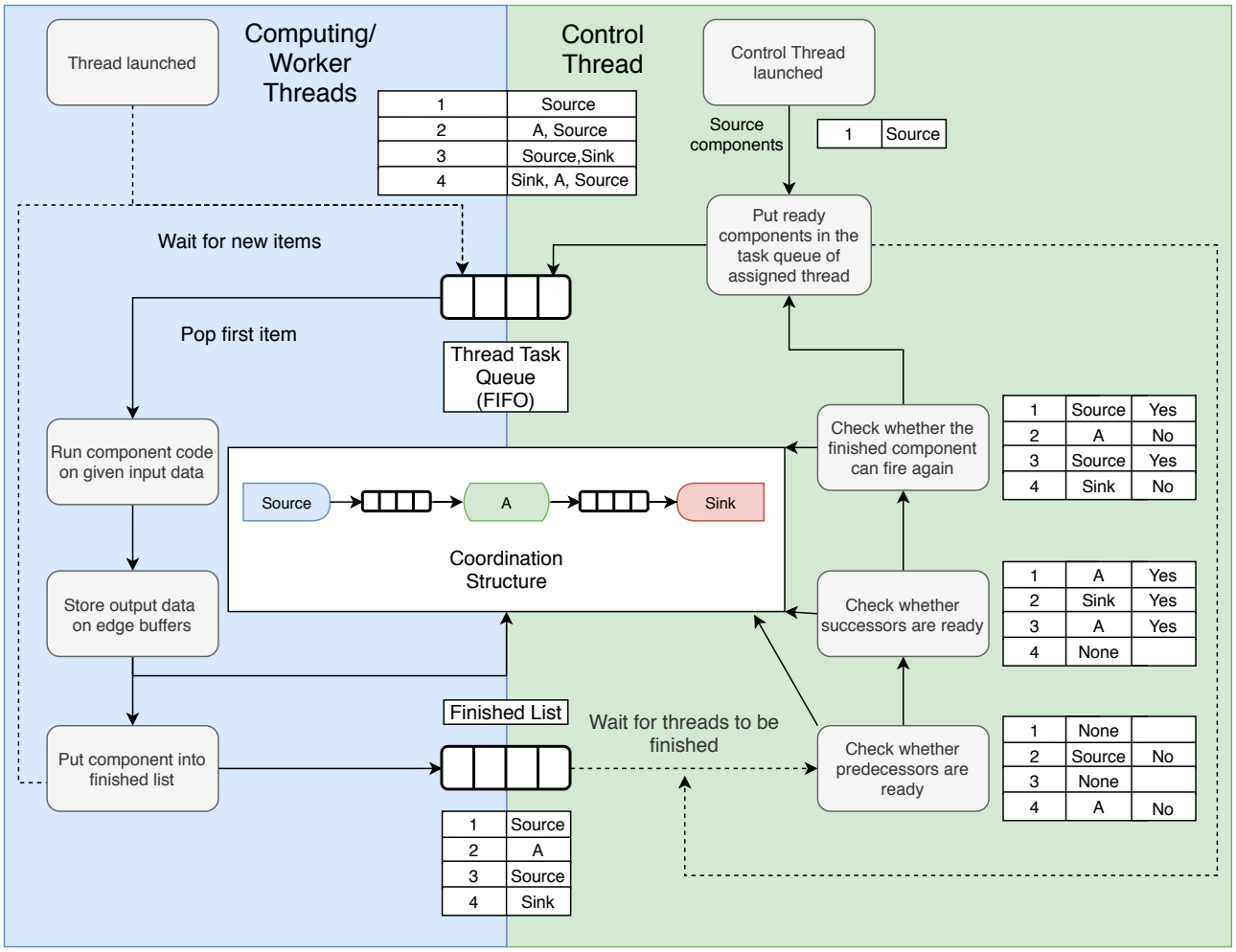


Figure 9: Schematic overview of the interaction between the computation components and the management component. The dotted lines indicate a waiting process via thread communication (e.g., condition variable or semaphore). The double-column tables show the components in the list per iteration. The triple-column tables show the components which are checked for firing, the third column shows whether they are added to the task queue.

4.3 Configuration file

Our simulation run-time uses a configuration file in which the user can specify options such as the number of threads and options related to fault-tolerance. Figure 10 shows an example of a configuration file. `numThreads` signifies the number of computation threads. Setting `debug (cont.)` to `true` turns on debug prints. `sleepTime` is the period of the heartbeat worker threads in microseconds. `controlSleep` is the period of the heartbeat control thread in microseconds. `heartbeatTries` is the threshold of the counter incremented by the heartbeat control thread, if the counter is higher than `heartbeatTries`, it is deemed to have crashed. `heartbeatCheckerPrio` and `heartbeatWorkerPrio` are the real-time scheduling priority of the control heartbeat thread

and heartbeat worker threads respectively. Setting `standbyEarlyTaskCompletion` to `true` allows standby threads which start computation after the primary thread (i.e., first finished thread) has finished to skip computing since the task is already delivered. The `edgeBufferSize` indicates the number of tokens an edge buffer can hold.

```
numThreads = 6
debug = false
sleepTime = 100
controlSleep = 1000
heartbeatTries = 10
heartbeatCheckerPrio = 10
heartbeatWorkerPrio = 15
standbyEarlyTaskCompletion = false
edgeBufferSize = 20
```

Figure 10: Simulator configuration file example.

4.4 Fault-Tolerance

In order to implement the suggested fault-tolerance mechanisms, we make several additions to our base simulator. We start out by describing how we detect crash faults. This is necessary as two out of our four chosen fault-tolerance methods, checkpoint/restart and primary-backup, require error detection mechanisms that trigger these fault-tolerance methods when a fault occurs. When a system suffers from a crash failure, we cannot assume that restarting the hardware component will solve the problem. As depending on the hardware configuration, data can be stored in non-volatile storage, which is lost after the restart. To prevent deviation from correct service caused by lost data, we introduce error tokens. For the reconfiguration process, mechanisms are required which reassign tasks to other threads with a compatible architecture. Furthermore, we introduce a mechanism to deal with heterogeneous architectures and spatial assignment of components to threads.

4.5 Crash fault detection

There are three major error detection methods for detecting node failure in distributed applications [36, 19]. The first of these methods is heartbeat, in which we actively ping the nodes to know whether they have not crashed. In the second method, one passively waits for messages to come in. This method is not predictable and thus not suitable for future real-time extensions. The third method is to provide a challenge-response protocol in which the node calculates a response to a provided input. This last method can also deal with value faults in addition to detecting node failure. We choose heartbeat to deal with crash faults in our simulator as we predict this is the most predictable method with the least amount of overhead.

In order to implement heartbeat, we need a way of checking whether a computation thread has crashed. This is not straightforward, as using a signal and signal handler on an individual thread does not guarantee which thread handles the signal [23]. Furthermore, we aim to treat

the components as black boxes, so we cannot intrude into the user code to send the signal periodically from there. To solve this problem, we add a separate heartbeat thread to each computation thread. The heartbeat and a computation thread in the simulator, correspond with one hardware component in the real world. We assume that the entire (real-world) hardware component dies at once, so if the computation thread dies, the associated heartbeat thread dies with it. Additionally, we introduce a main heartbeat thread or heartbeat checker thread which periodically checks whether the heartbeat threads are still alive. The checker does this by periodically looping trough the threads, incrementing a counter in memory shared with each heartbeat thread, followed by a sleep. After the increment of the counter, the heartbeat checker checks whether the counter is higher than a specified threshold, if it is higher, the worker heartbeat thread (and by extension the hardware component) is deemed unresponsive, i.e. it has crashed or is hanging. The worker heartbeat thread periodically resets the variable incremented by the checker thread, this is also followed by a sleep.

With this mechanism, we have to take care that we do not get any false positives, i.e., threads that are detected as having crashed but are still alive. Since the scheduling of the threads in the system running the simulator does not guarantee that the heartbeat worker threads wake up immediately after their sleep, we enable the user to set the amount of sleep each thread type takes as well as the threshold used by the heartbeat checker. This approach requires us to balance these three values. Choosing the worker thread sleep time too close to the heartbeat checker sleep time, will increase false-positives but will decrease the time it takes before an error is detected. Lowering the threshold has the same result, the lower the threshold, the faster errors are detected, but it also increases the chance of false-positives.

In order to further decrease the amount of false positives, we changed the scheduling type of both types of heartbeat threads to use real-time scheduling policies supported by pthreads [22]. `pthread_setschedparam` supports two scheduling policies: FIFO and round robin. FIFO scheduling (`SCHED_FIFO`) runs a thread to completion in first-in-first-out order. Round robin scheduling (`SCHED_RR`) aims to give each thread an equal execution time, but involves a larger number of context switches. FIFO scheduling does not work in our system because the heartbeat threads are continuous tasks. Thus, we enabled round-robin scheduling on both types of heartbeat threads. Additionally, we enable the user to set the priority of the heartbeat checker thread and heartbeat worker threads as can be seen in Figure 10. In our case, choosing a higher priority for the heartbeat worker threads lowers the chance of having false positives, since it lowers the chance that the worker moves after the heartbeat control thread.

4.6 Error tokens

Channels in the coordination language do not only signify data streams, but also dependency relations between components. When a thread fails and cannot be recovered, the components that follow miss tokens as the computation failed and the input tokens are lost. We explain this process and solution using the simple coordination application illustrated in Figure 12. This example contains 4 components signified by a letter. Component *Source* produces a pair of numbers and distributes them over its two outports. These outports lead to component A and B respectively. The paths from components, A and B join towards component *Sink*. What is important in the example is that the pair of numbers stay aligned, i.e. that the halves of the pair created in *Source* are consumed by the same execution of *Sink*.

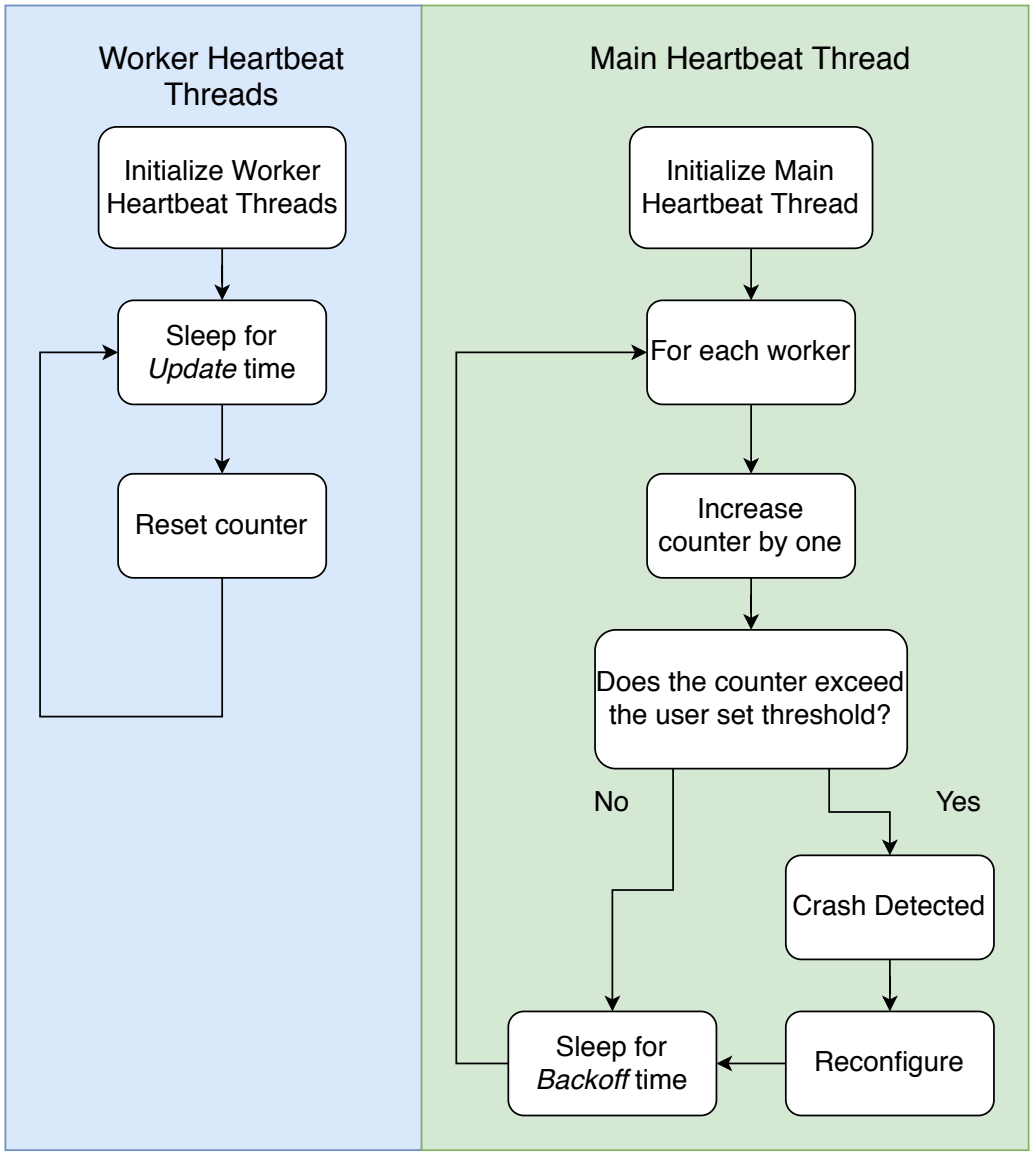


Figure 11: Overview of the heartbeat error detection mechanism. The left, blue coloured area in the image details the procedure for the worker threads while the right, green coloured area details the main heartbeat thread which checks the other threads.

