ADEPTNESS – Design-Operation Continuum Methods for Testing and Deployment under Unforeseen Conditions for Cyber-Physical Systems of Systems

EUROPEAN COMMISSION
Horizon 2020
H2020-ICT-01-2019
GA No. 871319

Deliverable No.	ADEPTNESS D3.3	
Deliverable Title	Report on recovery mechanisms for CPSoS and overall resilience improvement	
Deliverable Date	2022-02-28	
Deliverable Type	Report	
Dissemination level	Confidential	
Written by	MGEP	2022-02-28
Checked by	TUW, MGEP	2022-03-09
Approved by		
Status	Final version	2022-03-30

H2020-ICT-01-2019 – 871319 – ADEPTNESS: Design-Operation Continuum Methods for Testing and Deployment under Unforeseen Conditions for Cyber-Physical Systems of Systems

Acknowledgement

The author(s) would like to thank the partners involved with the project for their valuable comments on previous drafts and for performing the review.

Project partners

1 – MGEP – Mondragon Goi Eskola Politeknikoa – ES

2 – ORO – Orona S. Coop – ES

3 – UES – Ulma Embedded Solutions S. Coop – ES

4 – SRL – Simula Research Laboratory S. Coop – NO

5 – BT – Bombardier Transportation Sweden – SE

6 – IKL – Ikerlan S. Coop – ES

7 – EGM – Easy Global Market SAS – FR

8 – MDH – Maelardalens Hoegskola – SE

9 – TUW – Technische Universitaet Wien – AT

Disclaimer:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871319.

Document Information

Additional author(s) and contributing partners

Name	Organisation
Miren Illarramendi Ibai Roman Zlatan Tucaković	MGEP MGEP TUW

Document Change Log

Name	Date	Comments
V1.0 V1.1	2022-02-25 2022-03-09	Initial draft Version with minor comments commented by Zlatan Tucaković

Exploitable results

Exploitable results	Organisation(s) that can exploit the result
Recovery Microservice	MGEP

CONTENTS

LIST OF FIGURES

1 PURPOSE OF THE DOCUMENT

1.1 Document structure

This document presents the research and experiments done on recovery mechanisms and overall resilience improvement of CPSoS.

Section 2 presents an introduction and context of the work.

Section 3 introduces some background on recovery mechanisms, as well as the state-of-the-art.

Section 4 is focused on Recovery Mechanisms used to avoid undesirable behaviours of CPSoS's at operation. The section starts with a detailed description of the formalisation of the rules to define the hazardous behaviours that must be avoided based on the information that comes from other microservices such as Oracles and Uncertainty. Then, based on these rules, the methodologies and tooling to generate the code that will enable this checking as a Recovery mechanism is presented. It finishes presenting the integration with ADEPTNESS Microservice Template.

Section 5 Presents an example that validates the research and development carried out during the work presented in the previous section. The experiment carried out is presented, starting with the setup, and concluding with some results.

The deliverable culminates with some conclusions on the experiments carried out regarding recovery mechanisms for CPSoS.

1.2 Deviations from the original Description in the Grant Agreement Annex 1 Part A

1.2.1 Description of work related to deliverable in GA Annex 1 – Part A

There are no deviations with respect to work of this deliverable.

1.2.2 Time deviations from original planning in GA Annex *1 – Part A*

This deliverable has been delayed for 3 months, which was amended in the Grant Agreement.

1.2.3 Context deviations from the original plan in GA Annex 1 – Part A

There are no deviations from the Annex 1.

2 INTRODUCTION

At runtime, a Cyber-Physical System (CPS) can reach an erroneous state, outside its normal operation. Once an uncertain or an erroneous situation has been detected it is crucial to react, preferably in an automatic way, to recover the correct system state.

In order to enhance the availability and reliability of CPSoS at runtime, the challenge in this task is to provide dynamic and safe recovery mechanisms. When activating the recovery and adaptation mechanism, several aspects need to be considered, such as the current status of the CPSoS itself or environment status. It must be ensured that no hazardous points are introduced into the system, and the runtime adaptation process itself must also be safe.

In this deliverable we propose and validate a recovery system that can be applied on the operational time of CPSoS, solving the aforementioned limitations

3 BACKGROUND AND STATE OF THE ART ON TEST ORACLES

In this section, we discuss the research done in order to accomplish this task. First, we introduce the background of the project, and then, we show different approaches found in the literature so far.

