D1.1 First Report on Coordination Language for Robust Adaptive Systems

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

Due Date:	Month 09
Delivery:	Month 09
Lead Partner:	UvA
Editor:	Clemens Grelck, UvA
Dissemination Level:	Public (P)
Status:	Submitted
Approved:	
Version:	1.1

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 number	Author	Comments
28/9/2020 ver. 1.1	Clemens Grelck	Internal review completed
15/9/2020 ver. 1.0	Clemens Grelck	First draft

List of Contributors

Date and version number	Author	Comments
21/9/2020 ver. 1.0	Clemens Grelck	Internal review
05/9/2020 ver. 0.4	Marcus Völp	Section 4
04/9/2020 ver. 0.4	Clemens Grelck	Introduction and Conclusions
04/9/2020 ver. 0.3	Martina Maggio	Section 3
01/9/2020 ver. 0.2	Clemens Grelck	Section 2
01/8/2020 ver. 0.1	Clemens Grelck	Initial report structure

GLOSSARY

APT Advanced and Persistent Threat

BFT-SMR Byzantine Fault Tolerant Statemachine Replication

CPS(oS) Cyber Physical System (of Systems) [38]

DSL Domain Specific Language [39]

FTA Fault Tree Analysis

PPC Power PC

SoS System of Systems [22]

TRL Technology Readiness Level [40]

VIS Vision System

Contents

Executive summary

This deliverable is a report on the consortium's work in Task 1.1 *Coordination Language Design*, in Task 1.3 *Specifying Formal Guarantees for the Adaptation Layer* and in Task 1.4 *Specification of Fault Model and Threat Indicators*.

1 Introduction

Deliverable D1.1 is the first deliverable of work package 1: Specification of Adaptive Systems. It contains the initial report on a coordination language for robust, adaptive systems. This includes three tasks of work package 1, namely

- Task 1.1: coordination language design, led by UvA;
- Task 1.3: specification of formal guarantees for the adaptation layer, led by ULUND;
- Task 1.4: specification of fault model and threat indicators, led by UNILU.

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

We build our work work in this area on previous and on-going work on the TeamPlay coordination language [31]. Work on the underlying coordination model and the core language have been developed in the context of the Horizon-2020 project TeamPlay[1], but continue to be subjects of on-going research, both in the TeamPlay project as well as here in the Admorph project. We expect both synergy effects and fruitful cross-pollination across projects from this setup.

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

The name of the coordination language is TeamPlay, which we deem an appropriate name for a coordination language. Whenever needed throughout this document we disambiguate the language TeamPlay from the sister Horizon-2020 project TeamPlay, from which the language originates.

The remainder of Deliverable D1.1 is organised as follows. The subsequent three sections describe the work in the above mentioned three tasks T1.1, T1.3 and T1.4, respectively. They are written from the perspective (and by) the respective lead partner. Eventually, we summarise our work and draw some conclusions in Section 5.

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

2 Task 1.1: Coordination Language Design

In this section we describe the design of the coordination language TeamPlay. As pointed out before, the underlying coordination model and the core language have been developed in the context of the Horizon-2020 project TeamPlay. We reiterate on both the coordination model in Section 2.1 and the core language in Section 2.2 for completeness and self-containedness of this deliverable report. The TeamPlay coordination language is under active development in both Horizon-2020 projects, TeamPlay and Admorph, with different primary objectives. In Section 2.3 we present our language extensions for fault-tolerance as a primary Admorph contribution. In Section 2.4 we discuss various language extensions that allow programmers to write more abstract code. These extensions have been exclusively motivated by the previous fault-tolerance extensions that substantially extend the amount of parameters and attributes compared with the core TeamPlay language.

2.1 Coordination model

The term *coordination* goes back to the seminal work of Gelernter and Carriero [11] and their coordination language Linda. Coordination languages can be classified as either *endogenous* or *exogenous* [1]. Endogenous approaches provide coordination primitives within application code; the original work on Linda falls into the category. We pursue an exogenous approach that completely separates the concerns of coordination programming and application programming. Software *components* serve as the central artefact in between.

2.1.1 Components

Our exogenous approach fosters the separation of concerns between intrinsic component behaviour and extrinsic component interaction. The notion of a component is the bridging point between low-level functionality implementation and high-level application design. We illustrate our component model in Figure 1. Following the keyword `component` we have a unique component name that serves the dual purpose of identifying a certain application functionality and of locating the corresponding implementation in the object code.

A component interacts with the outside world via component-specific numbers of typed and named input ports and output ports. As the Kleene star in Figure 1 suggests, a component may have zero input ports or zero output ports. A component without input ports is called a *source component*; a component without output ports is called a *sink component*. Source components and sink components form the quintessential interfaces between the physical world and the cyber-world characteristic for cyber-physical systems. They represent sensors and actors in the broadest sense. We adopt the firing rule of Petri-nets, i.e. a component is activated as soon as data (tokens) are available on each input port.

Technically, a component implementation is a function adhering to the C calling and linking conventions whose name and signature can be derived from the component specification in a defined

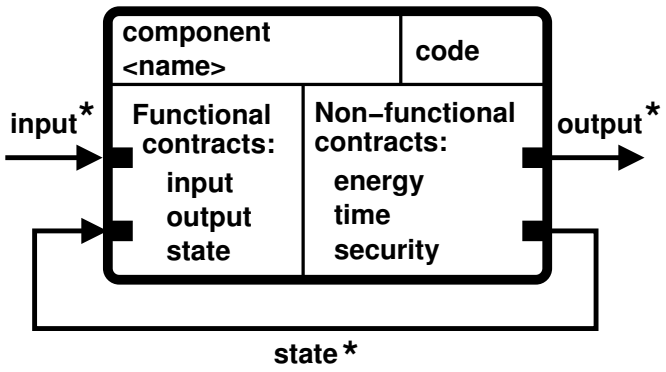


Figure 1: Illustration of component model

way. This function may call other functions using the regular C calling convention. However, the execution of the function, including execution of all subsidiary functions, must not interfere with the execution environment. Exceptions to the former restriction are source and sink components that are supposed to control sensors and actors.

2.1.2 Stateful components

Our components are conceptually stateless. However, some sort of state is very common in cyber-physical systems. We model such state in a functionally transparent way as illustrated in Figure 1, namely by so-called state ports that are short-circuited from output to input. In analogy to input ports and output ports, a component may well have no state ports. We call such a component a (practically) *stateless* component.

Our approach to state is in an interesting way not dissimilar from main-stream purely functional languages, such as Haskell or Clean. They are by no means free of state either, for the simple reason that many real-world problems and phenomena are stateful. However, purely functional languages apply suitable techniques to make any state fully explicit, be it monads in Haskell or uniqueness types in Clean. Making state explicit is key to properly deal with state and state changes in a declarative way. In contrast, the quintessential problem of impure functional and even more so imperative languages is that state is potentially scattered all over the place. And even where this is not the case in practice, proving this property is hardly possible.

2.1.3 Non-functional properties

We are particularly interested in the non-functional properties of code execution. In the TeamPlay project these are energy, time and security while the Admorph project adds fault-tolerance/robustness. Hence, any component not only comes with functional contracts but additionally with non-functional contracts concerning energy, time, security and/or fault-tolerance, and potentially more in the future. These non-functional properties are inherently different in nature. Execution time and energy consumption depend on a concrete execution machinery and vary between different hardware scenarios. In contrast, security, more precisely algorithmic security,

depends on the concrete implementation of a component, e.g. using different levels of encryption, etc. However, different security levels almost inevitably incur different computational demands and, thus, are likely to expose different runtime behaviour in terms of time and energy consumption as well. In Section 2.3 we will discuss the novel fault-tolerance contracts that form our primary contribution to Admorph.

2.1.4 Multi-version components

As illustrated in Figure 2, a component may have multiple versions, each with its own energy, time and security contracts, but otherwise identical functional behaviour. More security requires stronger encryption which requires more computing and, thus, more time and energy. However, many systems do not need to operate at a maximum security level at all times. Take as an example a reconnaissance drone that adapts its security protocol in accordance with changing mission state: low security level while taking off or landing from/to base station, medium security level while navigating to/from mission area, high security level during mission.

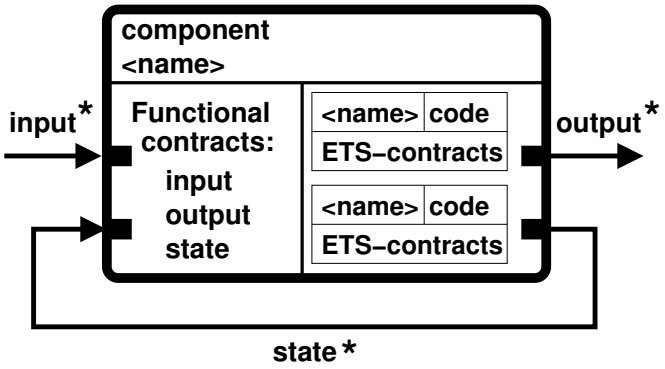


Figure 2: Multi-version component with individual energy, time and security contracts

When performing in low security mode, the drone can use a less resilient encryption when communicating with the base station while highest possible security is paramount in a potentially hostile environment. Continuous adaptation of security levels results in less computing and, thus, in energy savings that could be exploited for longer flight times. Our solution is to embed different versions of the same component that are all functionally equivalent but expose different trade-offs regarding non-functional properties (similar to [33]) and to select the best versions regarding mission state and objectives.

2.1.5 Component interplay

Components are connected via FIFO channels to exchange data, as illustrated in Figure 3. Depending on application requirements, components may start computing at statically determined time slots, when all input data is guaranteed to be present or may be activated dynamically by the presence of all required input data. Components may produce output data on all or on selected output ports.

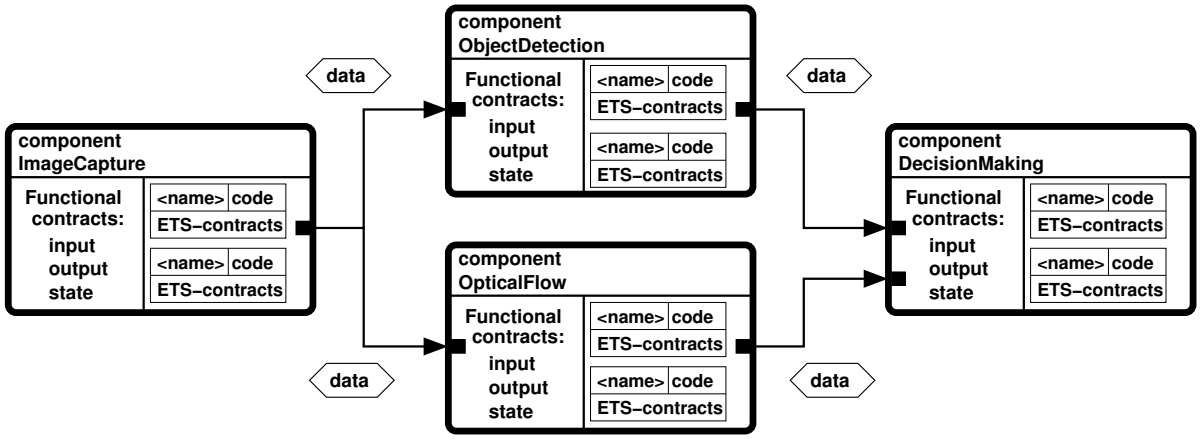


Figure 3: Illustration of data-driven component interplay via FIFO channels

2.2 Coordination Language TeamPlay

We illustrate the TeamPlay coordination language by means of the example shown in Figure 4. Our introductory example is an imaginary subsystem of a car with two sensors feeding messages to a decision controller. This decision controller synchronises the messages pair-wise and sends commands to two subsequent actuators. Figure 5 shows the TeamPlay coordination code that implements this example.

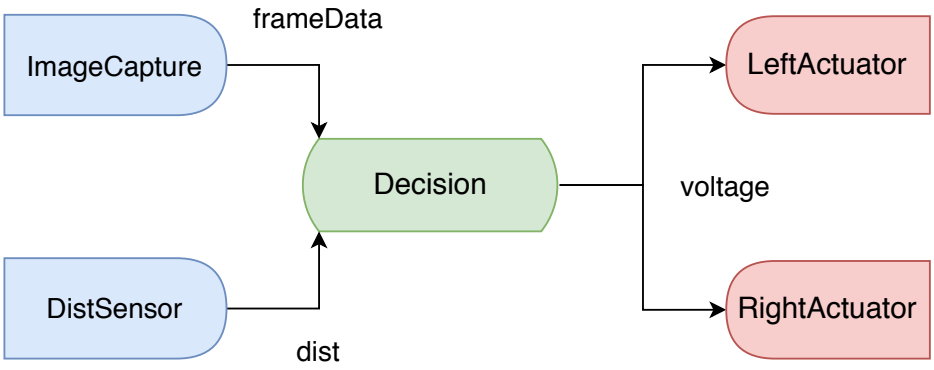


Figure 4: Example for TeamPlay component coordination.

TeamPlay is a component-based streaming language [37] developed primarily with real-time cyber-physical systems in mind. It implements the component coordination model laid out in the previous section. A TeamPlay application definition (line 1) starts with the keyword `app` followed by an identifier that serves as the application's name. Enclosed within curly brackets we can identify three major code regions: `datatypes` starting at line 2, `components` starting at line 6 and `edges` starting at line 24. Data types are declared as pairs of identifier and string, where the former will be used as symbolic type name throughout the TeamPlay code while the latter denotes the C language type definition and merely is used by our compiler during the final code generation step.