When component B crashes and the input data is not recoverable, B cannot fire. This introduces a problem since component *Sink* depends on the crashed component B. Since B cannot fire, we cannot reassign it to another hardware component. This causes the pair of numbers to be misaligned, resulting in non-desired behaviour. One could argue that this kind of behaviour works for some applications, in which alignment doesn't matter but in the general case, misalignment can cause service failure. Simply skipping one firing of component *Sink* would work in this case but if there was another branch earlier in the application that leads to after component *Sink*, then this later branch would be misaligned with the result of component *Sink*.

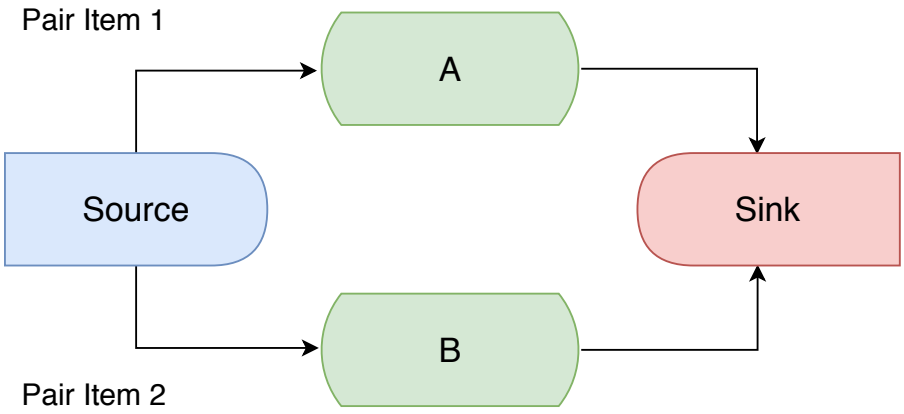


Figure 12: TeamPlay Coordination graph illustrating a branch. Component *Source* produces a pair of numbers sent to components A and B respectively. A and B are then synchronised with each other by *Sink*. If a thread executing either component A or B would crash without fault-tolerance specified one needs a mechanism to prevent misalignment to occur between the token going to the top-branch and the bottom-branch. For this we use error tokens.

Our solution to this problem is the introduction a fault-tolerance method, named error tokens. These error tokens indicate that one of the previous dependence relations could not be satisfied, i.e., that the following components cannot be executed without misalignment. When a thread executing a component encounters an error token it will skip the computation and produce sufficient error tokens on all outports of the component. This error token will propagate through the entire graph, invalidating the tokens that result from the same source component firing, i.e., invalidating one iteration of the application. In our example, this would mean that all numbers produced in a single firing of *Source* would be invalidated with a crash.

4.7 Checkpoint/restart

Checkpoint/restart can be implemented in the coordination language by checkpointing the FIFO buffers on the edges between the components, as the state of the entire application resides in these buffers. This is done in practice by adding an extra buffer on each outport that leads to a component. After the execution of the previous components, (i.e., the dependency components), copies of the output tokens ares made. For primitive types, this is an easy task but for user-defined types for which only a pointer is passed, the user needs to provide a copy function. When the thread executing the component fails, a new structure of input tokens is created from the checkpointed buffer and assigned to the task which takes place during the rejuvenation phase. The entire rejuvenation process, in which checkpoint/restart plays a role, is illustrated in Figure 14. We will visit this figure in full during the rejuvenation section. In normal operation, we need to remove the checkpointed data upon finishing execution and delivering the output, in order to prevent the buffers from overflowing.

4.8 Primary-backup

In primary-backup, a standby component takes over the main component when a failure is detected. Usually, this is done directly as the backup component synchronises with the active component to ensure a quick switch. In our coordination language, we do not need this behaviour as, again, the state of the application is completely saved in the FIFO buffers. We assign copies of the input tokens of the component to a number of threads equal to the number of replicas. The first thread that finishes the computation actually delivers the output. If a thread starts the computation after another thread has already delivered its answer, the thread starting the second computation can skip the task. However, it is unlikely that this behaviour is schedulable on real-time systems. Thus, we build an option into the simulator whether this form of task completion is allowed. Disabling this setting gives us the worst case, all threads compute even if the task is already delivered. When this setting is enabled, the task will only be computed multiple times if the backup threads start the computations while the thread that will finish the earliest has not yet finished. Again, the primary-backup rejuvenation process is illustrated in Figure 14.

4.9 N-version programming & N-modular redundancy

Due to time constraints we were not able to implement N-version programming and N-modular redundancy into into the simulator. We aim to explain a possible implementation of these methods into the simulator. Here we explain how N-modular redundancy can be implemented since N-version programming is a simple extension of N-modular redundancy in which you run different versions instead of identical processes.

N-modular redundancy requires more control over the processes compared to primary-backup since NMR utilises a voting process at the end. What we can use from the primary-backup system, is that these processes do not have to execute at the same time. When all processes are done or when one of the threads executing the processes has crashed or sustained a timeout conform the coordination settings, the voting process is executed on a designated voter node. This voter node requires a copy of all output tokens in order to execute the majority voting process. Our application requires that the output of NMR is a single answer, since the next component may not have NMR specified. This is why we cannot have a voter array without having an extra voting step afterwards to choose one correct answer from the voter replicas.

4.10 Component assignments & heterogeneous architectures

In order to simulate heterogeneous rejuvenation, we present an extension to the base system. Before we can create the rejuvenation mechanism we need to know which thread can be reconfigured to do which tasks. For this we need a mapping which defines which thread can simulate what component.

In time-critical systems, TeamPlay components are assigned in both the time and space dimensions to a hardware component. As the timing dimension is out of the scope of this work, we introduce a spatial assignment of the components to threads. This way, the system provides the minimum to test rejuvenation mechanisms for our fault-tolerance methods.

In addition to this, we introduce the concept of a thread class. This class mirrors a specific hardware architecture (as threads mirror hardware components) in heterogeneous systems. When a component needs to be reconfigured because it has crashed, it needs to be assigned to a component of the same class to ensure the software (e.g., component code) is compatible. This is a simplified model as in reality it is sometimes possible to run components on hardware in a different class, e.g., on a faster CPU (provided that it is compatible, as GPU code cannot run directly on a CPU). The mapping of components to threads are passed to the simulation run-time. It is not embedded in the coordination language as the number and types of threads are hardware architecture-specific.

4.11 Memory sharing

Usually, using threads on a system means that data is passed using shared memory. In cyber-physical systems (of systems) in the real world, this is generally infeasible. In our runtime, this raises the question about where the input and output data of the components should be stored. For this, we aim to solve the problem in a conceptual manner. First of all, we want to keep the strain on the internal network that connects the hardware components low. Thus, we aim to avoid sending the full content of the tokens back to the management thread for storage as it can be a CPU and network bottleneck. This is especially true with larger data-structures, e.g., images and videos, which are common in CPS(oS) [30]. This requires us to save the data on the hardware components that execute the component code and let others that need the data request it over the network.

However, this proposed solution causes problems with fault-tolerance methods like checkpoint/restart, as they require a copy of the data. This copy cannot reside on the same node, as it is not guaranteed that the data is in volatile storage and it is not guaranteed that the system can successfully restart, for example in the case of a physical failure. Our solution to this problem is the assignment of a memory-companion to each node. This memory-companion holds a copy of the checkpoint from the checkpoint/restart fault-tolerance method.

4.12 Rejuvenation

Strategies that deal with crash faults, checkpoint/restart and primary-backup require this mechanism as it is not guaranteed that crashed nodes operate normally when restarted.

Our final simulation architecture can be found in Figure 13. A hardware component from the real world is mimicked by two threads, a computation thread and a heartbeat thread. These hardware components form a group of heterogeneous workers, managed by the management component. The management component consists of two threads. The control thread checks and adds components to the task queue of the worker threads. The main heartbeat thread monitors the heartbeat threads associated with the workers and launch the rejuvenation process.

The rejuvenation process is illustrated in Figure 14. The process can be split into two parts: the invalidation and recovery of the task queue of the crashed thread and the reassignment of the threads' assigned components. The path of the rejuvenation process depends on which fault-tolerance mechanisms are specified. First, we explain the middle path which is taken when no fault-tolerance method is specified on the component. On this path, error tokens are produced on the outports of the components in the task queue. The component is marked as finished

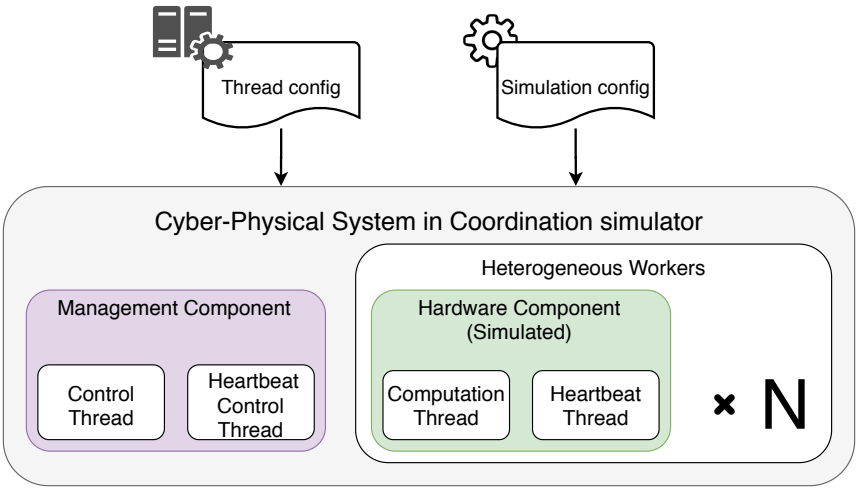


Figure 13: Illustration of the simulation architecture. The thread configuration and simulation configuration are input for the simulation runtime for the component to thread assignment and simulation settings respectively. The cyber-physical system has four thread types in two different component types. The management component consists of a control thread and a heartbeat control thread. The control thread which checks whether components are ready. The heartbeat control checks whether the heartbeat threads are still updating. The n heterogeneous worker threads consist of computation threads and heartbeat threads. The computation threads or worker threads execute the component code and interact with the control thread. The heartbeat thread updates a variable to show to the heartbeat control thread that it is still alive.

as the computation could not be saved by a fault-tolerance method. Then we arrive at the rejuvenation process. This rejuvenation process works by finding a non-crashed thread in the same architecture class with as extra condition that it is the thread with the fewest assigned components, to prevent from one thread taking all crashed tasks, consequently becoming a bottleneck for the application. We do not produce error tokens if a source component is present in the task queue of the crashed thread as it can simply fire again since it does not have any input tokens that need to be invalidated.

In checkpoint/restart the computation can be saved by re-running the task with the checkpointed input tokens. Before this can occur, we need to reassign components to a different thread, before the checkpointed data can be rerouted there. This rejuvenation step is the same as without fault-tolerance methods except that with checkpoint/restart, tasks exist that could potentially be saved. Note that a task cannot be saved with checkpoint/restart if it crashed twice.

In primary-backup, a component is assigned to multiple threads. The number of threads depends on the number of replicas defined in the coordination language. During the invalidation of the task queue, we need to check whether a task has been delivered or not. If it has been delivered then the computation does not need to be saved or invalidated. However, if a task has not been delivered we need to check if the crashed thread is the last replica of this task. If it is the last, the computation cannot be saved and we execute the same steps as without a fault-tolerance mechanism. If there are still replicas assigned to this task we do not have to do anything since they can deliver the task as they have copies of the input data, assuming

