



D2.3: Final Report on Adaptation Methods

Project acronym: ADMORPH

Project full title: Towards Adaptively Morphing Embedded Systems

Grant agreement no.: 871259

Due Date:	Month 33
Delivery:	Month 33
Lead Partner:	UniLu
Editor:	Federico Lucchetti, UniLu, Marcus Völz, UniLu
Dissemination Level:	Public (P)
Status:	final
Approved:	
Version:	1.2



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

This deliverable reflects only the author's view and the European Commission is not responsible for any use that may be made of the information it contains.

DOCUMENT INFO – Revision History

Date and version number	Author	Comments
21/05/2022 ver. 1.0	Marcus Voelp	First draft

List of Contributors

Date and version number	Beneficiary	Comments
24/01/2023 ver. 1.0	Federico Lucchetti	Initial report structure
16/02/2023 ver. 1.1	Marcus Voelp	First iteration
03/03/2023 ver. 1.2	Lukas Miedema and Marcus Voelp	Second iteration

GLOSSARY

BFT-SMR Byzantine Fault Tolerant Statemachine Replication

CPS(oS) Cyber Physical System (of Systems)

FIT Fault and Intrusion Tolerance

IPC Inter-process communication

QoS Quality of Service

SoS System of Systems

Contents

Executive summary	4
1 Adaptation as Key Enabler of Individualistic and CPSoS-wide Resilience	5
2 State-of-the-Art since D2.2 (month 33)	6
3 Adaptation to Achieve Control-Aware Fault and Intrusion Tolerance	7
4 Adaptation to Recover and Make Resilient Control Tasks	9
5 Bounding the Time of Adaptation and Reconfiguration	10
6 Dependable network monitoring and support for adaptation	12
6.1 Datasets for network monitoring	12
6.2 Exploring Diversity through Multi-Views	14
6.3 Comprehensive Evaluation Platform	15
6.4 Open Aspects	16
7 Interfacing with the TeamPlay Coordination Language Compiler Infrastructure	17
7.1 Exchange Format	17
7.2 Runtime Configuration changes	18
8 Testing Runtime Systems	20
9 Integration	31
10 Conclusions	32
11 References	32

Executive summary

Work package WP2 sets out to develop the adaptation building blocks necessary for maintaining or, in extreme situations, gracefully degrading the systems' quality of service guarantees. The focus is on methods, protocols, tools and techniques, to increase the resilience of controllers of Cyber-Physical-Systems of Systems (CPSoS), to optimize the mapping, partitioning and scheduling of system components, to automate the design transformation towards a reliable, resource and physical requirement aware system, and to analyze and limit system reconfiguration times.

This deliverable D2.3 reports the adaptation methods and their integration with the coordination language and runtime system, involving Task 2.1 - 2.6. This deliverable builds upon and extends deliverable D2.2.

Adaptation serves four main purposes:

1. to evade faults, including accidental ones and adversaries in their ongoing attacks;
2. to improve the resilience of systems after experiencing faults or during ongoing attacks, possibly by degrading the systems' quality of service, if necessary;
3. to return the system to a state that is considered safe and secure; and
4. to optimize the system whenever the perceived threat level drops (e.g., because the CPS entered less harsh environments or because adversaries lost interest).

We conclude our achievements in adapting individualistic CPS and CPSoS-wide resilience (Section 1), iterate on the SOTA since the last deliverable (in Section 2), look at adaptation to achieve control-aware fault and intrusion tolerance (in Section 3) and to recover control tasks and make them resilient (in Section 4). We report on our work to bound adaptation and reconfiguration times (in Section 5), including how we deal with situations where reconfiguration cannot be bounded. We report on challenges of integrating low-level control into the ADMORPH exchange format and the coordination language compiler infrastructure (in Section 7) as well as how to test control systems with a feedback loop for adaptation (in Section 8). We report our work on the integration of the adaptation methods with the coordination language and runtime systems in Section 9. Section 10 concludes.

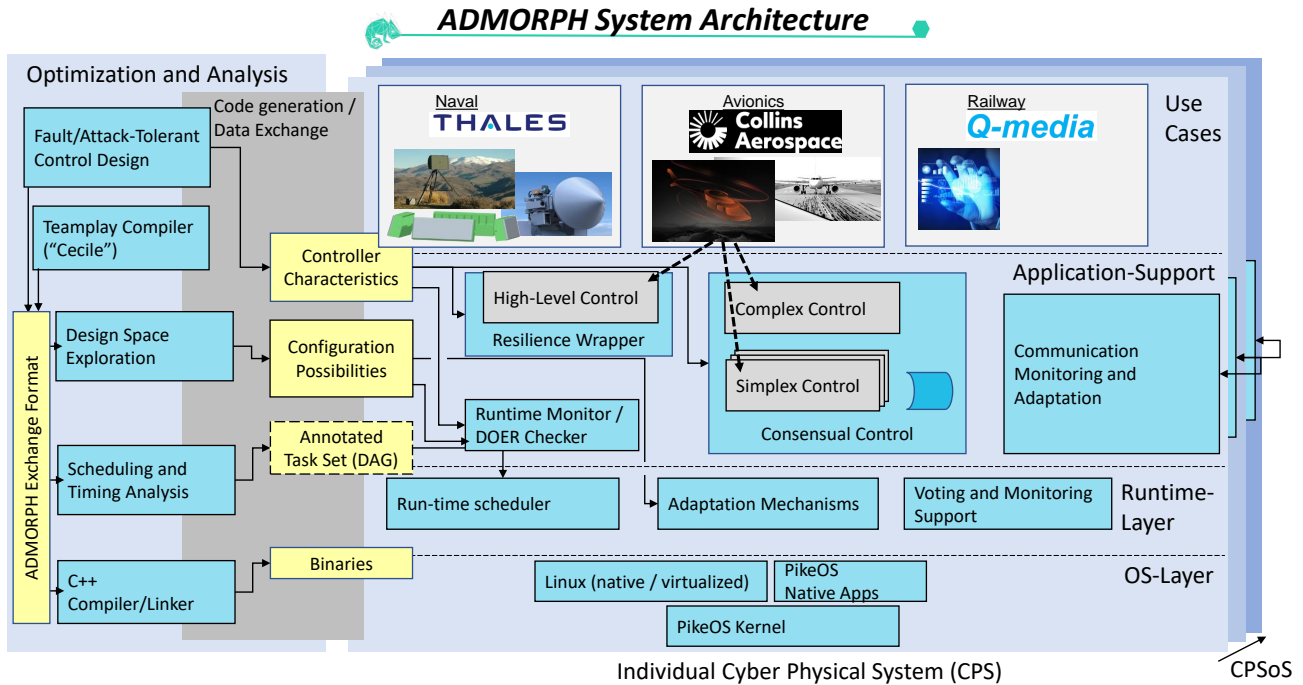


Figure 1: ADMORPH architecture: The figure shows the main components of the ADMORPH architecture and their interplay in relation to the three use cases. Shown is the tool support (left), the runtime components (right), and the interaction with other CPS of the CPSoS (back).

1 Adaptation as Key Enabler of Individualistic and CPSoS-wide Resilience

In ADMORPH and WP2 specifically, we take a holistic view on CPSoS and its components. We assume fault and threat models of incremental and possibly changing strength. This can be in terms of accidental faults or intentionally malicious faults, such as targeted attacks mounted by adversaries. Accidental faults are induced by the environment and change when the CPSoS changes where it is operating. Intentionally malicious faults change based on their intent, access to attack tools and skills of the team (see D2.1b for a more detailed discussion of fault and threat models).

Adaptation is the key to cope with faults, on the long run, provided the system can tolerate faults long enough for adaptation to become effective. In addition, adaptation will serve optimizing functional and non-functional properties. In this WP and Deliverable, we focus on adaptation in relation to faults and threats. As mentioned above, adaptation serves four purposes in this context:

1. to evade faults by relocating services to a different set of resources;
2. to improve resilience by including more resources in the tolerance of faults;

3. to rejuvenate the system by recovering faulty or compromised resources when possible; and
4. to match the systems' resilience to the perceived threat by allocating more or less resources to the defense.

We will report adaptation methods and their interplay by zooming into the runtime elements of the ADMORPH architecture, illustrated in Figure 1. ADMORPH supports both internal and external adaptation of tasks. Internal adaptation implies that the task or component knows by itself how to morph in order to tolerate faults and will trigger such measures autonomously. External adaptation are governed by the ADMORPH toolchain and happen in a coordinated manner at predefined points in time. We use the former for time-critical configurations, such as restarting low-level control replicas to absorb the faults of compromised ones within a few control epochs. The latter governs more involved reconfiguration, such as transitioning to a new configuration at the end of a schedule's hyperperiod. In particular, internal adaptation is tasked to tolerate faults long enough so that external adaptation has the time to evade the cause and return the system to a secure state.

In the following, we will report on

- adaptation to achieve control-aware fault and intrusion tolerance (in Section 3),
- recovery and resilience of control tasks (in Section 4)
- bounding reconfiguration times (in Section 5) and in particular keeping them low enough so that timeliness can be maintained in spite of faults and subsequent reconfiguration to recover from them,
- network-level monitoring and adaptation (in Section 6),
- interfacing with the coordination language compiler infrastructure (in Section 7), and
- testing runtime systems and adaptation strategies (in Section 8).

We start by reporting on the SOTA advances since deliverable D2.1b.

2 State-of-the-Art since D2.2 (month 33)

In addition to the threats reported in D2.2, Sargolzaei et al. [25] identified a new type of emerging threats: Time-Delay-Switch (TDS) faults. These faults are either injected by an adversary or the result of a an accidentally faulty communication channel (sensor-to-actuator). In the former case, attacks are injected purposefully and follow some planned strategy in order to inflict maximal damage to whereas for the latter faults are typically transient and stochastic. TDS commonly manifest as unknown time variable delays into a control signal such as the feedback signal imparted from the control regulator to the plant or the controller internal state update. TDS have been reported to cause significant harm in networked control systems (NCS). Because the latter are composed

of an array of parallel plants exchanging sensor data and feedback signals with a controller over multiple TDS-vulnerable communication channels, an induced delay in the actuation can severely compromise the overall stability of the system [64]. Even though, the effects of TDS on NCS have only been studied when only one controller actuates in a network of multiple plants, the same reasoning applies to a system with replicated controllers such as the one described in the next section. Arguably, negative effects of injected delays are expected to be amplified in a CPS that require real-time guarantees.

Several works have been dedicated to monitoring faults and attacks, including TDS attacks, by estimating the state in the control system [55].

Machine learning techniques have been developed to detect and estimate time delay in real-time but without being able to mitigate them [19].

Zhang et al. [71] developed a framework to lay out the worst-case behavior during design time such that TDS faults can be mitigated.

Since D2.2 we identified different types of recovery methods that enable the CPS(oS), in the presence of a fault at the level of the sensor system and/or controller to return to a state as secure and safe as a previous verified one. Herein we distinguish between shallow and deep recovery schemes. The former entails methods which repair the CPS behavior under attacks with minimal or no operation on the system states. [18, 20] propose an example of this where the compromised component is merely restarted and substituted with a surrogate of the original without sacrificing the systems stability. Another instance of a shallow recovery is to leverage redundancy where the output of multiple replicas are fused together and upon attack detection isolate the faulty contribution [60]. Another variation of a shallow recovery has been investigated by [40, 35] where a state checkpointer stores historic and verified states and are subsequently used to predict what the current state post failure time is. Deep recovery strategies extract information about system states and use this knowledge to steer the corrupted CPS back to a verified target state. [72, 73] propose a linear programming recovery controller to avoid a large deviation of the CPS trajectory from the desired one and steer it back to a safe endpoint where safety and stability can be guaranteed.