Components and edges are at the core of TeamPlay, and so we devote the following two sections to them.

```
app car {
 datatypes {
  (num, "uint32_t")
  (frame, "jpegFrame*")
 }
 components {
  DistSensor {
   outports [ (dist, num) ]
  }
  ImageCapture {
   outports [ (frameData, frame) ]
  }
  Decision {
   inports [ (dist, num) (frameData, frame) ]
   outports [ (voltage, num) ]
  }
  LeftActuator {
   inports [ (voltage, num) ]
  }
  RightActuator {
   inports [ (voltage, num) ]
  }
 }
 edges {
  DistSensor.dist -> Decision.dist
  ImageCapture.frameData -> Decision.frameData
  Decision.voltage -> LeftActuator.voltage & RightActuator.voltage
 }
}
```

Figure 5: TeamPlay coordination code for example of Figure 4

2.2.1 Components

Components serve as representations of stateless computations that map input data tokens on typed incoming streams to output data tokens emitted on type output streams. Following the conceptual approach of exogeneous coordination the actual computation is outside the scope of the coordination code. As pointed out in the previous section, we expect to link our compiled coordination code with independently provided and compiled component implementation code.

Given our focus on cyber-physical systems we assume component implementations to be written in the C language, or possibly in C++.

Coming back to Figure 5 we can identify the definitions of the five components of our example, illustrated in Figure 4. They are enclosed in curly brackets following the key word `components` in line 6. A component definition starts with a name followed by a pair of curly brackets enclosing further information about the component.

As components communicate with other components via FIFO channels, the corresponding ports are the most vital functional properties (or contracts) of components. Following the key words `inports`, `outports` or `state` (The latter is not shown in the running example.) we have a list of pairs of port name and port type with the pairs enclosed in round brackets and the whole list of ports enclosed in square brackets. For example, the `Decision` component has two input ports, i.e. port `dist` of type `num` and port `frameData` of type `frame`, and one output port `voltage` of type `num`. Port types must refer to types previously defined in the `datatypes` section of the coordination code. Port names are freshly introduced identifiers. The number of ports a component may have is fixed but unbounded.

Optionally, ports can specify a multiplicity other than the default of one token. Syntactically, multiplicity is indicated by a triplet as, for instance, in `inports [ (in, 3, frame) ]`. The multiplicity of an inport indicates the number of tokens required for firing. Likewise, outport multiplicity indicates the number of tokens produced on this port in a single firing.

Depending on their inports and outports we can classify components into three classes: source components, transformer components and sink components. Source components are characterised by an empty set of inports, in which case also the key word can be omitted. In cyber-physical applications source components typically represent sensors, like in our example from Figure 4. Sink components do not have outports; they merely consume data. Sink components typically control actuators or data transmitters outside of the scope of the coordination model. Last not least, transformer components (or just components) receive data on their input ports, compute output data solely based on input data and emit the output data on their outports. While transformer and sink components are activated by the availability of data tokens on their respective inports following a Petri-net inspired firing semantics, source components are either interrupt-driven or timer-driven.

2.2.2 Edges

Components are connected via so-called edges, i.e. FIFO channels, as can be seen on line 24-27 in the example code of Figure 5. Following the keyword `edges` and embraced within curly brackets we can identify a sequence of edge specifications consisting of a component/outport specification, an arrow and a component/inport specification. These simple edges are also illustrated in Figure 6.a for the code and visually illustrated in Figure 6.c. On either side component name and port name are separated by a dot. For the sake of concise code, port names can be omitted in the not uncommon case that a component only has a single inport or a single outport, and thus the port name is unambiguous.

There are a number of constraints on edge specifications. They must refer to valid outport and

inport specifications from the components section of the coordination code. Furthermore, the port types of outport and inport must coincide, thus defining the token type of the edge. Any port specified in components section must be connected by exactly one edge. Last not least, cycles are not permitted.

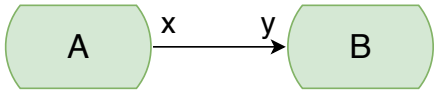
A special case are broadcast edges as shown in Figure 6.b for the code and visually illustrated in Figure 6.d). A broadcast edge, as the name suggests transmits copies of tokens emitted by the left hand side component's outport to all component inports specified on the right hand side and separated by ampersands. Our car subsystem example in Figure 4 makes use of a broadcast edge to send the same value to both the left and right actuator.

```
A.x -> B.y
```

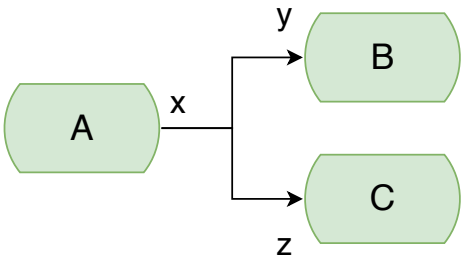
(a) Simple channel

```
A.x -> B.y & C.z
```

(b) Broadcast channel



(c) Simple edge visualisation



(d) Broadcast or duplication edge visualisation

Figure 6: Code specification and visual illustration of simple edges (left) and broadcast edges (right)

2.2.3 Non-functional properties

As pointed out before, one of the goals in the design of the TeamPlay coordination language is the active management of non-functional properties. In the TeamPlay project three such non-functional properties are in the focus of interest: energy, time and security. Both energy and time can only be considered in relation to some concrete execution machinery. Thus, any mentioning of energy or time in the coordination source code would inherently make the code hardware-specific, which is not what we want. In our current implementation of the TeamPlay language we make use of a so-called non-functional properties file (NFP) that functions as a sort of data base storing per component time and energy consumption values for the whole variety of hardware execution units of interest. Depending on the concrete hardware properties concrete values can be derived from static code analysis, dynamic profiling or simply asserted by the user.

The third non-functional property of interest, security, differs from energy and time as security (in our interpretation of the word) is an algorithmic or code property that is independent of the actual execution machinery. As demonstrated in Figure 7, line 5, TeamPlay supports the

```
components {
 Encryption {
  inports [ (original, frame) ]
  outports [ (encrypted, frame) ]
  security 4
  arch "arm/big"
}
```

Figure 7: Example of a component with non-functional properties: security and architecture

specification of a security level in form of a natural number, using the `security` key word. Here, we interpret higher numbers as denoting better security. Any concrete meaning of security levels, however, are application-specific.

In a similar way we can specify an class of hardware on which the component must be scheduled for execution, using the `arch` key word and a string representation. This feature is relevant for the particularly targeted heterogeneous architectures, and the string refers to an architecture specification that again is outside the scope of the TeamPlay coordination language. In the concrte example of Figure 7 the component `Encryption` is (for whatever reason) to only be scheduled on the big cores of an ARM big.LITTLE platform.

2.2.4 Multi-version components

As sson as non-functional properties rule, it becomes particularly interesting to have multiple versions of a component that expose identical functional behaviour but implement different trade-offs of the non-functional properties of interest. Figure 8 demonstrates how this can be accomplished in TeamPlay.

```
components {
 Encryption {
  inports [ (in, frame) ]
  outports [ (out, enc) ]
  version WeakerEncryption {security 4}
  version MediumEncryption {security 6}
  version StrongEncryption {security 9}
 }
}
```

Figure 8: Example of a multi-version component. The Encryption component has three different implementations, each with a different security value.

We see an alternative specification of the `Encryption` component from the previous section. It now features three different versions, namely `WeakerEncryption` with security level 4,

`MediumEncryption` with security level 6 and `StrongEncryption` with security level 9. Different versions of one component all share the same port specifications and must behave identically from the outside functional perspective. As the example demonstrates, this does not mean bit-wise equivalence of the input/output relation.

Our original design foresees the addition of more non-functional properties, and we will make extensive use of this in the next section for the sake of introducing fault-tolerance.

2.3 TeamPlay Language Extensions for Fault-tolerance

Our first extension of the TeamPlay language is in the specification of selected fault-tolerance methods known in literature. Some components or groups of components may be more important than others, depending on the target hardware, application domain, and other factors. We opt for a user-directed approach where the user can specify which of the predefined options to apply in different parts of the application. This is due to major challenges in having a compiler or scheduler analyse the criticality of a component in the application as a whole. Furthermore, the way fault-tolerance is implemented and achieved needs to be transparent to the programmer in order to make sure they they fit the application requirements.

As in the previous section we illustrate the TeamPlay language extensions for fault-tolerance by example. For completeness we also provide a formal syntax specification of the complete language in EBNF form in Appendix A.2.

2.3.1 Checkpoint/restart

Checkpoint/restart lets the system return to a stable (backup) state when a fault has occurred [36, 35]. Generally, the downside of checkpoint/restart methods is the concrete state of some failing software unit is difficult to assess and, thus, in the worst case the entire process image needs to be saved at each checkpoint. That is prohibitively expensive, both in storage space and execution time.

Here the architecture of our coordination-based approach pays off. It creates a middleware layer where our system software can precisely keep copies of the arguments of an individual component invocation before giving control to the third-party provided component implementation. The stateless nature of TeamPlay components ensures that no other data affects the computation. Note here that this property remains valid even if state ports are used as described in Section 2.1.2. The backup copies of the argument values only need to be stored while the component is computing. As soon as it emits its output data on its outports, the backup copies can be discarded.

The benefits of using checkpoint/restart are (potentially) three-fold: implementation is straightforward, coordination between hardware components is not needed, and it only requires extra memory and copy time but no redundant active components. Figure 9 shows how checkpoint/restart can be specified on the `Decision` component from Figure 5. Currently, our specification of checkpoint/restart has no supported options, hence the empty pair or curly brackets.

```
Decision {
 inports [(frameData, int) (dist, int)]
 outports [(voltage, int)]
 checkpoint {}
}
```

Figure 9: Defaults of the checkpoint/restart specification.

2.3.2 Standby or primary-backup

In standby or primary-backup methods, standby components can take over the active computing component in case of failure, as illustrated in Figure 10. Initially, the output of the primary process is used. Should a fault be detected, the output of the standby component is used instead. A distinction can be made between cold, warm and hot standby which differ in the amount of synchronisation the backup components have to the active components [23]. The distinction between these types can be defined as follows:

Cold The backup component(s) are only initialised and then turned off.

Warm The state of the active component is mirrored to the backyp component(s) at specified points.

Hot The backup component(s) are actively synchronised with the primary component so that the backup component can take over immediately.

In case of crash failures, components with long startup times benefit from this type of synchronisation because the working component can be taken over faster when using primary-backup [23]. Generally, the number of required computing resources for primary-backup are lower compared to methods like NMR. Especially using cold standby can save a lot of energy resources, which is important in (often) battery powered systems like those in the cyber-physical class. The main disadvantage of this method is that a fault needs to be detected before the redundant component can take over. Furthermore, primary-backup cannot detect value faults (i.e., there is no voting) and it relies on an error detection method to detect faults, so that the active component can be taken over. Thus, this method is primarily useful in systems which require fast switch times while keeping a state close to the state of the original, crashed process [23].

In primary-backup, the state of the primary and standby component is synchronised. The degree of this synchronisation depends on which flavour, cold, warm or hot is implemented. This allows the application to quickly switch outputs when a fault is detected. In TeamPlay, the firing of a component is discrete, i.e., every execution produces output tokens only once. This, together with the fact that state is made explicit in the buffers of the edges, means that it is not necessary to run the primary and standby components at the same time, i.e., they do not have to be synchronised. We can simply provide copies of the input tokens to the hardware units and take the first unit who delivers an output as the primary component. If it fails, one of the standby components will

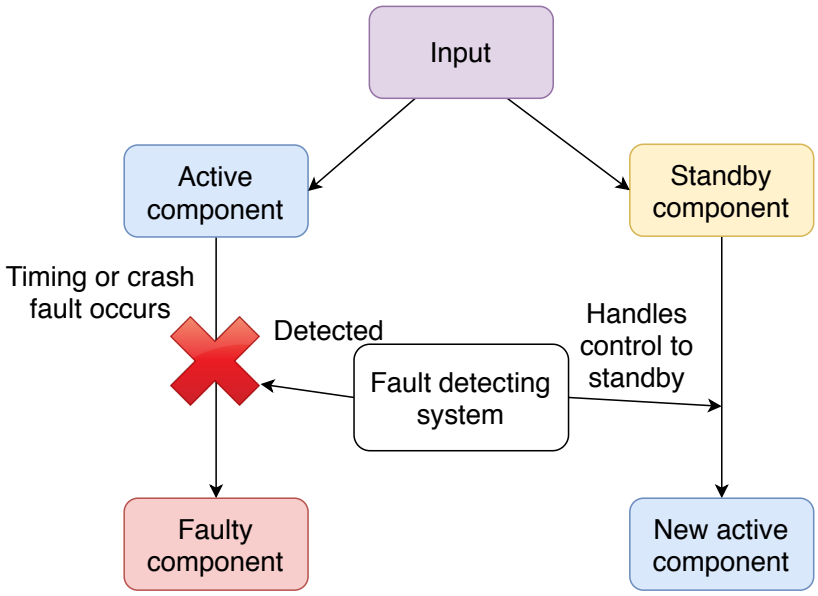


Figure 10: Illustration of recovery of a failing active component using the hot standby method. When a fault is detected, the output of the active component is routed to that of the standby component, so that the service can continue.

deliver output instead. This makes this method predictable as one does not have to account for switching from the primary to the replica component or synchronisation mechanisms.