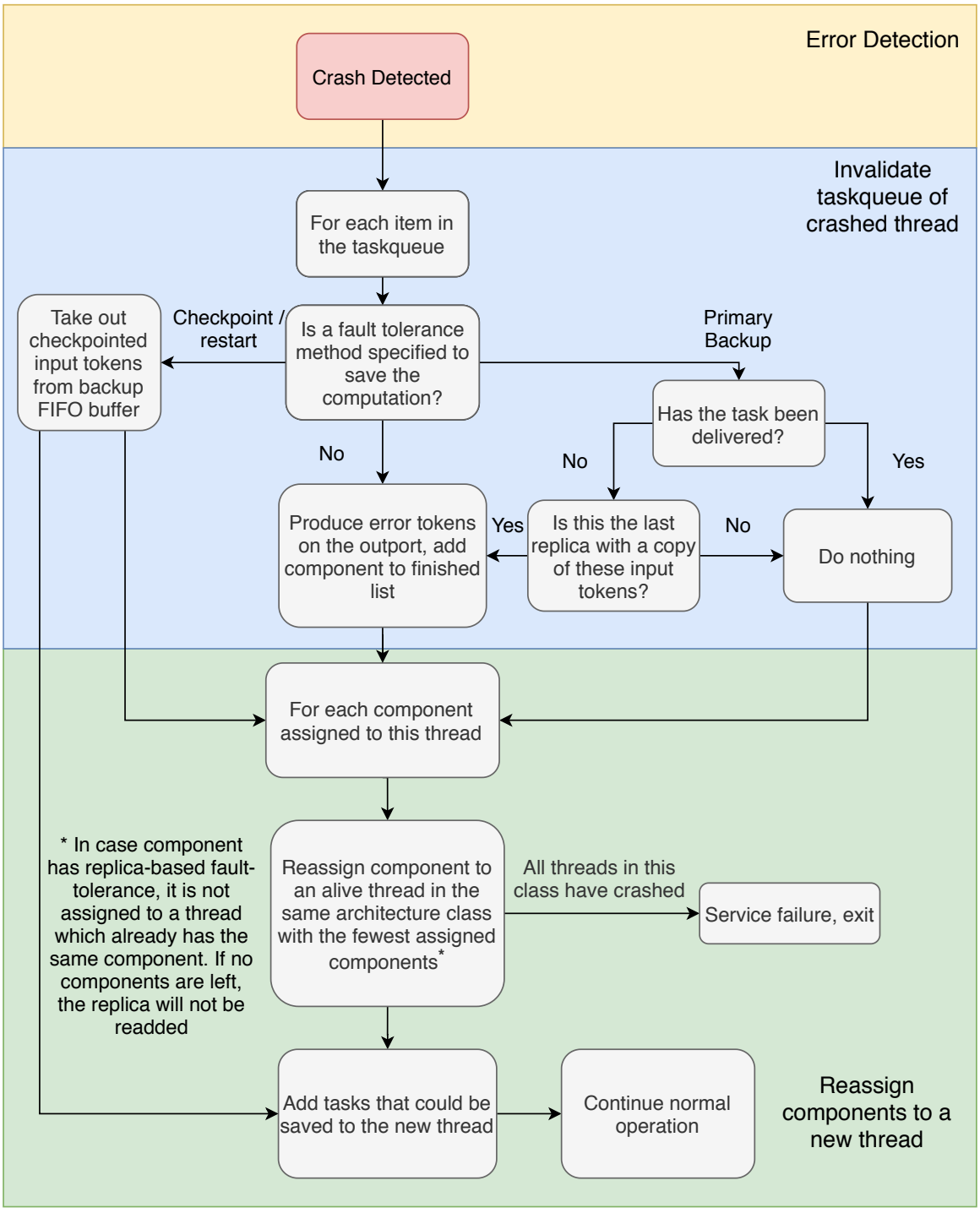


Figure 14: Illustration of a crashed thread rejuvenation mechanism. The process can be split into two parts. The first part is the invalidation of the task queue of the crashed thread (the flowchart contained in the blue box up top). The second part is the reassignment of the threads assigned components which is contained in the green box below.

that they do not also crash. In primary-backup, we add an extra requirement for finding a new thread to which the task can be reconfigured to. This requirement is that there cannot

be multiple assignments of the same component to the same task, as it defeats the purpose of primary-backup. If there are no threads left that follow these requirements the replica is not reassigned.

In our simulation, the task queue is in shared memory as the management component needs to assign new components with input data to the task queue of the thread. In a real system, you would send the task, (i.e., the component id) and where to find the input data. However, we also need to save this information on the management thread. During rejuvenation, we require a copy of this task queue in order to reassign the components of the crashed thread.

5 YASMIN: Yet Another Scheduling MIddleware for explorati0N

Commercial-Off-The-Shelf heterogeneous platforms provide immense computational power, but are difficult to program and to correctly use when real-time requirements come into play: A sound configuration of the operating system scheduler is needed, and a suitable mapping of tasks to computing units must be determined. Flawed designs may lead a sub-optimal system configurations and thus to wasted resources, or even to deadline misses and failures.

We propose *YASMIN*, a middleware to schedule end-user applications with real-time requirements in user space and on behalf of the operating system. *YASMIN* provides an easy-to-use programming interface and portability. It treats heterogeneity on COTS heterogeneous embedded platforms as a first-class citizen: It supports multiple functionally equivalent task implementations (versions) with distinct extra-functional behaviour.

The work described in this section is a synergy between the ADMORPH project and the Horizon-2020 project TeamPlay *Time, Energy and Security Analysis for Multi/Many-core Heterogeneous Platforms* (2018–2021, grant agreement no. 779882).

This work has led to a publication in MIDDLEWARE 2021 [31].

5.1 Application model

We consider non-safety-critical real-time systems composed of a set of tasks where each task represents an indivisible (or atomic) feature of the end-user application. To embrace heterogeneity we adopt recent task models representing each task with a set of versions ($v \in V_t$) [30], or variants [18]. All versions of a task expose the same interface (i.e. inputs, outputs), but each version has its own worst-case execution time (WCET), energy consumption, etc. In other words, all versions are functionally equivalent, but may exhibit distinct extra-functional behaviour. Versions can be created using different compilation flags, targeting different architectures or coming from different implementations.

The immediate motivation for multi-version tasks lies in the usually different ISAs between computing cores forming a heterogeneous platform. Given the complex architectures of the platforms we target, it is commonly not a-priori decidable which tasks should exclusively run on the CPU and which should exclusively run on one (or more) of the various accelerators. Consider the example of an application with at least two tasks A and B, where each task has two versions: one running only on a CPU core and one using a GPU. These two tasks are independent and exhibit the same timing properties, i.e. period. Hence, they could potentially run in parallel. On the targeted platform, however, only one GPU is available. Therefore, both versions of A and B targeting the GPU cannot execute in parallel. However, the presence of different versions allows us to run the GPU version of A at the same time as the CPU version of B, or vice versa.

We empirically demonstrated in [29] that deciding which version to execute at each instance of each task is not straightforward. This question is rather part of the overarching addressed scheduling problem, and it is common that depending on global circumstances and objectives, the same task may sometimes preferably be executed on the CPU and in other cases on the GPU, see [30] for details.

The versatility of multi-version tasks goes even beyond the above: They can likewise provide

task implementations particularly optimised for execution on a specific hardware unit, even in the presence of generic ISA compatibility. Furthermore, application designers could easily play with implementation variants that expose different non-functional behaviour (e.g. energy, time, security) and let YASMIN select the one best suited under concrete context and objectives.

YASMIN supports sporadic tasks as well as periodic tasks. The minimal time interval T (or period) separating two consecutive activations of a task is provided to our scheduler. We also allow aperiodic tasks where activations are the responsibility of the end-user, as no regular pattern can be given to the scheduler. Real-time tasks must complete their execution before a deadline D relative to the period. We support the three main deadline schemes: implicit ($D = T$), constrained ($D \leq T$) and arbitrary to the period.

YASMIN further supports tasks divided into subtasks with precedence constraints, thus forming a task graphs. We only deal with Directed Acyclic Graphs (DAG). Other graph-based task models, such as Synchronous DataFlow (SDF) [20], must *a-priori* be transformed (or expanded for SDF) to comply with a DAG task model. Each edge in a graph represents a causal dependence between two tasks. This causal constraint may be a data dependency, or it can be used to prevent side-effects between them. The source node of an edge must complete its execution before activating the sink. As in most graph-based task models, YASMIN supports activation patterns and relative deadlines described at the graph level: The whole graph is considered sporadic or periodic. Only the root node (which has no predecessors) is activated at an activation event triggering all subsequent nodes while the leaf node (which has no successors) must complete before the deadline. When dealing with graph-based task models, YASMIN considers versions attached to nodes of the graph, i.e. subtasks.

5.2 YASMIN design & implementation

We designed YASMIN as a library to be compiled individually and linked to the end-user program. YASMIN is highly modular and allows (1) the use of various scheduling policies and (2) easy switching between them at compile time using macros. The internal structure of our library adaptively morphs to meet the requirements of each scheduling strategy through macro definitions given in a configuration file.

We implemented YASMIN in structured C-code following real-time coding guidelines to enable the use of WCET analysis tools, such as AbsInt's aiT [12] or Heptane [17]. We systematically refrain from using dynamic memory allocation, and loops are statically bounded. To accomplish this, we make use of C-header configurations to define constants used throughout the library, e.g. the number of threads or the number of tasks.

YASMIN is compatible with any POSIX compliant OS. However, we also rely on the non-POSIX *pthread_set_affinity_np* function that binds a thread to a specific core. Similar requirements can be found in previous works [26, 33].

5.3 YASMIN API

The library is configured at compile time using a configuration file. In this file, pre-processor definitions set, among others, the type of scheduling, the type of mapping and the priority assignment. Each different scheduling strategy requires different mandatory information to perform adequately. The code of the different functions of the API is morphed to adapt its

behaviour following the configuration, e.g. information to declare a task differs between off-line and on-line strategies. The configuration is applied to the whole compiled binary, only one scheduling policy is allowed at a time. In order to switch to another policy the application must be recompiled with new parameters.

Table 1: Full API of YASMIN

struct TData { char *name, u64 period, u64 deadline, u16 virt_core_id, u64 release_offset}	Structure to describe a task. Some fields are optional depending on the configured scheduling policy.
void init(void)	Initialise the coordination runtime library.
void cleanup(void)	Wait for all worker threads to finish and close.
bool start(void)	Start to execute the tasks of the application.
void stop(void)	Stop pushing new tasks into the ready queue. All tasks already pushed will be executed.
TID task_decl(TData *d)	Declare a task to the scheduler.
void task_activate(TID t)	Activate a non-recurring task for immediate schedule.
VID version_decl(TID t, FuncPtr f, void *f_static_args, VSelect props)	Add a version to the task with user specific properties.
HID hwaccel_decl(char *name)	Declare a hardware accelerator
void hwaccel_use(TID t, VID v, HID a)	Declare a hardware accelerator used by a task version.
channel_decl(cid, datatype, size)	Macro to declare a channel of type *type* identified by *cid* containing *size* items of type *datatype*.
channel_connect(TID src, TID dst, cid)	Macro to connect a source and a destination task using the specified channel identified by *cid*.
channel_push(cid, datatype d)	Macro to push a value of *datatype* in the FIFO identified by *cid*. To be used in user function body.
channel_pop(cid, datatype *d)	Macro to pop a value of *datatype* in the FIFO identified by *cid*. To be used in user function body.

Table 1 presents the API of YASMIN. All functions are prefixed with `yas_`, which we left out in the paper for conciseness. This API is common to all scheduling strategies, allowing for an easy switch at compile time with modifications of the user code if all information are provided.

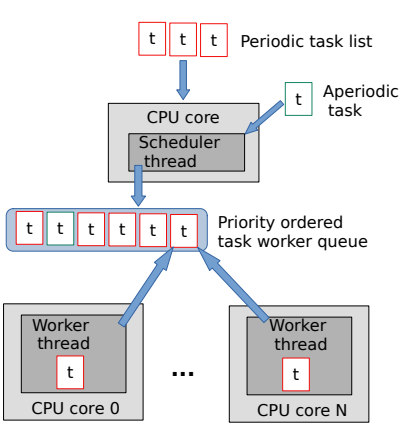
The end-user program must first call the *init* function that initialises different structures of

our library. Then, the user must declare the various tasks using *task_decl* and their associated versions with *version_decl*. See Section 5.4 for details.

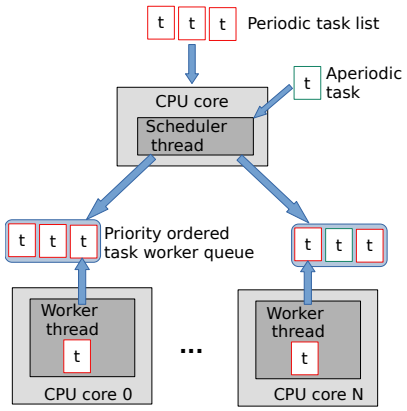
YASMIN supports graph-based tasks. We provide a mechanism to declare and manage FIFO channels required between causally dependent tasks within a graph. The pre-processor macro *channel_decl* defines the FIFO channel buffer. Connecting two tasks to use this channel is done with *channel_connect*. The channel can be accessed from within user tasks with the *channel_push* and *channel_pop* functions. With graph-based tasks only the root nodes need to have a period attached. Subsequent nodes are automatically activated by the scheduler, once all required incoming data are present in their input channels.

Hardware accelerators can be declared with *hwaccel_decl* and linked to a task version with *hwaccel_use*.

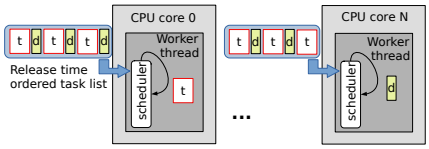
At this stage no user code has yet been executed, and no scheduling has been performed. It is after the call to *start* that the scheduler starts to run the application. Calling the *stop* function stops the scheduler. Then, either the main program performs the finalisation of the application with *cleanup*, or the schedule can be resumed with a new call to *start*. It is only possible to alter the task set while the schedule is not running, hence enabling multi-mode scheduling[15]. Functions to alter the task set are for conciseness removed from the following API tables.



(a) Global on-line scheduling strategy. The ready task queue is shared among worker thread.



(b) Partitioned on-line scheduling strategy. A scheduler thread pinned to another core feeds each worker thread ready task queue.



(c) On-line dispatcher with an off-line scheduling strategy. Each worker thread is pinned to a core, and a scheduling loop iterates on its ready task queue.

Figure 15: Overall architecture for each scheduling class.

5.4 Heterogeneity & multi-version components

With embedded platforms hardware accelerators are usually a scarce resource, i.e. there is typically only one GPU. If multiple tasks need access to the accelerator then they might need to wait for the resource to become available. To avoid this form of congestion we introduce multi-version tasks. A task may have one implementation targeting the GPU, one using some other specific hardware accelerator and yet another one targeting the CPU. Our scheduler detects that the computing unit targeted by the task is busy, and that it is preferable to use another version of the task running on a readily available compute unit.

Should our scheduler not be able to determine a matching version where all hardware resources are available and if the current task has a higher priority than the one currently using the targeted compute resource, we apply a Priority Inheritance Protocol (PIP) [27] and reschedule the task.