3.1 Background

The research carried out in this task began by looking at potential approaches for dynamic software adaptation such as the one presented by Gomaa et al. [9]. They describe software adaptation patterns that consist of interaction models and state machine models. In this solution, they describe the pattern to perform the adaptation: a Generalized Adaptation State Machine. They modelled a basic adaptation by a state machine with three states: Active (normal-mode of operation), Passive (stopped the initiated transition and will not initiate new transitions) and Quiescent (no longer operational). The initial idea for this task was to implement and adapt a similar solution for the context of CPSoS.

In MGEP, related to this work or this idea of recovery mechanisms, at the beginning of the project there was already a solution that was able to give information in terms of control model at runtime. There was also a solution capable of making adaptations at runtime but the limitation was that this adaptation or action was always to make use of another predefined control in any undesired situation [8].

Therefore, there was a need to investigate the possibility of making or activating adaptation or recovery actions in operation, but depending on the detected case, so that the decision would be more dynamic. Based on this situation, this area of work was identified.

3.2 State of the Art

During the task, different aspects about the recovery mechanisms have been analysed before starting the implementation of the final solution. Once an undesirable or an uncertain situation is detected, some actions have to be taken and different approaches have been studied to see which one could be used in the project.

3.2.1 RCV (Automatic Runtime Error Repair and Containment via Recovery Shepherding)

RCV [1] is used to tackle divide-by-zero and null-dereference errors, so they do not stop with a fatal error the execution of a program. For doing so, it uses recovery shepherding, which is attached to the execution of a program that generates the previously mentioned errors while invoking from the command line. The application runs unmodified until the two possible errors happen, where the lightweight runtime monitor of RCV calls the recovery shepherding to act.

The lightweight runtime monitor sets two signal handlers for the SIGFPE and SIGSEGV errors (divide-by-zero and null-dereference) and ensures that the information does not leave the process in which originate (if some data tries to be sent between processes or data is tried to be stored, it takes certain measures), as well as wrapping heap management calls for ensuring correct memory management (without memory leaks). When one of the two errors occurs, it invokes the recovery shepherding. This component ensures that all the subsequent errors generated by the first do not stop or suppose a problem for the execution of the program. The influenced variables are also monitored by RCV to ensure they do not deviate much from the expected execution. Once the influence is detected to be cleared, the recovery shepherding detaches from the program. It also contains errors from communicating with other processes apart from the one with the error (making some exceptions or adjustments to avoid malfunctioning). Therefore, it could be said that the recovery shepherding is formed by the recovery runtime, the influence tracking and the error containment.

This recovery mechanism is not directly applicable to the environment that we are considering in ADEPTNESS but it could help as an inspiration to how to react once an error or uncertain situation is detected.

3.2.2 RACE project RTE (A Safety Aware Run-Time Environment for Adaptive Automotive Control Systems)

To achieve fault tolerance in critical features of software functionalities and features of modern cars, the electrics and electronics architecture implementing them must provide built-in mechanisms to do so. For that, in the RACE project [2], the Plug&Play technique is used to provide additional services or integrate new components, like in a SOA. It is necessary, however, to have a RTE that provides a set of safety-related mechanisms, like health monitoring and diagnosis, benefiting from early error detection and a unified error handling. The RTE must provide functionalities to allow obtaining reliability, safety and security. Instead of following a sender-receiver approach, publish-subscribe is used to handle the data. As dynamic data makes it difficult to provide real-time guarantees, during the Plug phase, data paths are reconfigurated, so that in the Play phase, static data routes are used to avoid inefficient route lookup. The different components of the approach (or, at least, the most significant ones) are the following ones:

- the Data Handler: to store system data corresponding to safety and quality attributes, from which all the communication between applications passes through (it also can restrict access to different data elements),
- the Data Flow Monitor; for plausibility and consistency checks of data, Scheduling and execution of functions for deterministic time-triggered execution of applications and RTE components in a cyclic manner with a predefined minor cycle length,
- the Health Monitoring: for various monitoring of components, tests and plausibility checks, built around many different concepts, a Platform supervision to handle hardware and software faults,
- a Platform Mode Manager: to monitor the presence and trigger reconfiguration of computing nodes in the network, the Application State Manager, which is based on application states, and lastly,
- the Safety Manager: a specialised configuration component, which is triggered by the Plug&Play Manager during instantiation of a new safety-critical application.

All this RTE considers techniques to satisfy the requirements of the ASIL D system, defined in ISO 262626.