Figure 11 shows the way primary-backup can be specified. The specified options again use default values. In primary-backup the `replicas` option can be specified as an integer denoting the number of replicas to run for this component.

```
1 Decision {
2  inports [(frameData, int) (dist, int)]
3  outports [(voltage, int)]
4  standby {
5   replicas 2
6  }
7 }
```

Figure 11: Defaults of the primary-backup specification.

2.3.3 N-Modular redundancy

A classic example of physical redundancy is N-Modular Redundancy (NMR). In this strategy, n independent identical processes are executed with identical input [36, 23]. These n processes are followed by voting processes, which vote which answer they will be outputting. This method

primarily focuses on masking transient faults. Depending on the fault-model for the application, it can be possible that the voter processes fail. In order to decrease the chance of this happening it is possible to increase the number of voters [3].

If a minority of the computational processes have faults, a majority vote will still result in the correct answer. Triple Modular Redundancy (TMR) [20, 36, 23] is a special case of NMR in which n is minimally chosen such that the computation does not have to be repeated (when a single fault is present). Since we can't know which process is likely to be faulty in case $n = 2$. However, this double-modular redundancy method has uses as an error detection method since it can be used to detect transient errors. Figure 12 illustrates N-Modular Redundancy with a pipeline consisting of two stages of components followed by voting processes.

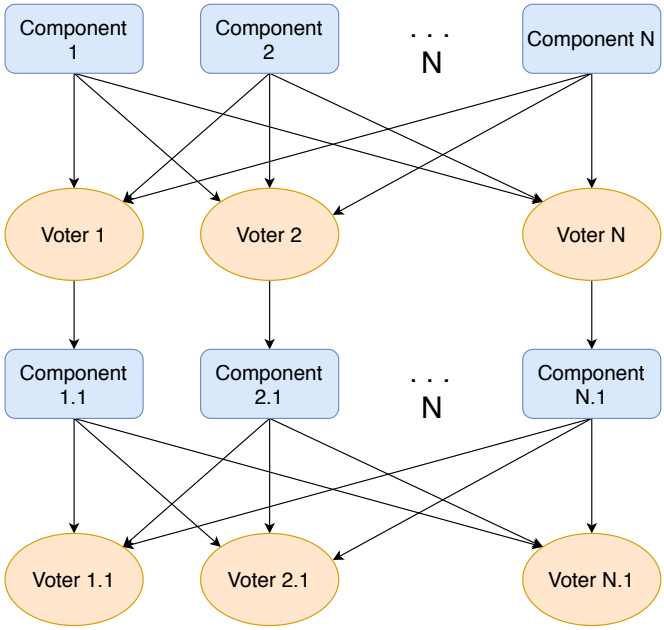


Figure 12: Illustration of N-Modular Redundancy (NMR)

NMR is a mechanism that deals with transient faults without employing low level (hardware) error detection techniques. Furthermore, NMR is a time-predictable method, i.e., it is suitable for use in real-time systems [29]. Figure 13 shows the defaults of the N-modular redundancy. We support the following options:

- `replicas` (line 6), integer signifying the number of replicas. Default is `3`, meaning a TMR setup.
- `votingReplicas` (line 7), integer signifying whether and how much the voting processes need to be replicated.
- `waitingTime` (line 8), how long processes should wait before initiating the voting process. Given as a percentage of the average execution time of the finished components, the percentage can be higher than `100%`.

- `waitingStart` (line 9), defines the starting point of waiting. When `waitingStart` is `majority`, processes start waiting based on the execution time when a majority of processes are done. In the case of `single`, the waiting will start when a single process is ready.
- `waitingJoin` (line 10), boolean defining whether processes that are finished later should be added in the waitingTime calculation. Can apply on both a `waitingStart` value of `majority` and `single`.

```
1  Decision {
2   inports [(frameData, int) (dist, int)]
3   outports [(voltage, int)]
4
5   nModular {
6    replicas 3
7    votingReplicas 2
8    waitingTime 30%
9    waitingStart majority
10   waitingJoin true
11  }
12 }
```

Figure 13: Default options for N-modular redundancy

2.3.4 N-version programming

In N-Version Programming (NVP) multiple functional equivalent implementations of the same component are created [28]. At runtime, they can be run in the same way as when using N-Modular Redundancy. The advantage of NVP over NMR is that software faults present in one implementation are caught the same way as transient hardware faults are caught.

The disadvantage of NVP is that it combines high runtime overhead with additional development cost. However, TeamPlay already supports the concept of multi-version components. What has primarily been intended to exploit different energy/time/security trade-offs, can now be reused for fault-tolerance. By leveraging our existing version mechanic, the programmer can kill two birds with one stone by reusing existing versions for NVP.

The options we support in NVP are similar to N-Modular Redundancy (Section 2.3.3) adding an option to specify `versions`, which defines which versions should be used and how many of each of these versions should be run. This is ilustrated in Figure 14.

2.4 TeamPlay Language Extensions for Ease of Programming

To assist in managing the fault-tolerance options defined in the previous section and reducing duplication in the coordination code we introduce new constructs into the TeamPlay coordination

```
components {
 Encryption {
  inports [ (in, frame) ]
  outports [ (out, enc) ]
  version Encryption1 {security 4}
  version Encryption2 {security 6}
  version Encryption3 {security 9}

  nVersion {
   versions [ (Encryption1, 2) (Encryption2, 1) ]
  }
 }
}
```

Figure 14: Example of `versions`. The first entry in the tuple specifies the version while the second specifies how many replicas of that version should exist. Not all versions have to be specified because the default value is `0`.

language. While the motivation for these additions originates from managing fault-tolerance options, they are also meant to work with existing options, like deadline and period specification, and future options of the language. Again we illustrate the TeamPlay language extensions for ease of programming by example. For completeness we provide the formal syntax specification of the complete TeamPlay language in EBNF form in Appendix A.2.

2.4.1 Profiles

In order to provide intuitive structures while keeping code duplication low, we pursue the idea of using profiles. These profiles are defined globally and can be added to components to prevent the user from having to specify the same options again and again. Figure 15 shows an example of adding profiles. The key word `profiles` opens a new code section where we define the profile named `TMR` as triple-modular-redundancy. Later we apply the `TMR` profile to the `ImageCapture` component (see Figure 4 for the complete example) by placing it inside the `profiles` attribute in the component definition on line 11.

Profiles aim to cover the general case in which the programmer wants to deal with certain components in the exact same way. However, we also envision a situation in which a programmer desires variations or additions in the use of a profile, possibly in conjunction with other profiles. This can range from changing a single setting through systematically overwriting the original profile. To do this we enable the user to combine and overwrite profiles in a cascading manner, overwriting settings that are previously set and adding settings that are not previously set. Figure 16 shows this in the `ImageCapture` component on line 17. On this line, two profiles are added. The order of these profiles matters as the profile given first (`TMR`) will be overwritten by the following profiles (in this case, `TMRRedundantVoting`).

```
1  profiles {
2   TMR {
3    nModular {
4     replicas 3
5    }
6   }
7  }
8  components {
9   ImageCapture {
10   outports [ (out, frame) ]
11   profiles [ TMR ]
12  }
13 }
```

Figure 15: Profile example, the options of the `TMR` profile will be applied on the `ImageCapture` component

```
1  profiles {
2   TMR {
3    nModular {
4     replicas 3
5     votingReplicas 1
6    }
7   }
8   TMRRedundantVoting {
9    nModular {
10    votingReplicas 3
11   }
12  }
13 }
14 components {
15  ImageCapture {
16   outports [ (out, frame) ]
17   profiles [ TMR TMRRedundantVoting ]
18  }
19 }
```

Figure 16: Cascading example, `votingReplicas` is equal to three on the `ImageCapture` component as the `TMRRedundantVoting` profile overwrites the `TMR` profile.

It is also possible to mix the globally defined profiles and inline settings. In Figure 17, the inline setting on line 13 overwrites the given `TMR` profile on line 12. The order of the keywords matters: inports is first, then outports, profiles, and then inline options can be defined. This order

was chosen to make it clear that profiles are overwritten by inline options just like earlier defined profiles are overwritten by later ones.

```
1  profiles {
2   TMR {
3    nModular {
4     replicas 3
5     votingReplicas 2
6    }
7   }
8  }
9  components {
10  Decision {
11   inports [ (dist, num) (frameData, frame) ]
12   outports [ (voltage, num) ]
13   profiles [ TMR ]
14   nModular { replicas 4 }
15  }
16 }
```

Figure 17: Cascading inline and separated profile definition. The inline option will overwrite the `replicas 3` option of the TMR profile because the inline option is more specific than a profile. Other options, in this case the `votingReplicas` option will be merged with the inline profile.

2.4.2 Controlling cascading options

In general-purpose languages, there are keywords that can be used to communicate certain intentions towards other programmers, for example to indicate that some function needs to implemented or that some variable must not be overwritten. To this end, we introduce two keywords to allow more expressiveness and control over options: `remove` and `vital`.

`remove` can be placed before a (fault-tolerance) option to signify that a method should be removed if it is specified. `remove` can be used when the programmer wants to cascade certain options using multiple profiles but not all. For now, we only allow removes on the first layer keywords of the fault-tolerance options and normal settings. When cascading multiple profiles the remove is handled per singular cascade. For example, given three profiles, the first of which adds `nModular` with `replicas 5`, the second profile removes `nModular` while the third one adds it again without specified options, i.e., the default settings will be used. In handling the cascade of the first and second profile, `nModular` will be deleted. When the result of the first cascading operation is cascaded with the third and final profile, `replicas` will be equal to 3.

The `vital` keyword signifies an option that is not allowed to be changed by cascading or removal. Figure 18 shows `remove` (line 13) and `vital` (line 4). If we were to have a component that

has these profiles in the same order they are defined, i.e., `profiles [ TMR TMRRedundantVoting RemoveNModular ]`, the TeamPlay compiler produces an error that the option with the `vital` keyword on line 4 cannot be overwritten on line 9, nor can it be removed by the `remove` keyword on line 13.

```
1  profiles {
2   TMR {
3    nModular {
4     vital votingReplicas 1 // will not be overwritten by cascading
       profiles
5    }
6   }
7   TMRRedundantVoting {
8    nModular {
9     votingReplicas 3
10   }
11  }
12  removeNModular {
13   remove nModular
14  }
15 }
```

Figure 18: Example for the usage of `vital` and `remove`. The option with the `vital` statement on line 4 cannot be changed or removed.

2.4.3 Basic sub-networks

Cascading profiles is a modular approach to applying profiles on individual components. However this approach can still become repetitive as groups of components responsible for a specific functionality may have a similar criticality, i.e., they require the same or similar fault-tolerance options. To support applying profiles on multiple components at once, we introduce sub-networks. This helps us in three areas:

1. applying potential future edge-based fault-tolerance methods;
2. applying profiles on multiple components via inheritance;
3. enabling programmers to reuse sub-networks of components in different parts of a program.

This section covers the first two points while the third one is treated in the next section.

Basic sub-networks are defined within the `components` section and can contain one or more components. Figure 20 shows a schematic example of a sub-network, which is detailed in code in Figure 19. Note that this example is similar to our previous example shown in Figure 4. The

Sensors sub-network is defined with two components: ImageCapture (line 9) and DistanceSensor (line 13). These two components inherit the TMR profile from the Sensors sub-network. In these components it is again possible to define profiles and inline settings as can be seen on line 11. These sub-networks are treated similarly to components as they have the same outports as normal components and they are automatically instantiated. Everything that is defined under the components keyword is instantiated under the name it has been given. The major difference between the interface of a normal component and a sub-network is that in sub-networks the programmer has to specify when an edge should be presented towards the outside of the sub-network. If an edge should go towards the outside of a sub-network the out keyword can be used instead of the component name, as can be seen on lines 17 and 18 in Figure 19. The in keyword can be used to allow an edge from the inport of the sub-network to connect to a component inside the sub-network, as is shown in line 31.

The advantage of specifying the edges inside the sub-network is that they are self-contained units. Thus, they can easily be copied in other parts of the program or even other programs entirely, provided the used types also exist there.

2.4.4 Templates

In order to have multiple instances of the same subgraph or component, i.e., have a single definition and use them multiple times in a graph, we have to decouple the component definition and their instantiation since there can be multiple instances of the same sub-network or component. To do this, we introduce an additional top-level code section in application definitions called templates. Sub-networks and components defined under this keyword are not automatically instantiated but serve as abstract definitions. These definitions must be instantiated explicitly in the components section.

In order to illustrate this templating mechanism we expand our previous sub-network example from Figure 20 into the one shown in Figure 22. The corresponding coordination code can be found in Figure 21. Here, we take the previous example and add another instance of the same Sensors sub-network to the application by moving the Sensors sub-network to templates. Two Sensors sub-networks, SensorFront and SensorRear, are instantiated on lines 26 and 27, respectively. Both are connected to an extended Decision component.

The split between components and templates makes explicit to the programmer what is already instantiated and what is not. Components defined in the components section are automatically instantiated. In contrast templates defined in the templates section must still be instantiated explicitly in the components section. This also reduces additional code overhead for smaller programs, as they can be automatically instantiated by placing their definitions directly into the components section.