3 Adaptation to Achieve Control-Aware Fault and Intrusion Tolerance

Figure 2 shows the control architecture of ADMORPH. In many situations, including our use cases, high-level controllers steer the behavior of more low-level control loops that run at much higher frequency and have as additional task the stability of the plant. A common example are advanced cruise control systems, which aim at keeping the distance, lane and velocity, while a lower-level controller follows the path they dictate. Similar controls can be found in our taxiing use case, whereby the high-level controller might as well reside outside the controlled CPS, communicating steering commands through wireless connections. Low level controllers can themselves consist of multiple instances of complex and simplex control components, the latter possibly replicated.

We have identified the following adaptation possibilities in such systems:

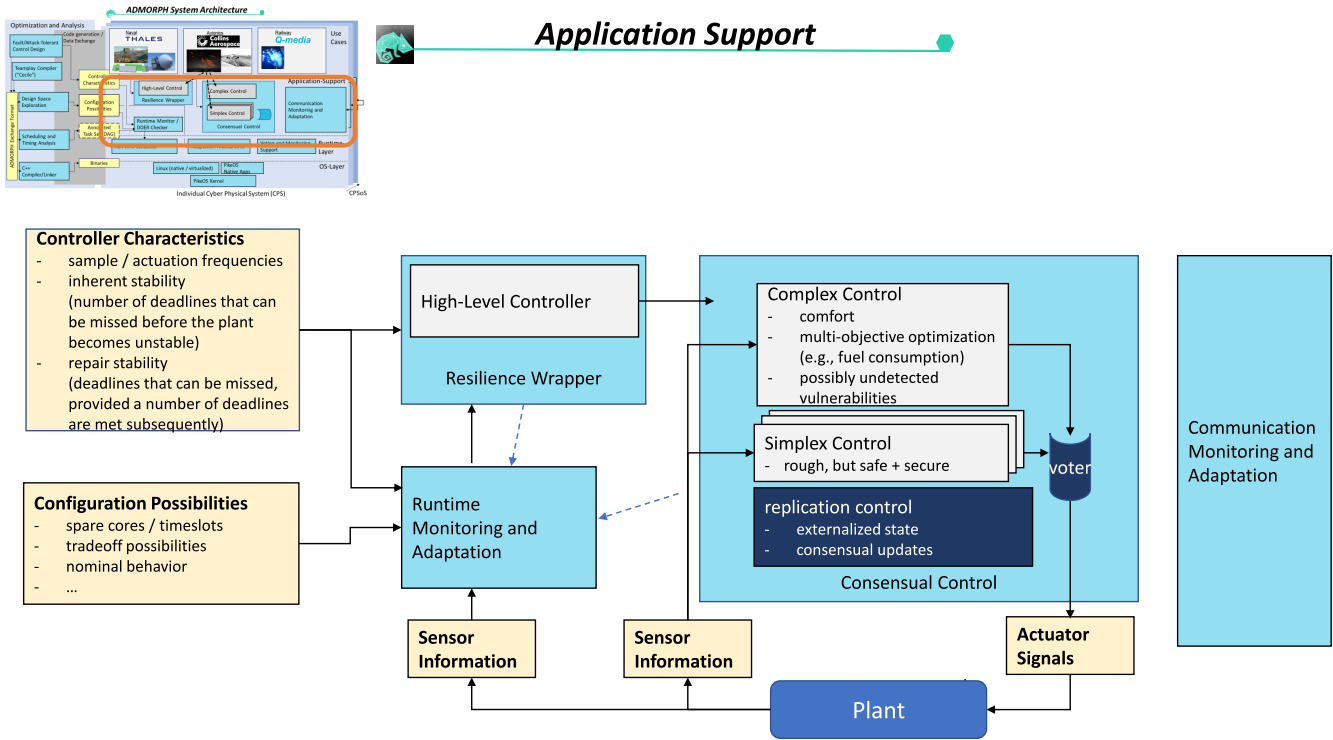


Figure 2: ADMORPH control architecture: The control architecture considers high-level controllers feeding commands into low-level controllers, which themselves are subdivided into a complex control task and simplex controllers, ready to take over in case complex fails. A replication control component coordinates the replication and hence the resilience of the simplex controller. It reacts to reconfiguration requests from the runtime monitoring and adaptation component.

1. *Functional adaptation*, by transitioning between multiple high-level controllers and the functionality they provide (e.g., manual vs. automated steering, but also internal vs. external control). Such adaptations typically happen in the form of configuration changes at pre-defined points in time and are coordinated by the coordination language, its compiler and tool chain. Functional adaptations at non-pre-defined points in time are event-triggered mode changes and must be considered ahead of time, including the transition period.
2. *Resource adaptations* of the current functionality (e.g., in response to changing loads or resource unavailabilities). Such adaptations are as well steered by the coordination language framework.
3. *Threat-related adaptations*, such as increasing / decreasing the internal resilience of components in response to observed higher or lower perceived threat levels. Adaptation at this stage needs to be a combination of proactive planning of the resources used in case of imminent stronger threats and fast runtime adjustments of the components itself to ensure the

system is in the desired state one the threat manifests [61].

4. *Adaptation of the runtime monitor*, in terms of strategies, depth of analysis and monitored components. Adaptation of the above kind typically entails also adapting the monitoring subsystem whose responsibility it is to observe components and identify abnormal behavior. Adapting also the observation and handling strategies allows monitoring to focus on the risks at hand.

The above strategies may lead to adaptations at runtime. These are internal to the control architecture to quickly respond to unforeseen situations and by leveraging excess resources and planned configurations that have already be anticipated during the design-space exploration. Such adaptations may include:

- take over by the simplex subsystem in case complex fails to provide correct information in time
- adjustment of the simplex replication policy by transitioning from a detection quorum, which establishes resilience over subsequent control epochs to immediate masking
- relocation of continuously failing controllers to spare resources
- adaptation of the frequency of rejuvenation

The above control-aware fault-and-intrusion tolerance techniques are enabled by the inherent stability of the plant.

We have implemented the above adaptation methods into the control infrastructure presented in D2.1b and evaluate them in our inverted pendulum demonstrator.

4 Adaptation to Recover and Make Resilient Control Tasks

Over time, control tasks fail or become compromised by adversaries aiming to take over the system and cause harm to the environment in which it operates. Recovery of such control tasks and the resources they use is essential to maintain healthy majorities and to continue tolerate failures. We have therefore extended both the state-capturing capability of the replicated controller and the its ability to restart replicas in a stateless manner with the means to also bring up additional replicas. These new replicas start from binaries that are drawn from a pool of pre-compiled and pre-analysed images that are sufficiently diverse to cancel adversarial knowledge how to attack replicas of this kind. Starting without state, the replication controller then injects the captured state and keeps these replicas up to date in the control tasks at hand before transitioning the responsibility to actually control the plant¹ to them. This way, additional replicas can be created and later brought

¹We shall use the term plant, as common in control-related articles to refer to the controlled device or cyber-physical system.

into the active voting group once they are up and running. In the next section, we detail why this two staged approach is required, i.e., why replicas must first be operational before we can consider their outputs in the majority decisions that control the cyber-physical system and systems of the same.

5 Bounding the Time of Adaptation and Reconfiguration

Normally, reconfiguration is limited to starting the new components and the tasks they implement it and transitioning responsibility to them. However, faults may also affect the resources that have been used to run these tasks and components. To not exhaust these resources, in particular in the presence of transient faults, resources must be rejuvenated and reconfigured themselves. The latter typically implies rebooting the failing resource and testing it to see whether the fault persists.

Similarly, reconfiguration at software level creates new tasks, merges them into the existing schedules and sets up their communication to coordinate with the remainder of the system. Such reconfiguration is highly task and system dependent and may have a runtime that may by far exceed the deadline of the re-configured tasks, even when several of them can be missed.

Primary objective of Task T2.3 is to limit configuration times in order to guarantee bounded down times. While the Multi-model model-of-computation MoC can be used in order to bound reconfiguration times in the general case, it considers coarse-grain reconfigurations. We refer the reader also to Deliverable 3.2 for a more detailed account of Multi-model MOCs. In T2.3 we go beyond by considering finer-grained reconfigurations and we do so by integrating fault models into the scheduling analysis of redundant dataflow tasks. This way, we determine the WCET of such tasks in the presence of errors down to a specific probability.

For instance, in the subway demonstration, we use a data transmission system running on a separation kernel with two partitions (A and B), and each partition has a modem that connects to an LTE network. When a decrease in the signal quality of one channel (run on partition A) is detected, then the switching time to a new channel (run partition B) is dependent on the gateway response. A data transmission failure may occur if the gateway reacts too slowly to the local information about switching to the new channel.

In case there is a security attack on the software running on partition A, then the data transmission system also switches to the LTE modem operated from partition B, and then partition A is reset. If this reset restores signal quality, we switch back to the system on partition A.

Conclusion: The reaction of the data transmission system can be affected by a slow response of the gateway to the data transmission system about switching to another channel and by the ramp-up speed of Linux partition A in case the quality of channel B is low and communication cannot be established.

If we are to design redundant communication for composed systems, where the limit for communication loss is 2 seconds, it will be necessary to take into account all delays, including partition ramp-up after reset. In both cases, the separation kernel response is negligible compared to the “technology” response.

ADMORPH technology allows specifying this reaction time. Distribution of task to more cores

(CECILE) allows to shortening and bounding reaction. The lack of security is eliminated by PikeOS with its partitioning.

Some aspects might however still lead to unbounded reconfiguration times, such as repeated reboots of a resource after a crash or reactivation of the same software vulnerability. These situations typically indicate a systematic and persistent failure in the hardware resource and or the software component that is re-instantiated. For this reason, we investigated software diversification and relocation possibly to computing resources of different characteristics and the implied consequences on the WCET of this software component. With pre-analyzed images pooled as well as a few resources on stand-by, the system can then adapt the schedule and reschedule entire subsystems, if necessary.

However, even though the above measures will bound the reconfiguration times of the system, we expect most of these bounds to be significantly larger than the worst case response times of tasks in the system. In particular, we consider the case where the system would become unsafe during reconfiguration if control is not maintained throughout the reconfiguration itself. This is where fault tolerance comes into play to buy the time that is necessary to bring up the new subsystems. A second challenge, related to the previous one, is the fact that although each component will be reconfigured in a bounded amount of time, component interdependencies may push the overall reconfiguration time beyond the limit what can be absorbed by our tolerance measures. Therefore, instead of creating reconfiguration cascades by accepting the downtime of a system while a depending system is reconfiguring (which may easily lead to transitive effects), we aim to decouple reconfiguration and inclusion into the operational group as much as possible.

We have demonstrated this decoupling of reconfiguration and inclusion in our control architecture by preparing our system to reconfigure in the following three steps: First, the new configuration starts up, by creating new replicas, by starting components implementing the new functionality or by starting new resources (kept as spares). Second, the new configuration connects with the existing setup, including by ensuring that the new configuration receives state updates and sensor inputs and is as well already monitored. Third, once this preparation phase concludes and all components report their readiness to be included, we transition control to them by atomically updating voters and associated components to consider the new subsystems instead of the previous configuration. This way, the voter will consider and apply proposals exclusively from the old configuration (leveraging its inherent tolerance to faults for a given amount of time), until the new configuration is ready to take over. Then, once the new configuration is in place, irrespective how many control epochs this takes, at the beginning of the next control epoch, the voter will start to only consider proposals of this new configuration. This decouples configuration, because timely control is maintained either by the old configuration (until the transitioning point) or by the new configuration (from the transitioning point onwards).