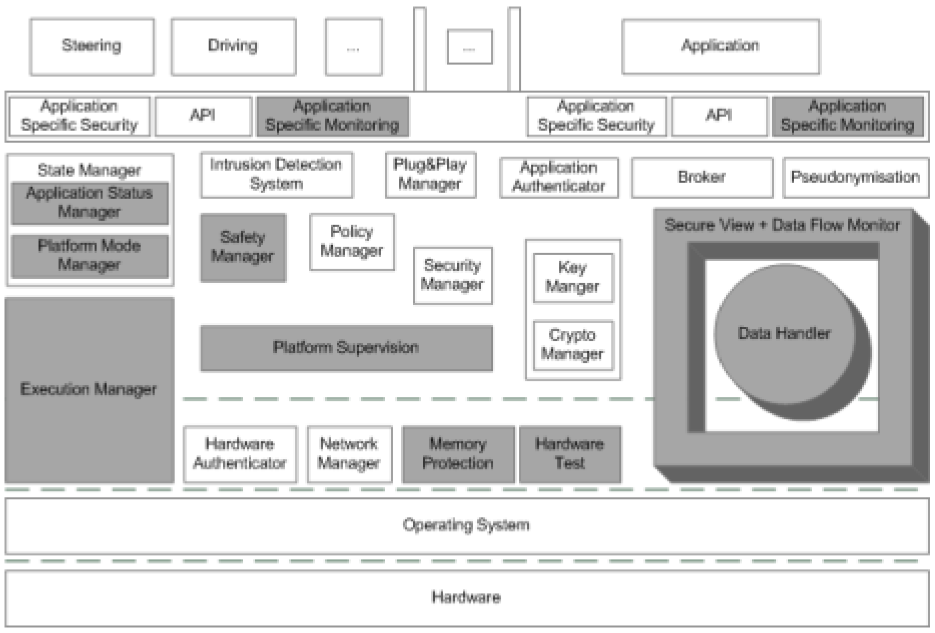


Fig. 1: Architecture of the solution

In this case, the work shows an architecture that could be used when developing the Recovery microservice for ADEPTNESS ecosystem.

3.2.3 *SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs*

SPECS [3] system is a low-overhead solution to the problem of processor implementation errata. SPECS employs a lightweight dynamic verification strategy for doing so. SPECS is implemented in an open-source processor as a hardware prototype. Its main functionality is to solve security-critical errata, working only ISA-level states.

The security-critical erratas are those bugs that have the potential to weaken the security of software by allowing unprivileged code to modify or control privileged processor state. SPECS adds a small amount of additional hardware to the processor that dynamically verifies a subset of processor functionality. The goal is to protect software from security-critical processor bugs, especially those that it cannot protect itself from efficiently.

SPECS works by enforcing instruction-set-specification-derived invariants on security-critical processor state dynamically. The invariants are enforced by a series of simple assertions in hardware. The assertions analyse ISA-level state and events to validate in-flight ISA-level state updates. If an in-flight state violates a SPECS invariant, a combination of assertions fires that then triggers an exception that squashes the inconsistent in-flight state. From here, existing software recovery/repair mechanisms take over, while SPECS continues to work in the background.

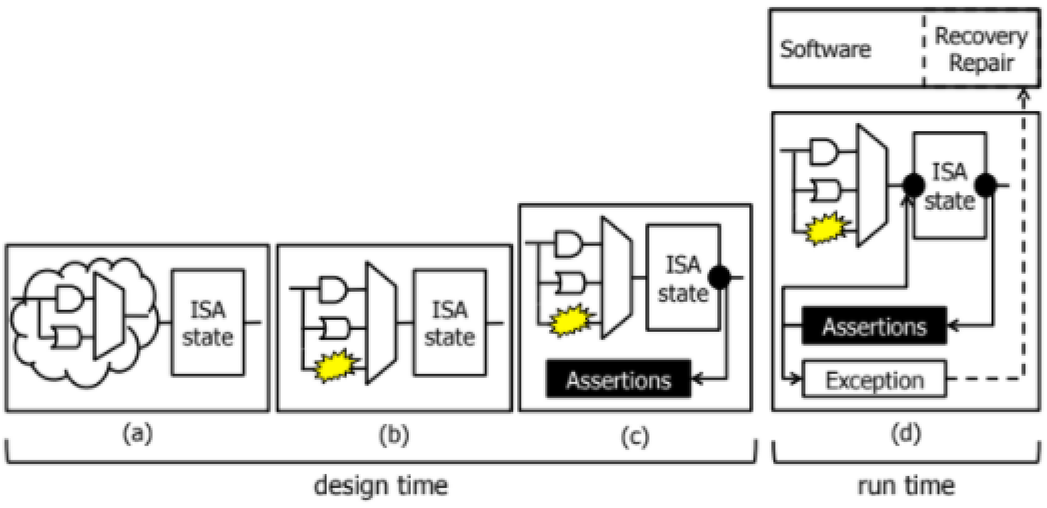


Fig. 2: SPECS: steps at different stages

In this case, the idea of having some assertions at runtime to check what is happening and how to repair the system helped to define the solution that is going to be shown in the document. The technology itself was not directly able to be applied in the solution.