Going further with versions, we provide multiple methods to select the version to use for the current job. At the time of writing it is possible to select the version depending on the current energy capacity of the platform, depending on an energy/time trade-off, depending on the current execution mode[2], depending on a bit mask permission or with a call to a user-defined function. Note that due to our technique to configure YASMIN only one method is actually used at runtime, but we can easily switch between the various options at compile time.

Each of these methods to select the version of a task requires different information from the user. These parameters are set when declaring a version using *version_decl* through the *VSelect props* argument. The type of this argument is a structure morphed to cope with the selected method. For example, if the method to select the version is based on the energy then the structure includes two fields to provide the energy budget of the task, and a user function to request the platform-dependent battery status. The different structure types are activated/deactivated at compile time using a macro defined in the configuration file.

Limitations: Usually task (versions) targeting a specific hardware accelerator nonetheless start on a CPU core before they move the main workload to the accelerator and eventually complete their run again on a CPU core. For the time being, we consider the resource busy from the beginning of the initial CPU part to the end of the final CPU part. In the near future we plan to add an asynchronous mechanism, where CPU cores can be used by tasks while the accelerator-bound task actually runs on the accelerator.

5.5 Partitioned & global on-line scheduling

We rely on the concept of shielded processor, as described in [33, 8]. The idea is to reserve cores to only execute real-time tasks in order to minimise interference with system tasks. On each of the reserved cores we spawn one thread responsible for executing the real-time (RT) tasks. These so-called *worker threads* serve as containers for the execution of the user RT tasks, which see them as virtual CPUs.

An on-line scheduler must activate tasks following their arrival time (period), decide which version of the task to execute and dispatch tasks to a worker thread (or virtual CPU). Two modes are available: (1) *Global* when all tasks can be executed on any virtual CPU and (2) *Partitioned* when all tasks have a predefined target virtual CPU. During compilation code is morphed to using the selected mode by means of configuration macros. Hence, only one of the two options effectively is compiled into the resulting binary and thus available at runtime. Switching between global and partitioned scheduling modes requires the modification of a single macro definition and a recompilation.

Figures 15a and 15b illustrate our overall architecture for the global and for the partitioned scheduling strategy, respectively. In either case each worker thread is pinned to a specific core. With global scheduling all worker threads share a common ready queue, whereas with partitioned scheduling each worker thread has its own ready queue.

[2]For example, multi-security mode where different implementations of an encryption algorithm can be switched at runtime by changing the mode of execution.

In either case, global or partitioned, the ready queue is filled by a separate scheduler thread that is likewise pinned to its private core. Unlike in [33], who also use an external scheduler thread, our scheduler thread does not constantly check for new tasks to activate. Instead, we only periodically check for new tasks to schedule, i.e. between two activations the scheduler thread *waits* in one of two ways, depending on the user-provided configuration:

- *sleep* (default): calls some kernel code, which is hardly timing-analysable,
- *spinlock*: enable a more precise overhead analysis at the cost of potential energy waste.

The period of the scheduler thread is determined using the greatest common divisor (GCD) of all the declared task periods.

Using a separate scheduler thread, that executes on its private core, decreases parallelism, as one core less is available to execute user RT tasks, but it increases predictability by minimising interference with user RT tasks. For example, in the Linux scheduler task preemption is realised by a periodic interrupt handler that stops the currently executing thread on each core. This interrupt handler of the kernel checks if there is a higher priority task to execute. This interruption mechanism must be accounted for in the worst-case response time (WCRT) of user real-time tasks. However, in practice, it is very difficult to estimate the time spent in this interrupt handler, introducing substantial pessimism in the WCRT of tasks. Using a separate scheduler thread to check for higher priority tasks avoids such pessimism. In Section 5.7 we elaborate further on how we deal with preemption.

In addition, it is possible to configure the Linux kernel to prevent the aforementioned periodic user thread interrupts: a value of -1 needs to be written in the virtual file */proc/sys/kernel/sched_rt_runtime_us*. We refer, the interested reader on how to increase user control over time in Linux to [32].

YASMIN supports static and dynamic priority assignments following task periods (rate monotonic), deadlines (deadline monotonic, earliest deadline first) or any statically user-defined priorities. Once RT tasks have been added to their assigned ready queue and dynamic priority assignment is enabled, the scheduler computes the priorities of all tasks present in each ready queue and reorders the ready queue by decreasing priority. Hence, the task with the highest priority is always at the head of the ready queue.

YASMIN supports recurring and non-recurring tasks. Both types of tasks must be declared before use, but only recurring tasks require a period to be given in the *TData* structure. Then a user-function may activate a non-recurring task at any time. Alternatively, YASMIN will do so in case of a graph dependency.

Limitations:

We do not yet handle arbitrary tasks in a conventional way using a periodic server. This is left for future work. However, the current state-of-the-art already allows us to further support graph-based task model.

We do not support job migration. A job (task instance) spawned on a virtual CPU cannot be migrated to another one. However, we do support task migration: job i of some task may run on one virtual CPU while job $i + 1$ runs on a different virtual CPU.

5.6 Off-line scheduling

Unlike any similar approach we have found in literature, YASMIN natively supports *off-line* computed schedules. An *off-line* schedule is computed before executing the application using the timing properties of the task set. In our run-time implementation an *on-line* dispatcher dispatches tasks at the predefined time following a given time table and a given mapping.

Figure 15c presents the overall architecture for the off-line scheduling strategy. Each worker thread is pinned to a specific core and has access to a predefined sequence of RT tasks ordered by increasing release time. Upon creation each worker thread starts executing a control loop running the RT task in order. To respect the release time of each task (computed off-line), special delay slots are added in between RT tasks that make the worker threads wait for a pre-computed duration. In analogy to our implementation of the on-line scheduling strategies, delay slots can be configured to *sleep* or to *spinlock*.

If the static scheduler is aware of multi-version tasks, the version can be pre-selected off-line. This has the advantage of reducing the size of the resulting binary size as it only needs to embed the actually required task versions.

Limitations: We do not support preemption in off-line generated schedules. This limitation could easily be overcome by splitting the preempted task into two separate subtasks. We consider heterogeneous resource management to be handled by the off-line scheduling step. A task can, hence, target an accelerator without requesting access to the on-line dispatcher.

5.7 Further implementation aspects

This section describes other design issues we encountered and how we addressed them in YASMIN.

Accessing time: We access time using the POSIX primitive *clock_gettime* where the given clock can be set using the configuration file. As default, *CLOCK_MONOTONIC* is employed. It gives a monotonically increasing clock with nanoseconds precision. The POSIX standard does not specify what the time 0 means. In Linux time 0 corresponds to system boot time. Our library stores the time at which the schedule is started using API call *start*. Afterwards, all timing information is computed using this initial starting time.

Pre-emption: YASMIN supports pre-emption with on-line scheduling policies only. Upon sorting, similar to [26], the scheduler thread sends a signal (PREEMPTION_SIGNAL), using the *pthread_kill* POSIX primitive, to each worker thread executing tasks with a lower priority than that of the head of the ready queue. This signal is caught by the thread which looks in the ready task queue for a higher priority task at the head of the ready queue. If a higher priority task is found, the execution context of the pre-emptee is saved (task with lower priority), and a context switch to the pre-emptor is operated (task with a higher priority). Upon pre-emptor completion the process of finding a higher priority task is repeated until the pre-emptee becomes the highest priority task and the context is switched back to this task.

Context switching: Similar to [26] we use an architecture-dependent *swapcontext* function (in assembly code), which is called when switching execution context upon pre-emption. We draw inspiration from the GLibC swapcontext implementation, but leave out extra syscalls. At the time of writing, our swapcontext implementation is available for ARM 32/64 bits as well as X86-64 architectures.

Locking: Internally we implement synchronisation primitives, i.e. mutex locks and barriers, in two different manners: A first implementation uses the POSIX API implemented in the kernel and GLibC. A second implementation relies on lock-free algorithms from [24]. It is possible to select one of the two options at compile time using the configuration file. We believe that lock-free algorithms form a superior choice for static WCET analysis, but spinlocks exhibit higher energy consumption. On the other hand, it is hard to analyse kernel and GLibC calls, but this solution offers better energy performance at the cost of predictability due to the kernel replacing the worker thread by an internal idle task. Selecting one or the other option depends on user preferences regarding predictability and energy conservation.

Protecting against page fault: Similar to [26] we lock our library code in memory using the POSIX primitive *mlockall*. This prevents swapping out the code of our library.

Interrupts: We set the kernel to use *threadirq*, and we shield the processor using *isolcpu*. Hardware interrupt handlers are composed of two parts, a top and a bottom part. We cannot do much about the top part that usually is pinned to a specific core. For the bottom part, if they are not pinned to a specific core, then the same configuration as for software interrupts applies. If they are specific to a core, and this core runs a worker thread, or the scheduler thread, then their schedule is left to the underlying OS. Care must therefore be taken to ensure that the priority of our worker threads, and/or scheduler threads allows these bottom part interrupt handlers to execute.

Standard compliance: We engineered our library to facilitate its analysis by standard WCET tools as much as possible. Nonetheless, our library is a middleware that acts in between the kernel and end-user application, forcing us to the kernel API with syscalls. But most of the syscalls are performed outside of the *start* and *stop*, hence not interfering with the real-time end-user tasks. In addition, we provide the options to either use lock-free implementations of synchronisation primitives or the POSIX versions. Thus, our library can be used in a deployed environment as much as prototyping a real-time system.

Moreover, we verified our code using PC Lint Plus [35] against MISRA-C 2012 rules. At the time of writing, 80% of the code base as been checked due to the various configuration possibility.

6 ADMORPH eXchange format

Multiple tools in the ADMORPH project provide or consume the same data. An example is the dataflow graph of the application, which is generated by the TeamPlay compiler and required by the scheduling analysis. Information like task execution times or the deadline for the overall graph execution is provided by the application specification.

To simplify the exchange of such data, a common format was developed, enabling the interconnection between different tools in the ADMORPH project. This format is named *ADMORPH eXchange Format* and a minimal set of node types and attributes was specified to exchange data between the TeamPlay compiler and the scheduling analysis. The format can easily be extended, for example with additional attributes, to connect more tools that provide additional data.

This work also concerns Task T2.5, *Interfacing the coordination language compiler infrastructure* of Work Package WP2. Nonetheless, we decided to already report on our on-going work in this Deliverable D1.2 and will provide an update in the upcoming Deliverable D2.2.

6.1 ADMORPH eXchange format design

The ADMORPH eXchange format (AXF) is based on the DOT format[3]. Graphs in the ADMORPH Scheduling Runtime Environment (RTE) contain two different kinds of nodes representing functional behaviour (*actor nodes*) and data (*data nodes*). These two kinds of nodes can be distinguished by their attributes so that no special keyword had to be introduced. Additional attributes, for e.g. processor assignment or execution times, can be introduced, as long as their names do not collide with the attributes described in this document or with the standard DOT attributes. DOT also provides keywords to assign attributes to all nodes, edges or the whole graph via *attribute statements*. To structure a graph, it can be subdivided into multiple *subgraph* blocks.

6.1.1 DOT Grammar

A slightly simplified version of the full DOT language is sufficient to specify the structure of graphs compatible to the ADMORPH Scheduling RTE. Features that are not part of the simplified version are ignored by the RTE.

Terminals are shown in **bold** font and non-terminals in *italics*. Literal characters are given in single quotes. Square brackets [and] enclose optional items and vertical bars | separate alternatives. Keywords (**digraph** and **subgraph**) are case-independent.

graph : **digraph** [*ID*] **'{'** *stmt_list* **'}'**
stmt_list : [*stmt* [**';'** *stmt_list*]]
stmt : *node_stmt* | *edge_stmt* | *subgraph*
attr_list : **'['** *a_list* **']'**
a_list : *ID* **'='** *ID* [**','** *a_list*]
node_stmt : *ID* [*attr_list*]
edge_stmt : *ID* **'->'** *ID* [*attr_list*]
subgraph : **subgraph** *ID* **'{'** *stmt_list* **'}'**

[3] http://graphviz.org/doc/info/attrs.html

An *ID* is one of the following:

- Any string of alphabetic ([a-zA-Z]) characters, underscores ('_') or digits ([0-9]), not beginning with a digit
- A numeral $[-]^{?}[0\text{-}9]^{+}(.[0\text{-}9]^{+})^{?}$
- Any double-quoted string ("...") possibly containing escaped quotes (\")

*ID*s are strings and so `abc_2` and "`abc_2`" are semantically equal. It is possible to use keywords as *ID*s but in this case quotes are mandatory.

6.1.2 Data Nodes

Data nodes contain information about the data which is transferred from one actor node to another. In our model, data can be either a single element or fixed-size list of homogeneous elements. Further, data nodes can have either zero or one preceding nodes and arbitrary many succeeding nodes. There are five custom attributes regarding data nodes. For which nodes an attribute is mandatory is noted in parentheses.

ptype: This attribute has four possible values, namely *input*, *output*, *constant* and *inner*, which correspond to the four types of data nodes. Input nodes and constant nodes are entry points in the graph and do not have predecessors. The difference is that the data of constant nodes stays the same for all graph executions. Output nodes represent data created by graph executions which can either be used externally or as input data in subsequent executions. Nodes of this type always have a predecessor but may also have successors. Inner nodes are those that do not fall into the other categories. These nodes always have a predecessor and one or more successors.

size: In order to be able to determine the memory requirements of a data node, there is another attribute to specify the size of a single element. The value of the *size* attribute represents the numbers of bytes required to store one data element.

pdata: (constant data nodes) The *pdata* attribute is exclusive for constant data nodes. Its purpose is to specify concrete data values. The correct format is a sequence of bytes in hexadecimal notation with two digits for each byte whose length must correspond to value of *size*.

pname: (input and output nodes) Input and output nodes have a name attribute which has the purpose to give programmers a more convenient access to graph nodes. Names can be arbitrary strings. The only restriction is that names must be unique within the class of data nodes, i.e. two input nodes must have distinct names but an input node may have the same name as an output node.