6 Dependable network monitoring and support for adaptation

In this section we describe the developed work on network monitoring, in particular for improving ML-based Network Intrusion Detection Systems (NIDS). Our primary focus is on preparing appropriate datasets to tackle the shortage of datasets available to evaluate complex scenarios, including stealthy and targeted attacks designed to evade detection, attacks that are only detected by multi-view systems of different features (see below), and adversarial machine learning (AML) attacks that exploit machine learning vulnerabilities. This step is essential as it sets the foundation for developing effective models in later stages. Our objective is to create datasets that accurately reflect the challenges faced in CPS Network Intrusion Detection and enable the creation of models that can effectively overcome these challenges. The following subsection presents a comprehensive overview of datasets for network monitoring, covering their formats and highlighting three significant datasets from the literature that serve as baselines for building a public evaluation platform. Then, work on evaluating the possibility of exploring multi-views to generate diversity is also described, including results from the performed analyses. Finally, the work towards the creation of a public evaluation platform is presented, followed by a discussion of open aspects that will require further work in the future.

6.1 Datasets for network monitoring

Network monitoring mechanisms rely on datasets to train and test their ML models. These datasets typically consist of network traffic data that includes both normal and attack behavior. The quality and diversity of the datasets used in such context play a crucial role in the performance and robustness of the network intrusion detection systems (NIDS) against different types of network attacks, including AML attacks. In this context, NIDS datasets should be representative of the network traffic in the monitored environment, diverse in terms of the types of attacks and normal behaviors, and labeled accurately. The availability of publicly available datasets for NIDS research and development is also essential for benchmarking and comparing different NIDS approaches [57].

Dataset type: Network data used in intrusion detection models can vary greatly in terms of characteristics, and according to [58], two prevalent datasets types for NIDS are packet-based and flow-based. Packet-based datasets typically consist of raw data packets captured by a network packet analyzer with minimal processing or manipulation. These datasets contain one instance for each packet and the payload information is used to compose the feature set [58, 52]. On the other hand, flow-based datasets work with traffic summaries, where for each instance, aggregate information that shares common properties (e.g., source IP, destination IP), using the protocol's header fields. The flow can be unidirectional or bidirectional (called sessions) [58, 52]. Some features extracted in flow-based datasets are the number of transmitted bytes, number of transmitted packets, duration, transport protocol, and others. It is possible to convert packet-based datasets to flow-based datasets, but not the other way around [58, 39].

Flow-based Datasets: Flow-based datasets play a crucial role in NID by offering a comprehensive overview of network traffic. This comprehensive view helps in detecting unusual behavior, which is crucial in identifying intrusions. Furthermore, flow-based datasets are more efficient in terms of storage and processing, as they retain only essential information about network traffic instead of retaining individual packets [[58]].

Packet-based datasets: Packet-based datasets capture all the individual packets of network traffic, providing a more detailed view of the data. This can be useful for identifying specific attack signatures, as well as for forensic analysis. However, packet-based datasets are more resource-intensive in terms of storage and processing, as they capture all the data. Additionally, they can be less effective at detecting abnormal behavior, as they do not provide as much context about the communication between hosts as flow-based datasets do [[58]].

Noteworthy Datasets: The use of datasets in the field of network security is crucial for evaluating and testing the performance of security systems. In this context, the *UNSW-NB15* [53], *BoT-IoT* [41], and *CIC-ID2017* [59] datasets play an important role as they provide diversity in network security by presenting a variety of scenarios, attack types, features and a large amount of network traffic.

The *UNSW-NB15* dataset, created by the Cyber Range Lab of UNSW Canberra, consists of 2.54 million records and 47 features of corporate network traffic. The *BoT-IoT* dataset also from the same lab focuses on the smart-home environment, with 72 million records and 32 features, including five IoT devices. The *CIC-ID2017* dataset, designed to resemble an ICS environment, includes the most popular attacks based on the 2016 McAfee report and boasts 700,000 records and over 80 features. Those datasets are accessible in both raw data (pcap) and flow-based formats, offering versatility in network security analysis.

The *UNSW-NB15*, *BoT-IoT*, and *CIC-ID2017* datasets were created in different environments and with various tools to simulate different types of attacks, making them ideal for testing the robustness and accuracy of network security systems. By using these datasets, researchers and practitioners can gain a better understanding of network security issues and develop solutions to prevent cyber attacks. The availability of pcap files also allows for the extraction of custom features, which we are leveraging in this work.

The varying feature sets utilized in different flow-based datasets in the literature have been observed by us, leading us to the conclusion that by consolidating the strategies employed by these works to construct their feature sets, we can produce a broad spectrum of views that can enhance the robustness of ML models against stealth attackers who attempt to evade NIDS systems that only use a single view. Furthermore, our aim is to create views that more effectively detect specific attack types. In the following section, we outline our initial experiment, which entails generating views of the *UNSW-NB15* dataset to generate diversity.

6.2 Exploring Diversity through Multi-Views

One approach to creating diversity is to utilize a multi-view approach. This approach involves generating multi-views or perspectives of the same data or problem, leading to a more comprehensive and diverse range of solutions. By exploring multiple angles and considering different viewpoints, a more well-rounded and diverse set of results can be achieved [68, 74].

We present our findings on multi-views, providing evidence on how to promote diversity in NIDS. The purpose of this experiment was to examine the potential impact of using multiple views to enhance diversity, taking into account our hypothesis that increased diversity will result in improved NIDS accuracy and robustness to AML attacks.

To investigate the effects of multi-views on enhancing diversity for network intrusion detection, we utilized the UNSW-NB15 dataset (original flow-based version, not pcap files), and we organized the features into three unique views, following the method described in [34]. These views were created based on their distinct features, including i) *Content view*, which encompasses information from the data payload; ii) *Basic view*, comprising attributes that represent protocol connections; and iii) *Traffic view*, consisting of traffic indicators of the current connections.

To evaluate the efficacy of each view, we trained three classifiers - Random Forest, Adaboost, and K-Nearest Neighbors - using the scikit-learn library version 0.20.3. To maintain the validity of the results, the models were trained with the default parameters without any optimization or customization. The dataset was divided into a training set (70%) and a testing set (30%) using the holdout method. The diversity was measured using the disagreement measure, calculated using the false positive and false negative as input.

The disagreement measure is a metric utilized to assess the extent of inconsistency between two or more classifiers. Simply put, it gauges the degree to which classifications produced by different sources differ from each other. The measure returns a numerical value ranging from 0 to 1, where a value close to 0 signifies low diversity, and a value close to 1 implies high diversity. In this study, we used the disagreement measure to evaluate the diversity between different views.

The results of the comparison among the views are shown in Figure 3. The disagreement measure was employed to quantify the degree of variation between the classifications produced by different views. False positives and false negatives of the classifier varied for each view, which is why we used all views as baseline comparisons. We first compared the Content View to the Basic and Traffic views. Then, we compared the Basic View to the Content and Traffic views. Finally, we compared the Traffic View to the Content and Basic views. The results of the experiment showed that there was evidence of diversity when using multiple views, as the disagreement measure varied from 25% to 71%.

The experiment yielded results that not only emphasized the diversity captured through multi-views but also highlighted the impact of classifier variability on diversity. This indicates that the utilization of different classifiers can also lead to diversity, which can be utilized to enhance the outcome. As depicted in Figure 3, the best disagreement performance varied between AdaBoost and K-Nearest Neighbors. The Random Forest classifier performed poorly in terms of generating diversity, with a median result of 44%, whereas AdaBoost and K-Nearest Neighbors achieved higher results of 59% and 60%, respectively. Furthermore, although AdaBoost and K-Nearest Neighbors

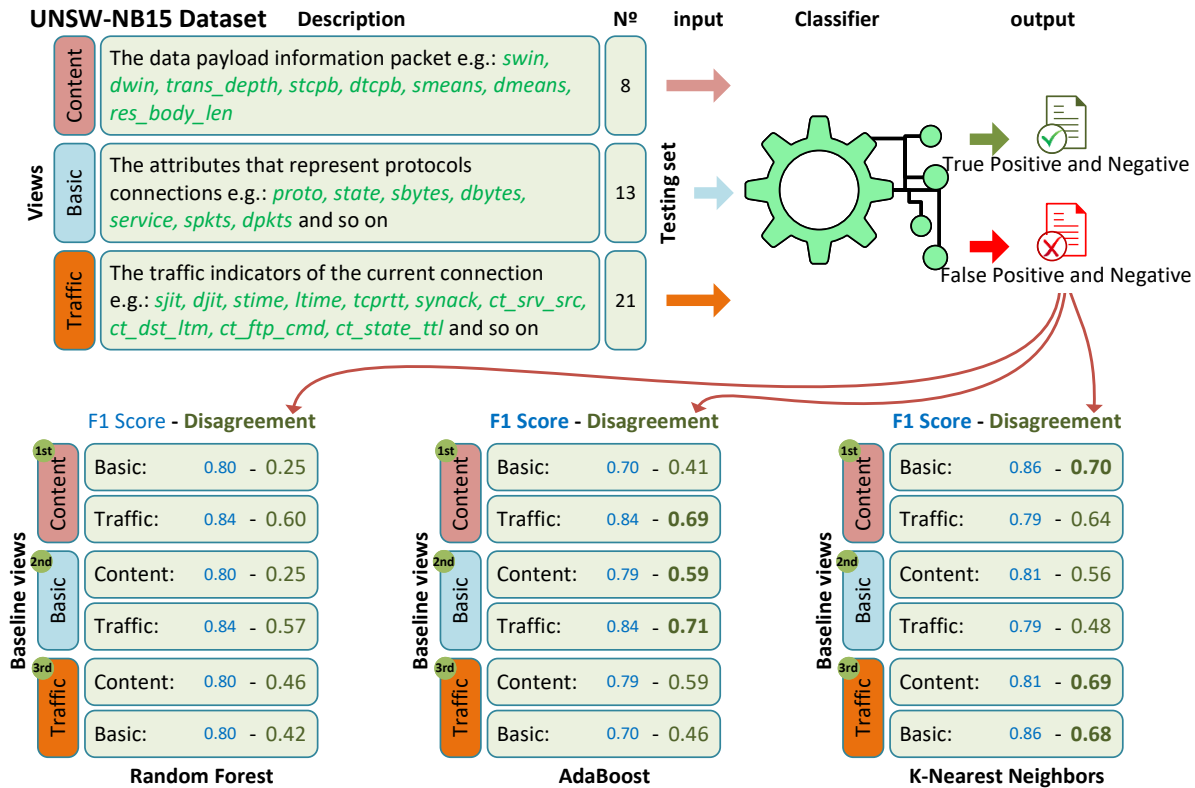


Figure 3: View Diversity.

produced similar results in terms of generating diversity in the mean, they leveraged different views, indicating that various classifiers can produce similar outcomes but generate diversity in unique ways, providing complementary results. This discovery supports the second aspect of diversity designed in our proposal, the diversity of models.

Afterward, we explored the concept further by building a public, comprehensive evaluation platform/benchmark for proposed architectures and other related literature approaches. Using the datasets previously mentioned, we employed four unique methods to construct multi-views, with the aim of uncovering the potential diversity in views.

6.3 Comprehensive Evaluation Platform

This report details our ongoing work on the development of a Public Comprehensive Evaluation Platform (PCEP). We aim to create a benchmark with multi-view and adversarial machine learning (AML) attacks to evaluate the proposed architecture documented in Deliverable D2.2 and make it available to the research and practitioner community.

A crucial part of our research is acquiring datasets that align with the specifications of our proposed architecture. The existing datasets in the field of Network Intrusion Detection are either outdated or do not cater to the requirements of multi-view and AML attacks [57]. Therefore, we aim to build or prepare new datasets that meet these criteria, and that can be used for benchmarking.