2.4.5 Parameterised templates

Templating is not limited to sub-networks as components can also be defined in templates so that they can be instantiated multiple times. However, having instances of the same component or

```
profiles {
 TMR {
  nModular { replicas 3 }
} }
components {
 Sensors {
  outports [ (dist, num) (frameData, frame) ]
  profiles [ TMR ]
  ImageCapture {
   outports [ (frameData, frame) ]
   nModular { replicas 4 }
  }
  DistanceSensor {
   outports [(dist, num)]
  }
  edges {
   ImageCapture -> out.frameData
   DistanceSensor -> out.dist
 } }
 Actuators {
  inports [ (voltage, num) ]
  profiles [ TMR ]
  LeftActuator {
   inports [ (voltage, num) ]
   nModular { replicas 4 }
  }
  RightActuator {
   inports [ (voltage, num) ]
  }
  edges {
   in.voltage -> LeftActuator.voltage & RightActuator.voltage
 } }
 Decision {
  inports [ (frameData, frame) (dist, num) ]
  outports [ (voltage, num) ]
} }
edges {
 Sensors.dist -> Decision.dist
 Sensors.frameData -> Decision.frameData
 Decision.voltage -> Actuators.voltage
}
```

Figure 19: Sub-network example which mirrors 20. The `TMR` profile applies to all sub-components of the `Sensors` subgraph, `ImageCapture` and `DistanceSensor`.

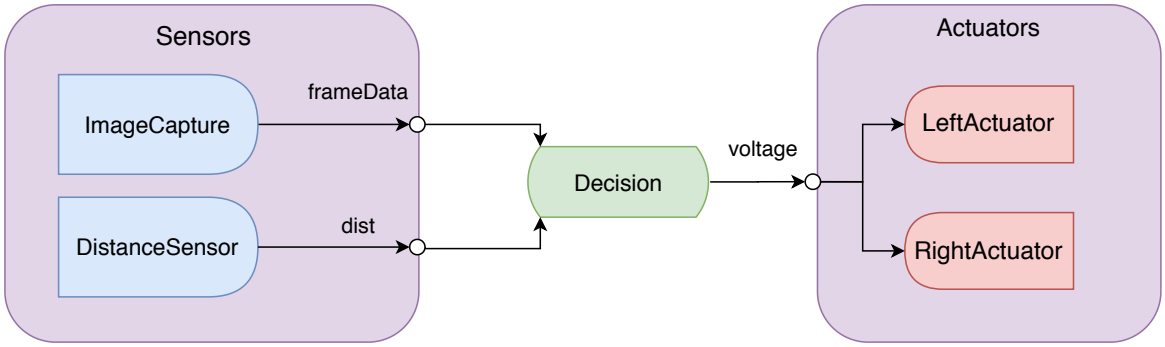


Figure 20: Conversion of Figure 4 to sub-networks. The edges originating from `ImageCapture` and `DistanceSensor` are presented towards the outside of the graph. From the outside of the `Sensors` sub-network, two edges go to the `Decisision` component. The `Decision` component calculates a voltage goes into the sub-network. From the edge of the `Actuators` sub-network, a duplicate edge copies the same voltage token to the `LeftActuator` and `RightActuator`. The coordination code that corresponds with this figure is listed in Figure 19.

sub-network does not mean that they have the same criticality (i.e., the same profiles), or, to move past the focus on fault-tolerance, have the same settings (e.g., `deadlines`). To satisfy the need for variations between instances of components or sub-networks we introduce parameterised templates.

Another version of the `Sensors` sub-network example we showed in Figure 20 is presented in Figure 23. Parameters can be added in the template definition, as can be seen on line 7. Occurrences of the parameter names in setting areas (e.g., fault-tolerance settings, deadline, period, etc.) are substituted by the value given during instantiation (lines 24 and 25). In the absence of parameters the brackets can be left out. On line 24 we add the TMR profile, but on line 25 we leave the parameter empty by giving an underscore as a parameter. For now, we only support adding parameters to options and profile names as these are the areas where we expect this variation between instances exists.

2.4.6 Component and function names

In TeamPlay, component names in the coordination domain are tied to function names in the computation domain. This creates problems with templates when having multiple sensors, when using the same function without any changes would just reuse the same sensor. To remedy this, we introduce the `cname` keyword to specify the name of the function inside a component. This becomes a useful mechanism when paired with parameter passing as the name of the function is then tied to its instance instead of its template.

```
profiles {
 TMR {
  nModular { replicas 3 }
 }
}

templates {
 Sensors {
  outports [ (distance, num), (frameData, frame) ]
  profiles [ TMR ]
  ImageCapture {
   outports [ (outFrame, frame) ]
   nModular { replicas 4 }
  }
  DistanceSensor {
   outports [ (dist, num) ]
  }
  edges {
   ImageCapture.outFrame -> out.outFrame
   DistanceSensor.dist -> out.dist
  }
 }
}

components {
 Sensors SensorFront // Instantiates a sub-network of type Sensors
 Sensors SensorRear // Instantiates a sub-network of type Sensors

 Decision {
  inports [ (frameFront, frame) (distanceFront, num)
       (frameRear, frame) (distanceRear, num) ]
 }
}

edges {
 SensorFront.dist -> Decision.distFront
 SensorFront.frameData -> Decision.frameFront

 SensorRear.dist -> Decision.distanceRear
 SensorRear.frameData -> Decision.frameRear
}
```

Figure 21: Example of multiple instances of the same sub-network in an application

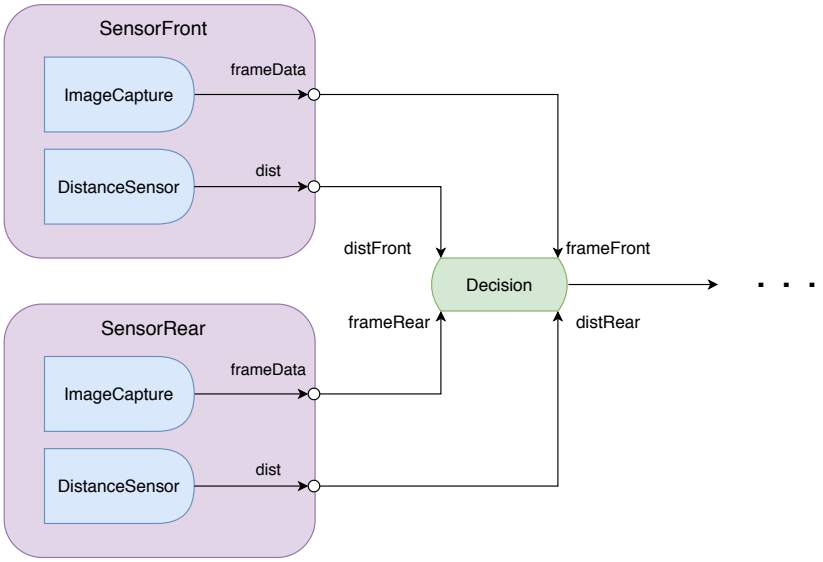


Figure 22: Extension of Figure 20 which adds another `Sensor` sub-network using templating. The `Actuators` sub-network is left out for brevity. The corresponding code snippet can be found in Figure 23.

```
profiles {
 TMR {
  nModular { votingReplicas 3 }
 }
}
templates {
 Sensors( numReplicas, rootProfile) {
  outports [ (dist, num), (frameData, frame) ]
  profiles [ rootProfile ]
  ImageCapture {
   outports [ (frameData, frame) ]
   nModular { replicas numReplicas }
  }
  DistanceSensor {
   outports [(dist, num)]
  }
  edges {
   ImageCapture -> out.frameData
   DistanceSensor -> out.dist
  }
 }
}
components {
 Sensors(3, TMR) SensorFront
 Sensors(4, _) SensorRear

 Decision {
  inports [ (distFront, num) (frameFront, frame)
       (distRear, num) (frameRear, frame) ]
 }
}
edges {
  SensorFront.dist -> Decision.distFront
  SensorFront.frameData -> Decision.frameFront

  SensorRear.dist -> Decision.distRear
  SensorRear.frameData -> Decision.frameRear
}
```

Figure 23: Example of specifying parameters in a template. In `SensorFront`, NMR will be utilised with three replicas and the TMR profile will be applied to all sub-components. `SensorRear` on the other hand will have four replicas and no other profile will be applied.

3 Task 1.3: Specifying Formal Guarantees for the Adaptation Layer

The aim of this task is to specify the formal guarantees that the adaptation layer has to meet in order to ensure the correct functioning even in the presence of an attack or of a fault. We focus on control systems that receive their sensor input and has to compute a corresponding control signal to actuate. In our initial research proposal, the task objective was defined as follows.

> *One of the objectives of this work package is to specify what kind of formal guarantees can be provided on the adaptation. An abstract example could be: "in response to event X, the predicate Y is false for a maximum time of T", which in concrete can be "if the temperature measured at the CPU level is higher than 90 degrees, after 30 seconds the system is able to recover and stops missing deadlines". This task will investigate how the guarantees can be specified.*

Suppose an event (fault or attack) happens at time t_{event}. The even start triggers anomalous conditions in which the control signal is not actuated correctly (either because the sensor message is not delivered, or because it is tampered with, or because the computation does not complete within its deadline). We assume that our adaptive system is able to resolve the anomalous situation at time $t_{\text{resolution}}$. We define the duration of the anomaly as

$$R_{\text{anomaly}} = t_{\text{resolution}} - t_{\text{event}}.$$

The aim of this task is to determine the formal guarantees that the adaptation layer has to provide with respect to R_{anomaly} in order to preserve the correct system behaviour. In particular, we accept guarantees expressed in two different forms:

1. The fault/attack should be resolved within R_{max} time. This means guaranteeing $R_{\text{anomaly}} \leq R_{\text{max}}$. In this case, we are stating that there can be subsequent events but each of them cannot last more than a given amount of time. Furtheremore, to simpify the analysis, we assume that only one active event can happen at a time (meaning that if a system is under attack it cannot experience a fault). This assumption is of course limiting and we will try to relax it in the future. However, the ability of determining the maximum duration of an anomaly is beneficial even in the presence of this limitation (if multiple faults can be experienced a bound to their number should be provided and the time R_{max} should be extended to ensure the resolution of all of them).

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

In fact, the first alternative allows us to guarantee that the system always operate in a fail-safe state as long as the event does not last more than $R_{\max}$, while the second alternative allows us to quantify a sensible recovery period after experiencing an event.

We provide some control background in Section 3.1. Translating the first requirement into the corresponding control properties, we want to study the stability of the system, ensuring that no harm is done to the system's execution. In Section 3.2 we summarise the preliminary results we obtained for this requirement.

The second case will be the subject of our future investigation in the work package.

3.1 Background on control theory

In this section we recap the basic concepts of control theory that are used in the rest of the paper. We analyse linear time-invariant models and controllers implemented as periodic tasks with implicit deadlines.

Plant Model The starting point for control design is always understanding the object that the controller should act upon. The control engineer obtains a model $\mathcal{P}_c$ of the plant to control. In most cases, this model is linear and time-invariant, and represents with ordinary differential equations the dynamics of the system in the following form.

$$\mathcal{P}_c : \begin{array}{l} \dot{x}(t) = A_c\, x(t) + B_c\, u(t) \\ y(t) = C_c\, x(t) + D_c\, u(t) \end{array} \tag{1}$$

The equations above are specifying how the internals of a physical object behaves. Typically, these equations are nonlinear, but the behaviour of physical systems can be approximated quite well with a linear system of equations. Each equation is specifying how some of the physical quantities involved in the behaviour of the physical object change over time. If we open a valve, letting some water flow in a bathub, the water flow depends on the section of the pipe and on how open the valve is. We can describe the quantity of water in the bathub by knowing how much water there was at the beginning of time and what was the water flow since then.

Another example, applying this to a specific problem – cruise control – allows us to map the specific items that appear in the equation with their meaning. We are trying to control the car speed, keeping it at a constant value specified by the user, and at the same time ensuring that the distance between the front of the car and the vehicle in front of it does not exceed a certain value. To do so, we control the gas pedal and the acceleration that is given to the car. We also need to measure the current speed of the car and the distance with the vehicle in front.

The values that are measured are considered the output of the system $y(t)$. At every point in time, the output vector represents a snapshot of what is known about the system. The output of the system depends on the internal state of the system – for example, the car speed depends on the previous speed – and on the value of the control signal that we apply (the gas pedal). We denote with $u(t)$ the value of the control signal and with $x(t)$ the state of the system. The state vector includes all the variables that are needed to fully describe the system behaviour – whether we can measure them or not. For example, the system state may include the slope of the road

that the car is currently running onto, as this makes a difference in describing how much power is needed to reach a certain speed. A common definition for the state of a system is "an encoding of evertything about the past that has an effect on the system at present time."

The system state $x(t) = [x_1(t), \ldots, x_p(t)]^{\mathrm{T}}$ evolves depending on the current state and the input signal $u(t) = [u_1(t), \ldots, u_r(t)]^{\mathrm{T}}$, where the superscript $^{\mathrm{T}}$ indicates the transposition operator. We denote with p the number of state variables (i.e., the length of vector x) and with r the number of input variables (i.e., the length of vector u). The matrices A_c, B_c, C_c, and D_c encode the dynamics of the system. In the following, we will make two assumptions: D_c is a zero matrix of appropriate size, and C_c is the unit matrix of appropriate size.[2] The first assumption means that the system is *strictly proper* and holds for almost all the physical models used in control. The second assumption means that the state is measurable. This does not always hold for real systems, but state observers can be built whenever this is not true [16], to estimate the state $x(t)$.[3] This means that, without losing generality, we can represent $\mathcal{P}_c$ as

$$\dot{x}(t) = A_c\, x(t) + B_c\, u(t), \tag{2}$$

and describe the system dynamics using only A_c and B_c.

