Deliverable D1.3

Final Report on and Second Software Prototype Release of a Coordination Language for Robust, Adaptive Systems

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

Due Date:	Nov 30th, 2022
Delivery:	Month 36
Lead Partner:	UvA
Editor:	Clemens Grelck
Dissemination Level:	Public (P)
Status:	Final
Approved:	
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
16/04/2022, v0.11	Martina Maggio	Section 5: Formal guarantees
21/10/2022, v0.12	Jeroen Kouwer	TNL Input for Section 3: Validation
28/10/2022, v0.13	Petr Novobilsky	Q-Media Input for Section 3: Validation
01/11/2022, v0.20	Clemens Grelck	Deliverable skeleton
08/11/2022, v0.21	Lukas Miedema	Section 2.1: Coordination language design
14/11/2022, v0.22	Lukas Miedema	Section 4: Strategy switching
16/11/2022, v0.23	Clemens Grelck	Section 2.2: Coordination language design
17/11/2022, v0.24	Clemens Grelck	Section 2.3: Coordination language design
18/11/2022, v0.25	Lukas Miedema	Section 3.3: Validation Q-Media
18/11/2022, v0.26	Clemens Grelck	Section 3.2: Validation TNL
21/11/2022, v0.27	Lukas Miedema	Section 2.4: Coordination language design
22/11/2022, v0.28	Clemens Grelck	Section 1: Introduction
24/11/2022, v0.29	Clemens Grelck	Section 8: Publications and dissemination
25/11/2022, v0.30	Clemens Grelck	Proofreading and polishing
30/11/2022, v1.10	Lukas Miedema	Section 6: State of the art
01/12/2022, v1.20	Clemens Grelck	Incorporation of internal feedback
05/12/2022, v1.30	Clemens Grelck	Section 7: Conclusions
06/12/2022, v1.40	Clemens Grelck	Final touches
09/12/2022, v1.50	Stefanos Skalistis	Section 3.1: Validation Collins
20/12/2022, v2.00	Clemens Grelck	Final proofreading and polishing

List of Contributors

Date and version	Author	Comments
16/04/2022, v0.11	Martina Maggio	Section 5: Formal guarantees
21/10/2022, v0.12	Jeroen Kouwer	TNL Input for Section 3: Validation
28/10/2022, v0.13	Petr Novobilsky	Q-Media Input for Section 3: Validation
01/11/2022, v0.20	Clemens Grelck	Deliverable skeleton
08/11/2022, v0.21	Lukas Miedema	Section 2.1: Coordination language design
14/11/2022, v0.22	Lukas Miedema	Section 4: Strategy switching
16/11/2022, v0.23	Clemens Grelck	Section 2.2: Coordination language design
17/11/2022, v0.24	Clemens Grelck	Section 2.3: Coordination language design
18/11/2022, v0.25	Lukas Miedema	Section 3.3: Validation Q-Media
18/11/2022, v0.26	Clemens Grelck	Section 3.2: Validation TNL
21/11/2022, v0.27	Lukas Miedema	Section 2.4: Coordination language design
22/11/2022, v0.28	Clemens Grelck	Section 1: Introduction
24/11/2022, v0.29	Clemens Grelck	Section 8: Publications and dissemination
25/11/2022, v0.30	Clemens Grelck	Proofreading and polishing
30/11/2022, v1.10	Lukas Miedema	Section 6: State of the art
01/12/2022, v1.20	Clemens Grelck	Incorporation of internal feedback
05/12/2022, v1.30	Clemens Grelck	Section 7: Conclusions
06/12/2022, v1.40	Clemens Grelck	Final touches
09/12/2022, v1.50	Stefanos Skalistis	Section 3.1: Validation Collins
20/12/2022, v2.00	Clemens Grelck	Final proofreading and polishing

Contents

Executive summary

Deliverable D1.3 is the third and final deliverable of work package 1: *Specification of Adaptive Systems*. This deliverable is a report on the consortium's work in

- Task T1.1 *Coordination Language Design*,
- Task T1.2 *Coordination Language Validation* and
- Task T1.3 *Specifying Formal Guarantees for the Adaptation Layer*.

Task T1.4 *Specification of Fault Model and Threat Indicators* was inactive throughout the reporting period.

1 Introduction

Since the previous Deliverable D1.2 Work Package 1 has progressed in various directions, that we present in the following sections.

As part of Task T1.1 we further develop the TeamPlay coordination language for the high-level specification of cyber-physical systems (of systems) [38]. Work on the underlying coordination model and the core language has commenced in the context of the Horizon-2020 project TeamPlay[1] and is refined and partially refocussed on aspects such as fault-tolerance and resilience in the ADMORPH project.

In Section 2 we present the latest developments around the TeamPlay coordination language. First, we introduce *next channels*. They augment the DAGs (directed acyclic graphs) specified by the coordination language by inter-iteration dependencies and data exchange. Next channels represent a generalization of the original state ports that permit components to maintain state in the formally state-free setting of TeamPlay. While state ports connect the output of a component to its own input as a closed circuit, next channels may connect any outport of some component to some inport of some other component, may it be before or after the first component in the DAG.

Supporting explicit communication across iterations of the DAG through both state ports and next channels raises the question of how to run components that rely on data from a previous iteration of the DAG for the very first time. We now solve this problem in a uniform way leveraging another feature of the TeamPlay coordination language, namely multi-version components. Components that depend on data from previous iterations must feature one or more tailor-made *initialization versions* that permit the component implementation layer to properly address initialization in a systematic way.

A managed runtime environment like that of the TeamPlay coordination language offers several opportunities to identify misbehavior of individual components, e.g. catching interrupt signals, detecting timing violations or simply by identification of divergence in the case of double or triple modular redundancy. We extend the TeamPlay coordination language by several means to catch such runtime misbehavior and react to it in an orderly fashion.

Last not least, we introduce hk-constraints to the TeamPlay coordination language that provide support towards weakly-hard real-time systems that are the foundation of the research conducted in the course of Task T1.3.

In Section 3 we report on our continuous efforts to validate the design of the TeamPlay coordination language in the context of the ADMORPH use cases. This work is done in close collaboration with the industrial use case providers in the ADMORPH project: Collins Aerospace for Autonomous Aerospace Systems, Thales Nederland for Radar Surveillance Systems and Q-Media for Railway Transport Systems. In alignment with the use case providers we pursue different approaches of validation: Together with Collins Aerospace and with Q-Media we follow a validation by construction approach, where we model (parts of) their use cases with the TeamPlay coordination language. With Thales Nederland we have embarked on a constructive dialogue about the fitness of TeamPlay for the Radar Surveillance Systems use case as well as the advantages and drawbacks

[1] European Union Horizon-2020 research and innovation programme grant agreement No. 779882 (TeamPlay), 2018–2021

of TeamPlay compared to their existing in-house coordination solution.

Section 4 elaborates on our work in the area of weakly-hard real-time systems following up on the introduction of hk-constraints to the TeamPlay coordination language. We introduce the concept of *strategy switching*, that aims to make the most effective use of available computing resources for replication of selected components in the presence of deadlines. The underlying assumption is that limitations in available computing resources and computing time (i.e. real-time deadlines) do not permit to run all components under some replication regime such as double or triple modular redundancy (DMR, TMR), but rather only some. Combined with the concept of weakly-hard real-time systems, that permit limited failure, we propose a technique of actively switching between multiple execution strategies depending on observed failure or success of those components that do run with protection (DMR or TMR). With strategy switching we demonstrate to improve the steady-state failure rate under limited resources. Our work primarily addresses transient hardware faults caused by single event upsets (SEU).

In Section 5 we stick to the realm of weakly-hard real-time systems, but move our focus towards control systems investigated in the course of Task T1.3. Here, our research aims at providing a stability analysis that can be applied to a class of generic weakly-hard models and deadline miss handling strategies. First, we formally extend the weakly-hard model to explicitly consider the strategy used to handle the miss events. By leveraging an automaton representation of the sequences allowed by (a set of) extended weakly-hard constraints, we use Kronecker lifting and the joint spectral radius to properly express its stability conditions. Using the concept of constraint dominance, we prove analytic bounds on the stability of a weakly-hard system with respect to *less dominant* constraints. Finally, we analyse the stability of the resulting closed-loop systems using `SparseJSR` [47], which exploits the sparsity pattern that naturally arises in the Kronecker lifted representation.

The contents of Sections 2 and 3 is originally presented in this deliverable. Section 4 is based on the following two publications:

- Lukas Miedema, Benjamin Rouxel and Clemens Grelck: *Strategy Switching: Smart Fault-tolerance for Resource-constrained Real-time Applications*. In: Workshop on Connecting Education and Research Communities for an Innovative Resource Aware Society (CERCIRAS 2021), Novi Sad, Serbia, CEUR-WS Proceedings vol. 3145, 2022.

- Lukas Miedema and Clemens Grelck: *Strategy Switching: Smart Fault-tolerance for Weakly-hard Resource-constrained Real-time Applications*. In: Software Engineering and Formal Methods, 20th International Conference, SEFM 2022, Berlin, Germany. Lecture Notes in Computer Science 13550, pp. 129–145, Springer, 2022.

Furthermore, Section 5 is based on the three following publications:

- Jie Wang, Martina Maggio and Victor Magron: *SparseJSR: A Fast Algorithm to Compute Joint Spectral Radius via SparseSOS Decompositions*. In: American Control Conference (ACC 2021), pp. 2254–2259, IEEE 2021.

- Nils Vreman, Paolo Pazzaglia, Jie Wang, Victor Magron and Martina Maggio: *Stability of Control Systems under Extended Weakly-Hard Constraints*. In: IEEE Control Systems Letters, vol. 6, pp. 2900–2905, 2022.
- Nils Vreman, Paolo Pazzaglia, Jie Wang, Victor Magron and Martina Maggio: *Stability of Control Systems under Extended Weakly-Hard Constraints*. In: 61st IEEE Conference on Decision and Control, CDC 2022, Cancún, Mexico.

We complete this deliverable report with a brief reflection on the evolving state-of-the-art in Section 6, and we draw some conclusions in Section 7.

2 Evolution of the coordination language design

In Deliverable D1.1 we discussed the TeamPlay coordination language and its concepts. In Deliverable D1.2 we evolved upon these concepts through an improved syntax as well as through a variety of language extensions addressing the specific demands of ADMORPH. In particular, we introduced *mode switching* as a new language feature.

In the following, we describe the further evolution of language features since the previous deliverable. We introduce *next* channels, a capability which helps to bridge the gap between stateless components and the need to act on historic data. This concept generalizes upon stateful components, and hence we take the opportunity to clarify various aspects around the initialization of stateful aspects, both next channels and stateful components. Furthermore, we propose robust error handling semantics, which allows explicit specification of behavior in the case of transient faults originating from the hardware. Finally, we extend the language to allow for the specification of *weakly-hard real-time* constraints on components.

2.1 Enhanced inter-iteration communication

The principal task model at the heart of the TeamPlay coordination language is the *Directed Acyclic Graph* (DAG). Components are structured as nodes in a DAG while channels between components specify both data exchange as well as precedence relations at the same time. Components themselves must be stateless, a constraint we leverage for the live migration of components between hardware resources.

State within a single component must be managed through the coordination language by means of *state ports*, a capability already presented in Deliverable D1.1. State ports form a combination of an inport, an outport of the same name and an immediate channel conneting the outport with said inport, so-called *state channels*. This way the inherent state of a component, where needed, is externalised and made known to the coordination layer. Such state channels go beyond the notion of a DAG and thus have semantics different from regular channels: instead of a precedence relation within the DAG, they specify a data exchange between one iteration (or instantiation) of the DAG and the subsequent iteration or instantiation.

We now extend this capability beyond single components and provide a more general inter-iteration communication facility. This extension of the TeamPlay coordination language is a direct response to collaboration between UvA and TNL on the radar surveillance system use case. The quintessential idea is to support special communication channels that connect the outport of one component with the inport of another component *in the subsequent iteration of the DAG*. Any inport/component can be the target of such a channel, which we call *next channel* motivated by the new keyword `next` introduced to qualify such channels. The target component can be both before or after the source component of the next channel (with respect to the structure of the DAG). In fact, next channels are not part of the DAG, neither do they create a dependency between the connected components within the structure of the DAG.

It is a common and crucial characteristic of both state ports/channels and next channels that they maintain the essential properties of a DAG while supporting inter-iteration data exchange in

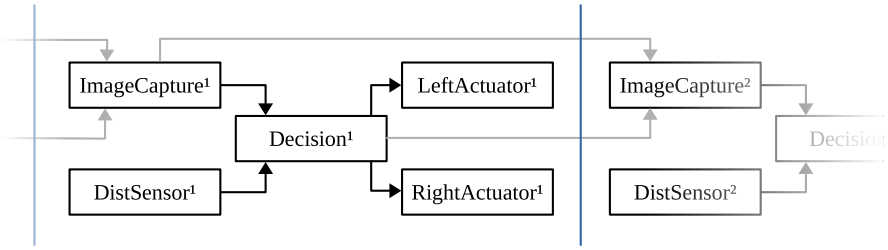

Figure 1: Example TeamPlay application as block diagram showing the interaction of five components across two iterations of a DAG

```
app car {
  components {
    DistSensor {
      inports { bool isMoving; }
      outports { num dist; }
      state { num prevDist; }
    }
    ImageCapture {
      outports { frame frameData; }
    }
    Decision {
      inports { num dist;
                frame frameData; }
      outports { num voltage;
                 bool isMoving; }
    }
    LeftActuator {
      inports { num voltage; }
    }
    RightActuator {
      inports { num voltage; }
    }
  }
  channels {
    DistSensor.dist -> Decision.dist
    ImageCapture.frameData -> Decision.frameData
    Decision.voltage -> LeftActuator.voltage & RightActuator.voltage
    Decision.isMoving -> next DistSensor.isMoving;
  }
}
```

Figure 2: TeamPlay coordination code of example application shown in Figure 1 illustrating inter-iteration communication via state channels and next channels

a controlled and orderly fashion.

Figure 1 illustrates both flavors of inter-iteration communication with a simple example. In this example, the `DistSensor` component takes its own previous distance value via the `prevDist` state channel as input for its subsequent instantiation. In Figure 2 we show the corresponding TeamPlay coordination code.

The novel form of inter-iteration communication can be identified by the *next* key word in the

last specified channel connecting the `isMoving` port of the `Decision` component with the `isMoving` port of the `DistSensor` component, but now in the subsequent iteration of the DAG. This way, the `DistSensor` component can take the latest moving status of the vehicle into consideration when running. This data cannot be computed in the current iteration of the (periodic) task graph. Instead, it is obtained from the previous iteration by the `Decision` component. The availability of data computed by previous iterations allows the `DistSensor` to provide a more accurate estimation of the distance (in this given small example).

2.2 Component and channel initialization

While Figure 1 shows steady-state behavior, there is (quite obviously) no previous data to provide to `DistSensor` in the very first iteration of the DAG. The same issue arises for state ports/channels, where components require some sort of initial data on the state channel to commence operation. We initially regarded the initialization of state ports/channels as merely an engineering problem in the interfacing between the coordination layer and the component implementation layer. Typical solutions could be to initially use a channel type-dependent default value. However, with the introduction of next channels we revisited the problem and revised our initial decisions.

In fact, we observe that component initialization can be generalised from the technical needs of state channels and next channels to application-specific initialization procedures, for instance to set up hardware sensors or actuators (e.g. setting GPIO pins or loading a configuration). If components need to perform such initialization actions, their initial runtime behavior in terms of time and energy may differ from those of subsequent instantiations. The same very likely holds for the initial instantiations of components that depend on data from previous incarnations of the DAG. In order to cover all such cases and provide a uniform programming interface for initialization, we leverage the multi-version component feature of TeamPlay (see Deliverables D1.1 and D1.2 for details) and introduce *initialization versions*.

The coordination specification, as shown in Figure 2, is accompanied by a set of *Non-Functional Properties* (NFP) contained a in CSV file, named *NFP file*. This concept was introduced with the revised syntax brought by Deliverable D1.2. We leverage multi-version programming for initialization of components: one or more versions may be flagged as *initialization versions*, requiring their use in the initial iteration of the task graph instead of one of the regular versions. As usual for multiple versions, we can specify individual WCET and other properties for these versions to accomodate for the extra (or less) time required for initialization. The semantics of changing versions at runtime are identical to those of *mode switching*, which is an explicit language feature introduced in Deliverable D1.2.

Employing multi-version components for the purpose of initialization requires that regular and initialization versions of a component have the very same interface to the component implementation. It lies in the nature of the solution we adopted that the coordination layer runtime environment cannot call the corresponding implementation of an initialization component with useful arguments for state and next channels. By convention component implementations must expect but not use the corresponding function arguments.

2.3 Error detection and handling

A managed runtime environment like that of the TeamPlay coordination language offers several opportunities to identify misbehavior of individual components:

- The component execution yields a segmentation fault or similar interrupt signal such as division-by-zero. This can be caught by a signal handler installed by the coordination runtime environment.
- The component implementation internally detects an inconsistent state and raises a user-defined signal, again to be caught by a signal handler installed by the coordination runtime environment.
- The execution time of a component instantion exceeds the component's worst case execution time (WCET) as specified in the NFP file or exceeds a time threshold below the WCET that likely indicates erroneous behavior of the component, again to be specified in the NFP file.
- The component runs under double modular redundancy (see Deliverable D1.1) and the results diverge.
- The component runs under triple (or higher) modular redundancy (see Deliverable D1.1) and all results diverge.

In all the above cases we assume that the origin for the observed misbehavior is not a systematic deficiency in the component implementation that would re-occur from DAG iteration to DAG iteration but instead a transient occasional misbehavior caused by single event upsets (SEU) or similar. Unexpected timing behavior may, for instance, also suggest a security threat in the system.