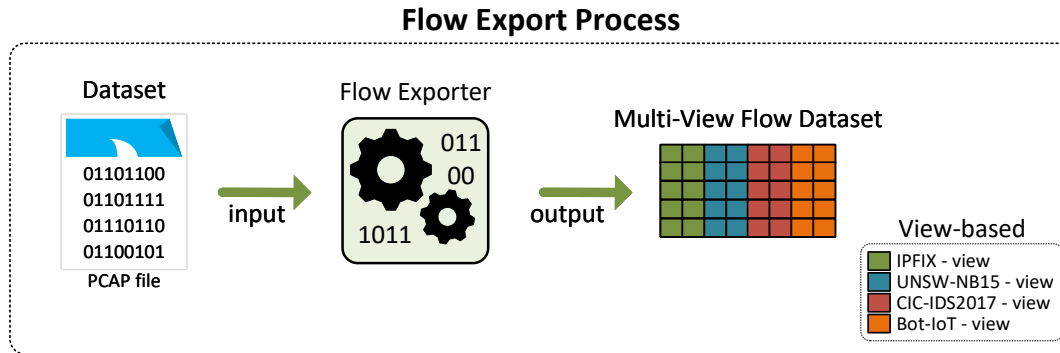


Figure 4: Flow Export Process.

After having the network packets flow with the four preliminary views, the next step is to generate the AML attacks. The generation of attacks is done through techniques widely used in literature. As a result of this phase, we will have three datasets with the necessary characteristics to evaluate the proposal and the literature approaches.

The preparation of the datasets involves utilizing three noteworthy datasets (UNSW-NB15, BoT-IoT, and CIC-ID2017) as baselines. The process begins with converting network packets, in the form of pcap files, into flows. Our flow export process is depicted in Figure 4. A pcap file is input into the flow exporter, which then converts the network packets into flows and extracts three views based on features on flow-based datasets formerly cited, as well as an additional view based on IPFIX features.

The datasets have been converted into network flow format with four views and properly labeled. However, our work is ongoing, and we will be outlining the remaining steps to complete the PCEP and to implement and to evaluate the proposed architecture.

6.4 Open Aspects

This section identified open aspects towards the implementation of a complete PCEP and evaluating the proposed architecture for network monitoring.

Concluding the PCEP: Once the network packets are converted to flows, the next step is to generate AML attacks to evaluate the performance of our proposed architecture. The generation of AML attacks can be carried out using popular methods commonly cited in the literature, including IDS Generative Adversarial Networks [[46]], Particle Swarm Optimization, and Genetic Algorithm [[22]].

As a result from this phase, one obtains three datasets that incorporate multi-views and AML attacks, and that share the the same single features. They can be used in scenarios as multi-font data and to evaluate platforms other than ours.

The PCEP results in three datasets that feature multi-views and AML attacks, providing a comprehensive evaluation platform for our proposed architecture and other relevant literature approaches. The consistency in features among the three datasets will also facilitate future studies,

such as evaluating the generalization ability of the models by training on one dataset and testing on another.

Design, Training, and Testing: After having the datasets ready, the next step is to create multiple scenarios to objectively evaluate various aspects of our proposed architecture and relevant literature approaches. Our implementation reflects the conceptual design.

A minimum of four evaluation scenarios has been devised to assess the impact of each proposed diversity. These scenarios are View Diversity, Model Diversity, Worker Diversity, and Overall Evaluation. These scenarios aim to uncover the contribution of each of the three diversities and the overall contribution of architecture for network monitoring to the field.

Evaluation: In this phase, the proposed architecture is evaluated to measure its performance and robustness against AML attacks. To evaluate the performance of the proposal, we consider the capacity to mitigate network attacks. Also, this phase reveals how each diversity proposed influences the robustness of the model.

To measure the effectiveness of the proposed architecture, a range of evaluation metrics can be used, including but not limited to accuracy, false positive rate, false negative rate, F1 score, and attack success ratio, to evaluate the architecture performance in AML settings.

7 Interfacing with the TeamPlay Coordination Language Compiler Infrastructure

Downstream tools, as shown in Figure 1, can interface with the coordination TeamPlay Coordination Language compiler and each other via the *ADMORPH Exchange Format* (AXF). The integration includes the *Scheduling and Timing Analysis* tools (developed for T3.3), the AROMA runtime environment (T3.3), the Design Space Exploration tool (T3.1, T3.3), and the TeamPlay compiler itself (T1.1, T1.2). The integration allows for turnkey processing of a TeamPlay-specified application with hardware and *Mean-Time to Failure* found through Design Space Exploration, with redundant schedules produced by Scheduling and Timing Analysis tools and capable of running on the AROMA Runtime Environment. The combined system is intended to be robust against hardware faults by migrating tasks when such a fault is detected. In this section, we discuss the integration via AXF, as well as outline the challenges faced when addressing robustness requirements via runtime mode-switches.

7.1 Exchange Format

The ADMORPH Exchange Format (AXF) is a text-based graph representation with additional attributes, allowing the expression of complex task graphs together with non-functional attributes in a single machine readable format. AXF task graphs are represented as a bipartite directed acyclic graph, where actor nodes (“tasks”) link to data nodes (channels). AXF files describe the hardware (number of cores, types of cores), and can optionally define a set of static schedules

for different scenarios (e.g. a schedule for dual-modular redundancy, a schedule for degraded hardware). The exchange format used multiple times. The *Scheduling and Timing Analysis* and *Design Space Exploration* tools (see Figure 1) incrementally add information to the AXF.

While both TeamPlay and AXF expect tasks to be implemented as C (or C++) functions, we create a simpler, uniform calling convention for AXF. TeamPlay communicates input and output data via the function signature (one argument per inport and outport), which may require a different signature for each task. As such, the function call differs per task, which complicates runtime execution environments. For AXF, the tasks are wrapped to support a homogeneous calling convention, where inports and outports are communicated via a one-argument data structure instead of separate arguments. Furthermore, TeamPlay supports multiple edges between task pairs as programming convenience, instead of requiring developer to merge them into one. AXF is restricted to only one edge between task pairs, simplifying further analyses and transformations at the cost of requiring the TeamPlay compiler to generate prologue and epilogue code for each task to unpack and pack the arguments.

We extend the AXF to allow for the specification with *sections*, where real-time mode switches can be made based on the state of the hardware. In particular, this allows reconfiguration of tasks as a result of failing hardware well before the end of the hyperperiod. The end goal is to use AXF to describe all possible configurations of the system under any anticipated fault situation that can still handle the minimum required service. For example, if a core of a certain type fails, the configuration to enable upon detecting such a core failure will be described using AXF and can be used to generate code for activating this configuration.

To aid in validating the chain of downstream tools, the TeamPlay compiler can be used to drive simulators, such as existing simulators like UPPAAL, or the fault-tolerance simulation runtime as presented in D1.2. The simulators can use (among others) the task graph, timing information, schedules and expected fault rate. The output of these simulators aids in the validation of the output of downstream tooling using the AXF format.

7.2 Runtime Configuration changes

Configuration changes in general may involve migrating existing tasks, starting new tasks (typically the ones that executed on the failed cores), and stopping tasks (e.g., to gracefully degrade the delivered service). However, in the presence of failing cores, some additional complexity is introduced as there is no control over which core is to leave the system. In a regular configuration change, the destination configuration might be viewed as a pool of identical cores and (from a reconfiguration perspective) interchangeable. But, when a core fails, merely the ability to switch to a destination configuration using one core fewer is not enough, as such a configuration change may make assumptions on which core it is to give up. Rather, the fault removes a core from the pool at random, and the tasks from that core need to migrate to meet system requirements. For example, assume a system with four cores of some type 'A'. Failure of any of these cores (or a similar anticipated fault) will activate a configuration where only three cores of type 'A' will be needed and the tasks-to-three-core mapping described in the configuration can then be realized by selecting any of the remaining non-faulty cores as core one of this mapping (proceeding accordingly

for core two and three of this mapping). However, depending on how this core-to-core mapping is realized, the runtime system needs to adapt a variable amount of tasks, while maintaining the timing guarantees of the old configuration until the new configuration can be enabled and cleanup completes.

We have identified two possible solutions to prepare the transitioning from source to target configuration with the help of the coordination language compiler infrastructure. Common to both is the assumption that each configuration is able to tolerate the occurring fault for a limited amount of time until a new configuration can be activated. This might imply a degradation of service until the new configuration can be enabled.

Solution 1 Construct a *transition schedule* for each source configuration and failure situation, which then facilitates the transition to the target configuration that should be enabled in the presence of a given failure type. This way, the compiler infrastructure can prepare the runtime to implement this mapping and ensure offline that all tasks continue to meet their deadline for as long as required, i.e., for all tasks of the source configuration until the transition to the target configuration can be performed (at a corresponding switching point) and for all tasks of the target configuration until cleanup completes. We expect this solution to offer faster transition times and more confidence in the transition, since it can benefit from the analytical rigor of all offline tools (including WCET analysis, long running optimizations, etc.). The drawback of course is the overhead in terms of the number of source/target pairs to consider, for each anticipated failure situation. In particular, we need to apply the same analysis recursively to the still operational resources to prepare the system to tolerate additional faults, although we expect the number of simultaneous faults under which the system will still be able to deliver the minimal desired service to be rather small.

Solution 2 The second solution avoids a-priori computation of transitions altogether by deferring it entirely to the runtime. The runtime environment must identify which tasks need migration, stopping and starting to transition from the source to the target configuration. To ensure that faults can be tolerated until the target configuration is enabled, and to allow the target configuration to properly handle the system, deadlines of all tasks on the surviving cores must be guaranteed for the source configuration until the point in time when the target configuration can be enabled. Also, while the target configuration is already running, we have to ensure that all tasks of the target configuration meet their deadlines despite unloading the residual tasks of the source configuration. The method to achieve this timeliness during the ramp up and ramp down phases differs significantly from the one presented for the first solution. Rather than computing a transition schedule offline, we anticipate (during their construction) that each configuration will have to tolerate limited interference from other activities that are not known in advance. For example, each core may have to tolerate memory accesses at a certain pre-defined bandwidth and may experience a limited interference through its caches. We assume that any such interference can be controlled (e.g., as described in MemGuard [69] and subsequent papers for resources other than the memory). Therefore, rather than scheduling ramp-up and ramp-down tasks, we would allow the runtime to use the anticipated interference to start, stop and migrate the required tasks. In the schedule,

this ramp-up / ramp-down interference can be included as virtual tasks, which exhibit such a behavior. Runtime interference control thereby ensures that ramp-up and ramp-down will not exceed the anticipated bounds of these virtual tasks.

For solution 2, the ramp-up and ramp-down activities will be severely hampered by the interference-control mechanisms (e.g., when limiting the memory bandwidth available to load new images into a processor’s tightly coupled memory during ramp-up). It will therefore be more difficult to estimate worst-case bounds, at least in the orders of magnitude required for giving ”normal” timeliness guaranteed (e.g., within the epoch lengths of control tasks). But this is also not needed, since the actual transitioning from source to target configuration happens at a different time scale (e.g., over multiple control task epochs). The only point where ”normal” scheduling and the ramp-up / ramp-down activities have to coordinate is when switching from the source to the target configuration. The AXF describes viable points for such a transition in the form of anticipated switch points. Therefore, even if the target configuration is ready to take over, the runtime needs to wait for the next such switching point to transition to the target configuration. Until this point in time, the surviving tasks of the source configuration maintain timely control and the other required real-time aspects of the system, while the target configuration is ramped up. And starting from this point in time, the target configuration will be active and responsible for guaranteeing timeliness, while the remainder of the source configuration is ramped down in the same interference constrained manner.

We are currently comparing and contrasting both solutions to evaluate which one (if any) can still be realized in the remaining time of the project.

8 Testing Runtime Systems

In control systems and in systems that adapts the behaviour of software, the software plays a crucial role of decision-making. Depending on the application, if this process is incorrect there can be dramatic consequences. Furthermore, modern applications include high levels of digitalisation and integration. For example, the software of a car executes several control systems in parallel (traction control, stability control, anti-lock braking system), also together with the infotainment systems [32, 37]. This makes control software complex, and prone to errors. Unsurprisingly, control software requires a long and costly verification and validation process [36].