From this starting point, control systems are usually designed and realised in one of these two ways:

1. The plant model $\mathcal{P}_c$ is used to synthesise a controller (model) $\mathcal{C}_c$ in continuous-time. Closed-loop system properties, like stability, are proven on the feedback interconnection of $\mathcal{C}_c$ and $\mathcal{P}_c$. However, when the controller is implemented, digital hardware is used. This means that the controller model $\mathcal{C}_c$ has to be discretised, obtaining $\mathcal{C}_d$. $\mathcal{C}_d$ describes the behaviour of $\mathcal{C}_c$ at given sampling instants.

2. The model of the plant in continuous time $\mathcal{P}_c$ is discretised, obtaining a discrete-time plant model $\mathcal{P}_d$. $\mathcal{P}_d$ describes the behaviour of $\mathcal{P}_c$ at given sampling instants. A controller $\mathcal{C}_d$ is designed directly in the discrete-time framework, using the discrete-time plant model $\mathcal{P}_d$. Closed-loop properties are proven on the feedback interconnection of $\mathcal{C}_d$ and $\mathcal{P}_d$.

In both cases, when an object (being it the plant or the controller) is discretised, a sampling period π is chosen. With either design methods, we can obtain a discrete-time model of the plant $\mathcal{P}_d$ and of the controller $\mathcal{C}_d$. In control theory, usually it is possible to prove properties of the interconnection between these two models. In particular, we use $\mathcal{C}_d$ rather than $\mathcal{C}_c$ to prove properties using the controller that is closer to the real implementation. However, on top of what is done in classical control theory, we want to take into account deadline misses.

We discretise $\mathcal{P}_c$ from Equation (2). From the representation in terms of ordinary differential equations, we obtain the system of difference equations $\mathcal{P}_d$ as

$$\mathcal{P}_d : x_{[k+1]} = A_d\, x_{[k]} + B_d\, u_{[k]}. \tag{3}$$

[2] A_c is a $p \times p$ matrix, B_c is a $p \times r$ matrix, C_c is a $p \times p$ matrix, and D_c is a $p \times r$ matrix.

[3] As a remark, if a state observer is present its dynamics should be taken into account in the analysis. This extension only requires to augment the system state with the rows and columns corresponding to the execution of the observer, but the analysis method remains the same.

Here, k counts the sampling instants (i.e., there is a distance of π [s] between the k-th and the $k+1$-th instant). The matrices A_d and B_d are the counterparts of A_c and B_c for the continuous-time system. They describe the evolution of the system in discrete-time, have the same dimensions of the corresponding continuous-time matrices, and their elements depend on the choice of the sampling period π.

We also consider noise in our model, which is relevant for performance analysis (and not for stability). We define the plant with noise as

$$\mathcal{P}_d : x_{[k+1]} = A_d\, x_{[k]} + B_d\, u_{[k]} + B_w\, w_{[k]}. \tag{4}$$

The noise variable $w_{[k]}$ is subject to its specific matrix B_w, which is not the same as the matrix B_d used for the input $u_{[k]}$. This allows us to distinguish system that behave differently depending on the input. Generally speaking, the inputs that are included in $u_{[k]}$ are under our control, meaning that we can choose them to drive the system state where we want. On the contrary, the inputs that are included in $w_{[k]}$ are not under our control and may come from imprecise sensors or from disturbances (like a gist of wind disrupting an helicopter flight).

Controller Model Once a model of the plant is available, control design can be carried out with many different methods. In this paper we tackle periodic controllers expressed as state feedback controllers, i.e., controllers that execute periodically with period π and whose discrete-time form is

$$u_{[k]} = K_k\, x_{[k]} \tag{5}$$

or

$$\begin{aligned} z_{[k+1]} &= A_k\, z_{[k]} + B_k\, e_{[k]} \\ u_{[k]} &= C_k\, z_{[k]} + D_k\, e_{[k]} \end{aligned}. \tag{6}$$

The control design problem is the problem of finding the matrix K_k (or the matrices A_k, B_k, C_k and D_k) that stabilises the system and obtains some desired properties. The state feedback formulation K_k is quite general, although the linear system A_k, B_k, C_k and D_k generalises it. State feedback controllers are not purely proportional controllers, although their update is proportional to the state vector. It is possible to augment the state vector of the system – for example introducing an error term and its integral – to achieve controllers that are not simply proportional but contain integral action.[4] Additionally, it is possible to use pole placement [16], or to compute optimal controllers using the Linear Quadratic Regulator [15] formulation. To properly represent the lack of update of linear controllers in case of lack of execution (and handle digital implementations of PID controller), we also consider the linear controller formulation in our analysis.

[4]The most widely adopted controllers in industry are the Proportional and Integral (PI) or the Proportional, Integral and Derivative (PID) controllers. These controllers can be expressed in state-feedback form, by augmenting the system state $x(t)$ with the difference between the desired state values and the obtained ones, i.e., the error, and its integral, or sum, over time. There is a small difference between the controller expressed in state-feedback form and the controller expressed as a state-space system. In the first case, when the controller misses its deadline, the update function for the state is still executed (as it is now part of the system equation). It is however possible to generalise the findings in this paper to handle controllers in state-space form.

In an industrial setting,[5] many controllers are still designed assuming zero latency and instantaneous computation [41], i.e., assuming that it takes zero time to retrieve the sensor measurement from the plant, compute the control signal, and apply it. When the dynamics of the plant are slow and the controller is able to sample and measure signals at a reasonable speed, this assumption does not significantly affect the behaviour of the system. However, in most cases, basic properties like stability can be violated because of the computational delays that are introduced in the loop. The controller job that is activated at time t_a completes its execution at time t_c, where t_c is in the controller period, i.e., $t_c \in (t_a, t_a + \pi]$, introducing a computational delay $t_c - t_a$.

Due to this computational delay, in industry, it became common practice to design control systems following the Logical Execution Time (LET) paradigm and to synchronise input and output exactly to the period boundary. In this case, the control signal is computed within a control period and applied at the beginning of the next period. This enhances the predictability of the system, allows the processor to execute other tasks without affecting the control properties, and ensures a consistent behaviour.

In control terms, this means that the controller actuates its control signal computation with a one-step delay. Assuming that the cycle of sampling, computing, and actuating can be always terminated within a control period, this allows the designer to synthesise an optimal controller regardless of the time-varying components of the computational delay such as activation jitter, unpredictable interrupts, uncertain computation times [18]. The equation for the state feedback controller then becomes

$$\mathcal{C}_d : u_{[k]} = K\, x_{[k-1]}, \tag{7}$$

where K is the designed controller, or

$$\mathcal{C}_d : \begin{array}{rcl} z_{[k+1]} & = & A_k\, z_{[k]} + B_k\, e_{[k]} \\ u_{[k+1]} & = & C_k\, z_{[k]} + D_k\, e_{[k]} \end{array} \tag{8}$$

where A_k, B_k, C_k and D_k represent the linear controller evolution and $e_{[k]}$ represents the error term. With very few exceptions, the vast literature on control design assumes that the deadlines to compute control signals are always met. Recently, Linsenmayer and Allgöwer started to connect the theory of m-K real-time systems (i.e., the $\tau \vdash \overline{\binom{m}{[k]}}$ model) with control design [19], showing that it is in some cases possible to design a state feedback controller that is robust (i.e., guarantees stability) to deadline misses. In this paper we will connect the amount of possible consecutive deadline misses (i.e., the $\tau \vdash \overline{\langle n \rangle}$ model) to the analysis of stability as a control design property.

Feedback Interconnection Assume there are no deadline misses. In this case, we can plug the value of $u_{[k]}$ obtained from Equation (7) into the plant Equation (3), obtaining

$$x_{[k+1]} = A_d\, x_{[k]} + B_d\, K\, x_{[k-1]}. \tag{9}$$

[5] In fact, a survey published in 2001 by Honeywell [8] states that 97% of the existing industrial controllers are PI controllers and use no delay compensation. This does not mean that the control community has not developed solutions to properly address delays in the control design. It simply means that in many industrial settings the design is still simple and limited to considering the computation instantaneous.

To analyse the closed-loop system, we define a new state variable $\tilde{x}_{[k]} = [x_{[k]}^{\mathrm{T}}, x_{[k-1]}^{\mathrm{T}}]^{\mathrm{T}}$ (the superscript $^{\mathrm{T}}$ indicates the result of the transposition operator). We recall that p denotes the order of the system (i.e., the number of state variables in vector $x_{[k]}$). Using the new state variable $\tilde{x}_{[k]}$, Equation (9) can be rewritten as

$$\tilde{x}_{[k+1]} = \begin{bmatrix} x_{[k+1]} \\ x_{[k]} \end{bmatrix} = \underbrace{\begin{bmatrix} A_d & B_d\, K \\ I_p & 0_{p\times p} \end{bmatrix}}_{A} \begin{bmatrix} x_{[k]} \\ x_{[k-1]} \end{bmatrix} = A\, \tilde{x}_{[k]}, \tag{10}$$

where I_p and $0_{p\times p}$ are respectively the identity matrix and the zero matrix of size of the number of state variables p.

Stability A discrete-time linear time-invariant system is asymptotically stable if and only if all the eigenvalues of its state matrix are strictly inside the unit disk. For the system shown in Equation (10), this means that the eigenvalues of A should have magnitude strictly less than one.

Another way of formulating the stability requirement uses the concept of *spectral radius* $\rho(A)$. The spectral radius is defined as the maximum magnitude of the eigenvalues of A. If we denote with $\{\lambda_1, ..., \lambda_n\}$ the set of eigenvalues of A, this means

$$\rho(A) = \max\{|\lambda_1|, \ldots, |\lambda_n|\}. \tag{11}$$

Requiring that all the eigenvalues have magnitude strictly less than one is equivalent to stating that the spectral radius of the A matrix should be less than 1.

This only proves the stability of the system in absence of deadline misses. However, we are aware that sporadic misses can occur, either due to faults [12] or to the chosen period π not satisfying the requirement of worst-case response time for the controller task τ being less than the controller period [10, 27, 26].

3.2 Maximum fail time $R_{\max}$

We consider that an event causes deadline misses, either because the control signal is not correctly received by the actuator, or because the computation is not carried on, or because the sensor message is not received.

In order to properly analyse the closed-loop system properties when deadlines can be missed, it is necessary to define a model of how the system reacts to deadline misses. There are two aspects of this reaction: (i) what is the chosen control signal when a miss occurs [19], and (ii) how is the operating system treating the job that missed the deadline [26]. In the remainder of this section, suppose that in the k-th iteration the controller task τ did not complete its execution before the deadline, i.e., it does not complete its computation before time $(k+1)\,\pi$ [s]. We denote time $(k+1)\,\pi$ [s] with t_m.

Control Signal At time t_m, a control signal should be applied to the plant. Two alternatives have been identified for how to select the next control signal [19]: zero and hold.

1. *Zero:* The control signal $u_{[k+1]}$ is set to zero.
2. *Hold:* The control signal $u_{[k+1]}$ is unchanged, i.e., it is the previous value of the control signal $u_{[k]}$.

The choice of these two alternatives often depends on the control goal that should be achieved.

When a controller is designed for setpoint tracking (i.e., to ensure that the value of some physical quantity follows a desired profile – e.g., to have a robot follow a desired trajectory), the control signal is usually zero in case the measured physical quantity is equal to its setpoint. In this case, setting the control signal to zero means assuming that the model of the plant is correct and the computation does not need correction. When a controller is designed for disturbance rejection (i.e., to ensure that the effect of some physical disturbance is not visible in the measurements – e.g., to keep the altitude of a helicopter constant despite wind) then the control signal is usually a reflection of the effort needed to counteract the disturbance. In this case, holding the previous value of the control signal means making the assumption that the system is experiencing the same disturbance.

System-Level Action The second decision to make is the choice of what to do with the job that missed the deadline. In this case three alternatives have been proposed [26]: kill, skip-next, and queue(1).

1. *Kill:* At time t_m the job that missed the deadline is killed and a new job is activated.
2. *Skip-next:* At time t_m the job that missed its deadline is allowed to continue with the same scheduling parameters (e.g., priority or budget) and carries on in the next period. The job that should have been activated at the deadline missed is not activated, and the next activation is set to $t_m + \pi$.
3. *Queue(1):* At time t_m the job that missed its deadline is continued. A new job is activated with deadline $t_m+\pi$. The two jobs share the scheduling parameters during the period interval $[t_m, t_m + \pi]$. At time $t_m + \pi$, the most recent update of the control value is applied. If both jobs finish their computation, the control variable is set to the value produced by the most recently activated job (i.e., the job that started at time t_m and was placed in the queue until the old job that missed its previous deadline finished). If only the first job finishes the computation the control variable is set to the value of the job that finished and the following one is continued in the subsequent period.

We analyse the system in all possible configurations. However, we point out that, from an implementation perspective, killing the control job may not be feasible in many industrial settings. In fact, the system has reached an inconclusive intermediate state. The internal state of the controller could have been updated and the system should implement a clean rollback of these

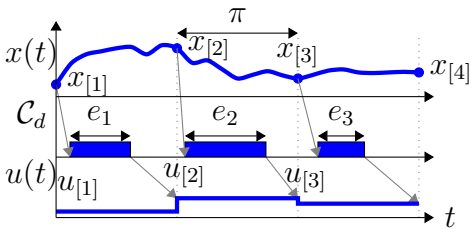


Figure 24: System evolution in case no deadline is missed. The state feedback controller $\mathcal{C}_d$ computes the control signal u based on measurements of the state x.