Regardless of the kind of misbehavior detection, the uniform consequence is absence of valid data (or presence of bogus data) on the outports of the misbehaving component. In the TeamPlay world of DAG-arranged dependent components any of the above error scenarios leads to subsequent components (with respect to the DAG) run on invalid input. Such invalid input might trigger further misbehavior further down the DAG. But even if dependent components run properly, invalid input data never leads to correct output data. Thus, erroneous behavior of actuators is the expected consequence.

The question is how can we deal with such scenarios on the level of the coordination language? For the purpose of explicit error handling we introduce the `on_error` clause, as illustrated in Figure 3. This can be added to a component specification or in the same way to a version specification to specify explicit error handling. The three dots in Figure 3 are a placeholder for one of a variety of potential actions to take upon detection of runtime misbehavior as sketched out above. The `on_error` clause can also be specified on application level, where it defines the corresponding behavior of all components uniformly. Such global specification may be overwritten by local exceptions for individual components or versions thereof.

In Figure 4 we illustrate the potential actions that can be taken upon the detection of erroneous operation of a component. In the following we will explain them one by one. While we generally

```
components {
  foo {
    inports {...}
    outports {...}
    on_error ...;
  }
}
```

Figure 3: Illustration of the novel `on_error` clause in the specification of a component

```
components {
  foo {
    ...
    on_error send_previous;
    on_error send_default;
    on_error skip;
    on_error skip_all;
    on_error switch_mode(<new mode value>);
    ...
  }
}
```

Figure 4: Illustration of potential actions associated with the novel `on_error` clause introduced in Figure 3

support multiple `on_error` clauses attached to a component, not all combinations of actions make sense and are supported. More precisely, the `switch_mode` action can be combined with any of the other actions and nothing else. In other words, the code in Figure 4 is merely for concise illustration, not literally legal (or useful) code. In the future we expect to add more actions as applications motivate their necessity or usefulness.

Action `send_previous` makes the coordination runtime environment send the failing component's output values of the previous instantiation to the subsequent components in the DAG.

Action `send_default` makes the coordination runtime environment send type-specific default values to the subsequent components in the DAG. Such default values are specified together with the type implementation in the NFP file.

Action `skip` makes the coordination runtime environment skip, i.e. not run, any of the subsequent components in the DAG that directly or indirectly depend on the failing component.

Action `skip_all` makes the coordination runtime environment skip, i.e. not run, all components of the DAG not yet started.

Action `switch_mode` makes the coordination runtime environment switch to the specified *mode*. This feature refers to TeamPlay modes as introduced in Deliverable D1.2. The mode to switch

```
modes {
  operationalState {Regular, UnderAttack}
}
components {
  foo {
    ...
    on_error switch_mode(UnderAttack) if Regular;
    on_error switch_mode(Regular) if UnderAttack;
    ...
  }
}
```

Figure 5: Illustration of combining the novel `on_error` clause introduced in Figure 3 with TeamPlay modes, as introduced in Deliverable D1.2

to must be a valid, previously defined mode value. The mode switch only becomes effective in the subsequent iteration of the DAG.

The `send_previous` action does (obviously) not work with initialization versions of components as introduced in the previous section. Neither does this action help any regular component upon first instantiation (in the absence of a specific initialization version, which might be regarded as the default and common case).

It might further be noteworthy that the `skip_all` action might lead to non-deterministic behavior. After all, it depends on the schedule (static or dynamic) which components still run and which not in case some component fails.

The new `on_error` clause integrates well with the TeamPlay modes, as proposed in Deliverable D1.2. The example in Figure 5 demonstrates this by means of a small example where we switch back and forth between two mode values upon detection of erroneous behavior of component `foo`. This might not be sensible to do in general, but the example nicely demonstrates how these two orthogonal features of the TeamPlay coordination language integrate with each other in a natural way.

2.4 Specification of weakly-hard real-time constraints

The field of weakly-hard real-time systems[7] recognizes that tasks can sometimes miss a deadline, and that such a deadline miss does not lead to catastrophic consequences provided that in subsequent iterations no further misses occur. The exogenous coordination approach of the TeamPlay language decouples components from their execution environment. As such, knowledge of the exact weakly-hard real-time tolerances of a task can allow for adaptation that is not possible otherwise. We discuss an example leveraging such capabilities under the name *strategy switching* in Section 4. For now, we limit ourselves to the introduction of the extended syntax for weakly-hard real-time constraints.

While a more formal definition of the task model captured by the weakly-hard real-time field is given in Section 3, our work in Task T1.1 has focused on expressing $\binom{h}{k}$ constraints. $\binom{h}{k}$ specifies

that for every k consecutive iterations at least h times the deadline is met. The values for h (hit) and k are specified at the component level using the following syntax:

```
components {
  foo {
    ...
    weakly-hard hk(1,2);
    ...
  }
  bar {
    ...
  }
}
```

Here, a $\binom{h}{k} = \binom{1}{2}$ constraint is specified for component `foo` while component `bar` has no such constraint. The proposed syntax allows us to support other weakly-hard real-time in future revisions of the coordination language. Components without weakly-hard constraint specification will implicitly receive a default $\binom{0}{1}$ constraint, i.e. their execution never needs to meet any deadline. This is a design decision aiming to aid in succinctly denoting large task graphs dominated by compute tasks (intermediate components neither connected to sensors nor actuators). Compute tasks themselves carry no intrinsic importance, rather their importance is exclusively derived from their (indirect) connection to one or more actuator tasks. As such, while they may be annotated with $\binom{0}{1}$ hk-constraints, that need not mean they never need to meet any deadline. Application designers can specify hk-constraints on their actuation components, and leave their compute components unannotated. It is then up to further processing steps applied to the coordination language to correctly propagate these constraints to compute tasks. We illustrate this in Section 4 by an example.

3 Validation of coordination language design

The objective of the TeamPlay coordination language is to allow the modeling of applications for cyber-physical systems (of systems) as a set of independent components, and allowing the specification of exogenous coordination between said components. A range of objectives such as reliability, security and performance can then be worked towards at the hand of *downstream* tools, techniques and methods developed by the ADMORPH consortium. While the goal of Task T1.2 is not to relate the coordination language to all downstream tools in the ADMORPH consortium, Figure 6 is presented to provide some context to the integration of the TeamPlay coordination language into the wider ADMORPH picture, with the *coordination specification* in the top-left corner.

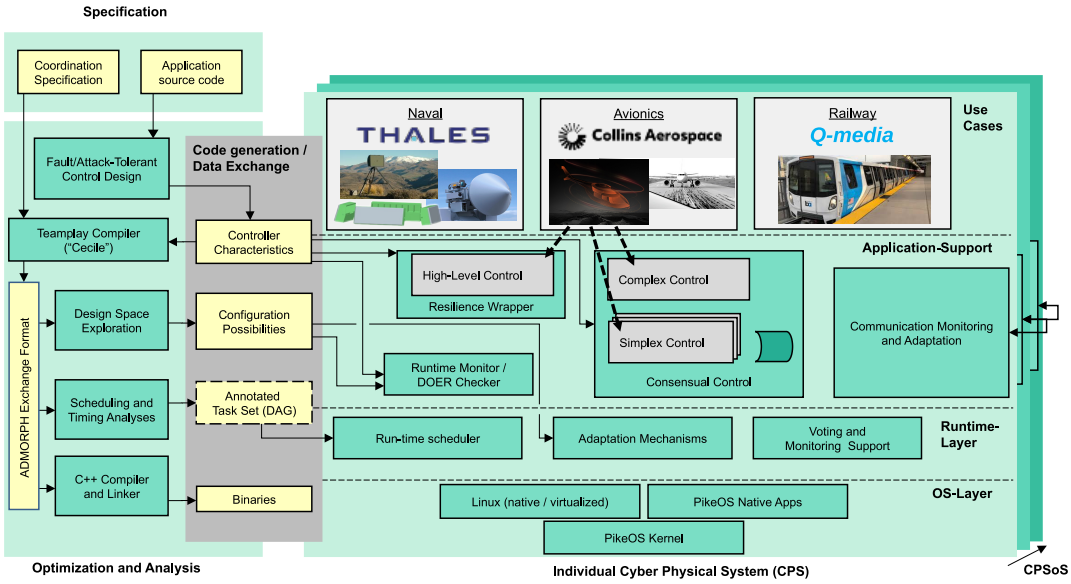

Figure 6: ADMORPH technology overview

Validation of the TeamPlay language is performed both from the view of the use case as well as the view of the coordination language. Each of the use case providers has provided input to this section, offering a use-case centric view of the coordination language. Likewise, the authors of the coordination language contribute a language-centric view on both the use case as well as further downstream tools developed within ADMORPH.

In alignment with our industrial partners, the use case providers, we pursue different approaches of validation. For the Autonomous Aerospace Systems use case (Section 3.1) and for the Railway Transport Systems use case (Section 3.3) we validate the TeamPlay coordination language by construction showing how (parts of) the use cases can be modelled. For the Radar Surveillance Systems use case (Section 3.2) validation is by critical review of design and features of the TeamPlay coordination language and its tool set.

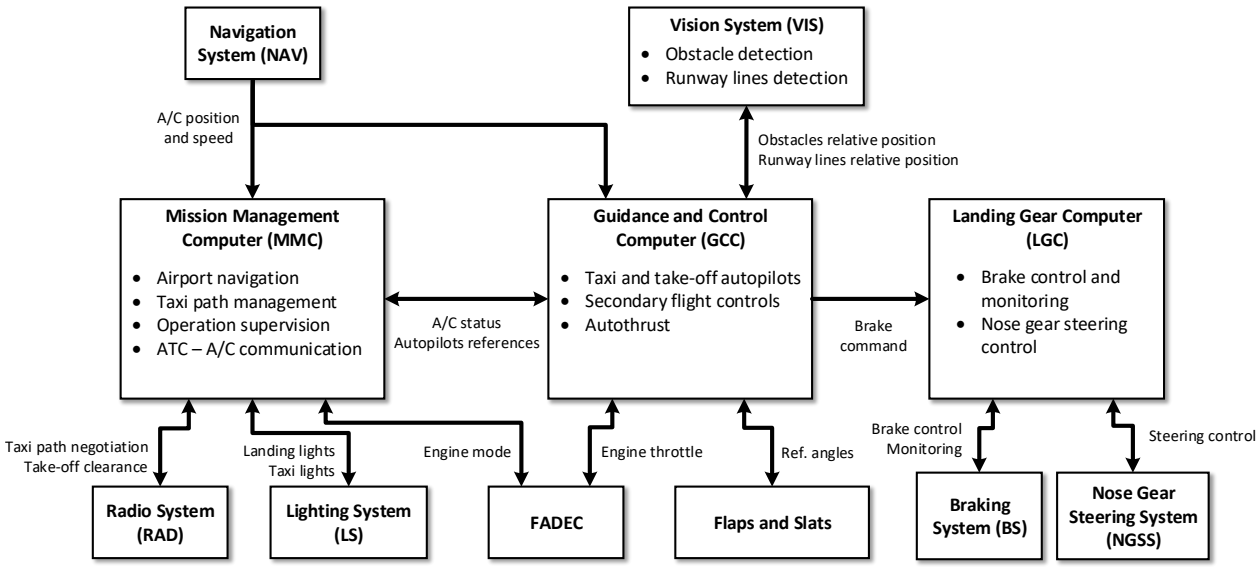

Figure 7: Functional architecture for autonomous taxi-out and take-off

3.1 Use case: autonomous aerospace systems

In Deliverable D5.1 *Requirement analysis and use case specification* [1] we outlined the adaptation requirements for the avionics use case. In this section we validate the TeamPlay coordination language with respect to those requirements. In particular, we study the expressivity of the language to capture, implicitly or explicitly, the adaptations required for a fault-tolerant safety-critical system capable of autonomous taxiing.

3.1.1 Example of execution of autonomous taxi

In a traditional aircraft the functionalities active during taxi-out and take-off are implemented and executed in three different computers. In an autonomous aircraft the tasks traditionally performed by the pilot are translated into new functionalities running in the avionics systems. The new functionalities for autonomous taxi-out and take-off were added to the functional architecture and the result is shown in Figure 7. They are comprised of the following computers:

- the Mission Management Computer (MMC),
- the Guidance and Control Computer (GCC),
- the Landing Gear Computer (LGC).

To illustrate the type of adaptation required we follow a mission execution example, as illustrated in Figure 8. When the aircraft is on the apron ready for taxi-out, a taxi-out clearance request is sent to the ATC by the Taxi Management System in the MMC. If the request is approved, ATC

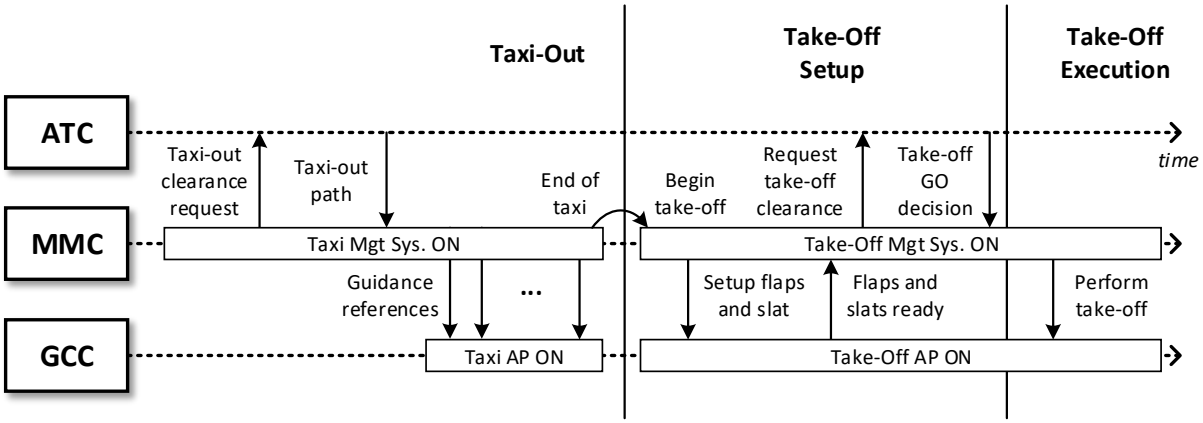

Figure 8: Mission phases and information exchange between ATC and aircraft computers.

replies with the path the aircraft should follow to reach the assigned runway. The path is used by the Taxi Management System in order to provide a series of reference positions to the Taxi Autopilot in the GCC. The reference points are used by the autopilot for generating the control signals for the engine, the brakes and the nose gear that will alow the aircraft to follow the assigned taxi path.

When the runway is reached and the aircraft is aligned with its direction, the Taxi Management System is deactivated and the *take-off setup* phase begins. The Take-off Management System in the GCC takes over the control of the aircraft and sends to the GCC the required flaps angle and slats deployment for take-off. The Take-off Autopilot configures flaps and slats as required and, once they are in position, notifies the Take-Off Management System. The latter can now send a take-off clearance request to the ATC. If the latter replies with a go decision then the aircraft enters into the *take-off execution* phase and the Take-off Autopilots performs the take-off.

Following this example it is clear that the involved computers of the autonomous avionics system use case are required to perform different functionalities in the various mission phases. For example, during taxi the GCC is responsible for executing the taxi autopilot, whereas during take-off GCC is responsible for executing the take-off autopilot. In addition, since these systems are safety-critical, they should be resilient against various and multiple occurences of faults and attacks. To do so the avionics use case defines the following adaptation requirements [1]:

1. the ability to adapt to the different mission phases according to the workload of each phase;
2. the ability to adapt to hardware faults by providing redundancy and/or degraded services;
3. the ability to adapt to software faults by providing degraded services;
4. the ability to adapt to attacks by providing redudancy and/or degraded services;
5. the ability to recover to a stable operation after one or more timing violations.

In the following subsections we validate the TeamPlay coordination language based on these requirements.

3.1.2 Adaptation to different mission phases

As outlined in the execution example (Section 3.1.1) of the auto-taxi and take-off use case, the involved computers are required to perform different functionalities in the various mission phases. Therefore, the system should be able to adapt when a mission phase change occurs in order to seamlessly transition from one phase to another. To be able to perform such adaptation at runtime, the system should be able to [1]:

1. load the set of tasks that correspond to the mission phase;
2. reconfigure the resource and time partitions according to the configuration decided at design time and the operational mode of the hardware components;
3. reconfigure the fault-tolerance mechanisms according to the demands of the mission phase and the operational mode of the hardware components, as needed.

With respect to reconfiguration of exeuted tasks depending on the mission phase, the Teamplay coordination language supports reconfiguration according to the opetation mode through the *modes* clause in the application and components. In addition, the coordination language supports reconfiguration of the fault-tolerance settings through the combination of the *modes* clause and the *profiles* that allow us to define the number of replicas of a particular functional component. With respect to reconfiguration of resource and time partitions, it is not the purpose of language to define such conctructs — and it should not be — but rather of the impementation. The CECILE compiler supports static scheduling of tasks for each of the different modes, hence this requirement is likewise satisfied.

3.1.3 Adaptation to hardware faults

Any critical system in the avionics domain is required to be able to sustain hadware faults. There are several ways of dealing with hardware faults at the task level. Typically these comprise redundancy, task replication and rescheduling. To be able to perform such adaptations at runtime, the system should be able to [1]:

1. stop tasks on a proccesing unit;
2. migrate tasks to the another proccesing unit, potentially of different type;
3. restart/replicate tasks on the same/different proccesing unit;
4. create/delete communication channels and modify their mapping;
5. modify the order of tasks and/or time partitions, whenever it is deemed safe.

These adaptation requirements effectively capture potential low-level hardware reconfigurations and their impact to task execution. While the coordination language is not meant to address such low-level reonfigurations, it does provides mechanisms to do so implicitly. In particular, the use of *modes* and error handling allows to restart/replicate tasks. In addition CECILE, with its capability for statically defined scheduling for the different *modes*, permits the halting and migration of tasks to other processing units as well as changing the scheduling order. Nevertheless, support for creating/deleting communication channels is limited. Finally, another important feature is that TeamPlay supports component checkpointing, that enables better task halt/restart capabilities.