During the verification and validation process, engineers spend most time on testing [75, 36, 28]. The main difficulty in testing control systems arises from the necessity of executing the system in a closed loop. Unit testing of the individual components is clearly important, but of limited effectiveness, and system testing is crucial [50, 21]. Given the tight coupling of components, it can be very difficult to identify a fault location. In fact, even when only one component is faulty, the malfunction spreads to all the components in the loop. Furthermore, the physics makes the execution of tests non-deterministic and costly both in time and resources.

To work around the tight coupling of the system and reduce the cost of executing system tests, it is common practice to *abstract* specific components and substitute them with executable models [48]. The choice of which components to abstract defines different testing setups [70, 31,

43]. Said setups are called *X-in-the-loop*, where “X” (e.g., software or hardware) describes which components are included as their final implementation and which components are abstracted. Despite being common industrial practice, the differences in fault-finding capabilities among X-in-the-loop setups have never been studied.

In particular, previous research started from the – often implicit – assumption that there exists a hierarchy among the testing setups. This hierarchy is supposed to manifest itself in terms of the testing capabilities and the coverage achieved with one or another setup. To mention some examples: [70, pp. 13–14] and [49, pp. 2] discuss of how each testing setup adds detail to the testing representativeness, [31, pp. 3] discusses the increasing level of integration of the different testing setups, [30] and [56] discuss the re-use of test cases across testing setups, and their incremental nature in approximating the real-world behaviour. Accordingly, previous literature uses the naming “*testing levels*” for the different setups, hence implying an ordering. A likely explanation of why this assumption has not been challenged, is that research on the topic is also limited by the development effort required by the implementation of the different setups.



With the aim of filling the gap in the study of the setups differences and of enabling further research, in the ADMORPH project we provide the following contributions: (i) a precise understanding of the testing abstractions in control systems’ testing, (ii) the development of four complete testing setups for a fully open-source case study, and consistent injection of different types of faults, (iii) comparison and discussion of said setups in terms of their ability to detect different types of software faults.













The system has three main components: (i) the physical process, (ii) the software implementing the control algorithm, and (iii) the hardware executing the software. The interaction between the controller and the physical process happens through actuators and sensors. The controller can also receive inputs from other software components or from human operators.

The overall structure of a CPS control system is usually represented with a block diagram similar to the ones shown in Figure 5. A cyber *controller* block is connected to a physical *process* block to form the closed loop.

Potentially, components can be abstracted – i.e. substituted with simulation models – so that the other components can be tested in isolation. Abstracting one or more components defines a testing setup [70]. When a component is abstracted, it is important that its simulation model and interaction with the other components are representative of the actual implementation. Said in other words, *for an abstracted system-level testing setup to be effective, there are associated assumptions that have to hold: these assumptions concern the validity of the models, their implementation, and their interaction.* Figure 5 provides a graphical representation of the closed loop for each testing setup: the dashed blocks are emulated and solid ones are implemented. Table 1 summarises the main testing setups and their fundamental assumptions.

At the model-in-the-loop abstraction level, all components of the system are simulated through models, as shown in Figure 5-MIL. The execution of said models requires a dedicated simulation environment. During the system development, MIL testing is performed in two ways: during the control design, and as part of the system testing (so called model-testing [29]). For control design, the control engineer develops an executable model of the physical process, represented in Figure 5-MIL by the function f , and a control law g , that uses measurement signals and control

Table 1: Abstractions for system-level testing of control systems. Comparison among: Model in the Loop (MIL), Software in the Loop (SIL), Hardware in the Loop (HIL) and Process in the Loop (PIL). The setup comprises controller code (C), hardware (H) and process (P);  indicates that a component is simulated,  that its real instance is used.

Name	Setup			Underlying Assumptions
	C	H	P	
MIL				(i) process model is accurate, (ii) controller model corresponds to implementation
SIL				(i) process model is accurate, (ii) hardware model captures the relevant properties (e.g., timing and instruction set)
HIL				(i) process model is accurate, (ii) execution of input/output hardware peripherals is not affected
PIL				—

commands to compute the actuation signals. The models are defined as differential equations, difference equations, and state machines [45], they can be implemented using common simulation software, like MATLAB [2] or Modelica [3]. In this way, the closed loop is tested to verify that the algorithm meets the expected performances and can be used to fine-tune the parameters [66].

MIL testing is completely simulation-based, and hence it fully relies on modelling assumptions. These can be divided in two categories depending on what they concern: (i) physical process-related assumptions, and (ii) controller-related assumptions.

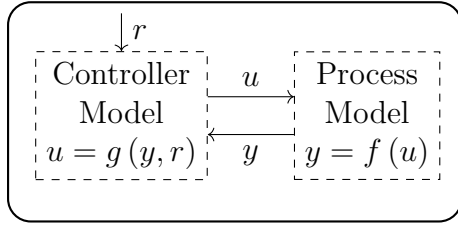
Examples of assumptions on the process are: neglected dynamics (like tyre dynamics in the vehicle model for cruise control), modelling approximations (like linearisation of nonlinear models), and neglected phenomena (like friction and road surface variability). Control theory provides metrics and rules-of-thumb to quantify the *robustness* of a control algorithm to non-ideal behaviour. However, these metrics also rely on assumptions, and hence need verification.

Examples of controller-related assumptions are: ideal timing (instantaneous execution) and infinite numerical precision. Moreover, not necessarily all of the required control features are implemented at this level. For example, a controller with different modes of operation may benefit (in terms of simplicity) from these modes being implemented and verified individually, neglecting the mode switching. In the cruise control example, a controller handling the distance from the vehicle ahead and a controller keeping the desired speed might be tested separately. If this is the case, the mode switching code has to be tested in other setups.

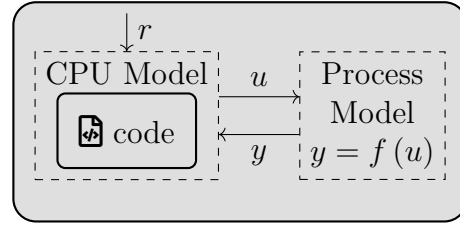
In the software-in-the-loop setup (Figure 5-SIL), we include the actual software implementation, while hardware and physical process are still abstracted. The physical process is implemented using models that are similar to (or the same as) the ones used for MIL testing.² On the hardware side,

²Here, models might need refinement. In the cruise control example, during the control design process, the engineer may assume direct control over the vehicle acceleration. In the SIL setup, on the contrary, the simulation

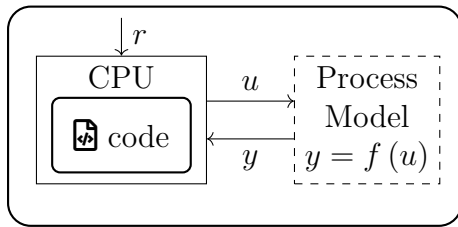
Notation: r (reference commands), u (actuation signals), y (measured signals), ■ (actuators), □ (sensors).



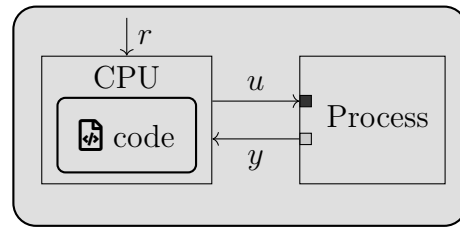
MIL: Model in the Loop.



SIL: Software in the Loop.



HIL: Hardware in the Loop.



PIL: Process in the Loop.

Figure 5: Testing setups for different testing levels. Dashed lines indicate that the corresponding component is simulated, while solid lines denote the component execution.

different choices are viable, from simulating only a few hardware components – like for example in our motivating example where we emulated the encoder and the pulse width modulation – to complete cycle-accurate hardware emulation. A simple alternative is to test the code in a general purpose machine. The code is then compiled for and executed on a machine different from the target one, hence abstracting the hardware and the execution environment. Under the associated assumptions, this enables testing of the functional component of the software, i.e., if the control law g is implemented correctly. However, other non-functional properties (e.g. execution time) cannot be verified, since they relate to system components that are abstracted. A more detailed alternative is hardware emulation: tools like gem5 [4] and Renode [5], can provide a higher degree of testing significance. Such a solution is often preferred in embedded systems (hence in control systems as well) given the strong coupling between hardware and software. In this way, the software is compiled for the target hardware. Among other things, hardware emulation enables the testing of the interaction with the Real-Time Operating System and possibly low-level software routines that interface with the sensor and actuator peripherals [26].

In SIL, the testing abstractions can still be divided into two sets: the first set is equivalent to MIL and relates to the process modelling, that needs to be accurate. The second set of abstractions is related to the environment in which the software is executed, varying significantly according to the specific choices made for the hardware abstraction. In general, these require that execution

needs to include the fact that the actuation signal is the voltage command sent to a digital-to-analog converter connected to a servo, that moves the throttle valve of the engine.

environment is representative of the actual one. Such abstractions mainly include: (i) software environment (meaning the interaction with other software components: for example the real-time operating system, if the code is executed on machine other than the target one), (ii) hardware (e.g. support for floating point arithmetic [47]), (iii) time modelling (the timing of the software execution has to be consistent with the physics simulation and possibly with other events, like user commands), and (iv) input and output definitions (measurement and actuation signals are representative of the real ones, e.g. with respect to measurement units).

The hardware-in-the-loop setup includes the target hardware in the testing process, as shown in Figure 5-HIL. The control software is now executed on the target computing platform – e.g. the microcontroller of the car in the cruise control example – and the model of the physical process is simulated on a different machine. The actuation signals produced by the software are extracted and fed to the physics simulator, while synthetic sensor readings from the simulator are fed to the hardware. The main design choices for this setup concern (i) the level at which the measurements and actuation signals are redirected, (ii) and the synchronisation between the controller execution and the physical process. For the first item, options range from using a debug port and accessing the memory registers of interest, to manipulating the software so that it interacts with the simulator instead of the actual peripherals. If signals are intercepted at lower levels, more details will be required for the model simulation: for example, in the cruise control, speed readings might have to be scaled to the encoder resolution instead of being in the physical units of measure. As an alternative, dedicated testing hardware can be developed so that it interfaces with the simulator at the physical connection level (i.e. I/O pins) instead of requiring that the software is redirected. This allows better coverage of the low-level firmware. Concerning the time synchronisation, the testing setup must ensure the consistency of time between the target hardware and the simulated physics; this can be done by performing the physics simulation and the I/O operations in real-time. Such a solution is however difficult to realise [44] and explicit synchronisation points might be needed—e.g. every millisecond the hardware is halted, then outputs are read, the physics is simulated, and sensor values are written before execution is resumed.

Apparently, also the HIL setup includes the abstractions associated to the modelling of the physical process. The two sets of design choices mentioned above are associated to respective testing abstractions. Intercepting the actuation and sensor signals at a higher level will possibly exclude more of the software that handles said signals in the control system. Consequently, this software is abstracted from the testing and assumptions have to be made about its behaviour. Analogously, the chosen synchronisation mechanism (if a real-time simulation is not implemented) can abstract timing phenomena from the test. For example, if the controller and physical simulator are synchronised every millisecond, events that happen at a higher rate are abstracted. To summarize, the HIL testing abstractions concern: (i) the input-output interactions of the hardware with the physical world, and (ii) the consistent evolution of computational time in the hardware and the evolution of time in the physical process.

In the PIL setup the physical process is included in the closed loop, therefore the full implementation of the CPS can be used and there are no testing abstractions. Extra sensors could be installed on the process and prototypes might be used in place of production models: such solutions are highly application-dependant and therefore excluded from this discussion. Hence,

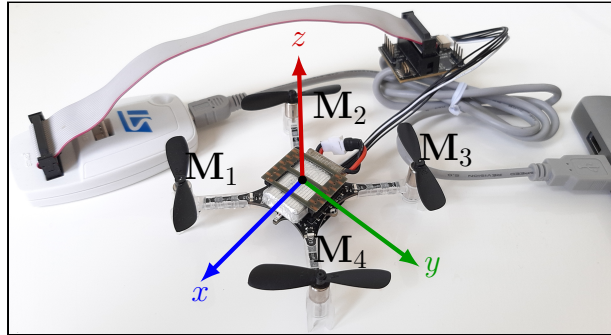


Figure 6: Crazyflie 2.1 with the STM debugger link.