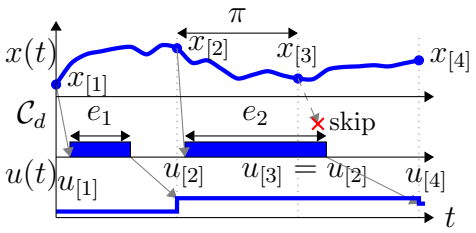


Figure 25: System evolution in case of a deadline miss with the hold policy and skip-next strategy. The controller misses the deadline and completes in the subsequent period.

changes. Implementing a clean rollback procedure is risky. Furthermore, if the lengthy computation (and subsequent deadline miss) is due to the received input values, it is likely that the next iteration will start from state values that are fairly close to the previous ones, with higher than normal risk of missing a deadline.

We also notice that enqueuing the task could be beneficial from the control perspective, because a computation with most recent measurements of the state variables could be applied. However, the scheduling parameters for τ have most likely been tuned for one single control job to be executed in a period. For example, if the control task is executed using reservation-based scheduling, its budget is selected to match one execution. When using fixed-priority scheduling, the controller priority has been selected. Executing a second control task may create ripple effects and have a disruptive effect on lower priority tasks.

Finally, if the deadline is missed, this means that the system is likely experiencing a transient overload state, which would make *skip-next* the best option to relieve some pressure from the system.

Analysis Fundamentals Here we present the general methodology that we apply to verify the stability of closed-loop systems with different strategies. We cast the problem into a switching-systems stability problem and show how real-world implementations behave.

Within one control period, there are two possible realisations. The controller job that was activated at time $k\,\pi$ can either hit or miss its deadline. Figure 24 shows the case in which no deadline is missed, while Figure 25 shows the behaviour of the system when a deadline miss occur with the *hold and skip-next* strategy. In the figures, we use e_i to indicate the execution time of the i-th job of the controller. The figure just provides a visual representation of a lengthy execution, but misses can occur due to other sources of interference, e.g., higher priority task being executed with a fixed-priority scheduling algorithm, interrupts being raised and served during the execution of the control task, or access to locked shared resources being requested. In Figure 25, the control signal $u_{[2]}$ is held as $u_{[3]}$. The next controller execution instance is skipped and the result of the completion of e_2 is applied as $u_{[4]}$.

The procedure that we follow to analyse the closed loop system is the following:

1. We express the dynamics of the closed-loop system in the cases of hit and of miss. Following a procedure similar to the one we used in Equation (10), we determine the state matrices for the closed-loop systems in case of deadline hit and deadline miss, respectively A_H and A_M. We then know that the system with (unconstrained) deadline misses can be expressed as a switching system [17] that arbitrarily switches between these two matrices. If the original system in Equation (3) was unstable, there is no hope that the switching system that arbitrarily switches between A_H and A_M is stable (as either an old or no control action is applied when a miss occurs). However, we still have not introduced any weakly hard constraint.

2. We determine the set of possible cases for the evolution of the system when $\tau \vdash \overline{\langle n \rangle}$ guarantees are provided, i.e., the possible realisations of the system behaviour. We denote with Σ the set of possible matrices that these realise. For $\tau \vdash \overline{\langle n \rangle}$ guarantees, the set of possible realisations is $\{H, MH, \ldots, M^n H\}$. The set contains either a single hit, or a certain number of misses (up to n) followed by a hit.[6] This means that $\Sigma = \{A_H, A_H A_M, A_H A_M^2, \ldots A_H A_M^n\}$. This can be written in a compact form as $\Sigma = \{A_H A_M^i \mid i \in \mathbb{Z}^{\geq}, i \leq n\}$ where $\mathbb{Z}^{\geq}$ indicates the set of integers including zero. Notice that matrices are multiplied from the right to the left (denoting the standard evolution of the system from a mathematical standpoint). This step introduces the weakly hard constraint for which we investigate the system stability.

3. We compute a generalisation of the spectral radius concept, called *joint spectral radius* $\rho(\Sigma)$ [32, 14], that allows us to assess the stability of the closed-loop system that switches between the realisations (i.e., the valid scenarios including a number of misses between 0 and n followed by a hit) included in Σ. More precisely, the closed-loop system that can switch between the realisations included in Σ is asymptotically stable *if and only if* $\rho(\Sigma) < 1$ [14, Theorem 1.2].

In order to generalise the spectral radius to a set of matrices, we introduce some notation. The following paragraphs are using the notation and sequential treatise proposed in [14] to introduce the concept of the joint spectral radius. We recap only what is needed for the purpose of understanding our analysis.

Joint Spectral Radius [32, 14] The first step for our definition is to determine what happens when some steps of evolution of the switching system occur. We then denote with $\rho_\mu(\Sigma)$ the spectral radius of the matrices that we find after μ-steps. Precisely,

$$\rho_\mu(\Sigma) = \sup\{\rho(A)^{\frac{1}{\mu}} : A \in \Sigma^\mu\}. \tag{12}$$

[6] The notation used to define Σ is slightly simplified here, as the matrix A_H may be different depending on how many deadlines have been missed (for example, with the skip-next strategy the controller uses an old measurement of the state to compute the control signal). We will be more precise in the following when we show how to apply the procedure to the different cases. Furthermore, notice that the matrices in Σ represent the evolution across a different number of time steps: A_H advances the time in the system of π, while $A_H A_M$ advances the system time of $2\,\pi$. This is not a concern for the system analysis.

In this definition we quantify the average growth over μ time steps, as the supremum of the spectral radius (elevated to the power $\frac{1}{\mu}$) of all the matrices that can be evolutions of the system after μ matrix multiplications (i.e., after μ evolution steps, where an evolution step is either a hit or a set of constrained misses followed by a hit). Equation (12) denotes the supremum of all the possible combinations of products of μ matrices that are included in Σ.

Using $\rho_\mu(\Sigma)$ we can define the joint spectral radius of a bounded set of matrices Σ as

$$\rho(\Sigma) = \limsup_{\mu \to \infty} \rho_\mu(\Sigma). \tag{13}$$

We are then looking at the evolution of the system for an infinite amount of time, i.e., pushing μ to the limit.

Determining that the switching system is asymptotically stable is equivalent to assessing that the joint spectral radius of the set of matrices Σ is less than 1. This condition is both sufficient and necessary [14, Theorem 1.2]. This means that if the joint spectral radius is higher than 1, there is at least a sequence of switches of hits and misses that destabilises the closed-loop system.

Joint Spectral Radius Computation On the practical side, the problem of computing if the joint spectral radius is less than 1 is undecidable [7]. In many cases it is possible to approximate the joint spectral radius with satisfactory precision [13, 6, 5, 24] and obtain upper and lower bounds for $\rho(\Sigma)$. Clearly, the closer the two bounds are, the more precise is the estimation of the true value of the joint spectral radius. We can safely say that our controller design is sufficiently robust to deadline misses if the upper bound on the joint spectral radius $\rho(\Sigma)$ is less than 1.

Joint Spectral Radius with at most n Consecutive Misses If the joint spectral radius of the set Σ is less than 1, the stability of all the combinations of realisations (of hits and misses, that include at most n consecutive misses) is proven, regardless of the window size.

For example, let us assume that we are analysing a system with the real-time guarantee that we cannot experience more than two consecutive misses. The realisations that we analyse are $\{A_H, A_H A_M, A_H A_M A_M\}$ and the joint spectral radius unfolds and checks all the possible (infinitely long) sequences of combinations of these realisations.

For a length of two, this means that we check: (1) $A_H A_H$ as the product of the first term twice, (2) $A_H A_H A_M$ as the product of the first two terms picking the first as final (in terms of time evolution of the system), (3) $A_H A_H A_M A_M$ as the product of the first and last terms picking the first as initial, (4) $A_H A_M A_H$ as the product of the first two terms picking the second as final, (5) $A_H A_M A_H A_M$ as the product of the second term twice, (6) $A_H A_M A_H A_M A_M$ as the product of the last two terms, picking the second as final, (7) $A_H A_M A_M A_H$ as the product of the last and first term, (8) $A_H A_M A_M A_H A_M$ as the product of the last and second term, (9) $A_H A_M A_M A_H A_M A_M$ as the last term twice. This procedure is repeated for more products, up to *infinitely long* sequences. From the computation side, the results are an upper and a lower bound on the value of the (true) joint spectral radius.

The analysis is sound on the control side, as stability is guaranteed if and only if the joint spectral radius is less than 1. On the theoretical side, this demonstrates that the $\tau \vdash \overline{\langle n \rangle}$ model

can elegantly provide a necessary condition for the stability of the system. The only if part means that there is at least a sequence of hits and misses (where at most we experience n consecutive misses) that causes the system to be unstable if the (true value of the) joint spectral radius is larger than 1.

Considering that we only compute an upper and lower bound on the joint spectral radius, what we can conclude is: if the lower bound that we obtain is above 1, we are entirely certain that such a sequence exists, while if the lower bound is below 1 and the upper bound is above 1 we have no mathematical certainty that the system is unstable. Nonetheless, on the practical side, the bounds obtained with modern approximation techniques [13, 6, 5, 24] are usually very close to one another, implying that they are a very good estimate of the true value of the joint spectral radius.

It is important to note that even if the control system is able to stabilise the system in the presence of n consecutive misses, this does not mean that executing the controller with a period of $n\,\pi$, rather than its original period π, is a sensible choice. In fact, the performance (measured for example using the integral of the squared error) of the controller that is executing with a larger period would be dramatically worse than the performance of the controller with the shorter period that can experience misses. Being able to tolerate misses is very different than performing well when these misses occur.

We presented the analysis of these systems in the paper [21] published at the Euromicro Conference on Real-Time Systems (ECRTS) 2020. In our analysis, we discover that depending on the physical characteristics of our process under control, a different number of deadlines can be missed without harming the stability of the system and we gave a quantification of the number, depending on the strategy used to handle the miss. We are then able to find $R_{\max} = n\,\pi$ where n is the maximum number of deadline misses that do not harm stability given the handling strategy that is defined.

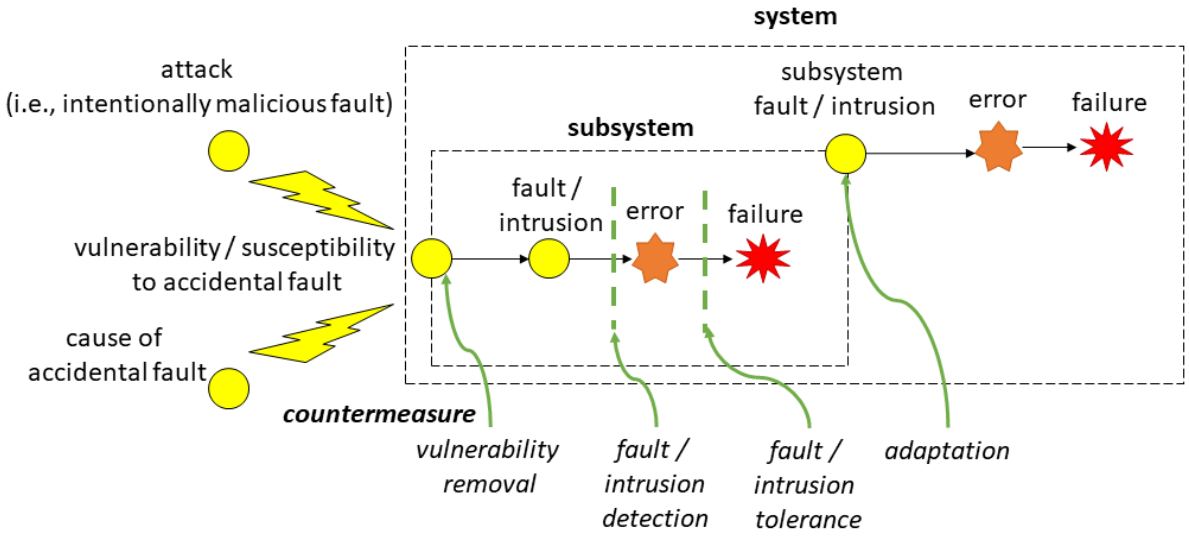


Figure 26: AVI model (see D.5.1 for further details): Systems of a certain complexity must be considered to contain undetected vulnerabilities. Accidental faults but also adversaries exploit vulnerabilities that have not been removed to intrude the system, where they may cause errors and ultimately the failure of a subsystem if the fault goes undetected and if it cannot be tolerated for extended periods of time. Adaptation improves the system's ability to tolerate accidental faults and partial compromise by evading the attack and by replacing failed components with healthy counterparts.

4 Task 1.4: Specification of Fault Model and Threat Indicators

For the DSL TeamPlay to anticipate faults and attacks, domain experts must be able to specify the fault categories (accidental and malicious) that the system should tolerate and the indicators that define the current threat level that the system is exposed to. Adaptation then happens as indicated in Section 2.1.4 by selecting one of the multiple versions that has been embedded to the system at design time. The role of threat indicators is therefore to identify the subset of embedded versions of the individual components that are capable of withstanding the perceived threats.