3.1.4 Adaptation to software faults

Any critical system in the avionics domain is required to be able to sustain software faults. There are several ways of dealing with software faults, typically by re-execution on a degraded service mode. That is, an alternative way to provide the functionality is provisioned, already at design-time, which may take longer time to execute and/or provide services of lower quality. To be able to perform such adaptations at runtime, the system should be able to [1]:

1. stop tasks on a proccesing unit;
2. migrate tasks to the another proccesing unit, potentially of different type;
3. replace tasks with functionally equivalent ones;
4. modify the order of tasks and/or time partitions.

As analyzed in previous subsections, the TeamPlay language and its implementation CECILE provides the aforementioned adaptation requirements. In particular, the use of *modes* and error handling allows us to halt and restart tasks. In addition, the ability to define multi-version implementations of a component allows us to replace tasks with functionally equivalent ones. Finally CECILE, with its capability for statically defined scheduling for the different *modes* supports the halting of tasks, their migration to other processing units as well as changing the scheduling order.

3.1.5 Adaptation to malicious attacks

Protection mechanisms against malicious attacks typically are either preemptive, i.e. change the attack surface of internal parts of the system in order to minimize the attack windows, or reactive, i.e. respond to detection mechanisms indicating that some part of the system is under attack or already compromised. Responding to under-attack detection events involves altering the attack surfaces and/or providing degraded services that are less prone to attacks. On the other hand, responding to compromisation detection events broadly involves isolating the part of the system that has been compromised and trying to recover that part while continuing to provide services with the trustworthy part. In order to be able to perform such adaptations, the system should be able to [1]:

1. stop tasks on a proccesing unit;
2. migrate tasks to the another proccesing unit, potentially of different type;
3. replicate tasks and/or replace tasks with functionally equivalent ones;
4. delete/create communication channels and modify their mapping;
5. modify the order of tasks and/or time partitions.

Apart from these requirements, that are well covered by TeamPlay through its different *modes*, error-handling, checkpointing, scheduling and replication, the ability to define multi-version implementations of a component is an another adaptation to isolate and address comprised software by changing the components' behavior. This not only makes it harder for an attacker to consistentenly exploit a component vulnerability, but also permits execution schemes with diversity where component replicas have different implementations. Hence, not all replicas can be comprimised at the same time due to a particular vulnerability.

3.1.6 Adaptation to timing violations

There are multiple reasons for timing violations of tasks, i.e. not finishing before the deadline, among which are improper timing analysis, faulty software or hardware and denial-of-service or similar attacks. Regardless of the reasons, the system should be able to recover once such a violation occurs, in order to avoid cascading timing violations affecting the most critical functionalities. In order for a recovery mechanism to be able to establish a steady state, the system should be able to [1]:

1. stop tasks on a proccesing unit;
2. migrate tasks to the another proccesing unit, potentially of different type;
3. modify the order of tasks and/or time partitions;
4. modify communication channels and/or resource partitions (to speed-up/slow-down task execution);
5. adapt to actual execution time of tasks and reclaim unused processing time, whenever it is safe to do so.

With respect to timing violations, the Teamplay coordination language allows us to define strategies through the use of modes and error handling. In particular, the use of *modes* and error handling allows to restart/replicate tasks. In addition, the CECILE compiler, with its support for statically defined scheduling for the different *modes*, permits the halting and migration of tasks to other processing units as well as changing the scheduling order. Nevertheless, support for creating/deleting communication channels is limited. Finally, another important feature is that TeamPlay language allows for component checkpointing that enables better task halt/restart capabilities.

3.2 Use case: radar surveillance systems

In this section we validate the design of the TeamPlay coordination language in the form of a critical but constructive dialogue between TNL and UvA based on the Radar Surveillance Systems use case provided by TNL. It is in our view important to understand that TNL already has a software architecture tailored to their needs, whereas the principle design of the TeamPlay coordination language is application-independent by nature. The original design of TeamPlay also predates the ADMORPH project and as such is not the result of a requirements engineering process based on the ADMORPH use cases.

TNL: The exogenous approach and the choice of software components as the central artifact matches with the approach used within TNL. Besides input and output TNL components provide grouping of input and output into protocols (including behavior), additional functional contracts for parameters (i.e. configuration), observability (i.e. logging, metrics and tracing), timers (i.e. periodic control to the component) and threads (i.e. non-deterministic control to the component) all controlled by a predefined availability of these functional contracts in a predefined life-cycle of the component.

> **UvA:** Indeed, TeamPlay does not support the grouping of input/output into protocols, and it remains unclear for the time being how protocols could fit with the design of TeamPlay. We see support for logging, metrics and tracing as a mostly engineering aspect, of course a very useful and important one, even more so for industrial deployment. The component-based design of TeamPlay would facilitate such support as an orthogonal aspect during code generation. However, it would be future research how this integrates with the real-time capabilities of TeamPlay, an aspect that the TNL use case right now does not have. TeamPlay, however, does already support periodic, timer-based activation of components.

TNL: TNL components do not have a means to model any non-functional contracts, nor do we recognize the state of the component as additional output and input.

> **UvA:** Non-functional properties are at the heart of the TeamPlay design and approach.

TNL: The design of TeamPlay making their components actually completely stateless is neat indeed. Control to the component is something we think is desirable as long as each component state changes is provided as output. About stateful components: the solution provided here is both elegant and simple!

UvA: TNL components are state machines whose internal behavior remains opaque to the control layer. This prevents (or least very much hinders) adaptation through migration of components between hardware execution units, which is the core idea behind the ADMORPH project: morphing embedded systems. The TeamPlay component design externalises state (where the presence of state cannot be avoided) and thus makes it visible to the coordination layer in a controlled way. This is in our view a strong prerequisite for morphing embedded systems, without ruling out state entirely, which would very much reduce the practical applicability.

TNL: Apart from the timers and threads, TNL components adopt the firing rule of Petri nets by activating the component as soon as data (tokens) are available on each input of an interface, where an interface is a grouping of inputs and outputs of the component. Besides that, timers activate the component periodically and threads can activate the component at any time (non-predictable), both can lead to data (tokens) on outputs. This means that each TNL component with timers and thread can also be source components.

UvA: Like TNL components, TeamPlay components adopt the firing rule of Petri nets while source components (with respect to the DAG) are triggered periodically through system timers. This aspect appears well aligned.

TNL: Technically a TNL component is a C++ class, adhering to the C++ calling and linking convention, whose name and signature can be derived from the component specification in a defined way.

UvA: Technically, TeamPlay components are C functions with their function signature (or prototype) serving as interface specification. This difference, C++ class versus C function, is a direct conseqence of the design choices made regarding state: the class encapsulates a state definition alongside the methods to manipulate said state. In contrast, the adoption of a C function reflects the state-free nature advocated by TeamPlay with a purely operational facette that maps input tokens to output tokens, very much like in Petri nets.
We see the choice between C and C++ as a minor difference. TeamPlay uses C as implementation language and even restricts the admissable language features in favor of real-time behavior and analyzability. Real-time is no concern for TNL components, and the focus on state machines as components makes C++ a quite natural choice.

TNL: We have a question on multi-version components: The different configurations / implementations are now modeled as different versions of the same component. Given that the component is a C function: how are these aspects made available to the C function?

UvA: Different versions of a TeamPlay component are implemented by individual C functions with version-specific names. All version implementations of one component share the same function signature.

TNL: With TNL components leaning on a dynamic (nano) service environment (like OSGi), this could be implemented as different versions of the services that provide the non-functional properties while using the exact same code for the functional implementation of the component. In TNL we aim to let the component be completely agnostic of the non-functional attributes / services required. For communication we have this already in place by having the ability to configure interceptor services implementing the non-functional contracts.

> **UvA:** TeamPlay does not and cannot rely on heavy-weight mechanisms or services for version selection given its targeting of real-time systems. The choice of version is taken either statically with no runtime overhead or dynamically in a very light-weight form to control runtime overhead, again in light of real-time requirements applying to TeamPlay. The non-functional properties or contracts we advocate for TeamPlay components are in fact guaranteed runtime behaviors, mainly in terms of time and energy requirements for execution. They are not extra-functional services that versions provide. In this sense TeamPlay components themselves are likewise agnostic of their non-functional properties/contracts.

TNL: TeamPlay is 'polluted' by code language specific attributes (type names in C lingo). The attribute has no meaning in TeamPlay, but is used one-on-one for code generation. This appears to be a leakage of abstraction. The TNL type/component language does not cross this boundary. Types defined in the type DSL are translated into potentially multiple target languages. The model can be annotated (marked in terms of MDA) with attributes meant for each individual code generator.

> **UvA:** Channel types in TeamPlay represent a light form of consistency checking in the sense that only ports with the same type may be connected by channels. They are actually not used one-on-one for code generation. There is a translation phase involved where the type names used on the coordination layer are mapped to C implementation types during code generation.
> TeamPlay does not have a type specification language as the TNL component layer, and in fact type implementations are opaque to the coordination layer. For the time being, we do not support other languages than C (due to our focus on real-time) and, hence, we neither have nor need a type compatibility or translation layer between different implementation languages.

TNL: Channels connect input to output ports. In TNL the connections are determined at runtime (although we would like to have additional modelling to model an orchestration of components, on which checking of sufficient connectivity can be performed). Dynamic resolution of connectivity does allow to add additional (monitoring) components on the fly, or change the implementation in specific situations without the need to re-edge the whole application. To determine the connections we utilize OSGi's Capability/Requirement resolution: A component can provide a set of services and require a set of services.

UvA: Channels in TeamPlay, are indeed static. This is a requirement for static analysis regarding time and energy consumption of an application. This restriction is somewhat eased by TeamPlay *modes* that allow us to activate and deactivate channels when switching modes. However, in the interest of analyzability switching from one application mode to another is rather equivalent to switching between an a-priori given number of configurations, where each individual configuration remains statically analyzable.
Nonetheless, TeamPlay modes could be sufficient to dynamically switch on and off monitoring components as mode switches are triggered dynamically. This would merely require the a-priori design of an application including monitoring components. Our focus on real-time properties and static analyzability rules out the adoption of any heavy-weight dynamic mechanisms such as OSGi's capability/requirement model.

TNL: In the TNL model, components with only required services are pure source components while components with only provided services are pure sink components. Transformer components (those with both provided and required services) can act as pure transformer components, but also as source or sink components of a subset of the data exposed on their required / provided services.

UvA: This description pretty much fits the notion of components in the TeamPlay coordination language as well.

TNL: Providing non-functional properties through separate NFP files is a neat solution. We may look into this again with TNL component language. We used to have these kind of constructions, but we moved to annotating the original source files, as this involved less typing. The downside of our solution becomes apparent once one needs a marking for a different platform or different application configuration.

UvA: In the TeamPlay context, non-functional properties, e.g. time and energy consumption of a component execution, are inherently specific to the architectural unit they run on. Hence, we decided early in the design process that such information should not be part of the coordination code, but rather reside in a kind of database: in essence, a 3-dimensional associative memory with component names, architectural unit names and non-functional property names as keys.

TNL: Specifying the target architecture of a compoent, or version thereof, in the coordination code (e.g. Figure 7 in Deliverable D1.1) is an abstraction breach. Security can be part of the component language as it has been completely de-coupled from any implementation by just using numbers.

UvA: The `arch` property of components or their versions is in our view not a breach of abstraction as it merely refers to an architectural unit within a chosen heterogeneous multi-processor system (on-chip or off-chip), not arbitary systems as a whole. Unfortunately, this is not made sufficiently clear in Deliverable D1.1. Nonetheless, we agree that the choice to have this information in the coordination code and not in the NFP file is debatable. We will reconsider and potentially revise this design choice in future versions of TeamPlay.

TNL: Multi-version components: For a next project: maybe rephrase to *multi-configuration*, *multi-aspect* or *multi-implementation* components? The term *version* is usually related to evolution of the functionality of a component and in this case the various levels of encryption being available is not an evolution, but it describes different configurations / implementations of the same functionality or maybe different aspects of the same functionality.

UvA: We agree that the term *version* is overloaded, but it is in our opinion not restricted to consecutive versions along a development timeline. We further believe that the proposed terms are equally overloaded as the term *version*. For instance, the term *aspect* reminds one of aspect-oriented programming. Multiple versions are not different aspects of one component, each version is a complete implementation of the functionality. In this sense, the term *multi-implementation* appears more fitting, but then multiple versions could also be derived from the very same implementation through different compilation paths or configuration parameters.

TNL: We had rather seen the various fault-tolerance methods (i.e. check-point restart, DMR, TMR, etc.) be specified in the NFP file instead of in the coordination code proper. After all, there is no need for the implementation to be aware of the fault-tolerance regime, if any.

UvA: In TeamPlay the component implementation would indeed not be aware of any fault-tolerance technique applied to the component. Whether to put the fault-tolerance specification into the coordination program or into the NFP file could, once more, be debated. The rationale to have this information in the coordination program is that we see fault-tolerance as an integral part of coordination to be separated from component implementation. However, it is also true that fault-tolerance is an orthogonal aspect compared to the specification of the core application, its components and communication channels.

TNL: A topic that we have not seen elaborated on so far is how to recognize when the middleware can discard the input, since not all outputs need to be triggered.

UvA: On the coordination layer we never discard inputs of a component. Generally, we assume that component implementations do need their input for operation. Of course, any component implementation is free to ignore its input, but then it is up to the component implementation what to do with the input data. (See, for instance, the discussion on initialization versions in Section 2.2.)
We might though need to discard output of a component if the corresponding outport is not connected to inports further in the DAG. In our YASMIN runtime system [41] data between components is exchanged via static buffers, due to our focus on real-time applications. Data in the static buffer is overwritten after each component instantiation, but never read if the outport is not connected to subsequent components.

TNL: Another topic that we would like to see elaborated on is how the generated code facilitates a "I'm ready producing outputs / state" notification. Or, is producing state regarded as the last action? How is production of *all outputs* made a discrete event?

UvA: This is indeed a very important topic. Considering components may fail at any time, we aim to keep the time period of transition of control between component implementation and coordination runtime environment as small as possible. For example, component implementations never incrementally generate output on their outports. More precisely, a TeamPlay component implementation C function expects, among others, pointers to buffers for each inport and each outport. Transfer of control between the coordination runtime environment and the component implementation boils down to the C function call and return protocol.
In the above discussion we refer to our YASMIN runtime system [41]. In other runtime systems that we indirectly support via the ADMORPH exchange format (AXF) this may be done differently.

TNL: TNL is interested in fault-tolerance features like check-point restart, standby and NMR, but since the TNL middleware does not support such features no real effort has been made into that direction Also the cost of replication in terms of processing power has led TNL to focus on creating a stable application in the first place thus far. We do see development on application level towards the implementation of standby solutions, so TNL could certainly benefit from applying this to their own modelling environment.

UvA: The processing cost of the typical fault-tolerance features is indeed non-negligible, but this is not an artifact of the TeamPlay coordination language but rather inherent to the concept of replication. Teamplay merely facilitates the employment of such features. However, at least in terms of latency today's abundance of parallel processing power mitigates the cost to a considerable degree.

TNL: How does TeamPlay implement voting in case of NMR fault-tolerance? Determining whether or not the output is correct requires application-level knowledge, so despite that the actual usage of NMR, for both the number of replications, which actual implementation to choose and how many voters, should be off-loaded to NFP. The component implementation should also indicate what voting implementation to use.

UvA: Voting can only be realised at the component implementation layer, not the coordination layer of an application because the actual data type implementations are unknown to TeamPlay. Technically, we insert implicit voter components into the DAG whose names are derived from the components to vote on. The component implementations can be constructed by the compiler based on type-specific voting functions provided

TNL: Profiles, cascading them and the keywords `remove` and `vital` are recognized from the way TNL handles configuration of components. It makes sense to allow this for NFP, but again: defined separately from the implementation of the component; the component implementation should not be aware of the NFPs applied to it (apart from enabling the ability to configure certain NFPs by providing the proper implementation, e.g. voting functions).

UvA: TeamPlay component implementations are fully agnostic of fault-tolerance features as they are of their position within the application DAG.

TNL: Sub-networks provide re-use of implementation only. TNL component language by requiring/providing services firstly relies on re-use of interfaces, and later when orchestrating the final solution we would like to add to that the re-use of implementation (collection of component implementations and their NFPs).
Even templates are at the bottom line a re-use of implementation, whereas re-use of interfaces is what would be needed at that level. For instance, one could have a Sensor with three components, or one with five, and they could still honor the same Sensor interface. I do not see (at the moment of writing) how I could achieve that using TeamPlay, other than to provide a ThreeComponentSensor and a FiveComponentSensor.

UvA: The observations on the restrictions of sub-networks and templates are correct. They are at the end of the day software engineering convenience features, but they do not allow to express programs that could not be written without these features, albeit with considerably more code and all the corresponding software engineering issues.
We do agree that a language extension towards interface reuse, as sketched out above, would be desirable for TeamPlay as well. In the light of the man-power we have for this kind of work in the context of the ADMORPH project, we must consider such extensions future work, unfortunately.

TNL: Component and Function names: Could this be the link between re-use of implementation and re-use of interface? But how does the cname defined in a profile that contains multiple components be linked to the correct component? Can I address individual components in a profile?

TNL: The evolution of the TeamPlay coordination language, as described in Deliverable D1.2, has moved some low level aspects (e.g. component implementation C function names) from the coordination specification to the NFP file. This is a good design choice. We further deem and the addition of modes and the ability to start certain versions of a component based on these modes good evolutions of the coordination language. Both evolutions are good examples of what we would like to see in a coordination language.
We also really like the modelling of the non-functional aspects, if only as inspiration for the DSLs we use internally. The way how "state" has been included in the modelling is a good example for this.