Data nodes do not store information about the type of data they contain. Instead, type information is restored by the RTE based on the parameters and return types of actor node functions at runtime. Some example data nodes are provided in Figure 16.

```
p_fork_in    [size=1, ptype="input", pname="input"];
p_fork_out_1 [size=4, ptype="inner"];
p_fork_out_2 [size=4, ptype="inner"];
p_left_out   [size=4, ptype="inner"];
p_right_out  [size=4, ptype="inner"];
p_middle_out [size=1, ptype="output", pname="output_1"];
p_join_out   [size=1, ptype="output", pname="output_2"];
```

Figure 16: Some data nodes, with one input node, four inner nodes, and two output nodes.

6.1.3 Actor Nodes

The purpose of actor nodes is to describe the application's functional behaviour. Actor nodes can have an arbitrary number of preceding data nodes (its inputs) but always have one succeeding data node (its output). With regard to actor nodes, the DOT language is expanded by four custom attributes.

atype: Specifies the operation the actor node executes on the input data. Beside functions like *map* or *reduce*, the type that is typically used within the AXF is *generic*. Actor nodes of this type simply execute the function specified by the following attribute.

afunc: Most types of actor nodes apply a function on their input data. This attribute is used to specify the function name.

In Figure 17, some example actor nodes are listed.

```
a_fork   [atype="generic", afunc="fork_function", shape=box];
a_left   [atype="generic", afunc="left_function", shape=box];
a_middle [atype="generic", afunc="middle_function", shape=box];
a_right  [atype="generic", afunc="right_function", shape=box];
a_join   [atype="generic", afunc="join_function", shape=box];
```

Figure 17: Five actor nodes, each calling the function specified in the attribute `afunc` during its execution.

6.1.4 Graph Edges

To specify compatible graphs, edges must always be directed. Since the order of parameters matters for most functions, two custom attributes had to be added. The *paramindex* attribute allows to specify the order of inputs for actor nodes with more than one predecessor. For actor nodes with only one predecessor the attribute is optional (but of course has to be 0 if set). To select one of multiple outputs of an actor node, the attribute *resultindex* has to be used. Example edges between the nodes of the previous examples are listed in Fig. 18.

```
1  p_fork_in ->          a_fork;
2  a_fork ->             p_fork_out_1   [resultindex=0];
3  a_fork ->             p_fork_out_2   [resultindex=1];
4  p_fork_out_1 ->       a_left;
5  p_fork_out_2 ->       a_middle;
6  p_fork_out_2 ->       a_right;
7  a_left ->             p_left_out;
8  a_right ->            p_right_out;
9  p_left_out ->         a_join         [paramindex=0];
10 p_right_out ->        a_join         [paramindex=1];
11 a_middle ->           p_middle_out;
12 a_join ->             p_join_out;
```

Figure 18: Data nodes and actor nodes are connected with directed graphs.

6.1.5 Example Graph

Combining the three previously listed parts builds a simple but complete graph, as depicted in the example in Figure 19.

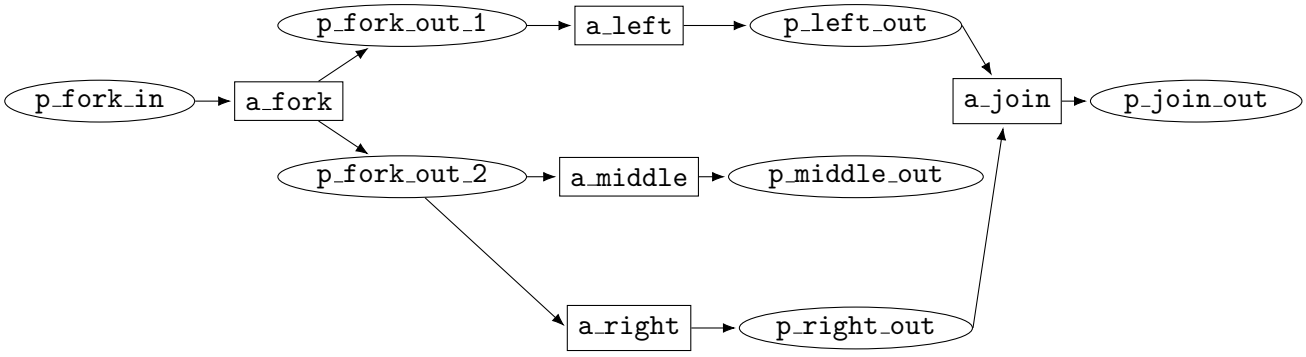


Figure 19: An example graph consisting of data nodes and actor nodes.

6.2 Compiling TeamPlay to ADMORPH eXchange format

While similar, the TeamPlay coordination language and the ADMORPH eXchange Format have a few distinct differences. Table 2 shows an overview of some of the major differences in encoding DAG applications.

Even for aspects where the formats are conceptually in agreement (e.g. both deal with task code as linkable C/C++ functions), there are differences. This lack of tight coupling is intentional, as it allows further evolution of the coordination language independent to the eXchange format. Furthermore, the goal of the exchange format is not to support every feature implemented in the coordination language. Instead, it exists to provide a way to exchange DAG applications between different tools. As such, only the intersection of features between the coordination language and downstream users of the language is relevant. Consequently, many

Feature		TeamPlay-encoding	AXF-encoding
Graph Encoding	Tasks	Nodes (“components”)	Actor nodes
	Dependencies	Edges (“channels”)	Data nodes
Task Identification	Start task(s)	Have no input channels	Are connected to initial data node
	End task(s)	Have no output channels	Are connected to final data node
Data Types	Name	In coordination language	Unnamed
	Size	In configuration file	As attribute on data node
	Callable	Each task a C function	Each task a C function
Calling Convention	Input provided as	Each input as a pointer	Input and output coalesced into a single argument
	Output retrieved as	Each output as a (mutable) input pointer	

Table 2: Various differences between the TeamPlay coordination language and the ADMORPH eXchange Format (AXF)

features found in the coordination language (e.g. modes or versions) do not have a representation in the exchange format.

Conversion between the formats is handled by the TeamPlay compiler. Edges are converted to data nodes, synthetic start and end data nodes are added, and special wrapper functions are generated to meet the calling convention of the exchange format. A diagram showing how compilation to the exchange format is shown in Figure 20. All TeamPlay input files are parsed as normally, and made available to the a dedicated *AXF Generator* component. While the TeamPlay compiler can perform analysis on the task graph encoded in the coordination file, any result must be communicated to a downstream user via the AXF file. At this time, features such as processor assignment of tasks (partitioning) or static schedules are not supported by the file format, and as such useful analyses are limited to e.g. schedulability analysis.

6.3 Runtime environment

The ADMORPH Scheduling RTE uses *directed acyclic graphs* (DAGs) as program representations. These graphs are imported from graph descriptions following the ADMORPH eXchange Format (AXF). To execute such programs, a schedule of the executions of all the tasks, i.e. the individual actor nodes in the graph, has to be calculated. Internally, the RTE consists of three parts, which are responsible for graph import, scheduling and graph execution. The different parts communicate only through exchanging graphs and have no additional dependencies between each other. This allows to replace parts of the RTE individually, for example to explore different scheduling techniques. Fig. 21 shows an overview of the structure of the RTE, where the lines connecting the parts represent interfaces for graph exchange.

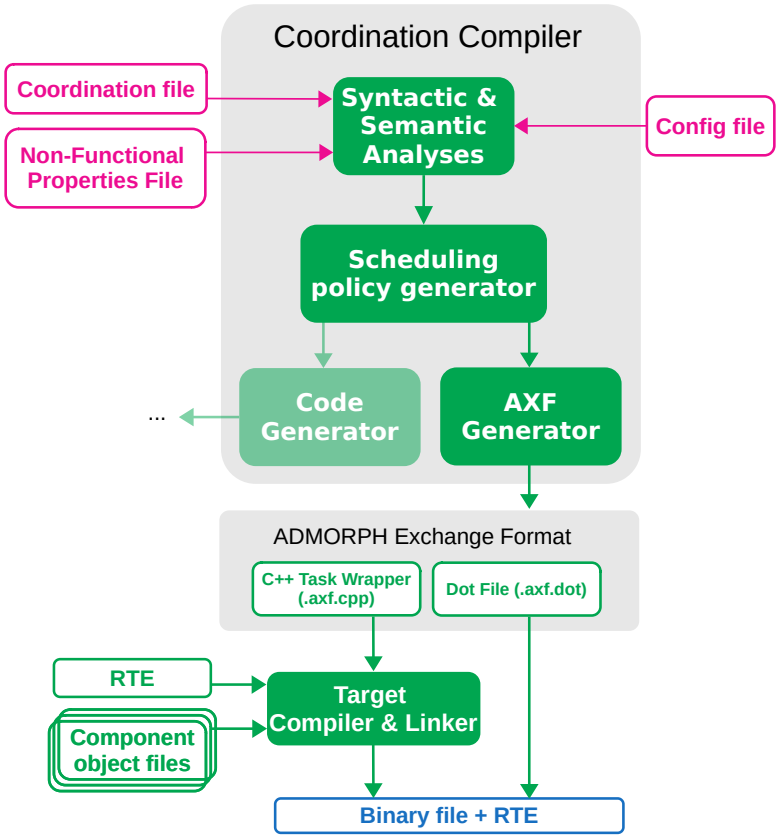


Figure 20: Overview of the ADMORPH eXchange Format compilation process

6.3.1 Graph Import

The RTE imports graphs from files in the ADMORPH eXchange Format, as described above. A graph description explicitly contains information about all nodes, data dependencies and memory requirements. Additional information can be provided through the AXF file, which are partially supported already, like actor worst-case execution time (WCET), and the maximum allowed execution time of the overall graph. Further attributes will be supported in the future, for example fixed mappings of actors to processors.

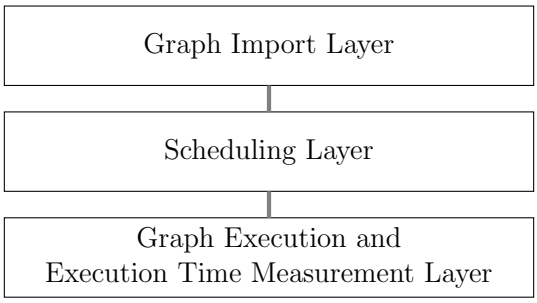


Figure 21: Structure of the ADMORPH Scheduling RTE

6.3.2 Scheduling

Based on the task dependencies specified in the graph, and the WCET of the individual tasks, schedules are calculated to find a feasible mapping of actors to processors. The current RTE implementation uses a variant of the HEFT scheduling heuristic (*heterogeneous earliest finish time*), but alternative scheduling algorithms could be used as well.

6.3.3 Graph Execution

This part of the RTE receives graphs from the scheduling part of the RTE and executes them according to the respective schedule. During the application development phase, the RTE can measure the execution times of the tasks to provide a maximum observed execution time for the tasks on the target hardware. These measured execution times can be used instead of an externally provided WCET, and can be inserted in the AXF as additional actor node attributes. The actor node execution time measurement also supports heterogeneous architectures, where the task execution times are determined on all of the different processor cores available.

7 Analysing the impact of various redundancy levels against single event upsets

The TeamPlay coordination language has evolved and provides new fault-tolerance capabilities. Certain kinds of faults lend themselves well for evaluation and validation within a wall-time simulator, using fault injection as discussed in Section 4. However, the effects of transient faults and the timing impact of their mitigation can only be analyzed by looking at long-running application behaviour. A wall-time simulator would take too much time to gather meaningful data over the behaviour of an application in the presence of transient faults. Instead, we analyse long-running application behaviour using simulated time. To that end, we have enlisted the help of the existing model checking tool *UPPAAL*. For seamless integration with the remaining tool flow, a UPPAAL compilation target has been added to the TeamPlay coordination language compiler.

This work has led to a publication in MCSoC 2021[25]

7.1 Need for a simulated-time simulator

Our simulator from Section 4 is capable of showing and validating the exact behaviour of the application when a permanent fault occurs. Faults are triggered by fault injection, which prompt any fault-tolerance capabilities to respond. While the simulator can show the correct behaviour of these capabilities, the simulator cannot tell *when* a fault will happen. The rate of permanent faults is a property of a combination of the environment and the hardware. However, the number of permanent faults that can be tolerated is limited as at some point no functioning hardware remains. As such, the simulator can be used to ascertain how many permanent faults can be tolerated. Then, this number can be used by the designer of the application to choose appropriate hardware, matching the fault behaviour of the hardware to the needs and properties of the application. If for example this application has to run for a long time, additional redundant hardware may be chosen, or hardware that is less likely to experience a permanent failure.

This method of extracting behaviour from the application and using it to choose hardware need not always apply when considering transient faults. A transient fault is, by its very definition, *transient*. It does not permanently leave the hardware in a degraded state. As such, after a fault-tolerance technique has handled the impact of the transient fault, the entire application returns to perfect working order. Consequently, the application has the capacity to handle an unlimited number of transient faults, provided they occur sufficiently spaced apart to allow for the mitigation of them.