3.2.4 ARES (Automatic Runtime Recovery via Error Handler Synthesis)

ARES [4] is a lightweight approach designed to support runtime error recovery, mainly focusing on handling unanticipated errors. Two lightweight strategies are followed to synthesise error handlers: error transformation and early return.

The approach followed by ARES to tackle the Automatic runtime recovery (ARR) necessity is the following one:

> The approach is built on a Java Virtual Machine (JVM), and executes the program where the errors should be detected in normal mode. When an error of interest for ARES occurs, the JVM is stopped and ARES takes over the execution to start the recovery process. Afterwards, a synthesiser generates a set of candidate solutions for recovery, which are known as error handlers, based on the thrown exception.
>
> After that, the Java PathFinder (JPF) is used as a sandbox, so it makes a copy of the state of the program just to enter the buggy state, and each of the different generated error handlers are tested, which are then ranked so the most promising one can be picked. The ranking heuristic is designed based on the observation that the most promising error handler usually outperforms others in that they have fewer discarded stack frames and longer JPF testing. Lastly, the patcher will realise the error handler returned by the ranker in the host JVM on the fly, leaving the program execution resume after that. If a synthesised error handler is applied for recovery, ARES does not persist it in the buggy method for future executions, as the error handler is synthesised based on the context in which the error manifests.

Two implementations exist: the first one is done on Java HotSpot VM, distributed as the default JVM vendor for popular Linux distributions, and all the functionalities have been integrated in this implementation. The second one is done on Android ART, but this implementation does not use JPF due to a different bytecode representation and a different layout of stack frames. Therefore, a conservative strategy has been followed.

In this case, this error was considered because they were also trying to detect and recover uncertain situations.

3.2.5 SmarterContext (Runtime Evolution of Highly Dynamic Software)

SmarterContext [5] is a service-oriented context monitoring infrastructure that exploits self-adaptation techniques to evolve at runtime with the goal of guaranteeing the relevance of monitoring strategies with respect to changes in monitoring requirements. SmarterContext evolves at runtime with the goal of monitoring the satisfaction of changing QoS contracts in an e-commerce scenario. Changes in contracted conditions correspond to either the addition or deletion of quality attributes to the contracted service level agreement, or the modification of desired conditions and corresponding thresholds.

SmarterContext relies on behavioural and structural self-adaptation techniques to realise runtime evolution. Behavioural adaptation comprises mechanisms that tailor the functionality of the system by modifying its parameters or business logic, whereas structural adaptation uses techniques that modify the system's software architecture. SmarterContext implements behavioural adaptation by modifying existing monitoring conditions or adding new context types and reasoning rules at runtime, and structural adaptation by (un)deploying context gatherers and context processing components. All these operations are realised without requiring the manual development or deployment of software artifacts, and minimising human intervention.

SmarterContext is composed of different components, each with its function to fulfil. The most important ones are

- GoalsManager: allow to modify system goals at runtime and define and modify context reasoning rules,
- ContextManager and DynamicMonitoringInfrastructure: both implement the context monitoring functionalities,
- ContextGatheringAndPrepocessing and ContextMonitoring: deployed as part of the runtime evolution process, and
- MFLController: includes the artifacts that implement the feedback loop in charge of controlling the runtime evolution of our context management infrastructure), among many others.