> **UvA:** We take these comments as encouragement that we are on the right way with the evolution of the TeamPlay coordination language.

TNL: What TNL expects from a coordination language:

- The ability to describe the contents of messages in a functional way.
  Say we have platform data with role and pitch in radians (between and not including minus and plus Pi). We expect the language workbench to be able to generate the proper code to handle this message in diverse computing environments like C, C++, Rust or Java.
- The ability to specify contracts (services?), preferably with protocol, i.e. behavior. In these contracts we want to refer to the functional messages to indicate that these messages are exchanged when realizing this contract. Teamplay links single messages as dependencies, while we would like to see dependencies between protocols, i.e. a component provides or requires a service (a set of messages and a protocol on in which order to send and receive these messages), which is probably a reflection of that TeamPlay originated in an environment of mainly stream-oriented processing while in our systems we usually deal with distributed asynchronous communicating components.
- The ability to specify an aggregation of contracts.
  We are not sure how to properly name this, maybe function? In one of our DSL solutions we have called this template, but it comes close to what TeamPlay describes as component. The only difference is that at this point it is not sure if the realization is a single component or a set of communicating components. Besides aggregating contracts / services into a single not-yet-component one should also be able to indicate if this element requires (functional) configuration, what it should do regarding observability and other generalized blocks of functionality (all at functional level).

- Components that a realizations of a function or template, linked to a language in which the realization is implemented.
  They may define additional elements besides those defined in the template / function, usually for test purposes and this time the level of information can also be technical).
  A component can require hardware capabilities, e.g. micro-architecture, processing power, number of cores, memory, network bandwidth, etc..
- Systems that are realizations of a function or template, but instead of components it contains templates / functions that later can be realized by either components or systems.
  In a system you could use channels to link the elements in the system, but they indicate (as in TeamPlay) nothing but a dependency between a provided and required protocol / service. The top-level system does not necessarily need to realize a function or template, but it could do so for external interfaces.
- A means to describe hardware, also with templates (functions) components and systems, providing capabilities like architecture, processing power (number of cores?), memory, network bandwidth, etc.
- A means to describe system realizations per system, i.e. which components or system realizations, are a realization for the functions / templates defined in the system, and a means to provide additional information for if there is ambiguity between the provided and required capabilities.
  This is the point where the decision is made if a function / template is realized by a single component or by a system realization. The capabilities required by the software components should be satisfied by the capabilities provided by the hardware components. If necessary, additional properties can be provided that are dedicated to the current solution, e.g. which software component runs on which hardware components in case of ambiguity in provided and required capabilities.

In the set of bullets above the more specific models can refer to the more generic models, and never vice versa. In that regard the evolution of TeamPlay, especially with the evolution of the non-functional properties file, is a step in the right direction. The elements mentioned in the list above are highly inspired by architectures based on functional decomposition.
Many aspects of the radar surveillance use case could be modelled with the TeamPlay language, but one would need a description next to the language workbench to map the concepts used in our systems to the concepts of the TeamPlay language (or the other way around). It could (almost) fit, if each element in the language can be interpreted slightly different. Given that we are talking about Domain Specific Languages, a different DSL may be better suited in order to describe a functionally decomposed system of asynchronous communicating components, and use TeamPlay to describe dataflow oriented synchronous (or at least processing in sequence) components.

UvA: During the intense discussions between TNL and UvA, of which the above dialogue is a condensed excerpt, it turned out that the TeamPlay coordination language, as is, is not a perfect match for the needs of TNL. This is not entirely surprising as the TeamPlay language was designed before the start of the ADMORPH project and the on-going collaboration between UvA and TNL.
Our collaboration has also led to the insight that the design goals of TeamPlay and those of the radar surveillance systems use case are overlapping but not identical. This manifests itself by the different design choices taken by TeamPlay and by the existing software architecture of the use case. TeamPlay targets (soft) real-time systems with requirements on energy consumption and fault-tolerance. The focus of TeamPlay is on static analyzability and dynamic adaptation, hence the emphasis on the handling of (unavoidable) state. In contrast, the coordination layer in the TNL software architecture is based on asynchronously operating state machines that may communicate with each other in arbitrary ways. The focus here is not on controlling resource consumption (time, energy) or static guarantees, but rather on keeping a dynamic system operational, though without making use of key ADMORPH concepts such as dynamic adaptions (i.e. morphing) or replication.
Despite these differences in perspectives there is also a surprising level of overlap and more importantly cross-pollination. The above list of bullet points will surely taken into account by UvA in the future evolution of TeamPlay. Likewise, TNL identified a number of attractive features in the TeamPlay coordination language, e.g. the handling of state, that may in one way or another find their way into future versions of the TNL middleware DSLs.

3.3 Use case: railway transport systems

As part of Task T5.4 Q-Media has developed a demonstrator that uses ADMORPH technology, including the TeamPlay coordination language. The goal of T5.4 is to exploit ADMORPH tools to create robust and reliable communication between moving trains ground stations alongside the railway tracks. In this deliverable, we present the demonstrator through the lens of the coordination language, thus providing validation by construction.

3.3.1 Architecture overview

The railway application runs on a T1040 PowerPC CPU, partitioned using PikeOS. A block diagram of the system architecture is shown in Figure 9. The software can be structured as a set of periodic tasks with a period of 100ms, shown in Figure 10. Compute tasks are shown in light green color while actuator tasks are shown in primary colors towards the right hand side of the figure.

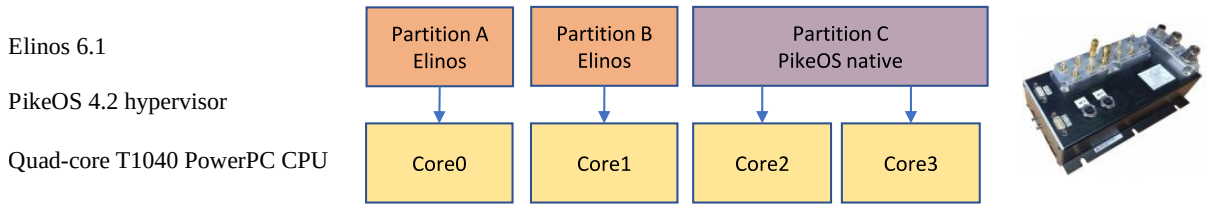

Figure 9: Hardware overview of the system

The application must be able to switch between a set of discrete modes. These modes relate to the domain of the application and are determined by the task `P1`, as shown in Figure 10. A mode

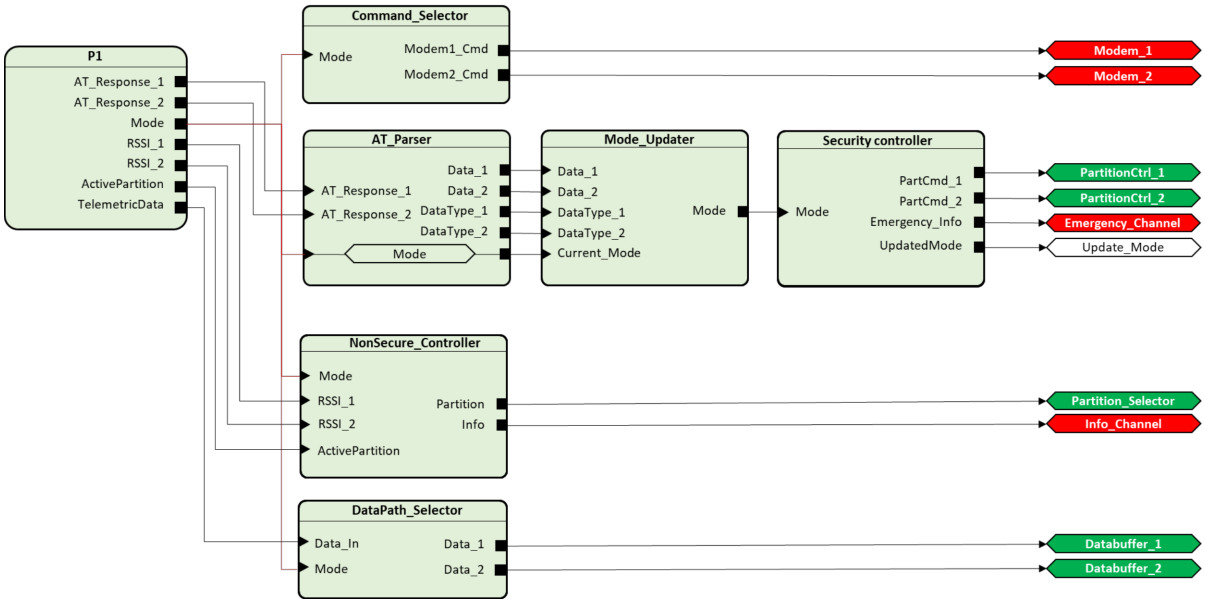

Figure 10: Train application structured as a set of periodic tasks with precedence relations

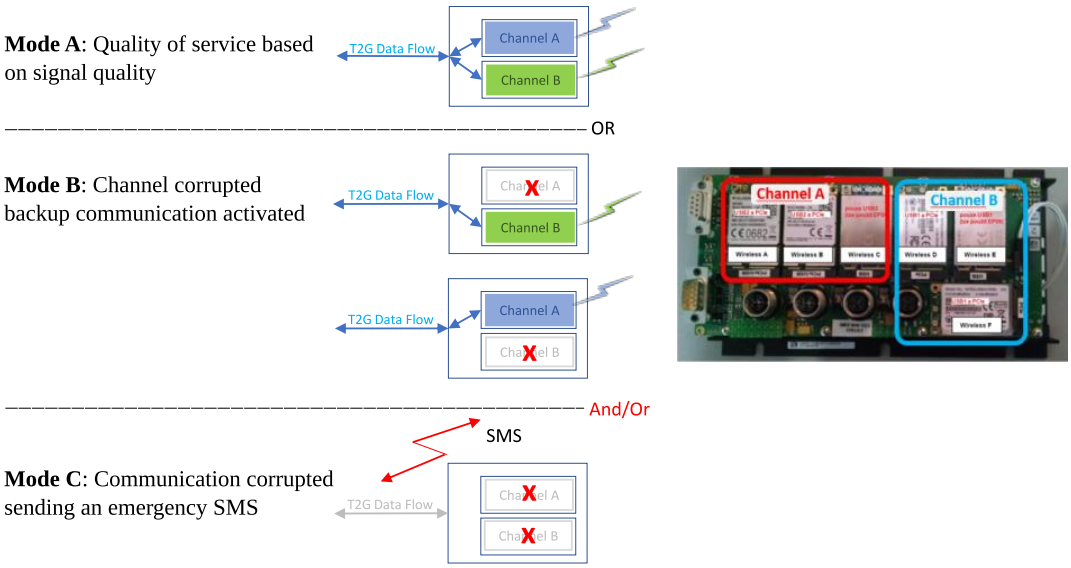

Figure 11: Supported operating modes showing different *Train to Ground* (T2G) paths

is only valid for one iteration of the task set. The modes that must be supported are shown in Figure 11.

3.3.2 Specification as TeamPlay coordination code

The structure of the task graph as shown in Figure 10 lends itself well to be translated to the TeamPlay component model. Edges are translated to channels. TeamPlay does not disambiguate between compute tasks and actuation tasks, as such all of these must be mapped to regular components. While TeamPlay has support for modes, it was decided not to use these TeamPlay modes as the very same schedule can be used for all application modes. Consequently, the modes shown in Figure 11 are purely application-specific and not present in the coordination file. With this setup, a specification in the TeamPlay coordination language can rather straightforwardly be derived from the application architecture. We show this specification in Figure 12.

3.3.3 Downstream tooling

The TeamPlay coordination file is read by the CECILE compiler, one of the *downstream tools* developed in the context of work package WP1. The output of Cecile could be used in combination with PikeOS, a workflow highlighted in Figure 13. Figure 14 shows a schedule produced by CECILE and its relation to the software architecture as shown in Figure 10.

3.3.4 Conclusion

In this section, we have demonstrated the ability to implement our application using the TeamPlay coordination language. We at Q-Media use channels to model both logical exchange and precedence

```
app Railway {
  period = 100ms;
  deadline = 150ms;
  components {
    P1{
      outports {Str_t AT_Response1;}
      outports {Str_t AT_Response2;}
      outports {u16 Mode;}
      outports {PST_TelemetricData_t
                  TelemetricData;}
      outports {s32 RSSI_1;}
      outports {s32 RSSI_2;}
      outports {u8 ActivePartition;}
    }
    CommandSelector {
      inports {u16 Mode;}
      outports {ET_ModemCmd_t Modem1CMD;}
      outports {ET_ModemCmd_t Modem2CMD;}
    }
    AT_Parser {
      inports {Str_t AT_Response1;}
      inports {Str_t AT_Response2;}
      inports {u16 ModeIn;}
      outports {u16 ModeOut;}
      outports {s32 Data_1;}
      outports {s32 Data_2;}
      outports {u8 DataType_1;}
      outports {u8 DataType_2;}
    }
    Security_Controller {
      inports {u16 Mode;}
      outports {u8 PartCmd_1;}
      outports {u8 PartCmd_2;}
      outports {u8 EmergencyInfo;}
      outports {u16 UpdatedMode;}
    }
    NonSecureController {
      inports {u16 Mode;}
      inports {s32 RSSI1;}
      inports {s32 RSSI2;}
      inports {u8 ActivePartition;}
      outports {u8 Info;}
      outports {u8 Partition;}
    }
    DataPath_Selector {
      inports {PST_TelemetricData_t
                 Data_In;}
      inports {u16 Mode;}
      outports {PST_TelemetricData_t
                  Data_1;}
      outports {PST_TelemetricData_t
                  Data_2;}
    }
    Mode_Updater{
      inports {s32 Data_1;}
      inports {s32 Data_2;}
      inports {u8 DataType_1;}
      inports {u8 DataType_2;}
      inports {u16 Current_Mode;}
      outports {u16 Mode;}
    }
    Modem_1 {
      inports {ET_ModemCmd_t ModemCMD;}
    }
    Modem_2 {
      inports {ET_ModemCmd_t ModemCMD;}
    }
    Update_Mode {
      inports {u16 Mode;}
    }
    Partition_Control {
      inports {u8 Command1;}
      inports {u8 Command2;}
    }
    Emergency_Channel {
      inports {u8 EmergencyInfo;}
    }
    Partition_Selector {
      inports {u8 Partition;}
    }
    Info_Channel {
      inports {u8 Info;}
    }
    Databuffer_1 {
      inports {PST_TelemetricData_t Data;}
    }
    Databuffer_2 {
      inports {PST_TelemetricData_t Data;}
  } }
  channels {
    P1.Mode -> CommandSelector.Mode;
    P1.AT_Response1 -> AT_Parser.AT_Response1;
    P1.AT_Response2 -> AT_Parser.AT_Response2;
    P1.Mode -> AT_Parser.ModeIn;
    P1.Mode -> NonSecureController.Mode;
    P1.RSSI_1 -> NonSecureController.RSSI1;
    P1.RSSI_2 -> NonSecureController.RSSI2;
    P1.ActivePartition
      -> NonSecureController.ActivePartition;
    P1.Mode -> DataPath_Selector.Mode;
    P1.TelemetricData -> DataPath_Selector.Data_In;
    CommandSelector.Modem1CMD -> Modem_1.ModemCMD;
    CommandSelector.Modem2CMD -> Modem_2.ModemCMD;
    Mode_Updater.Mode -> Security_Controller.Mode;
    AT_Parser.Data_1 -> Mode_Updater.Data_1;
    AT_Parser.Data_2 -> Mode_Updater.Data_2;
    AT_Parser.DataType_1 -> Mode_Updater.DataType_1;
    AT_Parser.DataType_2 -> Mode_Updater.DataType_2;
    AT_Parser.ModeOut -> Mode_Updater.Current_Mode;
    NonSecureController.Partition
      -> Partition_Selector.Partition;
    NonSecureController.Info -> Info_Channel.Info;
    DataPath_Selector.Data_1 -> Databuffer_1.Data;
    DataPath_Selector.Data_2 -> Databuffer_2.Data;
    Security_Controller.PartCmd_1
      -> Partition_Control.Command1;
    Security_Controller.PartCmd_2
      -> Partition_Control.Command2;
    Security_Controller.EmergencyInfo
      -> Emergency_Channel.EmergencyInfo;
    Security_Controller.UpdatedMode -> Update_Mode.Mode;
  } }
```

Figure 12: Complete TeamPlay coordination code of the railway use case

Figure 13: Workflow using downstream tools

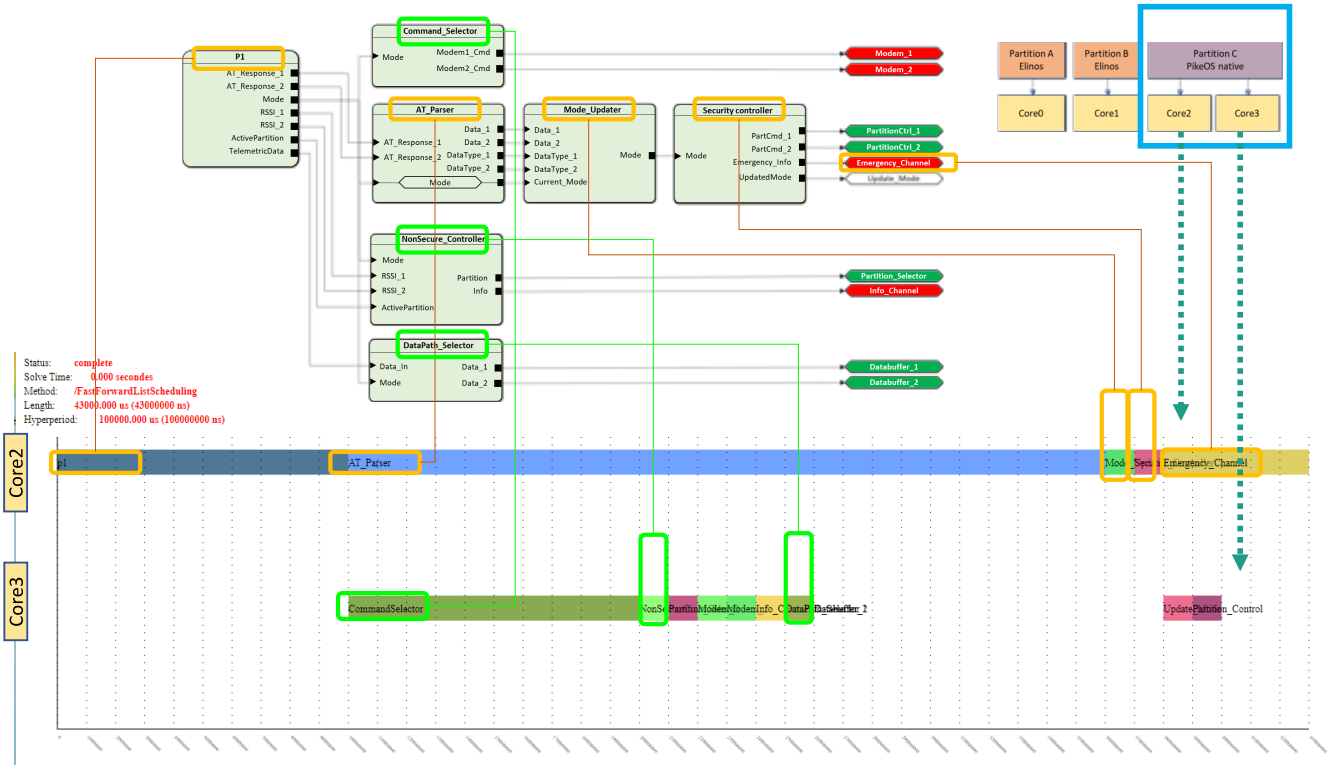

Figure 14: Schedule annotated with corresponding parts in the software architecture shown in Figure 10

relations between tasks/components within our application. Mode switching was not done via the dedicated coordination language syntax as it does not match our needs. Acording to the semantics of the coordination language a mode switch engages only on the next iteration of the task set, whereas we rather need a mode switch to take immediate effect, i.e. to affect the current iteration. At the same time, we have no need for complex machinery such as schedule changes, but can stick to a static schedule where some tasks (depending on the mode) are not active. Our case for mode switching does not have the complexity to benefit from a coordination language approach, and as such we decided to implement this aspect directly in application code.