As such, the rate of transient faults and the software need to be evaluated together. Any fault-tolerance techniques protecting against transient faults may easily be overwhelmed if the number of such events exceeds a certain threshold within a given time window, or if the events occur at particular unfortunate moments. We simulate the timing effects of transient faults on each task, together with an online scheduler and a particular fault-tolerance implementation. This allows us to compare the long-running behaviour of different fault-tolerance techniques, as well as approximate the probability of encountering catastrophic behaviour (i.e. deadline overrun) as a result of transient faults.

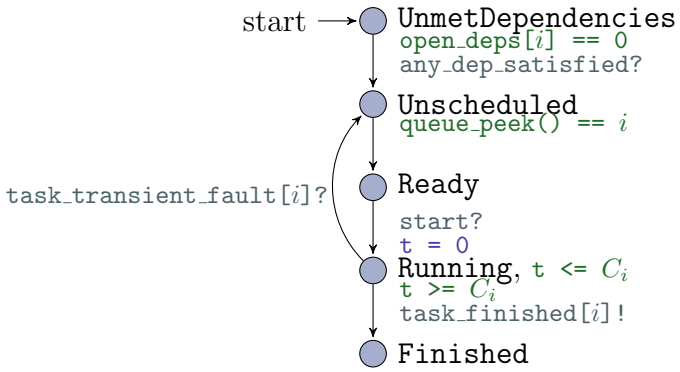


Figure 22: `SWIFT-CR` task model: the task runs under SWIFT instruction-level fault detection, and uses checkpoint-restart to reschedule the task after a fault has been detected.

Type	# replicas	Fault detection	Fault mitigation
`PLR-CR`	2	Process Level Redundancy[34]	Checkpoint-Restart at the task level
`PLR-3`	3		Internal
`FUNC-CR`	2	Process Level Redundancy (sync at function calls)[34]	Checkpoint-Restart at the task level
`FUNC-3`	3		Internal
`TTMR`	3	Time Triple Modular Redundancy[2]	Internal
`SWIFT-CR`	2	Instruction Level Redundancy[10, 28]	Checkpoint-Restart at the task level
`SWIFT-R`	3		Internal (SWIFT-R[10])

Table 3: Modelled fault-tolerance implementations

7.2 TeamPlay applications as UPPAAL models

UPPAAL is an existing model checking tool for modeling, validating and verifying real-time systems modeled as networks of timed automata. As we are only interested in the timing effects of various fault-tolerance techniques, our UPPAAL compilation target does not consider the actual implementation of each task. Rather, it generates a task *process model* based on the non-functional properties of the task.

Figure 22 shows an example of such a UPPAAL process model. Multiple of such models, together with models of processors and a scheduler, are composed together to simulate runtime timing behaviour. While the exact description and nature of a UPPAAL model is out of the scope of this document, one way to think about it is as a state machine, where state transitions can be additionally constrained on time. The model in Figure 22 models timing behaviour when implementing SWIFT[28] for fault detection, together with checkpoint-restart for fault mitigation. Faults are detected instantly, which is reflected by the immediate transition from `Running` to `Unscheduled` when a transient fault occurs. The overhead introduced by using SWIFT is included in the model parameter C_i.

Table 3 shows an overview of modeled fault-tolerance implementations. We differentiate between fault-detection (two replicas) and fault mitigation (three replicas) techniques. Where fault mitigation techniques typically offer completely transparent fault mitigation, the added overhead of the third replica may make them uninteresting for many applications. As such, we also consider fault detection using only two replicas, combined with CR to allow mitigation

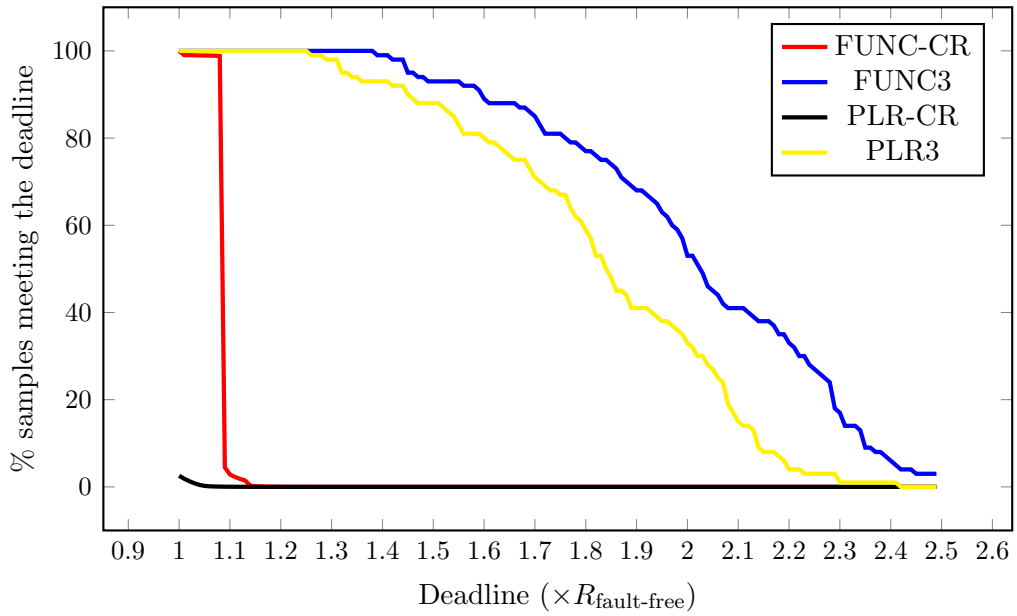


Figure 23: Comparison of various fault-tolerance implementations' timing behaviour

when a fault has actually occurred.

Figure 23 demonstrates the output of the toolchain. To construct this figure, 100 task graphs have been generated using *Task Graphs For Free* (TGFF). This figure plots the capacity of each fault-tolerance technique to meet the deadline across a range of deadlines. The deadline is shown as a multiple of the fault-free WCRT (worst-case response time) $R_{\text{fault-free}}$. A fault-tolerance implementation which would have a miss percentage of 0% at deadline $D = 1.0 \times R_{\text{fault-free}}$ would indicate that the use of fault-tolerance has no impact on the WCRT.

8 Specifying formal guarantees for the adaptation layer

The aim of this task is to specify the formal guarantees that the adaptation layer has to meet in order to ensure the correct functioning even in the presence of an attack or of a fault. We focus on control systems that receive their sensor input and has to compute a corresponding control signal to actuate.

In Deliverable D1.1 we provided background on control theory and focused on two requirements in terms of when a fault should be resolved.

(i) The fault/attack should be resolved within R_{max} time. This means guaranteeing $R_{\text{anomaly}} \leq R_{\text{max}}$. In this case, we are stating that there can be subsequent events but each of them cannot last more than a given amount of time.

(ii) Given R_{anomaly}, we would like to find R_{recovery} such that, if the adaptation layer can guarantee that the system is in a fail-safe state for R_{recovery} then it has returned to nominal conditions in which another event can happen. Notice that we are in no way constraining that $R_{\text{anomaly}} \leq R_{\text{max}}$ and the two requirements are independent.

We previously discussed the first of these two requirements. In our remaining time, we focused on the second requirement.

8.1 Anomaly and recovery time R_{anomaly} and R_{recovery}

The question that we try to answer is the following: given a duration R_{anomaly} we want to find the recovery time needed to return to a safe state, R_{recovery}.

Specifically, given a system, we define a *burst interval* $\mathcal{M}$ ($R_{\text{anomaly}}/T_{\text{period}}$, where T_{period} is the sampling period of the control system) as an interval of controller activations in which the control task executing $\mathcal{C}_d$ consecutively misses n deadlines, regardless of the strategy used to handle the misses. We assume that the burst interval $\mathcal{M}$ is followed by a *recovery interval* $\mathcal{R}$ ($R_{\text{recovery}}/T_{\text{period}}$), defined as an interval in which the control task consecutively hits h deadlines.

During the burst interval, the deadline misses of the control task are handled using a *deadline handling strategy* as specified in the analysis above. From an industrial viewpoint, the proposed fault model is highly relevant.

We look at both stability and performance. For stability we define two types of stability: static-cyclic stability and miss-constrained stability.

We now describe how the system matrices above can be used to analyse stability, starting with the cyclic burst case. Recall that a closed-loop control system under normal operation is stable if and only if the (fixed) system matrix A is Schur stable. This criterion is also valid for cyclic patterns, where A now represents the product of all closed-loop state matrices experienced in a full burst–recovery cycle. We can hence use the criterion in order to find the smallest number of deadline hits h such that n consecutive misses do not destabilise the system. It is important to note that h is the number of consecutive hits necessary for the system to recover to such an extent that another burst interval would not destabilise the system. This is a sufficient condition and not necessary, meaning that if a miss occurs during the recovery interval this does not immediately imply that the closed-loop system is destabilised. We summarise the analysis in the following definition.

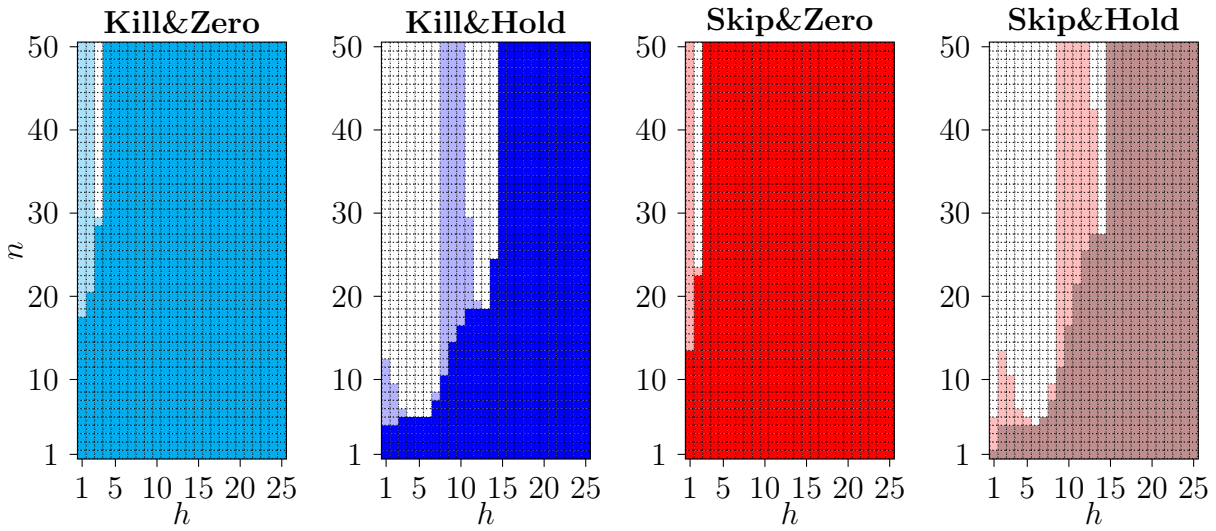


Figure 24: Miss-constrained stability (dark coloured area) and static-cyclic stability (light coloured area) when different strategies are used. Each square represents a window of size $\ell = n+h$. The dark area satisfies both the stability criterion whilst the light area only provides static-cyclic stability. The white squares denote unstable combinations of hits and misses.

Definition 1 (Static-Cyclic stability analysis)**.** We denote the stability analysis presented above with the term *static-cyclic stability analysis*. The system under analysis cycles through a sequence of n misses followed by a sequence of h hits. The sequence repeats and the eigenvalues of the closed-loop state matrix have absolute value less than 1.

We now turn to the second type of stability analysis. Given the same system and two possible models of computation, with n deadline misses or with $n_{\subset}$ deadline misses, where $n_{\subset} < n$ guaranteeing static-cyclic stability of the first one (n misses) does not immediately imply that static-cyclic stability is guaranteed also for the second one ($n_{\subset}$ misses). This motivates a second stability definition.

Definition 2 (Miss-constrained stability analysis)**.** To guarantee *Miss-constrained stability*, a system has to be stable under arbitrary switching between all the possible n realisations (i.e., closed-loop matrices) that comply with all the possible task models $n_{\subset} \leq n$ and also include the case in which the system does not miss deadlines.

Checking stability under arbitrary switching is unfortunately quite involved. We use the joint spectral radius to check this condition (including the system under normal operation among the possible switching alternatives), similar to [21]. It should be noted that, for the same parameters n and ℓ, static-cyclic stability is always implied by miss-constrained stability; hence the latter analysis is more conservative, that is, it guarantees stability in fewer cases. We take a system as example and show the corresponding results in Figure 24.

Alongside stability, it is common to look at the *performance* of the closed-loop system. Performance can be defined in different ways, often depending on the application [4]. Whichever way is chosen, a common way to quantify performance is to define a cost function and evaluate the cost function during the execution of the controller. In our work, we use a quadratic cost

function

$$J_{[k]} = \mathrm{E}\left(e_{[k]}^T Q_e e_{[k]} + u_{[k]}^T Q_u u_{[k]}\right), \tag{1}$$

where ^{T} is the transpose operator. The cost function penalises deviations from the reference value as well as usage of the control signal. E denotes expected value, and the positive semidefinite weighting matrices Q_e and Q_u weigh the different terms against each other. A small cost value means that the controller successfully makes the error approach zero, using a small control signal.