PIL testing effectiveness mostly depends on the testing strategy. However, in this work we focus on the design of the setups rather than of the testing strategy.

We developed and implemented the MIL, SIL, HIL and PIL testing setups for the Crazyflie 2.1 quadcopter [6], shown in Figure 6. Our code is available on github at <https://github.com/dummy-testing-abstractions/testing-abstractions>. We developed the setups with the objective of allowing consistent injection of the software faults in each of them. We did so by allowing only minimal modifications in the drone software when implementing the different setups. We provide a detailed report on the modifications and the setup design choices behind them. This is crucial to avoid biases in the study caused by the differences in the fault implementations and to assess the general validity of the results.

With the different setups, we first run the control software. We then inject faults in it, and run tests in each abstraction configuration. We use this procedure to expose the different fault-revealing capabilities of the setups. The choice of the Crazyflie case study is motivated by two main reasons. First, the control system of the quadcopter is both not trivial and based on the most used control algorithms, making it a practically relevant case study. In fact, the Crazyflie is known to the research community; it is used for both education and research, e.g., quadcopter control design [33], swarm robotics [24, 42, 51], distributed [65] and robust control [54]. Second, both the Crazyflie software [7] and hardware [8] are completely open-source. We therefore have complete knowledge about the design of the system, which allows us to build a testing infrastructure for all the MIL, SIL, HIL, and PIL setups. In particular, using the open source hardware specification, we can build the hardware emulator for the SIL testing. Similarly, we use the open source code for both SIL and HIL testing. To ensure reproducibility of the results and make the artefact available to the research community for further investigation we used only open source tools for the implementation of the infrastructure needed in the different setups.

We implemented in Python a physical model to describe the Crazyflie and its controller [38]. We use the SciPy module [9] to integrate the differential equations describing the physics. In MIL, several aspects are abstracted with respect to the software implementation of the controller. Some examples are: (i) the computation of the matrix exponential is performed using the NumPy [10] linear-algebra library, while, in the real Crazyflie software, the calculation is approximated, (ii) each floating point variable has double precision, while the firmware uses single-word floats, and (iii) our

model implementation is single-threaded, while in the software the algorithm is distributed over different threads. The physics model is based on first principles, however it also abstracts different phenomena. Some examples are: (i) the flexibility of the structure of the drone is abstracted (hence assumed infinitely rigid), (ii) the dynamics of the electric motors is abstracted (the relation between the voltage fed to the motors and the vertical thrust is assumed quadratic), and (iii) the differences between the two horizontal axes are abstracted (the drone is assumed symmetric).

In our SIL setup, we rely on the open-source hardware emulator Renode [5]. Bitcraze [11] maintains its own fork of Renode [12] and of the Renode-Infrastructure [13] which contains the emulators of the peripherals. We implemented the platform emulator, which is able to execute the binaries as they are compiled for the target hardware. We also implemented the infrastructure to allow communication between Renode and our simulator of the physics. Said infrastructure leverages the possibility of exposing, along with a Renode emulation, an OpenOCD [14] interface. Some changes were required in the software to interface with the physic simulator: (i) in the Flow deck driver, the low-level interaction with the camera is disabled, (ii) in the Z-ranger driver, the low-level interaction with the ranging sensor is disabled, (iii) in the motor driver, no output is written to the motors, (iv) in the IMU driver, the sensors calibration is skipped, (v) in the Kalman filter, a division by zero check has been added. (vi) debug variables are added in `mm_flow.c` and `mm_tof.c`. For the interested reader, the exact changes can be found in the patch file. When compiling the code, our changes are triggered by defining the preprocessor macro `SOFTWARE_IN_THE_LOOP`.

The most frequent interaction with the physics is the sampling of the IMU sensors, which happens every 1 ms . This periodic event is triggered by the IMU itself which sends an interrupt to the CPU. In our SIL setup, we use a python script to iteratively: (i) simulate the physics for 1 ms , (ii) feed the synthetic sensor data to the hardware emulator, (iii) trigger the sensor interrupt, and (iv) run the emulator. We empirically observed that the virtual time in the emulator is dilated. More specifically, the 1 ms software tick of the real-time operating system does not always increase when the emulator is issued to run for one millisecond. For this reason, at each iteration our script checks whether the software tick has increased or not and run the emulator until the tick increases. This check suffices to keep the simulated physic time and the real-time operating system time synchronised, at least to the resolution at which the sensors are sampled. Differences from execution on the real platform can still happen in other tasks that are timed on something else than the real-time operating system tick.

To summarise, our SIL setup for the Crazyflie is based on the following assumptions and abstractions: (i) the physical model is representative of the physical process and of the sensors, (ii) the emulator of the CPU is accurate, (iii) the synchronisation between the physical model and the emulator is representative of the actual interaction, and (iv) the hardware of the Flow deck is not emulated.

In our HIL setup, to enable low-level access to the hardware, we used the debugger link ST-LINK/V2 [15], also depicted in Figure 6. We used OpenOCD [14] to interface with the debugger, and communicate with the CPU. OpenOCD exposes a Telnet port through which it is possible to read and write to specific memory addresses, or insert breakpoints. We introduced the following changes in the software to interface with the physics simulator: (i) in the Flow deck driver, the low-level interaction with the camera for optical flow is disabled, (ii) in the Z-ranger driver, the

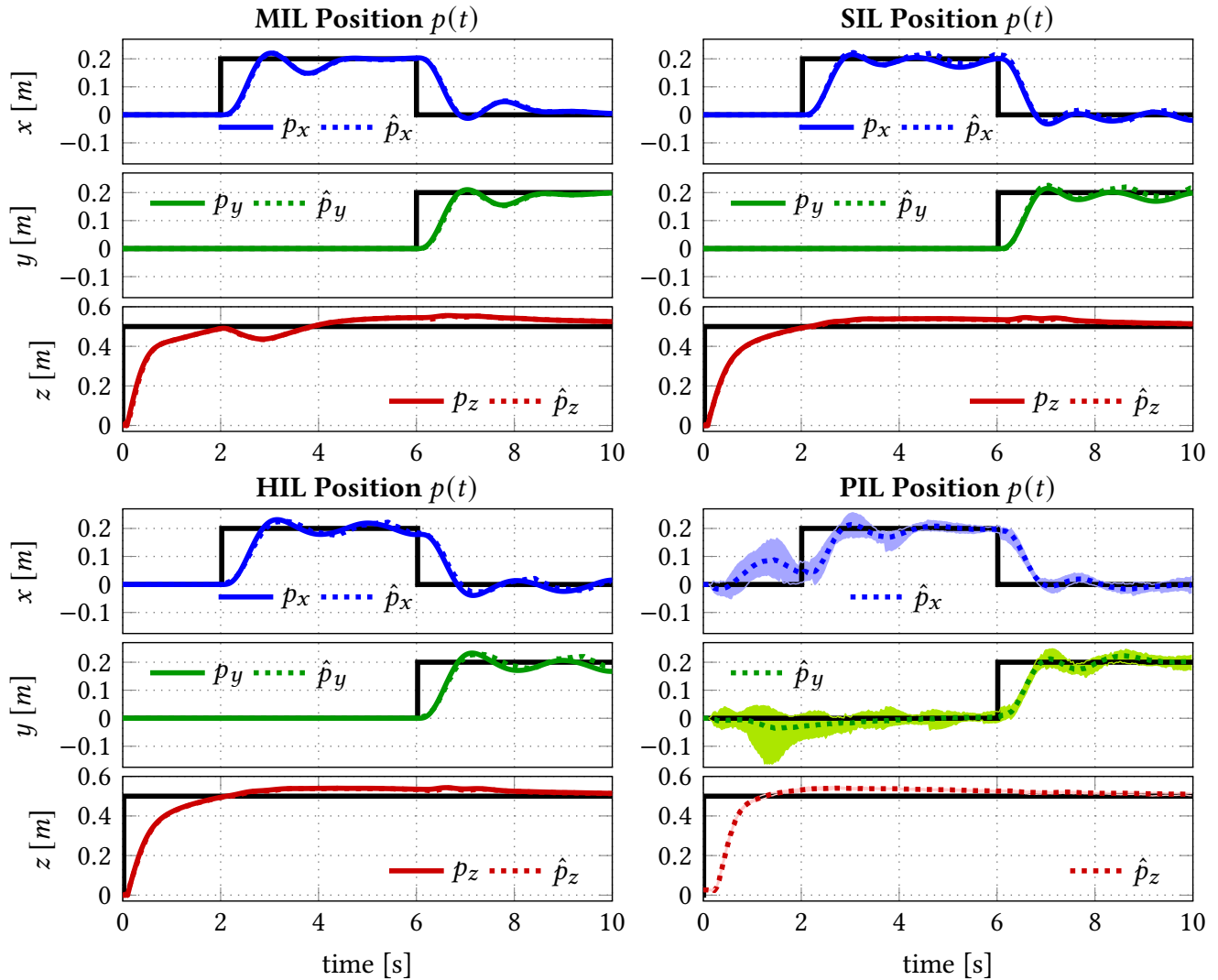


Figure 7: Nominal flight tests in the MIL, SIL, HIL, and PIL setups. For each axis x , y and z , the solid coloured lines show the drone’s true position (when available). The black lines show the step references. The dashed lines show the estimated position. For the 30 repeated PIL flights, at each time point, the dashed lines show the average over the 26 successful flights of the estimated state. Furthermore, the shades show the area between the maximum and minimum value measured at each time step.

low-level interaction with the laser ranging sensor is disabled, (iii) in the motor actuation, no output is written to the motors, (iv) the IMU sensor is never read, (v) the sensor thread is timed on the real-time operating system ticks instead of the external IMU interrupt, (vi) in the Kalman filter implementation, a check for division by zero has been added. (vii) debug variables are added in the files `mm_flow.c` and `mm_tof.c`, (viii) two assert statements in `uart_syslink.c` are

skipped.³ These changes are introduced with the provided patch file and triggered by defining the preprocessor macro `HARDWARE_IN_THE_LOOP`.

To synchronise the hardware with the physics simulator, we issue a breakpoint when the IMU sensor is read. When the breakpoint is hit, our python script performs the following operations: (i) read the motor values, (ii) simulate 1 ms in the physics, (iii) feed the sensor readings to the CPU, and (iv) issue the CPU to resume execution.

To summarise, our HIL setup for the Crazyflie is based on the following assumptions and abstractions: (i) the physical model is representative of the physical process and of the sensors, (ii) the synchronisation between the physical model and the emulator is representative of the actual interaction (in a different way compared to the SIL abstraction), (iii) the IMU interrupt is not used, and (iv) the hardware of the IMU sensors and of the Flow deck is not executed in the same way as in normal flight.

Finally, our PIL testing setup consists of running the Crazyflie with its nominal software. We use the Micro SD card deck to log flight data [16]. When compiling the code, the changes needed for the logging are triggered by defining the preprocessor macro `PROCESS_IN_THE_LOOP`. Our MIL, SIL, and HIL setups are deterministic, meaning that, when executed twice with the same inputs they will generate the same output. Instead, the PIL setup is not deterministic, because of the uncertainties related to the physical part of the system. For this reason we performed 30 test flights in PIL with the nominal software to assess the repeatability of the PIL experiments.