While the development of a runtime system environment is not in the scope of work package WP1, UvA nonetheless investigated the feasibility of porting our existing YASMIN runtime environment to PikeOS on PowerPC, the architecture chosen by Q-Media for this use case. Unfortunately, this turned out infeasible with the resources allocated as well as with the focus of work package WP1 on other aspects.

UvA and Q-Media discussed potential adoption of the coordination technology, but the relatively low TLR of the coordination language ecosystem hinders direct economic/industrial exploitation. However, Q-Media may nonetheless be able to extract value from coordination compiler CECILE as one key aspect of UvA's tool chain. Currently, CECILE is GPL-licenced, but Q-Media and UvA are in touch regarding potential relicensing.

4 Strategy Switching

As transistor sizes decrease and voltages are lowered, modern computer systems have to content with an increasing probability of encountering a *Single Event Upset* – or SEUs. Single Event Upsets are momentary or *transient* faults in integrated circuits, typically caused by cosmic rays. While no lasting damage is done to the hardware, the effects of the SEU modify the state of the program running on said hardware. Mitigation against SEUs can roughly be split into two categories: (1) hardware-based mitigation, e.g. by radiation hardening, and (2) software-based mitigation. Software-based mitigations typically employ some form of redundancy. Some unit of code is executed multiple times, allowing majority voting on the output to mask a fault. Software-based mitigations are attractive as they can be used to protect widely-available and price-competitive *Commercial Off The Shelf* (COTS) hardware.

Naively applying software-based fault-tolerance leads to a considerable reduction in performance due to the need for replication, negating part of the cost advantage of using COTS hardware. At the same time, many applications do not need complete protection against SEUs. Some tasks can naturally tolerate deadline misses captured by the weakly-hard model [7], as also described in Section 5. In an application structured as a set of periodic weakly-hard real-time tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ (e.g. the ADMORPH task model), not all tasks need to be protected at the same time. Instead, a subset of τ can be protected, and this subset can be varied in consecutive iterations of the task set (effectively *mode switching* between different schedules). This approach forms the basis of *strategy switching*. By protecting varying subsets of the task set with redundancy, the effective fault rate can be lowered substantially without the performance overhead that is typically associated with software-based fault-tolerance. We leverage the TeamPlay coordination language, whose exogenous coordination approach allows the transparent implementation of redundancy by task replication (*task-level redundancy*) on varying subsets of the task graph.

4.1 Task- and fault model

Our method expects a set of tasks Γ with one global period P and deadline D such that $P \leq D$ (no pipelining). Each task $\tau_i \in \Gamma$ is assumed to be annotated with a weakly-hard constraint of $\tau_i \vdash \binom{h}{k}$. Our method currently is limited to $k \leq 2$. Furthermore, each task can be scheduled at any moment in each iteration (epoch) of the task set. For each task $\tau_i \in \Gamma$, a worst-case execution time C_i is known, such that static schedules can be constructed, or such that schedulability tests can be performed. Each upset event is independent and modeled by the Poisson distribution. The probability of a task experiencing a fault depends on the fault rate λ and the WCET C_i of that task. Finally, we do not assume that the fault-tolerance technique is bullet-proof. The strategy switching technique assumes that the use of fault-tolerance measures only lowers the expected fault-rate. As software-based fault-tolerance is typically implemented by redundancy, a fault would manifest itself as a failure to reach a consensus between replicas. Consequently, we assume that the success status of each task under protection is available at the end of each period. This status is used to make an informed decision regarding which tasks to protect during the subsequent iteration of the DAG.

TeamPlay components communicate via channels. Channels make the data being communicated

between tasks explicit as well as the predecessor/successor relation between tasks. As discussed in Section 2.4, the effects of communication between tasks within one period of the task set must be considered when viewing hk-constraints. When a given task fails and cannot provide data to its successor, that successor also fails. As such, while the preceding task may have a relaxed hk-constraint, its failing may jeopardize the ability of the successor task to meet its constraint.

To capture the effects of a fault in a chain of tasks, we assume faults cascade across channels. When predecessor task τ_A fails, any data it produces and sends to successor task τ_B is considered corrupted. As such, regardless of the behavior of τ_B, we assume that it fails as well given a fault in τ_A. Similarly, the probability of a fault in τ_B is a composite of the probability of a fault during the logical execution of τ_A and τ_B.

4.2 Strategies and results

A *strategy* is a set of tasks under fault tolerance. Each strategy s applies fault-tolerance to $\Gamma_s \subseteq \Gamma$, while the other tasks $\Gamma - \Gamma_s$ are executed without fault-tolerance. A strategy is selected at the beginning of each period of the task set. At the end of the period, data is collected regarding the performance of each task, which is captured in a result r. Our fault model assumes that the success status of each task under protection is available, and nothing more. As such, a result r for strategy s must exist for each bitmask over Γ_s.

To guarantee schedulability and ensure only viable strategies are selected, all strategies are analyzed ahead of time. Likewise, for each result the next best successor strategy is determined. This construction gives rise to the *strategy state machine* – a bipartite state machine consisting of strategy and result states. A strategy s is selected at the start of a period, and the task set is executed according to the redundancy specified by that strategy. The success status of each task under protection maps to exactly one result r of s per the strategy state machine. This result state then maps to exactly one successor strategy, which was determined ahead of time. The approach of using a strategy state machine to encode all decisions minimizes the runtime overhead and improves analyzability of this method.

4.2.1 Strategy generation

To construct the strategy state machine, all strategies are enumerated in a set $\mathcal{S}$. Our method aims to minimize the effective fault rate, and to that end the set of strategies can be reduced by removing *redundant* strategies. When there exists a strategy s_x that protects a strict superset of another strategy s_y, and both strategies are schedulability, there is never a situation in which applying s_y is more favorable than applying s_x. The strict superset relation between strategies forms a lattice over $\mathcal{S}$. The strategy switching technique is designed for resource-constrained systems, i.e. systems where the strategy that protects all tasks is not schedulable. Assuming such a system, the greatest element (where $\Gamma_s = \Gamma$) in the lattice is not schedulable. Strategies from the total set of strategies are subjected to a schedulability test. When found schedulable, all inferior strategies per the lattice relation are removed and need not be evaluated by a schedulability test. Likewise, when found to be unschedulable, all superior strategies are removed. The final set $\mathcal{S}$ is guaranteed to contain at

least one strategy given that the empty strategy $s_\emptyset$ with $\Gamma_{s_\emptyset} = \emptyset$ is schedulable, i.e. the task set without any form of fault-tolerance is schedulable.

4.2.2 Strategy linking

To produce the bipartite strategy state machine, each result of the remaining strategies needs to be linked to a single successor strategy. This assignment is performed in a step called *linking*. When considering the transition from a result to a potential successor strategy, information about two consecutive iterations is available. The first iteration is the past iteration described by the result (and its associated strategy), and the second iteration is under the application of the successor strategy. Our method currently considers $\binom{m}{k} = \binom{1}{2}$ constraints, as such these consecutive iterations exactly match our window size of 2. The method aims to minimize the number of faults, as such we link any result r to a strategy s such that the expected number of faults across the two iterations is minimal, as per Definition 1.

Definition 1. Determining a successor strategy $\Delta(r) \in \mathcal{S}$

$$\Delta(r) = \underset{s \in \mathcal{S}}{\operatorname{argmin}} \sum_{\tau \in \Gamma} P(\tau \text{ violates hk-constraint} | r, s)$$

Computing the expected number of faults is straightforward as both the result r and the strategy s provide fault probabilities according to the Poisson distribution, any predecessor relations, or any information in the result. There are effectively three hk-constraints that we consider:

(i.) $\binom{0}{1}$ constraints, where the probability of failure is always 0.

(ii.) $\binom{1}{1}$ constraints, which must simply consider $P(\tau \text{ fails } | s)$. The previous result does not impact it.

(iii.) $\binom{1}{2}$ constraints, which are only violated if τ failed as per r and s: $P(\tau \text{ fails } | r) \cdot P(\tau \text{ fails } | s)$. Note that r does not need to have a definitive answer as to whether τ has failed: if τ was not protected, it can only give the probability of success as we do not consider faults to be detectable under all possible circumstances.

The probability of a fault $P(\tau \text{ fails } | x), x \in \{s, r\}$ must take all predecessors of τ into consideration. The effect of this prioritizes preceding tasks, even if those tasks have weaker mk-constraints.

4.3 Analytical evaluation

The strategy-switching approach lends itself well for analytical evaluation as the strategy state machine can be expressed as a *Discrete-Time Markov Chain* (DTMC). Rates are attached to strategy-result relations as given by the probability that said result is selected given the strategy. This rate is guaranteed to sum to 1 across a strategy, as there is exactly one result for each possible outcome of a strategy. To perform the conversion, all result states are coalesced into their successor strategies, collapsing the bipartite nature of the state machine. The emerging strategy-strategy

edges assume rates provided by the sum of rates set on previous strategy-result edges, finishing the conversion to a DTMC.

The DTMC can be used to obtain the steady-state distribution, providing the fraction of task graph iterations spent in each strategy. Each of these fractions, when multiplied with the technique for calculating the expected number of faults across a transition from Definition 1, provides the rate of expected faults experienced in that strategy in the steady state. By summing all these rates together, we can obtain the total expected number of faults experienced per iteration of the task graph. This evaluation technique was applied on a wide range of task graphs in [32], and numerically confirmed by a simulation using UPPAAL.

For more details on our work we refer the interested reader to our original publications:

- Lukas Miedema, Benjamin Rouxel and Clemens Grelck: *Strategy Switching: Smart Fault-tolerance for Resource-constrained Real-time Applications*. In: Workshop on Connecting Education and Research Communities for an Innovative Resource Aware Society (CERCIRAS 2021), Novi Sad, Serbia, CEUR-WS Proceedings vol. 3145, 2022.

- Lukas Miedema and Clemens Grelck: *Strategy Switching: Smart Fault-tolerance for Weakly-hard Resource-constrained Real-time Applications*. In: Software Engineering and Formal Methods, 20th International Conference, SEFM 2022, Berlin, Germany. Lecture Notes in Computer Science 13550, pp. 129–145, Springer, 2022.

5 Specifying formal guarantees for the adaptation layer

Robustness is an essential concern in the design of control systems. Controlled systems must be able to reliably handle nonlinear effects, unmodeled dynamics, external noise, as well as delays in signal transmissions and dropped packets. A lesser known problem concerns the assessment of robustness to *computational issues* when controllers are implemented as periodic tasks in cheap embedded platforms. Such tasks are expected to execute with real-time guarantees, i.e., their execution must be completed before a well defined *deadline*, usually corresponding to the activation of the next periodic instance of the task when the control output must be published to the actuator. However, it is common in practical cases [2] that tasks do not always complete within their deadline, causing what is called a *deadline miss*. This may be caused by delays in computation and memory accesses, transient overload conditions, bugs and further issues.

A popular model to describe real-time systems allowing deadline misses is the *weakly-hard* model [7]. Weakly-hard tasks feature constraints defining a maximum number of deadlines that can be missed (alternatively, a minimum number to be satisfied) in a given number of consecutive periods. This model will be also the focus of this work. To analyse the effects on the controlled plant, it is necessary to specify also *what happens when the miss is experienced*, both in terms of changes to the control signal and in terms of actions taken to deal with the failed computation [35]. An instance that experiences a deadline miss can be allowed to continue executing until completion (and possibly used later) in some application, whereas in other applications it is stopped and discarded instead.

There is, however, quite a mismatch between the guarantees that can be obtained for real-time tasks and platforms [20, 11] and the analysis that is available for *control* tasks under the weakly-hard model. Fewer works deal with *stability* analysis of weakly-hard real-time control tasks, often targeting specific use-cases or simpler properties like convergence rates [14]. For instance, the analysis in [31] is limited to constraints that specify a maximum number of *consecutive* deadline misses. The results in [29, 30, 21] obtained for networked control systems having packet dropouts (or system faults) bounded using the weakly-hard model, can not be generalized for the case of *late completions*. Other works [28, 27] studied safety guarantees of weakly-hard controllers, but only considering a miss as a discarded computation, and with a known periodic pattern. In [23, 22] an over-approximation-based approach is proposed to check the safety of nonlinear weakly-hard systems, with misses representing discarded computations and no actuator updates.

Our research aims at filling the gap, by providing a stability analysis that can be applied to a class of generic weakly-hard models and deadline miss handling strategies. First, we formally extend the weakly-hard model to explicitly consider the strategy used to handle the miss events. By leveraging an automaton representation of the sequences allowed by (a set of) extended weakly-hard constraints, we use Kronecker lifting and the joint spectral radius to properly express its stability conditions. Using the concept of constraint dominance, we prove analytic bounds on the stability of a weakly-hard system with respect to *less dominant* constraints. Finally, we analyze the stability of the resulting closed-loop systems using `SparseJSR` [47], which exploits the sparsity pattern that naturally arises in the Kronecker-lifted representation. The proposed analysis calls for modularity and separation of concerns and can be a useful tool to decouple the constraint specification and the

control verification.

We consider a controllable and fully observable *discrete-time* sampled linear time invariant (LTI) system, expressed as

$$P: \quad \begin{cases} x_{t+1} &= A_p\, x_t + B_p\, u_t \\ y_t &= C_p\, x_t + D_p\, u_t, \end{cases} \tag{1}$$

where $x_t \in \mathbb{R}^n$, $u_t \in \mathbb{R}^r$ and $y_t \in \mathbb{R}^q$ are the plant state, the control signal and the plant output, sampled at time $t \cdot T$, T is the sampling period, and $t \in \mathbb{N}$. The plant is controlled by a stabilising, LTI, one-step delay discrete-time controller defined as

$$C: \quad \begin{cases} z_{t+1} = A_c\, z_t + B_c\, e_t \\ u_{t+1} = C_c\, z_t + D_c\, e_t, \end{cases} \tag{2}$$

where $z_t \in \mathbb{R}^s$ is the controller's internal state, $r_t \in \mathbb{R}^q$ is the setpoint, and $e_t \in \mathbb{R}^q$, $e_t = r_t - y_t$ is the tracking error. Without loss of generality, we consider $r_t = 0$.

5.1 Control tasks that may miss deadlines

The controller in (2) is implemented as a real-time task τ and is designed to be executed periodically with period T in a real-time embedded platform. Under nominal conditions the task releases an instance (called *job*) in each period that must be completed before the release of the next instance. We denote the sequence of activation instants for τ with $(a_i)_{i\in\mathbb{N}}$, such that, in nominal conditions, $a_{i+1} = a_i + T$, the sequence of completion instants $(f_i)_{i\in\mathbb{N}}$, and the sequence of job deadlines with $(d_i)_{i\in\mathbb{N}}$, such that $d_i = a_i + T$ (also called *implicit* deadline). This requirement can be either satisfied or not, leading to deadline hits and misses, respectively.

Definition 2 (Deadline hit and miss)**.** The i-th job of a periodic task τ with period T hits its deadline if $f_i \leq d_i$ and misses its deadline if $f_i > d_i$.