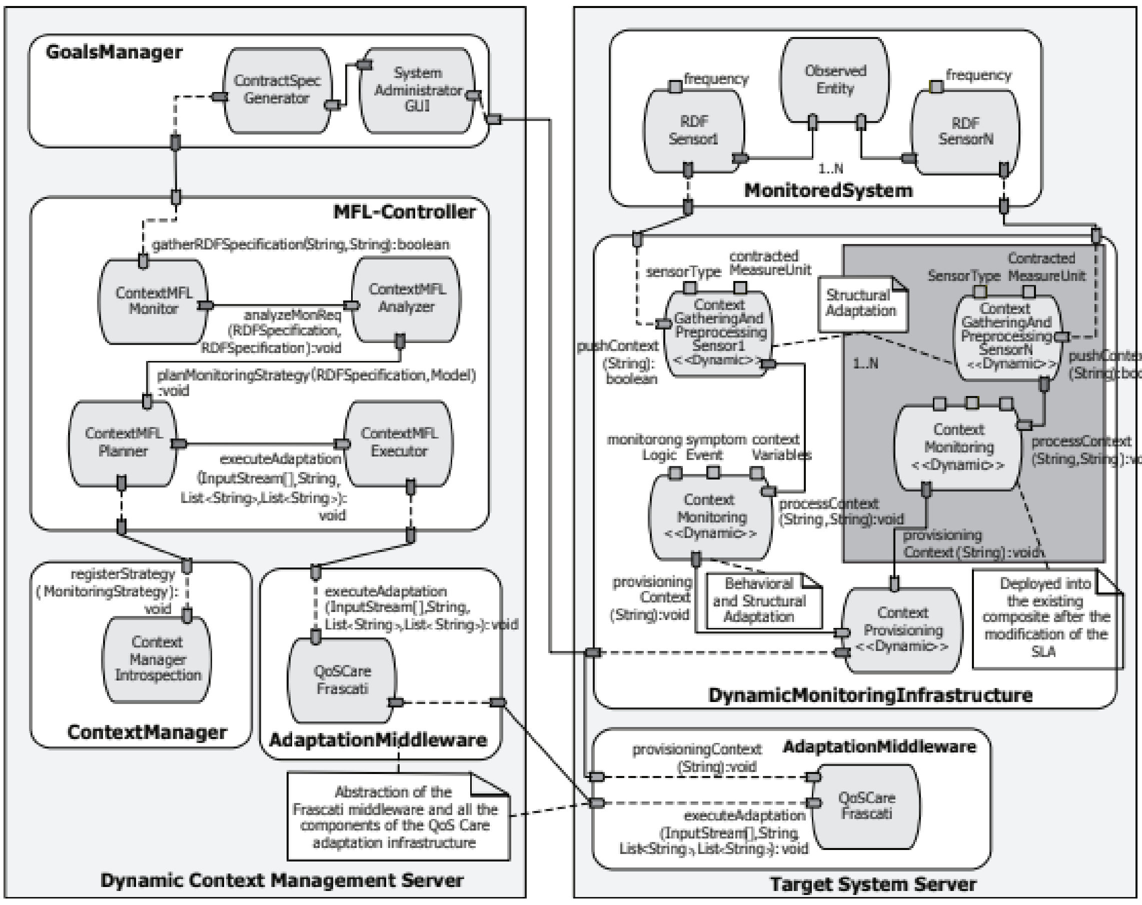

GoalsManager
ContractSpec Generator
System Administrator GUI
MFL-Controller
gatherRDFSpecification(String,String):boolean
ContextMFL Monitor
ContextMFL Analyzer
analyzeMonReq (RDFSpecification, RDFSpecification):void
planMonitoringStrategy(RDFSpecification,Model):void
ContextMFL Planner
ContextMFL Executor
executeAdaptation (InputStream[],String, List<String>,List<String>): void
registerStrategy (MonitoringStrategy): void
Context Manager Introspection
ContextManager
executeAdaptation (InputStream[],String, List<String>,List<String>):void
QoSCare Frascati
AdaptationMiddleware
Abstraction of the Frascati middleware and all the components of the QoS Care adaptation infrastructure
Dynamic Context Management Server
frequency
Observed Entity
RDF Sensor1
RDF SensorN
1..N
MonitoredSystem
sensorType
contracted MeasureUnit
Context GatheringAnd Preprocessing Sensor1 <<Dynamic>>
Structural Adaptation
SensorType
Contracted MeasureUnit
Context GatheringAnd Preprocessing SensorN <<Dynamic>>
pushContext (String): boolean
1..N
Context Monitoring <<Dynamic>>
processContext (String,String):void
monitorong symptom context Logic Event Variables
processContext (String,String):void
provisioning Context(String):void
provisioning Context (String):void
Behavioral and Structural Adaptation
Context Provisioning <<Dynamic>>
Deployed into the existing composite after the modification of the SLA
DynamicMonitoringInfrastructure
provisioningContext (String):void
AdaptationMiddleware
QoSCare Frascati
executeAdaptation (InputStream[],String, List<String>,List<String>): void
Target System Server

Fig. 3: Architecture of SmarterContext

This work was considered as an interesting input because they were also analysing the dynamics of recovery systems. It is not only to recover to predefined states but also they consider the situation where the system has the capability to adapt or evolve.

3.2.6 ClearView

ClearView [6] is a system designed for automatically correcting errors in deployed commercial off-the-self (COTS) software systems with high availability requirements. It patches running applications without requiring restarts or perturbing the execution with no human interaction or intervention. It is composed of 5 components. The first one is Learning, which observes the application during normal execution and infers a model from it. The model (which is a collection of properties or invariants) becomes more accurate the more it is observed. Monitoring is the second component, which classifies each execution as either normal or failed by using a set of monitors.