Figure 7 shows plots of flight with the software as released by Bitcraze in our different setups. The flight sequence consists of a take-off phase (from $t = 0$ to $t = 2$), followed by a setpoint step change in the x direction, $r_x(2) = 0.2$, followed at time $t = 6$ by a setpoint step change in both the x and the y directions, $r_x(6) = 0.0$ and $r_y(6) = 0.2$. Those *step responses* expose the main properties of a control algorithm thanks to their broad frequency spectrum [27]. Furthermore, a recent paper on the automatic detection of software faults in CPSs showed that the majority (in the case of said paper 80%) of control-related software faults appear in normal operation nor they need specific environmental conditions (and therefore trajectories) [63]. Apparently, exhaustive testing of the controller implementation requires more tests, and test case generation for CPSs is an active research topic [70]. In this work, we focus on the differences among the testing setups, rather than how to achieve exhaustive testing.

In the figure, the three top-left plots show the position of the quadcopter in the x, y, z coordinates in the MIL setup.⁴ For each plot, the figure shows both the actual position from the simulated physics (coloured solid lines) and the drone’s estimation (coloured dotted lines). The plots also include the reference position r (dark solid lines). This test shows that the model of the controller is able control the model of the process. Guarantees on the behaviour of the actual control system are however subject to the validity of both process and controller models, and on the implementation details [23]—i.e. the testing abstractions.

The top-right and bottom-left three plots in Figure 7 show the same test flight respectively

³The assert statements are related to the communication with the onboard microcontroller. In HIL they might be triggered and halt the CPU because the breakpoint interferes with the communication.

⁴More comprehensive plots for all the nominal and faulty test scenarios can be found at: <https://github.com/dummy-testing-abstractions/cps-testing-abstractions>

in the SIL and HIL setups, using the same conventions. The bottom-right three plots show the results of the repeated tests obtained with the physical process in the PIL setup. In PIL, there is no physics model involved and ground truth is not available, so we only display the position estimated by the quadcopter. Among the 30 PIL flights performed 4 failed without apparent reason, resulting in immediate crash. One possible explanation, as the producers suggest on their website, is that the IMU moving parts can get stuck at times. Using the successful 26 flights, we plot the average over the different flights of the estimated position (dotted lines), and the range between the maximum and minimum estimation. The PIL flights show consistent results, with the exception of the first 2 seconds. At take off, the turbulence caused by the ground effect can make the drone unpredictably oscillate. We also note that the z direction control is more accurate. This is due to the higher performance of the laser sensor compared to the optical flow.

While the general behaviour is consistent across the setups, few differences arise. In the SIL, HIL and PIL setups, the drone oscillates around the reference position in the x and y directions: this is due to the optical flow quantisation caused by the camera pixels. Movements smaller than the resolution of the camera are not detected. When the flow reading changes, the controller reacts at once, and the drone oscillates. This quantisation is abstracted in the MIL setup, hence not seen. In the MIL setup, the drone loses some elevation (z position) while performing the step in the x direction. This is caused by the loss of vertical thrust when the drone tilts to move laterally. Our tests show that the software implementation of the controller is robust to this disturbance. Finally, the ground effect is not captured in the physics model hence observed only in the PIL setup. Such phenomenon is chaotic and difficult to model hence often neglected in simulated setups.

We inject faults in the control software to expose the differences between the testing abstractions and highlight the capacity of each of them to unmask errors in the controller implementation. Unfortunately, it was not possible to mine the Bitcraze repository [7] for faults, as the developers do not use consistent practices to mark issues and commits associated with the control software faults, and frequently squash commits losing part of the version history. Furthermore, to the best of the authors knowledge, there exists no database of faults in control software.

Therefore, for obtaining faults to inject in the software we used different methods: (i) We selected two solved issues in the Bitcraze repository: the faults we used were suggested by Bitcraze engineers, because they struggled to reproduce and identify them. (ii) We took faults types from the close research field of faults in robotics systems: specifically, we considered [62, 67] to retrieve common types of faults and used the descriptions and examples to develop faults to inject. The scopes of the cited works are wider than ours as it relates to the whole robotic system and not just the control system. Hence, we manually filtered fault types that do not relate to the control system implementation—e.g. faults in communication protocols.

In [67], the authors use a practitioners survey to identify different categories of faults and provide some example for each category. Said categories are (with an example from the original work): (i) algorithms and logic (e.g. erroneous mathematical computations), (ii) resource leak (e.g. not closing a no longer needed connection), (iii) skippable computation (e.g. executing the same computation multiple times), (iv) configuration (e.g. erroneous initialisation of an address), (v) threading (e.g. incorrect timing code), and (vi) communication (e.g. incorrect address in the radio communication stack). Among said categories we excluded communication, as it apparently

Table 2: List of injected faults and corresponding test results. The fault names correspond to the patch files and flight plots in the repository [1]. For each fault, we report the corresponding categories covered among the types of faults in robotics faults highlighted in previous literature [62, 67]. For each fault and setup, we report in the last three columns whether the test flight was impaired ✖ or not ✔. We note naming discrepancies between the two works: e.g. a missed deadline is considered a threading fault in [67] and algorithmic in [62]. This does not affect our use of those classifications as we independently want to cover the relevant classes proposed by the two studies.

Fault Name	Category from [67]	Category from [62]	SIL	HIL	PIL
voltageCompCast	—	batteries/low-level drivers	✖	✖	✖
initialPos	configuration	algorithm: configuration	✖	✖	✖
flowGyroData	threading	sensors: communication	✖	✖	✖
motorRatioDef	—	motors driver/low-level drivers	✖	✖	✔
simUpdate	algorithms & logic	algorithm: wrong estimation	✔	✔	✔
byteSwap	—	sensors: connectors/config.	✖	✔	✖
gyroAxesSwap	—	sensors: connectors/config.	✖	✔	✖
timingKalman	threading	algorithms: missed deadlines	✖	✔	✔
flowDeckdtTiming	threading	software: computer vision	✖	✔	✔
slowTick	configuration	platform: controller board	✔	✔	✖

does not relate to the implementation of the control system performance. We also exclude resource leak, and skippable computation since they concern the embedded computing performance of the system rather than the control loop. For example, a memory leak is likely not seen in the control system performance, since it should not affect the functional properties of the software. Similarly, a repeated computation is not harmful, as the control software is supposed to be executed in an infinite loop. Such faults can become an issue when affecting the execution timing of the code, timing faults are however included in the threading class.

In [62], the authors surveyed the participants to the RoboCup [17] competition about faults encountered during the robot development. The practitioners were asked about faults concerning: the robotic platform, the sensors, the control hardware (where “control” refers to the communication with a master device that monitors and provides commands), sensors, robot software (the control software), and algorithms. Among those components we exclude the control hardware since, as mentioned, “control” is used with a different meaning than in this work, and refers to the user interface. For each of the remaining we report the main sources of faults mentioned by developers: (i) platform: batteries, motor drivers, and controller board, (ii) sensors: connectors, configuration, and communication, (iii) robot software: computer vision, inter-robot communica-

tion, and low-level device drivers, (iv) algorithms: configuration, wrong estimation, and missed deadlines. Among those fault types we exclude “inter-robot communication” since we consider a single system.

We manually develop and inject faults on the base of the descriptions and examples of the categories mentioned above. We cover all of the categories listed by the two surveys that relate to control software. Table 2 reports the list of the developed faults: the second and third columns map them to the different categories of [67, 62]. For each fault, we provide a patch file that injects it in the software [1].⁵ After injecting a fault, we perform tests in the SIL, HIL and PIL setups with the same flight sequence from Figure 7. The drone software used is the same in each setup, ensuring consistent injection of the fault. By setting one compilation macro (respectively `SOFTWARE_IN_THE_LOOP`, `HARDWARE_IN_THE_LOOP`, and `PROCESS_IN_THE_LOOP`) the code is compiled for the desired setup.

Table 2 reports the test results for each injected fault and each setup. We report whether the fault affects flight performance (✖) or not (✔) in the corresponding setup by comparing to the nominal behaviour observed in Figure 7. Complete flight data and pre-generated plots are available at [1], respectively inside the `flightdata` and `pdf` subfolders for each setup.

9 Integration

With the above in place, we can already provide a first brief report on the interplay of all technologies. Design-space exploration will start by determining a system configuration that exhibits the best possible survivability chances under the classes of faults and other reconfiguration causes we anticipate. A such identified base configuration will then be further investigated to determine the sequences of configurations through which the system needs to transition in case certain failures manifest. Techniques like alternated re-execution, the simplex-complex split of control tasks and the replicated execution of control tasks (in particular while leveraging the inherent stability of plants to execute with a detection quorum only), ensure a sufficient inherent fault tolerance of each configuration for long enough until the system can be adapted. Should a fault be detected (e.g., through our AI-based evaluation of safety monitors), the runtime will select the corresponding target configuration for this fault type and transitions to this configuration, using one of the methods described in Section 7. The transition phase will thereby either be already anticipated in a transition schedule or constrained such that the timeliness of the still active current or future configuration can be guaranteed before respectively after the scheduled transition point.

We are currently preparing a publication, in which we will further detail this interplay.

⁵The repository contains information about the specific version of the software that we used, together with detailed instructions on how to retrieve the correct version and inject the faults.

10 Conclusions

In this deliverable, we have presented the recent advances made in Task T2.1 – T2.6 of WP2 and our findings in terms of adaptation methods and how to embed them into the ADMORPH architecture. Although adaptation is governed by offline-computed strategies, possibly originating from design-space exploration, the actual adaptation performed at runtime requires care to secure fast response times of the individual CPSoS building blocks. In particular tasks with stringent timing requirements may need to be equipped with internal resilience mechanisms to absorb faults of an accidental and malicious nature. We have identified and already partially implemented several strategies for adapting to different situations, for bounding reconfiguration times and, in particular, for decoupling reconfiguration from the operational behavior of components, specifically controllers. The latter secures low response times by already bringing the new configuration into an operational state before responsibility is transferred.

Next steps include investigating how to more closely integrate internal adaptation and the coordination language and how to secure the monitoring and adaptation layer from faults.

11 References

- [1] <https://github.com/dummy-testing-abstractions/cps-testing-abstractions>, 2022.
- [2] <https://www.mathworks.com/products/matlab.html>, 30/06/2022.
- [3] <https://modelica.org>, 30/06/2022.
- [4] <http://www.gem5.org>, 30/06/2022.
- [5] <https://renode.io>, 30/06/2022.
- [6] <https://www.bitcraze.io/products/crazyflie-2-1/>, 30/06/2022.
- [7] <https://github.com/bitcraze/crazyflie-firmware>, 30/06/2022.
- [8] <https://github.com/bitcraze/hardware>, 30/06/2022.
- [9] <https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html>, 30/06/2022.
- [10] <https://numpy.org/doc/stable/reference/routines.linalg.html>, 30/06/2022.
- [11] <https://www.bitcraze.io/>, 30/06/2022.
- [12] <https://github.com/bitcraze/renode/tree/crazyflie>, 30/06/2022.
- [13] <https://github.com/bitcraze/renode-infrastructure/tree/crazyflie>, 30/06/2022.
- [14] <http://openocd.org/>, 30/06/2022.