We refer to both deadline hits and deadline misses using the term *outcome* of a job. Intuitively, each job's outcome is dependent on the characteristics of the remaining tasks executing in the real-time system and the chosen scheduling algorithm. Given a taskset and a (worst-case) schedule, it is possible to bound the worst-case behavior of the job outcomes [7, 20]. This bound is generally denoted the *weakly-hard model* [7]. Following such a model, a task τ may satisfy any combination of these weakly-hard constraints, defined as follows: (i) $\tau \vdash \overline{\binom{m}{k}}$: in any window of k consecutive jobs, at most m deadlines are missed; (ii) $\tau \vdash \binom{h}{k}$: in any window of k consecutive jobs, at least h deadlines are hit; (iii) $\tau \vdash \overline{\left\langle \begin{smallmatrix} m \\ k \end{smallmatrix} \right\rangle}$: in any window of k consecutive jobs, at most m *consecutive* deadlines are missed; and (iv) $\tau \vdash \left\langle \begin{smallmatrix} h \\ k \end{smallmatrix} \right\rangle$: in any window of k consecutive jobs, at least h *consecutive* deadlines are hit, with $m, h \in \mathbb{N}$, $k \in \mathbb{N} \setminus \{0\}$, $m \leq k$, and $h \leq k$. A generic weakly-hard constraint is, hereafter, denoted with the symbol λ while a set of L constraints will be referred to as $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_L\}$.

We define a *string* $\omega = \langle \alpha_1, \alpha_2, \ldots, \alpha_N \rangle$ as a sequence of N consecutive outcomes, where each outcome α_i is a character in the alphabet $\Sigma = \{\mathtt{M}, \mathtt{H}\}$. We use $\omega \vdash \lambda$ to denote that ω satisfies the

constraint λ. Stating that $\tau \vdash \lambda$ means that all the possible sequences of outcomes that τ can experience satisfy the corresponding constraint λ. The set of such sequences naturally results from the definition of λ and is formally defined as the *satisfaction set* as follows [7].

Definition 3 (Satisfaction set $\mathcal{S}_N(\lambda)$)**.** We denote with $\mathcal{S}_N(\lambda)$ the set of strings of length $N \geq 1$ that satisfy a constraint λ. Formally, $\mathcal{S}_N(\lambda) = \{\omega \in \Sigma^N \mid \omega \vdash \lambda\}$.

Taking the limit to infinity, the set $\mathcal{S}(\lambda)$ contains all the strings of infinite length that satisfy λ. The notion of *domination* between constraints [7] then follows.

Definition 4 (Constraint domination)**.** Constraint λ_i *dominates* λ_j (formally, $\lambda_i \preceq \lambda_j$) if $\mathcal{S}(\lambda_i) \subseteq \mathcal{S}(\lambda_j)$.

If a control task τ is implemented on an embedded platform with limited computational power running alongside other applications, it is not uncommon for it to experience deadline misses, even in case of simple control designs (PID, LQG, etc.) [2, 34]. Computational overruns may be caused by, e.g., bursts of interrupts, cache misses, variable execution times of ancillary functions, or other complex interactions. If such events are rare or temporary, choosing a longer period for the controller to avoid them may result in worse performance and stability margins for nominal conditions [35].

Characterising the stability and performance of such controllers requires knowing what happens when a control deadline is missed [35, 31, 45]. In particular, we need a *deadline miss handling strategy* to decide the fate of the job that missed the deadline (and possibly the next ones), and an *actuator mode* to deal with the loss of a new control signal, for example by holding the previous value constant or zeroing it [42]. A few handling strategies for periodic controllers have been proposed in literature, the most interesting being *Kill* and *Skip-Next* [10, 35, 31].

Definition 5 (Kill strategy)**.** Under the Kill strategy a job that misses its deadline is terminated immediately. Formally, for the i-th job of τ either $f_i \leq d_i$ or $f_i = \infty$.

Definition 6 (Skip-Next strategy)**.** Under the Skip-Next strategy a job that misses its deadline is allowed to continue during the following period. Formally, if the i-th job of τ misses its deadline d_i, a new deadline $d_i^+ = d_i + T$ is set for the job, and $a_{i+1} = d_i^+$.

In [31] the authors identify a set of subsequences of hit and missed deadlines, which can be arbitrarily combined to obtain all possible sequences in $\mathcal{S}(\overline{\langle {m \atop k} \rangle})$. The stability analysis of the resulting arbitrary switching system is then obtained by leveraging the *Joint Spectral Radius* (JSR) [40].

Given $\ell \in \mathbb{N} \setminus \{0\}$ and a set of matrices $\mathcal{A} = \{A_1, \ldots, A_\ell\} \subseteq \mathbb{R}^{n \times n}$, under the hypothesis of arbitrary switching over any sequence $s = \langle a_1, a_2, \ldots \rangle$ of indices of matrices in $\mathcal{A}$, the JSR of $\mathcal{A}$ is defined by:

$$\rho(\mathcal{A}) = \lim_{k \to \infty} \max_{s \in \{1,\ldots,\ell\}^k} \mathbb{N}orm{A_{a_k} \cdots A_{a_2} A_{a_1}}^{\frac{1}{k}}. \tag{3}$$

The number $\rho(\mathcal{A})$ characterizes the maximal asymptotic growth rate of matrix products from $\mathcal{A}$ (thus $\rho(\mathcal{A}) < 1$ means that the system is asymptotically stable), and is independent of the norm

$\|\cdot\|$ used in (3). Existing practical tools such as the JSR Matlab toolbox [44] include multiple algorithms to compute both upper and lower bounds on $\rho(\mathcal{A})$.

When the switching sequences between the dynamics of $\mathcal{A}$ are not arbitrary, but constrained by a graph $\mathcal{G}$, the so-called *constrained joint spectral radius* (CJSR) [13] can be applied. Introducing $S_k(\mathcal{G})$ as the set of all possible switching sequences s of length k that satisfy the constraints of a graph $\mathcal{G}$, the CJSR of $\mathcal{A}$ is defined by

$$\rho(\mathcal{A},\mathcal{G}) = \lim_{k\to\infty} \max_{s\in S_k(\mathcal{G})} \mathbb{N}orm A_{a_k}\cdots A_{a_2}A_{a_1}{}^{\frac{1}{k}}. \tag{4}$$

In general, computing or approximating the CJSR is harder than using the JSR. The authors of [36] propose a multinorm-based method to approximate with arbitrary accuracy the CJSR. Other works [25, 49] propose the creation of an arbitrary switching system such that its JSR is equal to the CJSR of the original system, based on a Kronecker lifting method. This will be also our approach, as detailed later.

The authors of [33] propose an efficient approach to compute upper bounds of the JSR based on positive polynomials, which can be decomposed as *sums of squares* (SOS). Finding the coefficients of a polynomial being SOS simplifies solving an SDP [26]. To reduce time and space complexity, a *sparse* variant has been proposed in [47] exploiting the sparsity of the input matrices, based on the *term sparsity* SOS (TSSOS) framework [48]. By contrast, the procedure in [33] will be denoted hereafter as *dense*. While providing a more conservative result, the sparse upper bound can be obtained significantly faster if the matrices from $\mathcal{A}$ are sparse [47], e.g., the matrices we analyse in Section 5.4.

5.2 Extended weakly-hard task model

To provide a comprehensive analysis framework, we need to examine what occurs in each time interval $(\pi_i)_{i\in\mathbb{N}}$, with $\pi_i = [a_0 + i\cdot T, a_0 + (i+1)\cdot T)$. In this context, an extension of the weakly-hard model is required to account for the given deadline miss handling strategy, denoted with the symbol $\mathcal{H}$.

Definition 7 (Extended weakly-hard model $\tau \vdash \lambda^{\mathcal{H}}$)**.** A task τ may satisfy any combination of the four *extended weakly-hard constraints* (EWHC) $\lambda^{\mathcal{H}}$:

(i) $\tau \vdash \overline{\binom{m}{k}}^{\mathcal{H}}$: in any window of k consecutive jobs, at most m intervals lack a job completion;

(ii) $\tau \vdash \binom{h}{k}^{\mathcal{H}}$: in any window of k consecutive jobs, at least h intervals have a job completion;

(iii) $\tau \vdash \overline{\left\langle {m \atop k} \right\rangle}^{\mathcal{H}}$: in any window of k consecutive jobs, at most m *consecutive* intervals lack a job completion;

(iv) $\tau \vdash \left\langle {h \atop k} \right\rangle^{\mathcal{H}}$: in any window of k consecutive jobs, at least h *consecutive* intervals have a job completion

with $m, h \in \mathbb{N}$, $k \in \mathbb{N} \setminus \{0\}$, $m \leq k$, and $h \leq k$, while using strategy $\mathcal{H}$ to handle potential deadline misses.

The definition above differs from the original weakly-hard model of [7], since (i) it explicitly introduces the handling strategy $\mathcal{H}$; and (ii) it focuses on the presence of a new control command at the end of each time interval π_i, instead of checking the deadline miss events, which guarantees its applicability also for strategies different than Kill.

We now require an expressive alphabet $\Sigma(\mathcal{H})$ to characterize the behavior of task τ in each possible time interval. For both Kill and Skip-Next strategies, each interval π_i contains at most one activated and one completed job. This restricts the possible behaviors to three cases:

(i) a time interval in which the same job is both released and completed is denoted by `H` (*hit*);

(ii) a time interval in which no job is completed is denoted by `M` (*miss*);

(iii) a time interval in which no job is released, but a job (released in a previous interval) is completed, is denoted by `R` (*recovery*).

By checking all unique combinations of job activations and completions in each interval, we obtain the alphabets for Kill and Skip-Next as $\Sigma(\text{Kill}) = \{\mathtt{M}, \mathtt{H}\}$ and $\Sigma(\text{Skip-Next}) = \{\mathtt{M}, \mathtt{H}, \mathtt{R}\}$, respectively. The recovery character `R` is used in the Skip-Next alphabet to identify the late *completion* of a job. As a consequence, `R` is treated equivalently to `H` when verifying the extended weakly hard constraints (EWHC).

The algebra presented in Section 5.1 is extended to the new alphabet. We assign a character of the alphabet $\Sigma(\mathcal{H})$ to each interval π_i. A string $\omega = \langle \alpha_1, \alpha_2, \ldots, \alpha_N \rangle$ is used to represent a sequence of N outcomes for task τ, with $\alpha_i \in \Sigma(\mathcal{H})$ representing the outcome associated to the interval π_i. To enforce only feasible sequences, we introduce an order constraint for the `R` character with the rule that: For any string $\omega \in \Sigma(\text{Skip-Next})^N$, `R` may only directly follow `M`, or be the initial element of the string.

The extended weakly-hard model also inherits all the properties of the original weakly-hard model. In particular, the satisfaction set of $\lambda^{\mathcal{H}}$ can be defined for $N \geq 1$ as $\mathcal{S}_N(\lambda^{\mathcal{H}}) = \{\omega \in \Sigma(\mathcal{H})^N \mid \omega \vdash \lambda^{\mathcal{H}}\}$, and the constraint domination still holds as $\lambda_i^{\mathcal{H}} \preceq \lambda_j^{\mathcal{H}}$ if $\mathcal{S}(\lambda_i^{\mathcal{H}}) \subseteq \mathcal{S}(\lambda_j^{\mathcal{H}})$.

5.3 Automaton representation

Any EWHC, as presented in Definition 7, can be systematically represented using an *automaton*. In this paper we build upon the `WeaklyHard.jl` automaton model presented in [46]. Here, a (minimal) automaton $\mathcal{G}_{\lambda^{\mathcal{H}}} = (V_{\lambda^{\mathcal{H}}}, E_{\lambda^{\mathcal{H}}})$ associated to $\lambda^{\mathcal{H}}$ consists of a set of vertices ($V_{\lambda^{\mathcal{H}}}$) and a set of directed labeled edges ($E_{\lambda^{\mathcal{H}}}$). Each vertex $v_i \in V_{\lambda^{\mathcal{H}}}$ corresponds to a string of outcomes of the extended weakly-hard task executions. Trivially, there exists no vertices for strings that do not satisfy the EWHC. A directed labeled edge $e_{i,j} = (v_i, v_j, \alpha) \in E_{\lambda^{\mathcal{H}}}$ (also denoted *transition*) connects two vertices iff the outcome $\alpha \in \Sigma(\mathcal{H})$ – the edge's label – appended to the tail vertex's string representation (v_i) would result in the string equivalent to the one of the head vertex (v_j).

Thus, a random walk in the automaton corresponds to a random string satisfying the EWHC. In particular, all the walks in the automaton corresponds to *all* strings in $\mathcal{S}(\lambda^{\mathcal{H}})$.

Since the `WeaklyHard.jl` automaton model only uses the binary alphabet $\Sigma = \{\mathtt{M}, \mathtt{H}\}$, we require the additional character `R` to handle the Skip-Next strategy properly. Recall that both a hit (`H`) and a recovery (`R`) are considered job completions. Thus, for the Skip-Next strategy, we post-process the automaton by enforcing that "a recovery hit can only occur after a miss" is honored and that the corresponding transitions are correct, i.e., switching the labels on some edges from `H` to `R`. We emphasize that despite the extended automaton model appearing similar for the Kill and Skip-Next strategies, the differing transitions of the two automata significantly affect the corresponding closed-loop systems, as will be clarified in Section 5.4.

The `WeaklyHard.jl` automaton model also allows for the case where the task τ is subject to a set of multiple constraints. Since the stability analysis presented in this paper is invariant to the type (and amount) of the constraints acting on the control task τ, we henceforth say that τ is subject to a set of EWHC $\Lambda^{\mathcal{H}}$ (unless stated otherwise).

Extracting all transitions in $E_{\Lambda^{\mathcal{H}}}$ corresponding to a character $\alpha \in \Sigma(\mathcal{H})$ yields what is generally known as a *directed adjacency matrix* [50], denoted here as a *transition matrix*.

Definition 8 (Transition matrix)**.** Given an automaton $\mathcal{G}_{\Lambda^{\mathcal{H}}}$, the *transition matrix* $F_\alpha(\mathcal{G}_{\Lambda^{\mathcal{H}}}) \in \mathbb{R}^{n_V \times n_V}$, with $n_V = |V_{\Lambda^{\mathcal{H}}}|$ and $\alpha \in \Sigma(\mathcal{H})$, is computed as $F_\alpha(\mathcal{G}_{\Lambda^{\mathcal{H}}}) = \{f_{i,j}(\alpha)\}$ with

$$f_{i,j}(\alpha) = \begin{cases} 1, & \text{if } \exists\, e = (v_j, v_i, \alpha) \in E_{\Lambda^{\mathcal{H}}} \\ 0, & \text{otherwise.} \end{cases}$$

Since *at most one* successor exists from each vertex with a transition labeled with $\alpha \in \Sigma(\mathcal{H})$, matrix F_α will have a column sum of either 1 or 0. We now introduce a vector $q_t \in \mathbb{R}^{n_V}$ called *G-state*, with $n_V = |V_{\Lambda^{\mathcal{H}}}|$, representing the state of the given automaton $\mathcal{G}_{\Lambda^{\mathcal{H}}}$ at interval π_t.

Definition 9 (G-state q_t)**.** Given an automaton $\mathcal{G}_{\Lambda^{\mathcal{H}}}$ and a string $\omega \in \Sigma(\mathcal{H})^N$, $\omega = \langle \alpha_1, \alpha_2, \ldots, \alpha_N \rangle$, for $k = |v|$, $v \in V_{\Lambda^{\mathcal{H}}}$, we define $q_t \in \mathbb{R}^{n_V}$, where the i-th element $q_{t,i}$ is:

$$q_{t,i} = \begin{cases} 1, & \text{if } \langle \alpha_{t-k}, \ldots, \alpha_{t-1} \rangle \equiv v_i \in V_{\Lambda^{\mathcal{H}}} \\ 0, & \text{otherwise.} \end{cases}$$

The G-state q_t is the vector representation of the vertex *left* at step t: here, $q_t = 0$ means that the transition at step $t-1$ was infeasible for the automaton. Given an arbitrary string $\omega = \langle \alpha_1, \ldots, \alpha_t, \ldots \rangle$, the G-state dynamics is defined as $q_{t+1} = F_\alpha(\mathcal{G}_{\Lambda^{\mathcal{H}}}) \cdot q_t$, and the following property holds [50].

Lemma 1 (Infeasible sequence)**.** *If* $\omega \notin \mathcal{S}_N(\Lambda^{\mathcal{H}})$*, then* $F_\omega(\mathcal{G}_{\Lambda^{\mathcal{H}}}) = F_{\alpha_N}(\mathcal{G}_{\Lambda^{\mathcal{H}}}) \cdots F_{\alpha_2}(\mathcal{G}_{\Lambda^{\mathcal{H}}}) \cdot F_{\alpha_1}(\mathcal{G}_{\Lambda^{\mathcal{H}}}) = 0$

Thus, if $q_t = 0$ for an arbitrary t, then $q_{t'} = 0$ for $t' \geq t$.

5.4 Closed-loop system stability

Using the alphabet $\Sigma(\mathcal{H})$ and the chosen actuator mode (i.e., zeroing or holding the previous value), we compute the closed-loop behavior of the controlled system. We identify one matrix for each dynamics corresponding to an interval π_t associated by $\alpha \in \Sigma(\mathcal{H})$, building the set $\mathcal{A}^{\mathcal{H}}$.