If we know the stochastic properties of the external input signals (setpoint $r_{[k]}$ and noise $w_{[k]}$) we can calculate the value of the cost function analytically. For simplicity and without loss of generality, we will from now on assume that $r_{[k]} = 0$ (i.e., we want to regulate the output to zero) and that $w_{[k]}$ is a zero-mean Gaussian white noise process with variance R. More elaborate disturbance models can be realised by introducing additional states in the plant model.

We now show how to evaluate (1). Let P_k denote the covariance of the closed-loop state vector at time k,

$$P_k = \mathrm{E}\left(\tilde{x}_{[k]}\tilde{x}_{[k]}^T\right). \tag{2}$$

The state covariance evolves according to

$$P_{[k+1]} = AP_{[k]}A^T + B_w R B_w^T. \tag{3}$$

where B_w is the input matrix that multiplies the noise. Given $P_{[k]}$, we can calculate the cost for time step k as

$$J_{[k]} = \mathrm{E}\left(\tilde{x}_{[k]}^T Q \tilde{x}_{[k]}\right) = \mathrm{tr}\left(P_{[k]}Q\right) \tag{4}$$

where tr computes the trace of the matrix. In (4)

$$Q = \begin{bmatrix} C_p^T Q_e C_p & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & Q_u \end{bmatrix} \tag{5}$$

is the total cost matrix. The stationary cost of the system is defined as J_∞. More specifically, this is the cost the system converges to when operating under normal conditions:

$$J_\infty = \lim_{k\to\infty} J_{[k]}. \tag{6}$$

Under normal operation, after a transient phase, the cost does not deviate from the nominal cost J_∞, meaning that there exists an instant $\bar{k}$ for which $J_{[k]}$ reaches the steady-state value J_∞, or formally $\exists \bar{k}$ s.t. $\forall k > \bar{k},\ J_{[k]} \approx J_\infty$.

We use the cost function in Equation (4) as a time-varying performance metric. Before a burst interval, we assume that the system is in the neighbourhood of its steady-state covariance P_∞ and performance J_∞. When a burst interval of n missed deadlines occurs, we track the performance function and wait until it returns in a small interval around J_∞, after which we can consider the system has returned to its nominal state.

Definition 3 (Performance recovery interval)**.** We define the recovery length interval h^* as the smallest h such that the cost threshold is satisfied for all $k \geq h$ when using a given strategy to handle deadline misses.

Definition 4 (Maximum normalised cost)**.** We denote the maximum normalised cost of the system by

$$J_M = \max_k \frac{J_{[k]}}{J_\infty}, \tag{7}$$

where $J_{[k]}$ is the cost computed according to (4) when using a given strategy to handle the deadline misses.

Despite the system being stable for long burst intervals and able to recover, the performance loss could still be intolerable. Consequently, we would like to find the greatest burst interval length such that the system does not violate some predefined performance threshold J_T.

Definition 5 (Performance-aware burst interval)**.** We define the number of misses that a control system can tolerate, given a performance threshold J_T, as n^*, the largest n such that

$$J_M \leq J_T. \tag{8}$$

We can find n^* by iterating the performance analysis until the smallest n violating the threshold is found. We then take a step back to get the value of n^*.

Compared to the stability analysis, the performance analysis also takes into account state deviations and uncertainty due to disturbances. The disturbance term w_k is neglected in the stability analysis as it does not influence the system stability. However, its presence (as the presence of any disturbance) changes the dynamic behaviour of the system. For the performance metric, the state covariance matrix evolves according to both the noise intensity and the system dynamics. The result is that the performance analysis provides us with a conservative (but more realistic) recovery interval, that takes system uncertainties into consideration.

8.2 Application to an example system

In this section, we apply the analysis to a case study, analysing stability and performance. We show detailed results with one single plant and a controller devised for that plant.

The chosen plant is part of a representative set of processes [3] for control problems that commonly arise in the process industry.[4] The goal is to regulate the control error to zero, and control is assumed to be cheap. A low-frequency load disturbance is assumed to be acting on the system, so we extend the discrete-time plant model with a disturbance state with near-integrator dynamics (eigenvalue 0.999). We then follow an established control design procedure [13] and design a PI controller for the plant. This results in the following matrices that completely

[4]The plant could for instance represent a chemical tank where the level is controlled by pumping chemicals through a pipe. The level of the chemicals in the tank and the uncertainty in the pipe diameter are the plant states $x_{[k]}$. Furthermore, the amount of chemicals pumped through the pipe is the control signal $u_{[k]}$.

Table 4: The recovery interval length h necessary to remain cyclic-stable stable after a burst of n consecutive misses for the different strategies.

n	h_{KZ}	h_{KH}	h_{SZ}	h_{SH}
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1
5	1	1	1	1
6	1	1	1	2
7	1	1	1	2
8	1	1	1	2
9	1	1	1	2
10	1	1	1	2
11	1	1	1	2
12	1	1	1	2
13	1	8	1	2
14	1	8	1	9
15	1	8	1	9

describe the example[5]:

$$\mathcal{P}_d : \begin{cases} A_d = \begin{bmatrix} 0.932 & -0.170 & -0.276 \\ 0.170 & 0.312 & 0.324 \\ 0 & 0 & 0.999 \end{bmatrix}, \\ B_d = \begin{bmatrix} -0.276 & 0.324 & 0 \end{bmatrix}^T, \\ C_d = \begin{bmatrix} -0.276 & -0.324 & 0 \end{bmatrix}, D_d = 0 \end{cases} \tag{9}$$
$$\mathcal{C}_d : \left\{ A_k = 1,\ B_k = 0.415,\ C_k = 0.796,\ D_k = 1.921 \right.$$

Analysing the static-cyclic burst stability of the system means finding the smallest h for which the matrix that represents the combination of misses followed by hits has eigenvalues in the stability region. Given a fixed n, the check is iterated, incrementing h until the conditions are satisfied. Table 4 shows the recovery interval lengths h when n varies from 1 to 15 for the different strategies Kill and Zero (KZ), Kill and Hold (KH), Skip and Zero (SZ), Skip and Hold (SH).

We notice the Zero actuation strategies are static-cyclic stable already for $h = 1$ for all the presented burst interval lengths. However, for the Hold strategies h depend on n.

We also tested the miss-constrained stability for the system for various amount of burst interval lengths n. Figure 24 summarises our findings. The vertical axis counts the number of misses n and the horizontal axis counts the number of subsequent hits h. The dark areas in

[5]The original plant from [3] is a first order system with a time delay. The stability analysis presented in this section is able to handle time delays, however, the performance analysis assumes that the system has no intrinsic delay (other than the one induced by the one-step delay controller). However, the delay can be approximated, for example using a first-order Pade approximation [14]. We approximate the delay to obtain the plant matrices in Equation (9).

Table 5: Performance Recovery Interval h^* required to recover from a burst of n consecutive misses for strategies.

n	h^*_{KZ}	h^*_{KH}	h^*_{SZ}	h^*_{SH}
1	38	12	38	17
2	38	16	38	18
3	38	17	38	25
4	55	25	55	26
5	55	25	56	27
6	56	26	56	27
7	56	26	56	27
8	56	26	56	27
9	56	26	57	27
10	56	33	57	34

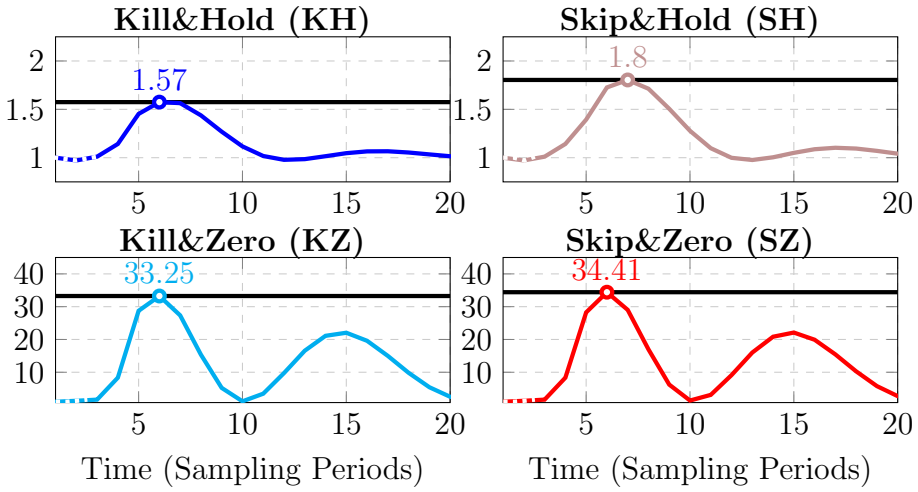


Figure 25: Time evolution of the normalised cost ($J_{[k]}/J_\infty$) when different strategies are used and the system is subject to a burst interval of length $n = 3$. The dashed part of the line indicates the period of time in which the burst interval occurs. The solid line represents the recovery interval, in which the deadlines for the control task are satisfied. The black line marks the peak, whose value is also written in the figure.

the figure show the combinations of misses and hits for which the system is miss-constrained stable. The light squares in the figure show combinations for which the system satisfies the static-cyclic stability condition but is not miss-constrained stable. The white squares show configurations for which the system is not stable, that is, does not satisfy the static-cyclic stability condition. Stability is primarily governed by the dynamics of the system. The Zero actuation strategies have some unstable configurations that can be related to the system order and the time delay in the system. Furthermore, the behaviour of KH appears to have a static-cyclic stable configuration for a fixed recovery interval $h = \{8, 9, 10\}$ regardless of the burst interval (the columns corresponding to these values of h are entirely marked as static-cyclic stable). Similarly, SH is static-cyclic stable for $h = \{9, 10, 11, 12\}$. This is presumably due to

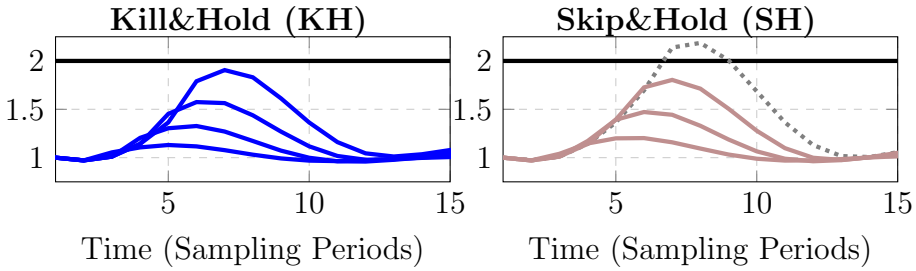


Figure 26: Time evolution of the normalised cost ($J_{[k]}/J_\infty$) for different burst interval lengths $n \in [1, 4]$. The solid lines represent the curves that satisfy the upper bound $J_T = 2$ on the performance degradation (represented by the black solid horizontal line). The dotted plot for the Skip&Hold strategy shows the performance for $n = 4$, which violates the threshold constraint.

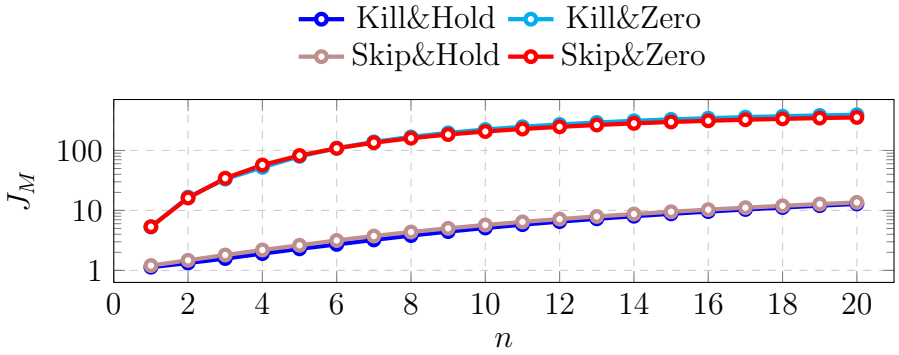


Figure 27: Maximum normalised cost J_M over the burst interval length $n \in [1, 20]$ for different strategies. The vertical axis is displayed with logarithmic scale.

n matching a stable oscillatory mode in the closed-loop system dynamics. However, this mode is not miss-constrained stable, thus lacking robustness towards the burst interval being *shorter* than expected.

If we apply the analysis presented in Deliverable D1.1, we find the maximum number of consecutive deadline misses for which the system can tolerate an arbitrary pattern of deadline hits and misses, whilst guaranteeing stability. Stability under consecutive deadline misses is guaranteed for $\{n_{KZ}, n_{KH}, n_{SZ}, n_{SH}\} = \{10, 4, 13, 1\}$. These guarantees are more conservative than the ones given by both the static-cyclic and miss-constrained stability case. Our investigation here asks a different question: how many hits should follow a sequence of n misses to return to nominal conditions (in which we are able to handle another fault)?

The stability analysis shows promising results for the recovery interval length. However, the following performance analysis motivates why stability alone is not enough to determine the behaviour of the system. In fact, we argue that the system is stable but does not behave well.

We now investigate the system cost. Figure 25 plots the normalised cost of the system ($J_{[k]}/J_\infty$) for $n = 3$ with different strategies. The dashed part of the plot shows when we experience the misses, while the solid lines corresponding to the cost that follows the misses (and the part of the plot in which the controller hits its deadlines). Higher normalised cost

values mean a decrease in performance (a normalised cost of 2 means that the performance is halved with respect to the case in which no deadline is missed). The black line shows the maximum we obtain, that is the maximum performance loss factor we experience due to the n deadline misses.