As highlighted in Deliverable D.5.1, we shall follow the fault taxonomy of Avizienis et al. [2] and build upon and extend the AVI model [9]: Attacks and accidental faults are treated under the same body of knowledge with the exception that statistical information about occurrence can only be used for the latter. It remains completely to the discretion of an adversary, when he triggers a fault of the former kind and his ability to do so is only limited by his skill, power and intention. Both types of faults affect the system through reachable vulnerabilities and we shall assume that several vulnerabilities go undetected in any system of reasonable complexity, even if the system has been tested extensively and even if it has undergone formal or semi-formal assurance processes. Our goal is to achieve dependability by adapting systems to compensate failures and to remove the

vulnerabilities that lead to those. However, to not overpay for dependability, we aim to adjust the system to withstand the current perceived threats, while optimizing quality of service. This way, when the system perceives its risk of failure and compromise increases, it will gracefully degrade performance to compensate for the changed risk situation.

4.1 Dependability

Avizienis et al. [2] defines dependability as "the ability to deliver service that can justifiably be trusted". However, there are many more possible definitions of depedability. For example, an alternative definition whould characterize dependability a "the ability of a system to avoid service failures that are more frequent or more severe than acceptable."

In our opinion, dependability can be better explained by defining its attributes, threats and means as shown in Figure 27.

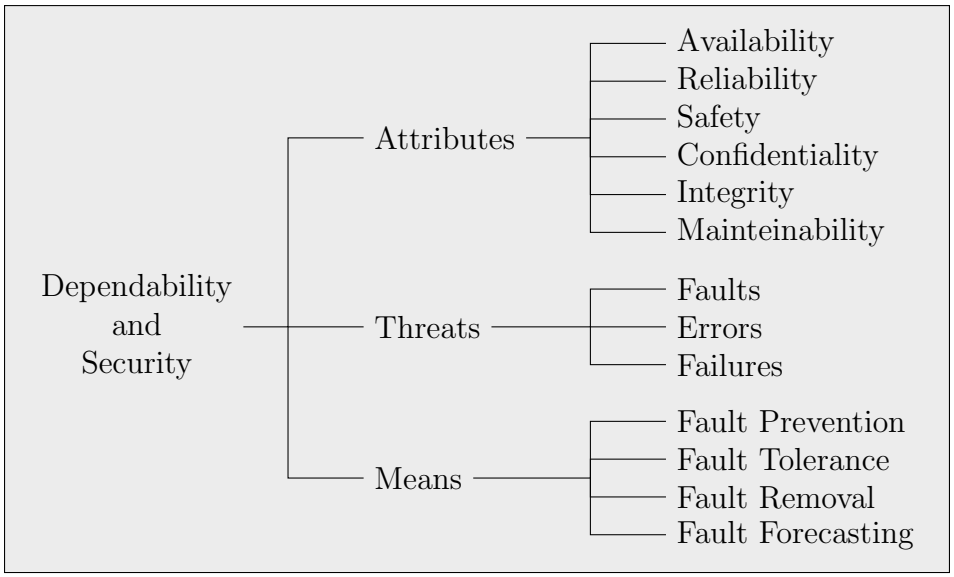


Figure 27: Definition of Dependability

1. **Dependability Attributes**

   (a) *Availability* is the readiness for correct service.

   (b) *Reliability* is the continuity of correct service.

   (c) *Safety* is the absence of catastrophic consequences on the user(s) and the environment.

   (d) *Confidentiality* is the the absence of unauthorized disclosure of information.

   (e) *Integrity* is the ability to detect improper system alterations.

   (f) *Maintainability* is the ability to undergo modifications and repairs.

2. **Dependability Threats**

   (a) *Vulnerability* is a condition that may lead to a *Fault* (e.g., insufficient shielding against radiation, which may lead to a bitflip if an α-particle hits the memory, a buffer, which may overflow if an adversary triggers this software bug).

   (b) *Fault* is an event (or system state) that has unintentionally occured, caused by external and/or uncontrollable factors (such as radiation induced bit flips for which only the probability of occurrance is known, largely unpreditable subsystem failures, or attacker induced state changes), which *may* manifest to an error.

   (c) *Error* is the deviation from the correct behaviour. An error is the part of the total state of the system that may lead to its subsequent service failure. They might not reach an external state so they are called dormant errors.

   (d) *Failure* occurs when the delivered service deviates from correct service. This means, an external state deviates from the nominal behaviour. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function. A service failure is a transition from correct service to incorrect service. The period of delivery of incorrect service is a service outage. The transition from incorrect service to correct service is a service restoration. The deviation from correct service may assume different forms that are called service failure modes and are ranked according to failure severities.

3. **Dependability Means**

   (a) *Fault Prevention* is the means to prevent the occurrence or introduction of faults

   (b) *Fault/Intrusion Detection* is a mean to identify faults that are happening

   (c) *Fault Forecasting* is the means to estimate the present number, the future incidence, and the likely consequences of faults.

   (d) *Fault Removal* is the means to reduce the number and severity of faults.

   (e) *Fault Tolerance* is the means to avoid service failures in the presence of faults or subsystem failure.

   (f) *Fault Masking* ensures continued service delivery (i.e., without interruption) despite components or subsystems failing. As such, masking implies tolerance.

4.2 Fault Model and Threat Indicators

In the following we detail the general fault model we follow in Admorph and the threat indicators that trigger adaptations to increase the resilience of the system.

We assume systems operate through phases where they are exposed to threats of different severity. This includes environments which differ in the statistics how frequent accidental faults happen. A prominent example of this kind are the different radiation situations experienced by

an airplane while it is flying versus when it is on ground at the airport. Phases also differ in the presence and strength of adversaries attacking the system. In the following, we detail the assumptions of our fault model as far as they concern this environment and the assumptions and threat indicators that affect internal components, after adversaries have been partially successful in compromising some of the system's components.

4.2.1 Environmental Threats

We assume individual systems are composed of interacting components, as described in Section 2. Systems act in their environment. Components with no input channel constitute input components. They sense the environment and are thus susceptible to receiving faulty inputs from this environment. Conversely, components with no output channel constitute output components. Since output components typically act on the environment, they as well may be susceptible to faults triggered by the environment.

For example, rogue packets on a wireless communication medium or physical attacks to the sensors of an individual cyber-physical system may lead to a fault in the input component, which interprets this packet or sensor value. A physically blocked rotor is an example of an environment fault that may jam an actuator and that in turn might trigger a fault in the actuating component.

Threat indicators for environment-triggered faults include reading implausible sensor values, receiving improperly signed packets, unexpectedly high network traffic, but also the CPS entering a zone of known high likelihood to experience accidental or intentionally malicious faults (e.g., an area with high radiation or a hostile area).

We take the safety criticality of the Admoprh use cases as justification to not exclude the general possibility of advanced and persistent threats (APTs). We therefore consider rare failure situations and prepare the system to withstand also worst case combinations of threats to the degree that this tolerance can be justified given the resource budget available to system components.

Naturally, systems normally do not face such worst-case combinations. Instead, it is much more likely that no component fails or that components fail only individually and by accident. Our fault model is therefore not tied to define a single threshold f over the number of failing components that can be tolerated, but rather adjusts this threshold according to the component considered and to the level of threat, the system thinks it is exposed to.

For input/output components, detecting the presence of an adversary in the environment justifies increasing the perceived threat level, whereas having applied a new patch to fix a known vulnerability supports lowering the perceived threat level, since adversaries are stripped from this attack possibility.

4.2.2 Internal Threats

However, threat levels apply equally well to internal components. In fact, they allow us to characterize the partial success adversaries might already have when we see indicators for a possibly successful compromise. Indicators of this sort include wrong outputs from an internal component, no or delayed reaction to inputs, the communication of corrupt, stale or otherwise suspicious data

over internal channels, and gained knowledge about detected but not yet fixed vulnerabilities. Since the systems Admorph investigates are cyber-physical systems and as such real-time systems, threat indicators are not only limited to the value domain, but may also concern the timing when information is provided. Moreover, because of the CPS nature of the Admorph use cases, controllers form the last line of defense against physical damage to the environment or to the humans that operate in close proximity to the CPS. Once the aversary compromises these controllers or once he overcomes the internal resilience of the controlled system (see Sec. 3), the adversary can reach out to the physical world and cause harm that no adaptation can prevent. We therefore set out to prevent such software side compromise, leaving physical attacks to the actuators the only threat that requires different mitigation strategies.

Adaptation gives rise to two general ways of defending against such attacks: by evading critical components from the attack paths through which adversaries seek to reach their ultimate targets — typically the actuators if adversaries intend to cause harm — or by reinstantiating components along this part to replace a possibly compromised version with a fresh instance that the adversary still has to compromise to advance in his attack. Of course, pracical defenses apply a combination of the two.

For the purpose of this project, we shall assume the real-time operating system (PikeOS) and the adaptation component to not be compromised by adversaries. Note that this does not constraint the adversary to spare these components, but we will make no guarantees once adversaries are successful in compromising these components. The design and implementation of fault and intrusion tolerant real-time operating systems is out of scope of this project.

4.3 Threat Levels and Adversarial Power

To operate in the above fault model, one must be able to characterize adversarial power and to put this strength in relation to the countermeasures we foresee through adaptation. We do so, by investigating components c_i individually and by anticipating a time T_A^i that reflects the best case time after which an adversary with his resources has compromised c_i. For interface components, we can then associate the presence of adversaries of a certain strength with a threat-level $tl \in TL$ and act to an increasing or decreasing tl by adapting the components to an embedded component that can withstand attacks at tl and that performs better than component instances, which could withstand the threats of higher levels. Similarly, we can adapt the path by which components can be reached.

For most situations, a totally ordered set TL will be sufficient (e.g., distinct numbers $1 - 9$ to match the security levels in Figure 14). However, in general, a lattice suggests itself to be able to express different aspects of adversarial strengths (e.g., the ability to compromise cryptographic operations vs. the possibility to tamper with the sensors) in one level and to quantify as least upper bound when multiple of these aspects come together.

Once the adversary is successful in compromising components, the adversarial strength increases, since the adversary now has additional resources, provided by the system to the components he has compromised. At the same time, attacking internal components, requires a foothold in the system from where it may attack other components. For this direct aspect of his attack the

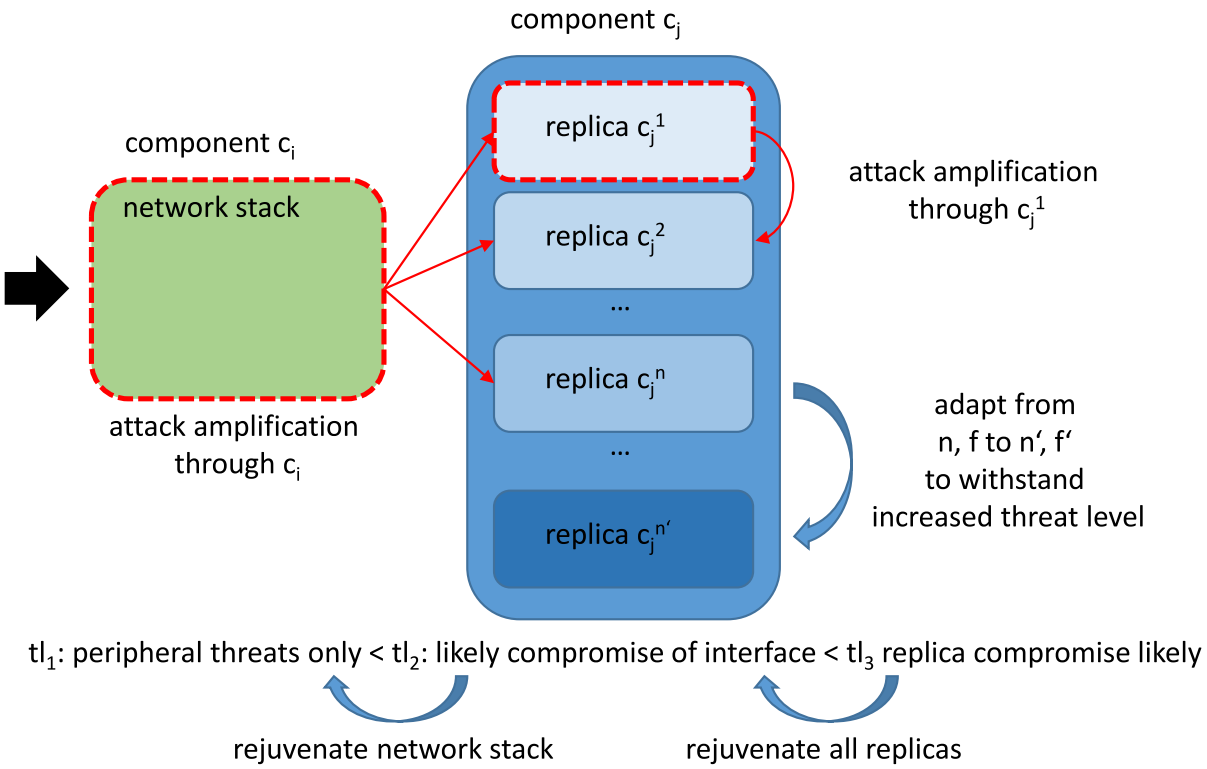


Figure 28: Threat levels and their implication on assumed accidental fault likelihoods/attack strengths.

adversary is therefore constrained to the resources he can muster inside the system and these are the resources associated with the components he has already compromised.

So far, we have no final notion to express this reduction of adversarial strength when his attack is limited to system-local resources. For now, we will work with abstract force factors F_A^i (initially $F_A^i = 1$) that are applied to T_A^j to reflect how long this type of adversary takes to compromise component c_j, reachable from c_i under the assumption that he has already compromised c_i.

Rejuvenation [34] with a novel diverse instance [30] resets the component to a state at least as secure as initially. Rejuvenating c_i faster than T_A^i outpaces adversaries in compromising components [4] to sustain security and safety for the lifetimes we expect from cyber-physical systems and systems of them [25].