Kill: Defining $\tilde{x}_t^{\mathrm{K}} = \begin{bmatrix} x_t^T & z_t^T & u_t^T \end{bmatrix}^T$ as the closed-loop state vector, we compute the discrete time closed-loop system dynamics $A_{\mathtt{H}}^{\mathrm{K}}$, corresponding to the character `H`:

$$\tilde{x}_{t+1}^{\mathrm{K}} = A_{\mathtt{H}}^{\mathrm{K}}\, \tilde{x}_t^{\mathrm{K}}, \quad A_{\mathtt{H}}^{\mathrm{K}} = \begin{bmatrix} A_p & 0 & B_p \\ -B_c C_p & A_c & -B_c D_p \\ -D_c C_p & C_c & -D_c D_p \end{bmatrix}.$$

For the case of `M`, the controller execution terminates prematurely and its states are not updated ($z_{t+1} = z_t$). Therefore, depending on the actuation mode, the controller output is either zeroed ($u_{t+1} = 0$) or held ($u_{t+1} = u_t$). The resulting closed-loop system in state-space form is denoted with $A_{\mathtt{M}}^{\mathrm{K}}$:

$$\tilde{x}_{t+1}^{\mathrm{K}} = A_{\mathtt{M}}^{\mathrm{K}}\, \tilde{x}_t^{\mathrm{K}}, \quad A_{\mathtt{M}}^{\mathrm{K}} = \begin{bmatrix} A_p & 0 & B_p \\ 0 & I & 0 \\ 0 & 0 & \Delta \end{bmatrix}.$$

Here, $\Delta = I$ (identity matrix) if the control signal is held and $\Delta = 0$ if zeroed. The set of dynamic matrices under the Kill strategy is then $\mathcal{A}^{\mathrm{Kill}} = \{A_{\mathtt{H}}^{\mathrm{K}}, A_{\mathtt{M}}^{\mathrm{K}}\}$.

Skip-Next: For the Skip-Next strategy, we introduce two additional states $\hat{x}_t$ and $\hat{u}_t$ storing the old values of x_t and u_t while the controller awaits an update. The resulting state vector then becomes $\tilde{x}_t^{\mathrm{S}} = \begin{bmatrix} x_t^T & z_t^T & u_t^T & \hat{x}_t^T & \hat{u}_t^T \end{bmatrix}^T$. When π_t is associated to `H`, the two additional states mirror the behavior of the states of which they are storing data. The resulting closed-loop system is described using $A_{\mathtt{H}}^{\mathrm{S}}$:

$$\tilde{x}_{t+1}^{\mathrm{S}} = A_{\mathtt{H}}^{\mathrm{S}}\, \tilde{x}_t^{\mathrm{S}}, \quad A_{\mathtt{H}}^{\mathrm{S}} = \begin{bmatrix} A_p & 0 & B_p & 0 & 0 \\ -B_c C_p & A_c & -B_c D_p & 0 & 0 \\ -D_c C_p & C_c & -D_c D_p & 0 & 0 \\ A_p & 0 & B_p & 0 & 0 \\ -D_c C_p & C_c & -D_c D_p & 0 & 0 \end{bmatrix}.$$

For the case of `M` in π_t, $\hat{x}_t$ and $\hat{u}_t$ maintain their previous values. The resulting closed-loop is described by $A_{\mathtt{M}}^{\mathrm{S}}$:

$$\tilde{x}_{t+1}^{\mathrm{S}} = A_{\mathtt{M}}^{\mathrm{S}}\, \tilde{x}_t^{\mathrm{S}}, \quad A_{\mathtt{M}}^{\mathrm{S}} = \begin{bmatrix} A_p & 0 & B_p & 0 & 0 \\ 0 & I & 0 & 0 & 0 \\ 0 & 0 & \Delta & 0 & 0 \\ 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 & I \end{bmatrix}.$$

Finally, for the case of `R`, the new control command is calculated using the values stored in $\hat{x}_t$ and

$\hat{u}_t$. The resulting closed-loop system is described by $A_{\mathtt{R}}^{\mathrm{S}}$:

$$\tilde{x}_{t+1}^{\mathrm{S}} = A_{\mathtt{R}}^{\mathrm{S}}\,\tilde{x}_t^{\mathrm{S}}, \quad A_{\mathtt{R}}^{\mathrm{S}} = \begin{bmatrix} A_p & 0 & B_p & 0 & 0 \\ 0 & A_c & 0 & -B_cC_p & -B_cD_p \\ 0 & C_c & 0 & -D_cC_p & -D_cD_p \\ A_p & 0 & B_p & 0 & 0 \\ 0 & C_c & 0 & -D_cC_p & -D_cD_p \end{bmatrix}.$$

The resulting set of matrices under Skip-Next strategy is then defined as $\mathcal{A}^{\text{Skip-Next}} = \left\{A_{\mathtt{H}}^{\mathrm{S}}, A_{\mathtt{M}}^{\mathrm{S}}, A_{\mathtt{R}}^{\mathrm{S}}\right\}$.

Combining the set of system dynamics $\mathcal{A}^{\mathcal{H}}$ with the associated automaton $\mathcal{G}_{\Lambda^{\mathcal{H}}}$, we seek to obtain an equivalent system model based on Kronecker lifting, characterized by a set of matrices denoted by $\mathcal{L}_{\Lambda^{\mathcal{H}}}$ and behaving as an *arbitrary switching system*, such that $\rho\left(\mathcal{L}_{\Lambda^{\mathcal{H}}}\right) = \rho\left(\mathcal{A}^{\mathcal{H}}, \mathcal{G}_{\Lambda^{\mathcal{H}}}\right)$. In this way, powerful algorithms applicable to arbitrary switching systems [44, 47] can be used to find tight stability bounds. We build upon the Kronecker lifting approach of [49]. Leveraging the vector q_t, we introduce the *lifted discrete-time state* $\xi_t \in \mathbb{R}^{n \cdot n_V}$, defined as $\xi_t = q_t \otimes x_t$, where $n_V = |V_{\Lambda^{\mathcal{H}}}|$ and $\otimes$ is the Kronecker product. By construction, ξ_t is a vector composed of n_V blocks of size n, where at most one block is equal to x_t and all other blocks are equal to the 0 vector. Then, we build a set of lifted matrices $P_\alpha(\mathcal{G}_{\Lambda^{\mathcal{H}}}) \in \mathbb{R}^{n \cdot n_V \times n \cdot n_V}$, which incorporates both the system dynamics and the possible transitions given a certain outcome $\alpha \in \Sigma\left(\mathcal{H}\right)$:

$$P_\alpha(\mathcal{G}_{\Lambda^{\mathcal{H}}}) = F_\alpha(\mathcal{G}_{\Lambda^{\mathcal{H}}}) \otimes A_\alpha^{\mathcal{H}}, \quad \alpha \in \Sigma\left(\mathcal{H}\right). \tag{5}$$

The lifted dynamics of the closed loop system then become $\xi_{t+1} = P_\alpha(\mathcal{G}_{\Lambda^{\mathcal{H}}}) \cdot \xi_t$. Formally, we obtain a system composed of a set of switching dynamic matrices, $\mathcal{L}_{\Lambda^{\mathcal{H}}}$.

Definition 10 (Lifted switching set $\mathcal{L}_{\Lambda^{\mathcal{H}}}$)**.** Given a set of dynamic matrices $\mathcal{A}^{\mathcal{H}}$ and an automaton $\mathcal{G}_{\Lambda^{\mathcal{H}}}$, the switching set $\mathcal{L}_{\Lambda^{\mathcal{H}}}$ is defined as: $\mathcal{L}_{\Lambda^{\mathcal{H}}} = \left\{P_\alpha(\mathcal{G}_{\Lambda^{\mathcal{H}}}) \mid \alpha \in \Sigma\left(\mathcal{H}\right)\right\}$.

Leveraging the mixed-product property of $\otimes$ and introducing a proper submultiplicative norm, it is possible to prove that $\rho\left(\mathcal{L}_{\Lambda^{\mathcal{H}}}\right) = \rho\left(\mathcal{A}^{\mathcal{H}}, \mathcal{G}_{\Lambda^{\mathcal{H}}}\right)$. For more details and a formal proof we refer the interested reader to [49].

We now provide a general relation between *all* EWHCs in terms of the joint spectral radii.

Theorem 1 (JSR dominance)**.** *Given $\lambda_1^{\mathcal{H}}$ and $\lambda_2^{\mathcal{H}}$ as arbitrary EWHCs, if $\lambda_2^{\mathcal{H}} \preceq \lambda_1^{\mathcal{H}}$ then $\rho\big(\mathcal{L}_{\lambda_2^{\mathcal{H}}}\big) \leq \rho\big(\mathcal{L}_{\lambda_1^{\mathcal{H}}}\big)$.*

Proof. From Equation (3), for a generic EWHC $\lambda^{\mathcal{H}}$,

$$\rho\left(\mathcal{L}_{\lambda^{\mathcal{H}}}\right) = \lim_{\ell \to \infty} \rho_\ell\left(\mathcal{L}_{\lambda^{\mathcal{H}}}\right), \;\; \rho_\ell\left(\mathcal{L}_{\lambda^{\mathcal{H}}}\right) = \max_{a \in \mathcal{S}_\ell(\lambda^{\mathcal{H}})} \|A_a\|^{1/\ell}.$$

Definition 4 gave us that $\lambda_2^{\mathcal{H}} \preceq \lambda_1^{\mathcal{H}}$ iff $\mathcal{S}(\lambda_2^{\mathcal{H}}) \subseteq \mathcal{S}(\lambda_1^{\mathcal{H}})$. Thus, if for a string b it holds that $b \in \mathcal{S}_\ell\left(\lambda_2^{\mathcal{H}}\right)$, then it also holds that $b \in \mathcal{S}_\ell\left(\lambda_1^{\mathcal{H}}\right)$. The set of all possible A_b is thus included in the set of all possible A_a, $a \in \mathcal{S}_\ell\left(\lambda_1^{\mathcal{H}}\right)$, thus:

$$\max_{b \in \mathcal{S}_\ell(\lambda_2^{\mathcal{H}})} \|A_b\|^{1/\ell} \leq \max_{a \in \mathcal{S}_\ell(\lambda_1^{\mathcal{H}})} \|A_a\|^{1/\ell}, \quad \forall \ell \in \mathbb{N} \setminus 0.$$

The theorem follows immediately when $\ell \to \infty$. □

Theorem 1 is the first result that provides an analytic correlation between the control theoretical analysis and the real-time implementation. Primarily, it implies that the constraint dominance from Definition 4 also carries on to the JSR, giving us a notion of *JSR dominance*. The results of Theorem 1 are strategy-independent, further reducing the coupling between the control analysis and real-time implementation, and are also independent of the controlled system's dynamics.

Two Corollaries of Theorem 1 are derived for the commonly used models $\overline{\binom{m}{k}}^{\mathcal{H}}$ and $\overline{\binom{m}{k}}^{\mathcal{H}}$, highlighting some practical relations between such constraints.

Corollary 1 ($\overline{\binom{m}{k}}^{\mathcal{H}}$ dominance)**.** *Given* $\lambda_1^{\mathcal{H}} = \overline{\binom{m}{k_1}}^{\mathcal{H}}$ *and* $\lambda_2^{\mathcal{H}} = \overline{\binom{m}{k_2}}^{\mathcal{H}}$*, if* $k_1 \leq k_2$ *then* $\rho\big(\mathcal{L}_{\lambda_2^{\mathcal{H}}}\big) \leq \rho\big(\mathcal{L}_{\lambda_1^{\mathcal{H}}}\big)$.

Corollary 2 ($\overline{\binom{m}{k}}^{\mathcal{H}}$ dominance)**.** *Given* $\lambda_1^{\mathcal{H}} = \overline{\binom{m}{k}}^{\mathcal{H}}$ *and* $\lambda_2^{\mathcal{H}} = \overline{\binom{m}{k}}^{\mathcal{H}}$*, then* $\rho\big(\mathcal{L}_{\lambda_2^{\mathcal{H}}}\big) \leq \rho\big(\mathcal{L}_{\lambda_1^{\mathcal{H}}}\big)$.

The conclusions drawn from Theorem 1 are theoretical, but its practical applicability lies in the algorithm used to find lower and upper bounds for the JSR value ρ^{LB} and ρ^{UB}. Using these bounds we can determine the stability of the corresponding switching systems, as follows:

$$\rho^{LB}\big(\mathcal{L}_{\lambda_2^{\mathcal{H}}}\big) \leq \rho\big(\mathcal{L}_{\lambda_2^{\mathcal{H}}}\big) \leq \rho\big(\mathcal{L}_{\lambda_1^{\mathcal{H}}}\big) \leq \rho^{UB}\big(\mathcal{L}_{\lambda_1^{\mathcal{H}}}\big).$$

Regardless of the algorithm used to find the bounds, if $\lambda_2^{\mathcal{H}} \preceq \lambda_1^{\mathcal{H}}$ and $\rho^{UB}(\mathcal{L}_{\lambda_1^{\mathcal{H}}}) < 1$, the system under $\lambda_2^{\mathcal{H}}$ is switching stable. A similar relation holds for the lower bound.

Theorem 1 can be further extended by relating the joint spectral radius of a single constraint to sets of constraints.

Theorem 2. *Given an arbitrary EWHC* $\lambda^{\mathcal{H}}$*, it holds that* $\rho\left(\mathcal{L}_{\Lambda^{\mathcal{H}}}\right) \leq \rho\left(\mathcal{L}_{\lambda^{\mathcal{H}}}\right),\ \forall \Lambda^{\mathcal{H}} \ni \lambda^{\mathcal{H}}$.

Proof. For an arbitrary EWHC set $\Lambda^{\mathcal{H}} = \{\lambda_1^{\mathcal{H}}, \ldots, \lambda_N^{\mathcal{H}}\}$, its satisfaction set is $\mathcal{S}_\ell\left(\Lambda^{\mathcal{H}}\right) = \bigcap_{i\in\{1,\ldots,N\}} \mathcal{S}_\ell\left(\lambda_i^{\mathcal{H}}\right)$. Thus, for any $\lambda_i^{\mathcal{H}} \in \Lambda^{\mathcal{H}}$ it holds that $\mathcal{S}_\ell\left(\Lambda^{\mathcal{H}}\right) \subseteq \mathcal{S}_\ell\left(\lambda^{\mathcal{H}}\right)$. If a string b is in $\mathcal{S}_\ell\left(\Lambda^{\mathcal{H}}\right)$ it also belongs to $\mathcal{S}_\ell\left(\lambda^{\mathcal{H}}\right)$. The set of all possible A_b is thus included in the set of all possible A_a, $a \in \mathcal{S}_\ell\left(\lambda^{\mathcal{H}}\right)$. As a consequence it holds that

$$\max_{b\in\mathcal{S}_\ell(\Lambda^{\mathcal{H}})} \|A_b\|^{1/\ell} \leq \max_{a\in\mathcal{S}_\ell(\lambda^{\mathcal{H}})} \|A_a\|^{1/\ell}, \quad \forall \ell \in \mathbb{N}^{>}.$$

The theorem follows immediately when $\ell \to \infty$. □

As in Theorem 1, the more we restrict the execution pattern of the control task with sets of constraints, the lower its JSR will be. Theorem 2 delivers the practical insight that enforcing tighter EWHC to a stable system will *never* destabilize it, as formally stated in the following corollary.

Corollary 3. *Given an arbitrary EWHC* $\lambda^{\mathcal{H}}$*, if* $\rho\left(\mathcal{L}_{\lambda^{\mathcal{H}}}\right) < 1$ *then* $\rho\left(\mathcal{L}_{\Lambda^{\mathcal{H}}}\right) < 1,\ \forall \Lambda^{\mathcal{H}} \ni \lambda^{\mathcal{H}}$.

We conducted experiments with numerical examples to validate our claims, but we omit them here for brevity. In our research, we propose a switching stability analysis framework for LTI systems with weakly-hard constraints, extending the weakly-hard model and providing an analytic stability bound. The analysis allows us to assess whether computational errors (present in industrial controllers) affect the stability of the controlled systems.

For more details on our work we refer the interested reader to our original publications:

- Jie Wang, Martina Maggio and Victor Magron: *SparseJSR: A Fast Algorithm to Compute Joint Spectral Radius via SparseSOS Decompositions*. In: American Control Conference (ACC 2021), pp. 2254–2259, IEEE 2021.
- Nils Vreman, Paolo Pazzaglia, Jie Wang, Victor Magron and Martina Maggio: *Stability of Control Systems under Extended Weakly-Hard Constraints*. In: IEEE Control Systems Letters, vol. 6, pp. 2900–2905, 2022.
- Nils Vreman, Paolo Pazzaglia, Jie Wang, Victor Magron and Martina Maggio: *Stability of Control Systems under Extended Weakly-Hard Constraints*. In: 61st IEEE Conference on Decision and Control, CDC 2022, Cancún, Mexico.

Future work will focus on the performance loss due to the presence of deadline misses. Furthermore, the cause of deadline misses could be the lack of computing resources (due, for example, to a security attack). We plan to investigate if it is possible to protect the execution of the controller against this threat.

6 Reflections on the state-of-the-art

In this section we revisit the state-of-the art as originally described in the ADMORPH proposal. In particular, we consider the current state-of-the-art as well as the immediate effects of coordination language research conducted within ADMORPH on coordination research in general.

6.1 State-of-the-art at beginning of ADMORPH

In the project proposal, we introduced the idea of coordination languages and discussed the then-current state-of-the-art as follows:

> "The idea of coordination models, languages and architectures goes back to the seminal work of Gelernter and Carriero [15] and their coordination language Linda[9]. Since those early days a plethora of coordination-based approaches have been proposed [12, 39]. Many refine the original concept of Linda's tuple spaces in a multitude of directions. They all target distributed systems in general and, apart from the early years, they are rooted in the Java ecosystem. Others, notably Reo [3, 24] and BIP (Behavior, Interaction, Priority) [5, 8, 4], focus on advanced interaction structures and devote a significant share of research to semantic models and verification of desirable properties of programming logic. BIP explicitly focuses on embedded systems and comes with a complete tool chain that includes generation of executable C/C++ code from abstract models. Our own prior work on S-Net [18, 19, 16, 51] embraces the concept of coordination as an abstract concurrency model as well as a concrete programming environment for compute-intensive applications. As such S-Net orchestrates the execution of components implemented in either C/C++ or the functional array language Single Assignment C (SAC) [17]. More recently we have been working on carrying over the concept of component coordination to energy-, time- and security-aware computing on heterogeneous multi-core platforms in the context of the Horizon-2020 project TEAMPLAY.
>
> To the best of our knowledge applying the concept of component coordination to manage system robustness remains entirely uncharted territory. The only previous work in this direction that we managed to find is on FT-Linda [43] and dates back to 1995. Apart from this single paper no further traces of publications or implementations could be found. In this sense, ADMORPH is not only promising to advance the state-of-the-art, but to open up an avenue towards an entirely new field of research. Furthermore, we expect a fruitful cross-pollination between TEAMPLAY and ADMORPH, where ADMORPH will immediately benefit from the research results obtained within the TEAMPLAY project."