- [15] <https://www.st.com/en/development-tools/st-link-v2.html>, 30/06/2022.
- [16] <https://store.bitcraze.io/products/sd-card-deck>, 30/06/2022.
- [17] <https://www.robocup.org/>, 30/06/2022.
- [18] Fardin Abdi Taghi Abad, Renato Mancuso, Stanley Bak, Or Dantsker, and Marco Caccamo. Reset-based recovery for real-time cyber-physical systems with temporal safety constraints. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2016.
- [19] Alireza Abbaspour, Arman Sargolzaei, Mauro Victorio, and Navid Khoshavi. A neural network-based approach for detection of time delay switch attack on networked control systems. *Procedia Computer Science*, 168:279–288, 2020.
- [20] Fardin Abdi, Chien-Ying Chen, Monowar Hasan, Songran Liu, Sibin Mohan, and Marco Caccamo. Guaranteed physical security with restart-based design for cyber-physical systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPs)*, pages 10–21. IEEE, 2018.
- [21] Afsoon Afzal, Claire Le Goues, Michael Hilton, and C. Timperley. A study on challenges of testing robotic systems. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 96–107, 2020.
- [22] Elie Alhajjar, Paul Maxwell, and Nathaniel Bastian. Adversarial machine learning in network intrusion detection systems. *Expert Systems with Applications*, 186:115782, 2021.
- [23] Nadia Alshahwan, Andrea Ciancone, Mark Harman, Yue Jia, Ke Mao, Alexandru Marginean, Alexander Mols, Hila Peleg, Federica Sarro, and Ilya Zorin. Some challenges for software testing research (invited talk paper). In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019*, page 1–3, New York, NY, USA, 2019. Association for Computing Machinery.
- [24] Brandon Araki, John Strang, Sarah Pohorecky, Celine Qiu, Tobias Naegeli, and Daniela Rus. Multi-robot path planning for a swarm of robots that can both fly and drive. In *2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore, Singapore, May 29 - June 3, 2017*, pages 5575–5582. IEEE, 2017.
- [25] Sargolzaei Arman, Federico Zegers, Alireza Abbaspour, Carl Crane, and Warren E. Dixon. Secure control design for networked control systems with nonlinear dynamics under time-delay-switch attacks. 5 2022.
- [26] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. A runtime verification tool for detecting concurrency bugs in freertos embedded software. In *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 172–179, 2018.

- [27] Karl Johan Åstrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, USA, 2008.
- [28] Antonia Bertolino, Pietro Braione, Guglielmo De Angelis, Luca Gazzola, Fitsum Kifetew, Leonardo Mariani, Matteo Orrù, Mauro Pezzè, Roberto Pietrantuono, Stefano Russo, and Paolo Tonella. A survey of field-based testing techniques. *ACM Comput. Surv.*, 54(5), May 2021.
- [29] Lionel Briand, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. Testing the untestable: Model testing of complex software-intensive systems. In *Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16*, page 789–792, New York, NY, USA, 2016. Association for Computing Machinery.
- [30] Eckard Bringmann and Andreas Krämer. Systematic testing of the continuous behavior of automotive systems. In *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems, SEAS '06*, page 13–20, New York, NY, USA, 2006. Association for Computing Machinery.
- [31] Eckard Bringmann and Andreas Krämer. Model-based testing of automotive systems. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 485–493, 2008.
- [32] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, Feb 2007.
- [33] Barbara Barros Carlos, Tommaso Sartor, Andrea Zanelli, Gianluca Frison, Wolfram Burgard, Moritz Diehl, and Giuseppe Oriolo. An efficient real-time NMPC for quadrotor position control under communication time-delay. In *16th International Conference on Control, Automation, Robotics and Vision, ICARCV 2020, Shenzhen, China, December 13-15, 2020*, pages 982–989. IEEE, 2020.
- [34] Jinyin Chen, Yi-tao Yang, Ke-ke Hu, Hai-bin Zheng, and Zhen Wang. Dad-mcnn: Ddos attack detection via multi-channel cnn. In *Proceedings of the 2019 11th International Conference on Machine Learning and Computing*, pages 484–488, 2019.
- [35] Hongjun Choi, Sayali Kate, Yousra Aafer, Xiangyu Zhang, and Dongyan Xu. Software-based realtime recovery from sensor attacks on robotic vehicles. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 349–364, 2020.
- [36] Sergio García, Daniel Strüber, Davide Brugali, Thorsten Berger, and Patrizio Pelliccione. Robotics software engineering: A perspective from the service robotics domain. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020*, page 593–604, New York, NY, USA, 2020. Association for Computing Machinery.

- [37] Iris Gräßler, Eric Bodden, Jens Pottebaum, Johannes Geismann, and Daniel Roesmann. Security-oriented fault-tolerance in systems engineering: A conceptual threat modelling approach for cyber-physical production systems. In Andrzej Bartoszewicz, Jacek Kabziński, and Janusz Kacprzyk, editors, *Advanced, Contemporary Control*, pages 1458–1469, Cham, 2020. Springer International Publishing.
- [38] Marcus Greiff. Modelling and control of the crazyflie quadrotor for aggressive and autonomous flight by optical flow driven state estimation, 2017. Student Paper.
- [39] Fariba Haddadi and A Nur Zincir-Heywood. Benchmarking the effect of flow exporters and protocol filters on botnet traffic classification. *IEEE Systems journal*, 10(4):1390–1401, 2014.
- [40] Fanxin Kong, Meng Xu, James Weimer, Oleg Sokolsky, and Insup Lee. Cyber-physical system checkpointing and recovery. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 22–31. IEEE, 2018.
- [41] Nickolaos Koroniotis, Nour Moustafa, Elena Sitnikova, and Benjamin Turnbull. Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset. *Future Generation Computer Systems*, 100:779–796, 2019.
- [42] Pierre Laclau, Vladislav Tempez, Franck Ruffier, Enrico Natalizio, and Jean-Baptiste Mouret. Signal-based self-organization of a chain of uavs for subterranean exploration. *Frontiers Robotics AI*, 8:614206, 2021.
- [43] Klaus Lamberg, Michael Beine, Mario Eschmann, Rainer Otterbach, Mirko Conrad, and Ines Fey. Model-based testing of embedded automotive software using mtest. In *SAE 2004 World Congress and Exhibition*. SAE International, mar 2004.
- [44] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. The MIT Press, 2nd edition, 2016.
- [45] William S. Levine. *The Control Systems Handbook*. CRC Press, Inc., USA, 2nd edition, 2009.
- [46] Zilong Lin, Yong Shi, and Zhi Xue. Idsgan: Generative adversarial networks for attack generation against intrusion detection. *arXiv preprint arXiv:1809.02077*, 2018.
- [47] D. Lohar, Clothilde Jeangoudoux, Joshua Sobel, Eva Darulova, and M. Christakis. A two-phase approach for conditional floating-point verification. *Tools and Algorithms for the Construction and Analysis of Systems*, 12652:43–63, 2021.
- [48] Paulo Henrique Maia, Lucas Vieira, Matheus Chagas, Yijun Yu, Andrea Zisman, and Bashar Nuseibeh. Dragonfly: a tool for simulating self-adaptive drone behaviours. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 107–113, 2019.

- [49] Abel Marrero Perez and Stefan Kaiser. Integrating test levels for embedded systems. In *2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, pages 184–193, 2009.
- [50] C. Menghi, P. Spoletini, M. Chechik, and C. Ghezzi. A verification-driven framework for iterative design of controllers. *Formal Aspects of Computing*, pages 1–44, 2019.
- [51] Derek Mitchell, Ellen A. Cappel, and Nathan Michael. Persistent robot formation flight via online substitution. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2016, Daejeon, South Korea, October 9-14, 2016*, pages 4810–4815. IEEE, 2016.
- [52] Borja Molina-Coronado, Usue Mori, Alexander Mendiburu, and Jose Miguel-Alonso. Survey of network intrusion detection methods from the perspective of the knowledge discovery in databases process. *IEEE Transactions on Network and Service Management*, 17(4):2451–2479, 2020.
- [53] Nour Moustafa and Jill Slay. Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set). In *2015 Military Communications and Information Systems Conference (MilCIS)*, pages 1–6, Canberra, Australia, 2015. IEEE.
- [54] Mark W. Müller and Raffaello D’Andrea. Stability and control of a quadcopter despite the complete loss of one, two, or three propellers. In *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*, pages 45–52. IEEE, 2014.
- [55] Miroslav Pajic, James Weimer, Nicola Bezzo, Paulo Tabuada, Oleg Sokolsky, Insup Lee, and George J. Pappas. Robustness of attack-resilient state estimators. 4 2014.
- [56] Jan Peleska. Hardware/software integration testing for the new airbus aircraft families. <http://www.informatik.uni-bremen.de/agbs/jp/papers/peleskaTestCom2002.html>, 01 2002.
- [57] Markus Ring, Sarah Wunderlich, Deniz Scheuring, Dieter Landes, and Andreas Hotho. A survey of network-based intrusion detection data sets. *Computers & Security*, 86:147–167, 2019.
- [58] Markus Ring, Sarah Wunderlich, Deniz Scheuring, Dieter Landes, and Andreas Hotho. A survey of network-based intrusion detection data sets. *Computers & Security*, 86:147–167, 2019.
- [59] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp*, 1:108–116, 2018.
- [60] Jongho Shin, Youngmi Baek, Jaeseong Lee, and Seonghun Lee. Cyber-physical attack detection and recovery based on rnn in automotive brake systems. *Applied Sciences*, 9(1):82, 2018.

- [61] Douglas Simoes Silva, Rafal Graczyk, Jeremie Decouchant, Marcus Voelp, and Paulo Esteves-Verissimo. Threat adaptive byzantine fault tolerant state-machine replication. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 78–87, 2021.
- [62] Gerald Steinbauer. A survey about faults of robots used in robocup. In Xiaoping Chen, Peter Stone, Luis Enrique Sucar, and Tijn van der Zant, editors, *RoboCup 2012: Robot Soccer World Cup XVI*, pages 344–355, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [63] C. S. Timperley, A. Afzal, D. S. Katz, J. M. Hernandez, and C. Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 331–342, 2018.
- [64] Mauro Victorio, Arman Sargolzaei, and Mohammad Reza Khalghani. A secure control design for networked control systems with linear dynamics under a time-delay switch attack. *Electronics*, 10(3):322, 2021.
- [65] Gang Wang, Weixin Yang, Na Zhao, Yunfeng Ji, Yantao Shen, Hao Xu, and Peng Li. Distributed consensus control of multiple uavs in a constrained environment. In *2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020*, pages 3234–3240. IEEE, 2020.
- [66] Michael W. Whalen, Anitha Murugesan, Sanjai Rayadurgam, and Mats P. E. Heimdahl. Structuring simulink models for verification and reuse. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering, MiSE 2014*, page 19–24, New York, NY, USA, 2014. Association for Computing Machinery.
- [67] Johannes Wienke, Sebastian Meyer zu Borgsen, and Sebastian Wrede. A data set for fault detection research on component-based robotic systems. In Lyuba Alboul, Dana Damian, and Jonathan M Aitken, editors, *Towards Autonomous Robotic Systems*, pages 339–350, Cham, 2016. Springer International Publishing.
- [68] Chang Xu, Dacheng Tao, and Chao Xu. A survey on multi-view learning. *arXiv preprint arXiv:1304.5634*, 2013.
- [69] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.
- [70] Justyna Zander, Ina Schieferdecker, and Pieter Mosterman. *Model-Based Testing for Embedded Systems*. 09 2011.
- [71] C.K. Zhang, L. Jiang, Q.H. Wu, Y. He, and M. Wu. Delay-dependent robust load frequency control for time delay power systems. 1 2013.

- [72] Lin Zhang, Xin Chen, Fanxin Kong, and Alvaro A Cardenas. Real-time attack-recovery for cyber-physical systems using linear approximations. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 205–217. IEEE, 2020.
- [73] Lin Zhang, Pengyuan Lu, Fanxin Kong, Xin Chen, Oleg Sokolsky, and Insup Lee. Real-time attack-recovery for cyber-physical systems using linear-quadratic regulator. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–24, 2021.
- [74] Jing Zhao, Xijiong Xie, Xin Xu, and Shiliang Sun. Multi-view learning overview: Recent progress and new challenges. *Information Fusion*, 38:43–54, 2017.
- [75] Xi Zheng, Christine Julien, Miryung Kim, and Sarfraz Khurshid. Perceptions on the state of the art in verification and validation in cyber-physical systems. *IEEE Systems Journal*, 11(4):2614–2627, 2017.