4.3.1 Example

Let us exemplify T_A^j and F_A^i on the example of the replicated subsystem, shown in Figure 28. To compromise a component c_j, which is composed of n replicas, reaching agreement through a Byzantine Fault Tolerant State-Machine Replication (BFT-SMR) protocol, the adversary must either find a flaw in the BFT-SMR protocol or it must compromise more than f replicas. f is here the number of faults that the protocol can tolerate with n replicas. Admorph will research

protocols that allow the parameters n and f to be adapted to match the perceived threat level.

To reach replicated component c_j, the adversary may have to go through a network stack, which we denote as component c_i. Therefore, the time T_A^j to compromise more than f replicas of c_j depends on the speed at which the adversary may apply attacks through c_i. F_A^i reduces the adversaries power from arbitrary many resources he may muster outside the system to the resources the system has allocated to component c_i. Of course, having compromised some of the replicas c_j^k of component c_j, gives the adversary additional resources he may muster for continuing his attack. We reflect this by increasing the threat-level of c_j to reflect the number of suspected replicas. This threat-level increase should then trigger adaptation (e.g., by increasing n and f to compensate the higher risk of failure). Rejuvenating replicas ceases adversarial control over the resources they provide.

4.4 Next steps

With a general characterization of threat levels in place, the next steps are to develop a fault-tree analysis (FTA) that is capable of anticipating partial adversarial success to derive the risk of faults under the condition of compromises. Such an FTA will then serve to quantify the residual safety of use-case scenarios after individual components have fallen in the hands of an attacker, and will give guidance how the system can and should adapt to defend such an ongoing attack and to ultimately throw out the adversary by changing components along its attack paths.

5 Conclusion

The previous three sections reported on the work accomplished by the various partners in the context of work package 1: Specification of Adaptive Systems. Quite a lot has already been achieved by the individual partners during this initial reporting period. While the three strands of work show clear relations and contact points, we see potential to strengthen the collaborative aspects of our work.

To this effect the Covid-19 pandemic and the corresponding lock-downs across Europe only shortly after the Admorph kick-off meeting have had a negative effect on collaboration in the work package. Planned physical meetings had to be cancelled or already given up in the planning phase. New staff for the project was difficult to find and once found difficult to train during the lock-down period that with exceptions and regional differences persists until today. Last not least, in particular academic staff has been overstrained to keep education and student services alive and running during these extraordinary times.

With the general Covid-19 situation slowly stabilising we see the compilation of this deliverable as a good starting point for the more structured exchange of ideas and solutions across consortium partners. In this sense we aim at strengthening the collaborative aspects of the work package throughout the coming reporting period.

6 References

[1] Farhad Arbab. Composition of interacting computations. In Dina Goldin, Scott Smolka, and Peter Wegner, editors, *Interactive Computation*, pages 277–321. Springer, 2006.

[2] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[3] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[4] A. Bessani, H. Reise, P. Sousa, I. Gashi, V. Stankovic, T. Distler, R. Kapitza, A. Daidone, and R. Obelheiro. Forever: Fault/intrusion removal trhough evolution and recovery. In *ACM/IFIP/USENIX Middleware*, pages 99–101, December 2008.

[5] V. Blondel and Y. Nesterov. Computationally efficient approximations of the joint spectral radius. *SIAM Journal on Matrix Analysis and Applications*, 27(1):256–272, 2005.

[6] Vincent Blondel, Yurii Nesterov, and Jacques Theys. On the accuracy of the ellipsoid norm approximation of the joint spectral radius. *Linear Algebra and its Applications*, 394:91–107, 2005.

[7] Vincent Blondel and John N. Tsitsiklis. The boundedness of all products of a pair of matrices is undecidable. *Systems & Control Letters*, 41(2):135–140, 2000.

[8] Lane Desborough. Increasing customer value of industrial control performance monitoring-honeywell's experience. *Preprints of CPC*, pages 153–186, 2001.

[9] Giovanna Dondossola, Geert Deconinck, Felicita Giandomenico, Susanna Donatelli, Mohamed Kaaniche, and Paulo Veríssimo. Critical utility infrastructural resilience. 11 2012.

[10] G. Frehse, A. Hamann, S. Quinton, and M. Woehrle. Formal analysis of timing effects on closed-loop properties of control software. In *2014 IEEE Real-Time Systems Symposium*, pages 53–62, December 2014.

[11] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[12] Saurav Kumar Ghosh, Soumyajit Dey, Dip Goswami, Daniel Mueller-Gritschneder, and Samarjit Chakraborty. Design and validation of fault-tolerant embedded controllers. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1283–1288, 2018.

[13] N. Guglielmi, F. Wirth, and M. Zennaro. Complex polytope extremality results for families of matrices. *SIAM Journal on Matrix Analysis and Applications*, 27(3):721–743, 2005.

[14] R. Jungers. *The Joint Spectral Radius: Theory and Applications*. Lecture Notes in Control and Information Sciences. Springer Berlin Heidelberg, 2009.

[15] Huibert Kwakernaak. *Linear Optimal Control Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1972.

[16] W.S. Levine. *The Control Handbook*. Electrical Engineering Handbook. Taylor & Francis, 1996.

[17] D. Liberzon. *Switching in Systems and Control*. Systems & Control: Foundations & Applications. Birkhäuser Boston, 2003.

[18] B. Lincoln and A. Cervin. JITTERBUG: a tool for analysis of real-time control performance. In *41st IEEE Conference on Decision and Control*, volume 2, pages 1319–1324, December 2002.

[19] S. Linsenmayer and F. Allgower. Stabilization of networked control systems with weakly hard real-time dropout description. In *IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 4765–4770, December 2017.

[20] R. E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

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

[22] David Nicholson. 153821-nwr-rep-ese-000002 digital railway – system of systems (sos) system definition.

[23] Manuel Oriol, Thomas Gamer, Thijmen de Gooijer, Michael Wahler, and Ettore Ferranti. Fault-tolerant fault tolerance for component-based automation systems. In *Proceedings of the 4th international ACM Sigsoft symposium on Architecting critical systems*, pages 49–58, 2013.

[24] Pablo A. Parrilo and Ali Jadbabaie. Approximation of the joint spectral radius using sum of squares. *Linear Algebra and its Applications*, 428(10):2385–2402, 2008.

[25] A. Paverd, M. Völp, F. Brasser, M. Schunter, N.Asokan, A. Sadeghi, P. Verissimo, A. Steininger, and T. Holz. Sustainable security and safety: Challenges and opportunities. Technical report, ICRI CARS, 2019.

[26] Paolo Pazzaglia, Claudio Mandrioli, Martina Maggio, and Anton Cervin. DMAC: Deadline-Miss-Aware Control. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:24, 2019.

[27] Paolo Pazzaglia, Luigi Pannocchi, Alessandro Biondi, and Marco Di Natale. Beyond the Weakly Hard Model: Measuring the Performance Cost of Deadline Misses. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:22, 2018.

[28] Z. Peng. Building reliable embedded systems with unreliable components. In *ICSES 2010 International Conference on Signals and Electronic Circuits*, pages 9–13, 2010.

[29] Stefan Poledna. *Fault-tolerant real-time systems: The problem of replica determinism*, volume 345. Springer Science & Business Media, 2007.

[30] R. Pucella and F. B. Schneider. Independence from obfuscation: A semantic framework for diversity. In *19th IEEE Work. on Computer Security Foundations*, pages 230–241, 2006.

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

[32] Gian–Carlo Rota and W. Gilbert Strang. A note on the joint spectral radius. *Indagationes Mathematicae*, 63:379–381, 1960.

[33] Cosmin Rusu, Rami Melhem, and Daniel Mossé. Multi-version scheduling in rechargeable energy-aware real-time systems. *Journal of Embedded Computing*, 1(2):271–283, 2005.

[34] Paulo Sousa, Nuno Ferreira Neves, and Paulo Verissimo. Proactive resilience through architectural hybridization. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 686–690. ACM, 2006.

[35] Nawrin Sultana. *Toward a Transparent, Checkpointable Fault-Tolerant Message Passing Interface for HPC Systems*. PhD thesis, Auburn University, USA, 2019.

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

[37] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 365–376. IEEE, 2010.

[38] `https://www.cpsos.eu` (main author Sebastian Engell). Cyber-physical systems of systems - definition and core research and innovation areas. `https://www.cpsos.eu/wp-content/uploads/2015/07/CPSoS-Scope-paper-vOct-26-2014.pdf`. Online; accessed 26 May 2020; included for definition of CPS(oS).

[39] wikipedia. Domain specific language. `https://en.wikipedia.org/wiki/Domain-specific_language`. Online; accessed 25 May 2020.

[40] wikipedia. Technology readyness level. `https://en.wikipedia.org/wiki/Technology_readiness_level`. Online; accessed 25 May 2020.

[41] Dirk Ziegenbein and Arne Hamann. Timing-aware control software design for automotive systems. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 56:1–56:6, 2015.

A Appendices

A.1 TeamPlay core language

App ⇒ **app** *Id* { *AppBody* }

AppBody ⇒ **deadline** *FrequencyConst*
period *FrequencyConst*
datatypes { [*(Type, StringConst)*]* }
components { [*Component*]+ }
edges { [*Edge*]* }

Type ⇒ *Id*

Settings ⇒ **period** *FrequencyConst*
| **deadline** *FrequencyConst*
| **arch** *StringConst*
| **security** *IntConst*
| **cname** *StringConst*

Component ⇒ *Id* {
[**inports** *PortList*]
[**outports** *PortList*]
[**state** *PortList*]
[*Settings*]*
[*Version*]* }

Version ⇒ **version** *Id* { *Settings* }

PortList ⇒ [[(*Id* , [*IntConst* ,] *Type*)]+]

Edge ⇒ *SimpleEdge* | *BroadcastEdge*

SimpleEdge ⇒ *OutPort* -> *InPort*

BroadcastEdge ⇒ *OutPort* -> *InPort* [& *InPort*]+

InPort ⇒ *Id* [. *Id*]

OutPort ⇒ *Id* [. *Id*]

A.2 TeamPlay language after Admorph extension

App ⇒ **app** *Id* { *AppBody* }

AppBody ⇒ **deadline** *FrequencyConst*
period *FrequencyConst*
datatypes { [*(Type, StringConst)*]* }
[**profiles** { [*ProfileDef*]+ }]
[**templates** { [*Templs*]+ }]
components { [*Components*]+ }
Edges

Type ⇒ *Id*

ProfileDef ⇒ *Id* { [*Settings*]* [*FTSettings*]? }

Settings ⇒ [**vital**] **period** (*FrequencyConst* | *Id*)
| [**vital**] **deadline** (*FrequencyConst* | *Id*)
| [**vital**] **arch** (*StringConst* | *Id*)
| [**vital**] **security** (*IntConst* | *Id*)
| **cname** (*StringConst* | *Id*)
| **remove** (*removeSetting* | *Id*)

FTSettings ⇒ **nModular** { [[**vital**] [*NModularSetting* | *ReplicaSetting*]]* }
| **standby** { [[**vital**] *ReplicaSetting*]* }
| **nVersion** { [[**vital**] [*NVersionSetting* | *NModularSetting*]]* }
| **checkpoint** { }

ReplicaSetting ⇒ **replicas** [*IntConst* | *Id*]

NModularSetting⇒
| **votingReplicas** [*IntConst* | *Id*]
| **waitingTime** [*PercentConst* | *Id*]
| **waitingStart** [**single** | *Id* | **majority**]
| **waitingJoin** [*BoolConst* | *Id*]

NVersionSetting ⇒ **versions** [[*(Id, IntConst)*]+]

RemoveSetting ⇒ **nModular** | **standby** | **nVersion** | **checkpoint**
| **deadline** | **period** | **cname**

Components ⇒ *Component* | *SubNetwork* | *Instantiation*

Templs ⇒ *Component* | *Template*

Component ⇒ *Id* {
[**inports** *PortList*]
[**outports** *PortList*]
[**state** *PortList*]
[*Profiles*]
[*Settings*]*
[*Versions*]* }

Profiles ⇒ **profiles** [[*Id*]+]

Edges ⇒ **edges** { [*Edge*]* }

Versions ⇒ **version** *Id* { *Settings* }

Template ⇒ *Id* [(*Id*[, *Id*]*)] {
[**inports** *PortList*]
[**outports** *PortList*]
[**state** *PortList*]
[*Profiles*]
[*Settings*]*
[*Templs* | *Instantiation*]+
[*Versions*]*
Edges }

SubNetwork ⇒ *Id* {
[**inports** *PortList*]
[**outports** *PortList*]
[**state** *PortList*]
[*Profiles*]
[*Settings*]*
[*Component* | *Instantiation*]*
Edges }

Instantiation ⇒ *Id* [(*Arg* [, *Arg*]*)] *Id*

Arg ⇒ _ | *StringConst* | *IntConst* | *Id* | *PercentConst* | *FrequencyConst*

PortList ⇒ [[(*Id*, [*IntConst* ,] *Type*)]+]

Edge ⇒ *SimpleEdge* | *BroadcastEdge*

InPort ⇒ *Id*[. *Id*] | **in** [. *Id*]

OutPort ⇒ *Id*[. *Id*] | **out** [. *Id*]

SimpleEdge ⇒ *OutPort* -> *InPort*

BroadcastEdge ⇒ *OutPort* -> *InPort* [& *InPort*]+