ClearView is designed to incorporate arbitrary monitors, but does not eliminate all failures, only those that a monitor detects. The next component is Correlated Invariant Identification, which detects some correlated invariants that are violated when a failure occurs but are satisfied during normal executions. Candidate Repair Generation generates a set of candidate repair patches for each correlated invariant. The patches must aim to correct the execution after the first error has occurred but before any effects have propagated. The last component is the Candidate Repair Evaluation, whose purpose is to check if the patch applied is effective. This is because a patch may have no effect, or even a negative effect, on the patched application. Therefore, ClearView evaluates the patches by continuously observing the patched application running, and ranks the different possible patches to choose the more adequate one, to minimise the possibility of negative effects. ClearView needs to check the exploit or error several times before it creates and applies patches. It is also designed to work for application communities, where if one machine finds the patch to an error, the other machines in the community also receive the patch so they can apply it, even if they were not attacked or affected.

This solution was used as inspiration for the work to be developed because our system will also have different oracles and uncertainty detectors so a similar way to evaluate the criticality of the situation and the actions in consequences were also taken into consideration.

3.2.7 ASSURE (Automatic Software Self-healing Using REscue points)

ASSURE [7] is a system that introduces rescue points that recover software from unknown faults while maintaining both system integrity and availability, by mimicking system behaviour under known error conditions. Rescue points are locations in existing application code for handling a given set of programmer-anticipated failures, which are automatically repurposed and tested for safely enabling fault recovery from a larger class of faults.

When a fault occurs at an arbitrary location in the program, ASSURE restores execution to an appropriate rescue point and induces the program to recover execution by virtualizing the program’s existing error-handling facilities. Rescue points virtualize error handling by creating a mapping between the potentially infinite set of errors that could occur during a program’s execution and the limited set of errors that can be handled by the program’s code. Thus, a failure that would cause the program to crash is translated into a “return with an error” from an error-handling function along the execution path in which the fault occurred. To reproduce a fault, ASSURE uses execution logging and periodic checkpointing to record and recreate the sequence of events that led to the manifestation of the fault. ASSURE first identifies candidate rescue points in an application offline using fuzzing, then implements, tests, and deploys rescue points online in response to the occurrence of faults using an Observe Orient Decide Act feedback loop.

ASSURE does many things, like operating without human intervention or working with polymorphic input behaviour or encrypted traffic, among many others. Assure works as follows: when a fault is detected during execution, the latest application checkpoint state along with the log of all the inputs since that checkpoint is transferred to a triage system, a shadow deployment of the application, where the fault is analysed. ASSURE then carries out an automated process whose goal is to identify a suitable rescue point to which the application can recover if the fault re-manifests. Once a candidate rescue point is selected, ASSURE confirms that it is suitable for deployment by verifying that it satisfies three criteria: survivability, correctness and performance. As soon as a suitable rescue point is verified, ASSURE produces a remediation patch that is dynamically applied to the software while the application is executing on the production system. The patch instantiates a rescue point inside the application to protect the application against the recurrence of the particular fault. The modified application will trigger a checkpoint whenever execution reaches the rescue point.

This work also was analysed to think about the recovery strategies once a no desirable situation is detected.

4 RECOVERY MECHANISMS: CPSOS'S UNDESIRABLE BEHAVIOUR AVOIDANCE AT RUN-TIME

Based on the literature revised, the following solution is proposed in order to enhance the availability and reliability of CPSoS at runtime.

In the ADEPTNESS context, the recovery microservice is responsible for managing the execution of recovery actions once an uncertain situation has been detected or a verdict in an oracle has failed. As can be seen in Figure 4, it uses the results obtained from the validation agent (Test oracles proposed in Task 3.1) and uncertainty agent (Investigated in Task3.2), in order to assess the status of the CPSoS. Then, based on the Recovery Rules and depending on the information it receives, the Recovery Microservice decides to launch or not a specific recovery action. The launched recovery actions are sent their corresponding microservices in order to be executed.