In Figure 25, the cost keeps increasing even when the system exits the burst interval after $n = 3$ deadline misses. During the burst interval, the plant deviates from its desired state, whilst the controller state is not being updated (as the controller does not complete its jobs). Once the recovery interval is entered, the control state is based on an old system state, resulting in the calculated control signal being outdated. The controller needs a few samples to converge to a state that better represents the correct course of action, thus reducing the control cost. However, despite the controller converging to a better state, the dynamics of the system still affect the performance. The secondary peaks seen in Figure 25 therefore correspond to the oscillatory effects the time delay imposes on the system.

From the figure it is apparent that the Zero actuation strategies lowers the performance of the system by factors $J_{M,KZ}$ and $J_{M,SZ}$ (from Definition 4) greater than 30. With the stability analysis presented in Deliverable D1.2 and with the stability analysis presented in this section (shown in Table 4) Zero would be the preferred choice. In fact, it seems to stabilise the system more easily. The performance analysis however shows that this comes with a performance loss that might be unacceptable. In comparison, Kill&Hold and Skip&Hold increase the cost by a factor $J_{M,KH} = 1.57$ and $J_{M,SH} = 1.8$ respectively.

The extraordinary decrease in performance of the Zero actuation strategies correlates with the system noise. The integrator in the control law mitigates the effect of the (near) integrated noise. However, once the controller misses a deadline and the output is set to zero, the disturbance state drives the plant rapidly away from the desired state. This does not affect the Hold actuation strategies due to the actuator still holding the last control signal, thus mitigating the noise effect.

We now look at the performance recovery length interval h^* from Definition 3. To find it, we increase h until we satisfy the performance condition for a recovery threshold $\varepsilon = \frac{J_\infty}{100}$.[6] Table 5 shows the minimal recovery interval length h^* corresponding to $n = [1, 10]$. Comparing Tables 4 and 5, we see that the stability analysis provides more optimistic bounds on the recovery interval length than the performance analysis. The performance should thus be taken into account when analysing the system behaviour.

We investigate how the performance-aware burst interval n^* from Definition 5 changes depending on the strategy used to handle the misses, given a threshold $J_T = 2$. Figure 26 shows the performance degradation given different miss interval lengths (i.e., $n \in [1, 4]$) for $\{KH, SH\}$. Selecting zero immediately violates the threshold constraint J_T even with a single miss. The figure therefore focuses on hold and shows the two alternatives for deadline handling strategy. Using Kill&Hold, we obtain $n^*_{HK} = 4$, while using Skip&Hold we observe $n^*_{SK} = 3$. The dashed line in the plot for Skip&Hold shows the normalised performance degradation plot for $n = 4$. The corresponding necessary recovery interval lengths h^* can be found in Table 5.

Going back to our initial aim, and recalling that T_{period} is the sampling period of the control system, we can then state that for a given $R_{\text{anomaly}} = n\,T_{\text{period}}$, and based on the strategy used, we can find $R_{\text{recovery}} = h^*\,T_{\text{period}}$.

[6] The results have been verified using JitterTime [9]. JitterTime evaluates the total state covariance of mixed continuous/discrete time linear systems driven by noise, with arbitrary timing behaviour.

9 Publication and dissemination

The work done in WP1 has led to the following publications during the reporting period:

1. Martina Maggio, Arne Hamann, Eckart Mayer-John, Dirk Ziegenbein: Control System Stability under Consecutive Deadline Misses Constraints; Euromicro Conference on Real-Time Systems (ECRTS 2020).

2. Lukas Miedema, Benjamin Rouxel, Clemens Grelck: Modeling Single Event Upsets in UPPAAL SMC for Real-time DAG Scheduling; 15th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2021), part of the 29th International Conference on Real-Time Networks and Systems (RTNS 2021).

3. Lukas Miedema, Benjamin Rouxel, Clemens Grelck: Task-level Redundancy vs Instruction-level Redundancy against Single Event Upsets in Real-time DAG Scheduling; 14th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2021).

4. Paolo Pazzaglia, Arne Hamann, Dirk Ziegenbein and Martina Maggio: Adaptive Design of Real-Time Control Systems subject to Sporadic Overruns; Design, Automation and Test in Europe Conference (DATE 2021); *Best Paper Award in the Embedded and Cyber-Physical Systems Track*.

5. Nils Vreman, Anton Cervin and Martina Maggio: Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses; Euromicro Conference on Real-Time Systems (ECRTS 2021); *Outstanding paper & Best Paper Award*.

6. Benjamin Rouxel, Sebastian Altmeyer, Clemens Grelck: YASMIN: a Real-time Middleware for COTS Heterogeneous Platforms; 22nd ACM/IFIP International Middleware Conference (MIDDLEWARE 2021).

This report has necessarily compressed (and left out) some of the details that are presented in the scientific publications above. This report also includes additional material, that complements the publications above. In particular some of the content presented in Section 8.2 is a deeper investigation on the topics of the ECRTS 2021 paper.

In addition to the dissemination events directly associated with above mentioned publications, various aspects of work conducted in the context of Work Package 1 have among others been presented at the following venues:

1. Design, Automation and Test in Europe Conference (DATE 2021), Special Initiative on Autonomous System Design (ASD 2021), Virtual.

2. ICT COST Action Meeting Connecting Education and Research Communities for an Innovative Resource Aware Society (CERCIRAS), Novi Sad, Serbia.

3. 21st Workshop on Programming Languages and Foundations of Programming (KPS 2021), Kiel, Germany.

10 Conclusion

This deliverable reports on the work accomplished by the various partners in the context of work package 1: *Specification of Adaptive Systems* since month 10. Besides the considerable achievements by the consortium partners individually, collaboration between partners is on a good way. The joint work on the ADMORPH Exchange Format AXF is an example of this.

Our work in Task T1.1 is progressing in various directions. We have carefully revised and extended the TeamPlay coordination language and adapted the coordination compiler front-end accordingly. Including support for UPPAAL modelling we have built four code generators and two fully-fledged runtime environments plus an additional runtime environment connected to TeamPlay via the ADMORPH eXchange Format (AXF).

The next steps are to complete these various lines of on-going research. For example, we plan to integrate fault-tolerance capabilities into the real-time runtime environment YASMIN. Likewise we plan to continue our modelling work with UPPAAL.

Task 1.2 on validation of the coordination language is still in its infancy. Our aim is to model essential parts of the ADMORPH use cases by means of the TeamPlay coordination language. For the time being this must be considered future work. Retrospectively, we should have formally postponed the start date of this task in line with the extension of the project, but in practice this issue remains a formality.

In Task 1.3 we analysed control systems and their behaviour in the presence of bursts of deadline misses We provided a comprehensive set of tools to determine how robust a given control system is to faults that hinder the computation to complete in time, with different handling strategies.

Our analysis tackles both stability and performance. In fact, we have shown that analysing the stability of the system is not enough to properly quantify the robustness to deadline misses, as the performance loss could be significant even for stable systems. We introduced two performance metrics, linked to the recovery of a system from a burst of deadline misses.

A limitation of the presented performance analysis is that it only applies to linear control systems. However, the approach can easily be extended to analyse *time-varying* linear systems and can also be used for local analysis of a nonlinear system that should follow a given reference trajectory. In fact, to illustrate the applicability to real (e.g., nonlinear) systems, we applied the analysis to a Furuta pendulum and compared the results of simulations obtained with a model of the process to the real execution data.[7] The results support our claim that the proposed performance analysis is a valid approximation of the real-world system performance. We performed additional tests on a large batch of industrial plants, using modern control design techniques. From our experimental campaign, we conclude that the choice of actuation strategy affects the control performance significantly more than the choice of deadline handling strategy. In the remaining time, we will focus on consolidating the scientific results and developing teaching and dissemination material based on the explored examples.

In the conclusions of the work package's previous Deliverable D1.1 we expressed our hopes and back then realistic expectations regarding the easining of the pandemic-induced restrictions

[7]The results are presented in the paper "Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses", presented at ECRTS 2021, co-authored by Nils Vreman, Anton Cervin and Martina Maggio. A paper preprint is publicly available at: `http://admorph.eu/wp-content/uploads/2021/06/ecrts21_preprint.pdf`. A video, showing the results of the Furuta pendulum experiments is available at: `https://www.youtube.com/watch?v=0P0K_7lvKVU`.

in autumn 2020. However, reality as we know it has proven these expectations to be too optimistic. Personal exchange has still been negatively impacted by the various measures taken locally and across Europe throughout the current reporting period. We remain confident that personal interaction and collaboration will soon become effective, and we will slowly return to a pre-Covid-19 style of working, just in time to still develop a positive impact on the ADMORPH project.

11 References

[1] ANTLR. ANother Tool for Language Recognition. `https://www.antlr.org/`.

[2] Seyyed Amir Asghari, Mohammadreza Binesh Marvasti, and Amir M. Rahmani. Enhancing transient fault tolerance in embedded systems through an OS task level redundancy approach. *Future Generation Computer Systems*, 87:58–65, 2018.

[3] Karl Johan Åström and Tore Hägglund. Revisiting the Ziegler—Nichols step response method for PID control. *Journal of Process Control*, 14(6):635–650, 2004.

[4] Karl Johan Åström and Tore Hägglund. *Advanced PID Control*. The Instrumentation, Systems and Automation Society, 2006.

[5] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The impact of control technology*, 12(1):161–166, 2011.

[6] Blaise Barney. POSIX threads programming. *National Laboratory. Disponível https://computing. llnl. gov/tutorials/pthreads*, 5:46, 2009.

[7] G. Bernat, A. Burns, and A. Liamosí. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, April 2001.

[8] Steve Brosky and Steve Rotolo. Shielded processors: Guaranteeing sub-millisecond response in standard linux. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 9–pp. IEEE, 2003.

[9] A. Cervin, P. Pazzaglia, M. Barzegaran, and R. Mahfouzi. Using JitterTime to analyze transient performance in adaptive and reconfigurable control systems. In *24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1025–1032, 2019.

[10] Jonathan Chang, George A Reis, and David I August. Automatic instruction-level software-only recovery. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 83–92. IEEE, 2006.

[11] Sven Efftinge and Miro Spoenemann. Why Xtext? *Xtext: Language Engineering Made Easy!* https://www.eclipse.org/Xtext/.

[12] Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004.

[13] Olof Garpinger and Tore Hägglund. Software-based optimal PID design with robustness and noise sensitivity constraints. *Journal of Process Control*, 33:90 – 101, 2015.

[14] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.

[15] Joël Goossens, Xavier Poczekajlo, Antonio Paolillo, and Paul Rodriguez. ACCEPTOR: a model and a protocol for real-time multi-mode applications on reconfigurable heterogeneous platforms. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, pages 209–219, 2019.

[16] Clemens Grelck and Frank Penczek. Implementation architecture and multithreaded runtime system of S-Net. In *Symposium on Implementation and Application of Functional Languages (IFL 2008), Lecture Notes in Computer Science 5836*, pages 60–79. Springer, 2011.

[17] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane static worst-case execution time estimation tool. In *Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, 2017.

[18] Zahaf Houssam-Eddine, Nicola Capodieci, Roberto Cavicchioli, Giuseppe Lipari, and Marko Bertogna. The HPC-DAG task model for heterogeneous real-time systems. *IEEE Transactions on Computers*, 2020.

[19] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*, volume 395. Springer Science & Business Media, 2006.

[20] Edward Ashford Lee and David G Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35, 1987.

[21] Martina Maggio, Arne Hamann, Eckart Mayer-John, and Dirk Ziegenbein. Control-System Stability Under Consecutive Deadline Misses Constraints. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[22] Linux man pages. pthread_setschedparam. `https://man7.org/linux/man-pages/man3/pthread_setschedparam.3.html`.

[23] Linux man pages. signal. `https://man7.org/linux/man-pages/man7/signal.7.html`.

[24] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[25] Lukas Miedema, Benjamin Rouxel, and Clemens Grelck. Task-level redundancy vs instruction-level redundancy against single event upsets in real-time DAG scheduling. In *14th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2021)*. IEEE, 2021.

[26] Malcolm S Mollison and James H Anderson. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 283–292. IEEE, 2013.

[27] Ragunathan Rajkumar. *Synchronization in real-time systems: a priority inheritance approach*, volume 151. Springer Science & Business Media, 2012.

[28] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. SWIFT: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[29] Julius Roeder, Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. Energy-aware scheduling of multi-version tasks on heterogeneous real-time systems. In *35th ACM Symposium on Applied Computing (SAC 2020)*. ACM, 2020.

[30] Julius Roeder, Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. Towards energy-, time- and security-aware multi-core coordination. In *Coordination Models and Languages, 22nd International Conference, COORDINATION 2020*, pages 57–74. Springer, LNCS 12134, 2020.

[31] Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. YASMIN: a Real-time Middleware for COTS Heterogeneous Platforms. In *22nd ACM/IFIP International Middleware Conference (MIDDLEWARE 2021)*. ACM, 2021.

[32] Benjamin Rouxel, Ulrik Pagh Schultz, Benny Akesson, Jesper Holst, Ole Jørgensen, and Clemens Grelck. Prego: a generative methodology for satisfying real-time requirements on cots-based systems: definition and experience report. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 70–83. ACM, 2020.

[33] N Saranya and RC Hansdah. An implementation of partitioned scheduling scheme for hard real-time tasks in multicore Linux with fair share for linux tasks. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–9. IEEE, 2014.

[34] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, 2009.

[35] Gimpel Software. PC-Lint Plus, 2012. `https://www.gimpel.com/pclp.html`.

[36] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.