6.2 Revisiting the state-of-the-art

During our work on ADMORPH we continuously monitor the state-of-the-art on coordination languages at the intersection between real-time cyber-physical (or embedded) systems (of systems)

and fault-tolerance, or reliability in general. Still, we have not come across notable new research activities during the life time of ADMORPH. However, we did find two relevant lines of research conducted before the start of the ADMORPH project, that we had not been aware of before.

In particular, Pradhan et al. [37] proposed *CHARIOT*, a *Domain Specific Language* for CPSoSes. While they do not describe their language as a *coordination language*, it provides capabilities such that it could easily be considered a coordination language. The authors also do not explicitly describe utility for systems-of-systems and, instead, focus on *fractionated* systems. A fractionated system is an architecture within which one system is composed of a set of networked systems at the same location (e.g. a satellite swarm). The advantage of such a system is the ease of extendibility: a system providing a new capability (e.g. sensor or processing platform) can be added to the general area of the other systems, thus enhancing the capabilities of the composite system.

The CHARIOT language aims at aiding the development of these complex, extensible cyber-physical systems-of-systems, primarily the integration between middleware produced by different vendors and managing complexity. A user specifies components, their communication channels (similar to inports and outports in TeamPlay), as well as their mapping to hardware in the coordination language. The presence of this mapping makes the final conversion step, where a CHARIOT specification is converted into instructions for a given middleware, rather straightforward. Their main contribution is in offering a consistent programming interface to address fractionated systems.

The composed system can switch between predefined modes referred to as *objectives*. Objectives are somewhat similar to TeamPlay modes, given that they are discrete system states between which the system may switch. However, TeamPlay modes attempt to serve as a front-end to a *coordination compiler* and further downstream tools, where the conversion between modes is non-trivial, yet handled automatically by design-time algorithms. At the same time, the CHARIOT objectives rather form a set of configuration points that are each written to a database. These points are available to the runtime, which handles them when they are selected. The contribution of Pradhan et al. is primarily in the management of software development complexity, where the conversion of their DSL to a mapping is rather trivial (from an academic point of view). In contrast, the TeamPlay coordination language aims not only to reduce software development complexity, but also to optimize for objectives (such as energy, time or robustness) that are not feasible by traditional means.

More work published under the "DSL" label shows considerable overlap with coordination languages. The contribution by Berger [6] proposes a DSL for the integration of sensors into a cyber-physical system. The platform is supplied in a *board specification file*. However, while the board can be changed, the work is limited to a single board. The DSL encodes the requirements for sensors attached to the board. Each sensor is attached by pins, which must be set to a particular mode (e.g. analog or use i2c). Not all pins support all modes, which forms the basis of a *constraint-satisfaction problem* to perform the assignment of sensors to pins. As such, this work is an example of using a language to formulate a problem that cannot be solved manually, similar to the TeamPlay language. However, the scope and focus on real-time of our coordination language remains unique.

6.3 Impact of our research

At the present time, the combined efforts of ADMORPH fronted with the proposed coordination language has not been published. However, parts of the work on the coordination language have been disseminated at a variety of places (see Section 8).

7 Conclusion

This deliverable reports on the work accomplished by the various ADMORPH consortium partners in the context of Work Package WP1: *Specification of Adaptive Systems* between month 23 and month 36. Besides the considerable achievements by the consortium partners individually, collaboration between partners is on a good way.

We continue to evolve the design of the TeamPlay coordination language in various directions, as elaborated on in Section 2. Our particular focus in this third deliverable of Work Package WP1 is on the validation of TeamPlay in close collaboration with the ADMORPH industrial partners and use case providers (Section 3). Throughout the reporting period a particular focus of our work has been on the research line of strategy switching to create support for weakly-hard real-time systems (Section 4). Our work on the specification of formal guarantees for the adaptation layer in the context of Task T1.3 (Section 5), actually pioneering the field of weakly-hard real-time systems, has come to an end early in the reporting period with some nice results and publications.

The research performed in the context of Work Package WP1 is competitive in the scientific community as exemplified by a decent number of international publications in renowned conferences and journals, as listed in Section 8.

There are strong ties between our work in ADMORPH Work Package WP1 and two other collaborative projects on the European scale: the COST Action CERCIRAS (*Connecting Education and Research Communities for an Innovative Resource Aware Society*) and the Erasmus+ Strategic Partnership for Higher Education SusTrainable *Promoting Sustainability as a Fundamental Driver in Software Development Training and Education*.

We presented our work during a (hybrid) COST Action plenary meeting in Novi Sad, Serbia, in autumn 2021 and published an article on strategy switching in the post-workshop proceedings. The coordination language TeamPlay and its associated implementation technologies will be one topic during the SusTrainable teacher training in Pula, Croatia, in winter 2023 as well as during the upcoming SusTrainable summer school in Coimbra, Portugal, in summer 2023. Both via CERCIRAS as well as via SusTrainable the research conducted in the ADMORPH project is exploited, further disseminated and generally made available to a larger scientific audience beyond the usual conference and journal publications.

Overall, we believe Work Package WP1 to be well on track for the remaining months of the ADMORPH project.

8 Publications and further dissemination activities

The work reported in this deliverable has led to the following publications:

1. Lukas Miedema, Benjamin Rouxel and Clemens Grelck: *Strategy Switching: Smart Fault-tolerance for Resource-constrained Real-time Applications*. In: Workshop on Connecting Education and Research Communities for an Innovative Resource Aware Society (CERCIRAS 2021), Novi Sad, Serbia, CEUR-WS Proceedings vol. 3145, 2022.

2. Jie Wang, Martina Maggio and Victor Magron: *SparseJSR: A Fast Algorithm to Compute Joint Spectral Radius via SparseSOS Decompositions*. In: American Control Conference (ACC 2021), pp. 2254–2259, IEEE 2021.

3. Nils Vreman, Paolo Pazzaglia, Jie Wang, Victor Magron and Martina Maggio: *Stability of Control Systems under Extended Weakly-Hard Constraints*. In: IEEE Control Systems Letters, vol. 6, pp. 2900–2905, 2022.

4. Nils Vreman, Paolo Pazzaglia, Jie Wang, Victor Magron and Martina Maggio: *Stability of Control Systems under Extended Weakly-Hard Constraints*. In: 61st IEEE Conference on Decision and Control, CDC 2022, Cancún, Mexico.

5. Lukas Miedema and Clemens Grelck: *Strategy Switching: Smart Fault-tolerance for Weakly-hard Resource-constrained Real-time Applications*. In: Software Engineering and Formal Methods, 20th International Conference, SEFM 2022, Berlin, Germany. Lecture Notes in Computer Science 13550, pp. 129–145, Springer, 2022.

In addition to the conference presentations 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. Clemens Grelck: *The TeamPlay Coordination Language for Mission-critical Cyber-physical Systems*. 42nd IEEE Real-Time Systems Symposium (RTSS 2021), Virtual from Dortmund, Germany, December 2021.

2. Clemens Grelck: *Exploring the energy/time/security/fault-tolerance trade-off with the TeamPlay coordination language*. SusTrainable Workshop, 8th International Conference on ICT for Sustainability (ICT4S 2022), Plovdiv, Bulgaria, June 2022.

3. Martina Maggio: *Control Systems in the Presence of Computational Problems*. Saarland Informatics Campus Lecture Series, Saarbrücken, Germany, October 2022.

4. Martina Maggio: *Control Systems in the Presence of Computational Problems*. Center for Trustworthy Edge Computing Systems and Applications (TECoSA), KTH Royal Institute of Technology, Stockholm, Sweden, October 2022.

5. Martina Maggio: *Control Systems Fault Tolerance in the Presence of Computational Problems*. Saarland University, lecture in first-year computer science lecture series *Perspectives of Informatics*, devoted to expose starting students to research topics in computer science, Saarbrücken, Germany, November 2022.

During the remainder of the ADMORPH project the following dissemination activities are already planned:

1. Clemens Grelck: *With the TeamPlay Coordination Language towards Adaptively Morphing Embedded Systems*. Workshop on Adaptive Cyber-physical Systems of Systems (WASOS), HiPEAC Conference on High Performance Embedded Architecture and Compilation (HiPEAC 2023), Toulouse, France, January 2023.

2. Clemens Grelck: *tba*. SusTrainable Teacher Training 2023, Erasmus+ Project *Promoting Sustainability as a Fundamental Driver in Software Development Training and Education*, Pula, Croatia, January 2023.

3. Lukas Miedema, Clemens Grelck: *tba*. SusTrainable Summer School 2023, Erasmus+ Project *Promoting Sustainability as a Fundamental Driver in Software Development Training and Education*, Coimbra, Portugal, July 2023.

Acknowledgments*

We would like to thank our internal reviewers Jeroen Kouwer (Thales Nederland) and Florian Haas (Augsburg University) for their more than valuable feedback on this deliverable.

References

[1] ADMORPH. Requirement analysis and use case specification. Technical report, 2020.

[2] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R.I. Davis. An empirical survey-based study into industry practice in real-time systems. In *Real-Time Systems Symposium*, 2020.

[3] Farhad Arbab and Farhad Mavaddat. Coordination through channel composition. In Farhad Arbab and Carolyn L. Talcott, editors, *Coordination Models and Languages, 5th International Conference, COORDINATION 2002, YORK, UK, April 8-11, 2002, Proceedings*, volume 2315 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 2002.

[4] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Softw.*, 28(3):41–48, 2011.

[5] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India*, pages 3–12. IEEE Computer Society, 2006.

[6] Christian Berger. Sensedsl: Automating the integration of sensors for mcu-based robots and cyber-physical systems. In *14th Workshop on Domain-Specific Modeling (DSM'14), Portland, USA*, pages 41—46, New York, NY, USA, 2014. ACM.

[7] G. Bernat, A. Burns, and A. Liamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50, 2001.

[8] Simon Bliudze and Joseph Sifakis. The algebra of connectors - structuring interaction in BIP. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.

[9] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[10] A. Cervin. Analysis of overrun strategies in periodic control tasks. *IFAC Proceedings Volumes*, 38(1), 2005.

[11] H. Choi, H. Kim, and Q. Zhu. Job-class-level fixed priority scheduling of weakly-hard real-time systems. In *Real-Time and Embedded Technology and Applications Symposium*, 2019.

[12] Paolo Ciancarini and Alexander L. Wolf. Issues in coordination languages and architectures. *Sci. Comput. Program.*, 46(1-2):1–3, 2003.

[13] X. Dai. A gel'fand-type spectral radius formula and stability of linear constrained switching systems. *Lin. Algebra and Applications*, 2012.

[14] M. Gaukler, T. Rheinfels, P. Ulbrich, and G. Roppenecker. Convergence rate abstractions for weakly-hard real-time control. *arXiv preprint arXiv:1912.09871*, 2019.

[15] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992.

[16] Bert Gijsbers and Clemens Grelck. An efficient scalable runtime system for macro data flow processing using s-net. *Int. J. Parallel Program.*, 42(6):988–1011, 2014.

[17] Clemens Grelck and Sven-Bodo Scholz. SAC - A functional array language for efficient multi-threaded execution. *Int. J. Parallel Program.*, 34(4):383–427, 2006.

[18] Clemens Grelck, Sven-Bodo Scholz, and Alexander V. Shafarenko. A gentle introduction to s-net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Process. Lett.*, 18(2):221–237, 2008.

[19] Clemens Grelck, Sven-Bodo Scholz, and Alexander V. Shafarenko. Asynchronous stream processing with s-net. *Int. J. Parallel Program.*, 38(1):38–67, 2010.

[20] Z.A.H. Hammadeh, S. Quinton, and R. Ernst. Weakly-hard real-time guarantees for earliest deadline first scheduling of independent tasks. *ACM Transactions on Embedded Computing Systems*, 2019.

[21] M. Hertneck, S. Linsenmayer, and F. Allgöwer. Efficient stability analysis approaches for nonlinear weakly-hard real-time control systems. *Automatica*, 2021.

[22] C. Huang, K.-C. Chang, C.-W. Lin, and Q. Zhu. Saw: A tool for safety analysis of weakly-hard systems. In *International Conference on Computer Aided Verification*. Springer, 2020.

[23] C. Huang, W. Li, and Q. Zhu. Formal verification of weakly-hard systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, 2019.

[24] Sung-Shik T. Q. Jongmans, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab, and Hamideh Afsarmanesh. Orchestrating web services using reo: from circuits and behaviors to automatically generated code. *Serv. Oriented Comput. Appl.*, 8(4):277–297, 2014.

[25] V. Kozyakin. The berger–wang formula for the markovian joint spectral radius. *Lin. Algebra and its Applications*, 448, 2014.

[26] J.B. Lasserre. Global optimization with polynomials and the problem of moments. *SIAM Journal on optimization*, 2001.

[27] H. Liang, Z. Wang, R. Jiao, and Q. Zhu. Leveraging weakly-hard constraints for improving system fault tolerance with functional and timing guarantees. In *Conference On Computer Aided Design*, 2020.

[28] H. Liang, Z. Wang, D. Roy, S. Dey, S. Chakraborty, and Q. Zhu. Security-driven codesign with weakly-hard constraints for real-time embedded systems. In *Int. Conference on Computer Design*, 2019.

[29] S. Linsenmayer and F. Allgower. Stabilization of networked control systems with weakly hard real-time dropout description. In *Conference on Decision and Control*, 2017.

[30] S. Linsenmayer, M. Hertneck, and F. Allgower. Linear weakly hard real-time control systems: Time-and event-triggered stabilization. *IEEE Transactions on Automatic Control*, 2020.

[31] M. Maggio, A. Hamann, E. Mayer-John, and D. Ziegenbein. Control-System Stability under Consecutive Deadline Misses Constraints. In *Euromicro Conference on Real-Time Systems*, 2020.

[32] Lukas Miedema and Clemens Grelck. Strategy switching: Smart fault-tolerance for weakly-hard resource-constrained real-time applications. In *International Conference on Software Engineering and Formal Methods*, pages 129–145. Springer, 2022.

[33] P.A. Parrilo and A. Jadbabaie. Approximation of the joint spectral radius using sum of squares. *Lin. Algebra and its Applications*, 2008.

[34] P. Pazzaglia, A. Hamann, D. Ziegenbein, and M. Maggio. Adaptive design of real-time control systems subject to sporadic overruns. In *Design, Automation & Test in Europe Conference Exhibition*, 2021.

[35] P. Pazzaglia, C. Mandrioli, M. Maggio, and A. Cervin. Deadline-Miss-Aware Control. In *Euromicro Conference on Real-Time Systems*, 2019.

[36] M. Philippe, R. Essick, G.E. Dullerud, and R.M. Jungers. Stability of discrete-time switching systems with constrained switching sequences. *Automatica*, 72, 2016.

[37] Subhav M. Pradhan, Abhishek Dubey, Aniruddha Gokhale, and Martin Lehofer. CHARIOT: A domain specific language for extensible cyber-physical systems. In *15th Workshop on Domain-Specific Modeling (DSM'15), Pittsburgh, USA*, pages 9—-16, New York, NY, USA, 2015. ACM.

[38] Julius Roeder, Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. Towards energy-, time- and security-aware multi-core coordination. In Simon Bliudze and Laura Bocchi, editors, *22nd International Conference on Coordination Models and Languages (COORDINATION 2020), Malta*, volume 12134 of *Lecture Notes in Computer Science*, pages 57–74. Springer, 2020.

[39] Davide Rossi, Francesco Poggi, and Paolo Ciancarini. Dynamic high-level requirements in self-adaptive systems. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 128–137. ACM, 2018.

[40] G. Rota and W Strang. A note on the joint spectral radius. *Indagatione Mathematicae*, pages 379—-381, 1960.

[41] 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.

[42] L. Schenato. To zero or to hold control inputs with lossy links? *IEEE Transactions on Automatic Control*, 54(5), 2009.

[43] Francis Tam, Mike Woodward, and G Toppong. Ft-linda: a coordination language for programming distributed fault-tolerance. In *Proceedings of IEEE Singapore International Conference on Networks and International Conference on Information Engineering'95*, pages 649–653. IEEE, 1995.

[44] G. Vankeerberghen, J. Hendrickx, and R.M. Jungers. Jsr: A toolbox to compute the joint spectral radius. In *International Conference on Hybrid Systems Computation and Control*, 2014.

[45] N. Vreman, A. Cervin, and M. Maggio. Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses. In *Euromicro Conference on Real-Time Systems*, 2021.

[46] N. Vreman, R. Pates, and M. Maggio. Weaklyhard.jl: Scalable analysis of weakly-hard constraints. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2022.

[47] J. Wang, M. Maggio, and V. Magron. SparseJSR: A Fast Algorithm to Compute Joint Spectral Radius via Sparse SOS Decompositions. *American Control Conference*, 2021.

[48] J. Wang, V. Magron, and J.B. Lasserre. Tssos: A moment-sos hierarchy that exploits term sparsity. *SIAM Journal on Optimization*, 2021.

[49] X. Xu and B. Acikmese. Approximation of the constrained joint spectral radius via algebraic lifting. *Trans on Automatic Control*, 2020.

[50] X. Xu and Y. Hong. Matrix expression and reachability analysis of finite automata. *Journal of Control Theory and Applications*, 2012.

[51] Pavel Zaichenkov, Olga Tveretina, Alex Shafarenko, Bert Gijsbers, and Clemens Grelck. The cost and benefits of coordination programming: Two case studies in concurrent collections and S-NET. *Parallel Process. Lett.*, 26(3):1650011:1–1650011:24, 2016.