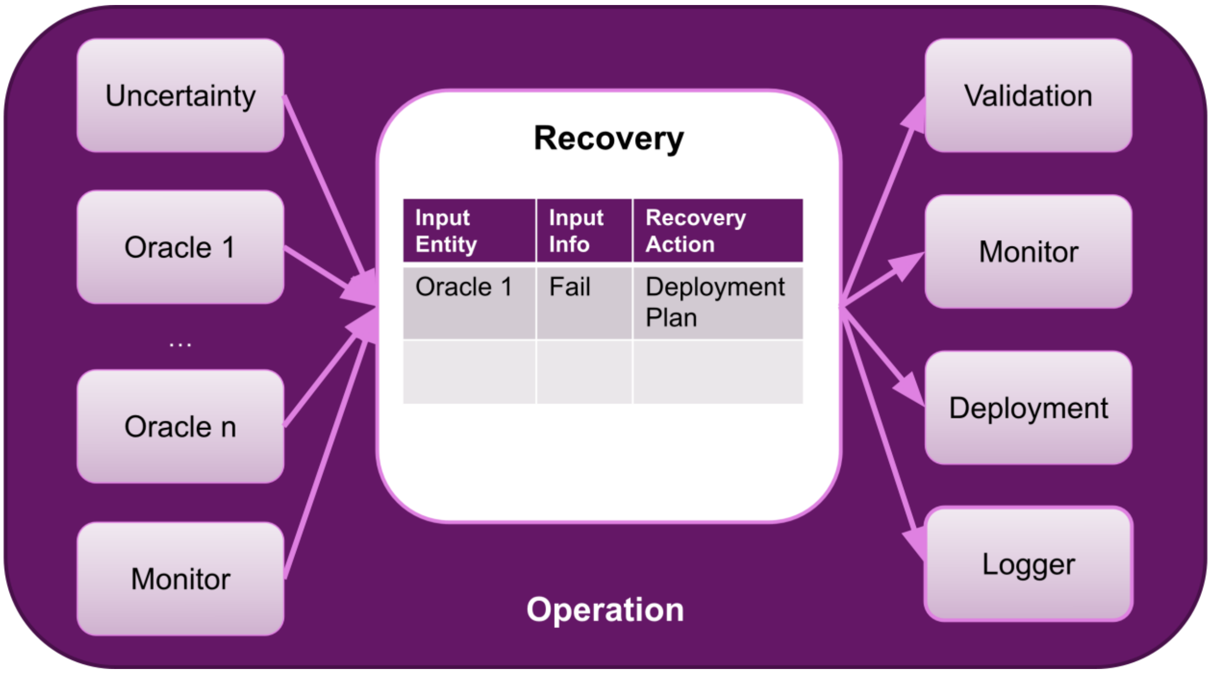


Fig. 4: Overview of the approach

The behaviour of the recovery microservice is described in a rule definition language. Once the rules have been defined, the source code of the rule engine is automatically generated and compiled. Finally, the resulting binary is integrated into a docker container in order to be delivered to the CPSoS. The following section describes this process in more detail.

4.1 Rule Definition

We have developed a rule definition language that will allow a practitioner with no programming knowledge to define the behaviour of the recovery agent.

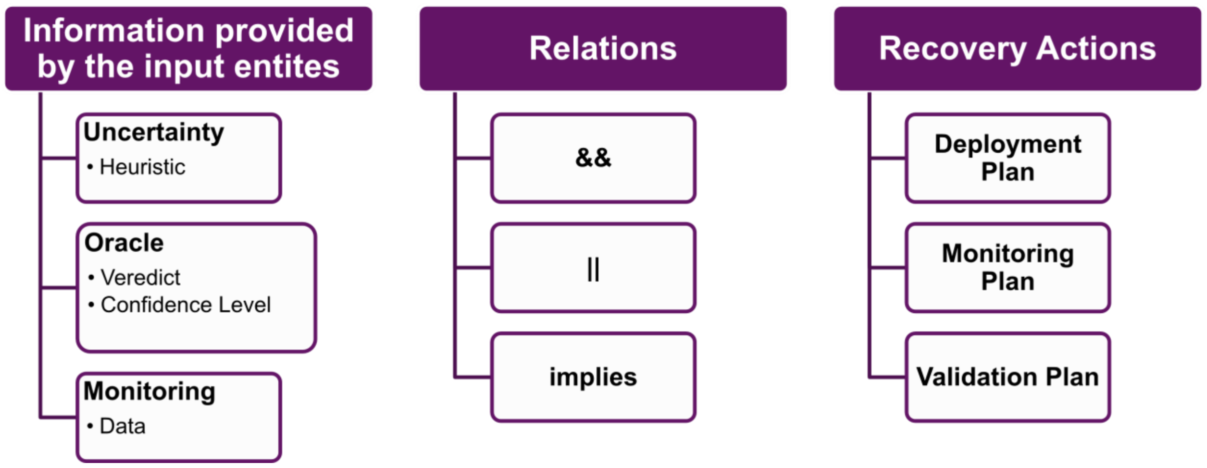


Fig. 5: Syntax of rule definition language

As shown in FIgure 5, the syntax of rule definition language is composed by input entities, recovery actions and relations between them. On the left-hand side of the expression, the information given by the input entities is transformed into boolean statements. Given a confidence level, oracle verdicts are reduced to true or false statements, as well as the uncertainty level (given by the uncertainty agent) or the monitoring data (provided by the monitoring agent). All those boolean statements are combined by means of AND (&&) and OR (||) logical operators. Additionally, parenthesis can be used in order to change the order of execution of these logical operators. Then, on the right-hand side of the expression, and separated by the implies keyword, the Recovery Action that will be used in case of a faulty situation is defined. Recovery Actions involve a logical and structural model reconfiguration, to represent newly activated behaviours. Five recovery actions are defined:

- validation_plan: triggers a validation plan executed by the validation agent.
- monitoring_plan: asks the monitoring agent to start a monitoring plan.
- deployment_plan: the deployment agent will start a new deployment.

The decision about which of them will be activated depends on the inputs that the Recovery Agent will receive from the oracles and uncertainty modules.

An example of a rule written in our rule definition language can be seen in the following line:

O1.fail.90 && (O2.pass.90 || U1.80) implies Validation_Plan

In this example, if the first oracle fails with a confidence level greater than 80 and the second oracle passes with confidence level greater than 90 or the uncertainty is greater than 80 the recovery actions will be triggered. In this case, a validation plan will be executed by the validation agent.

4.2 Code generation and checker

Once the use-case-specific rules are defined using the aforementioned rule definition language, the rule generator provides the corresponding C++ code. Then, the generated code is integrated into the checker which can be used stand-alone. Figure 6 shows this process.

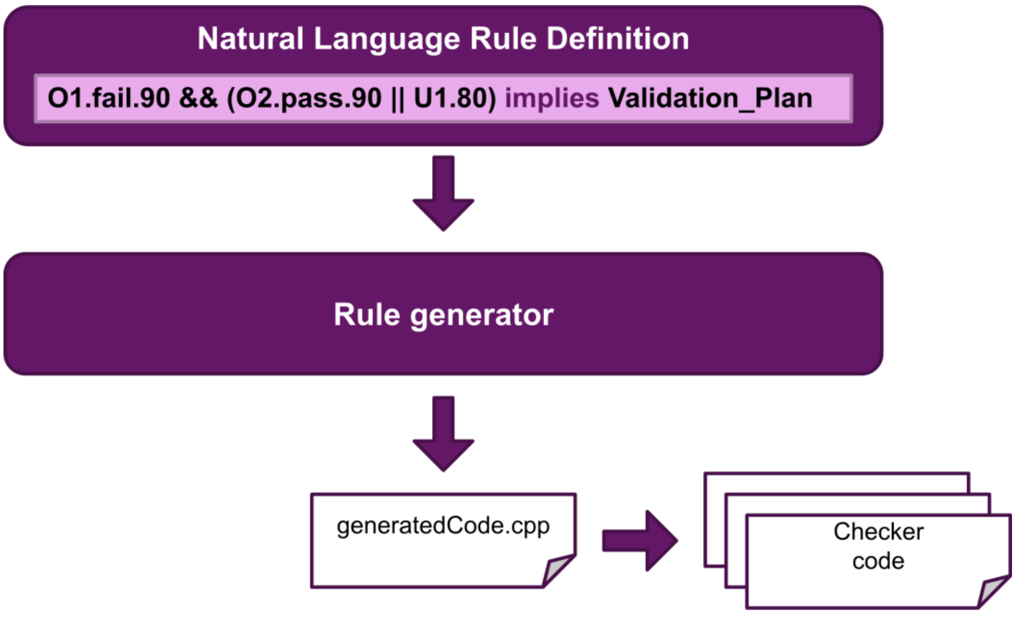

Natural Language Rule Definition
O1.fail.90 && (O2.pass.90 || U1.80) implies Validation_Plan
Rule generator
generatedCode.cpp
Checker code

Fig. 6: Code generation process

However, in order to act as a correctness checker in the ADEPTNESS context, this code must be integrated into the Recovery Microservice template.

4.3 Recovery MS generation based on the ADEPTNESS template

In order to create the microservice, we need the checker code together with the generated code, and the Recovery Microservice template.

As can be seen in Figure 7, we created a Gitlab pipeline that compiles those sources of code and generates the microservice binaries for several architectures. Then, those binaries are included in different docker containers, ready to be deployed in their corresponding hardware.

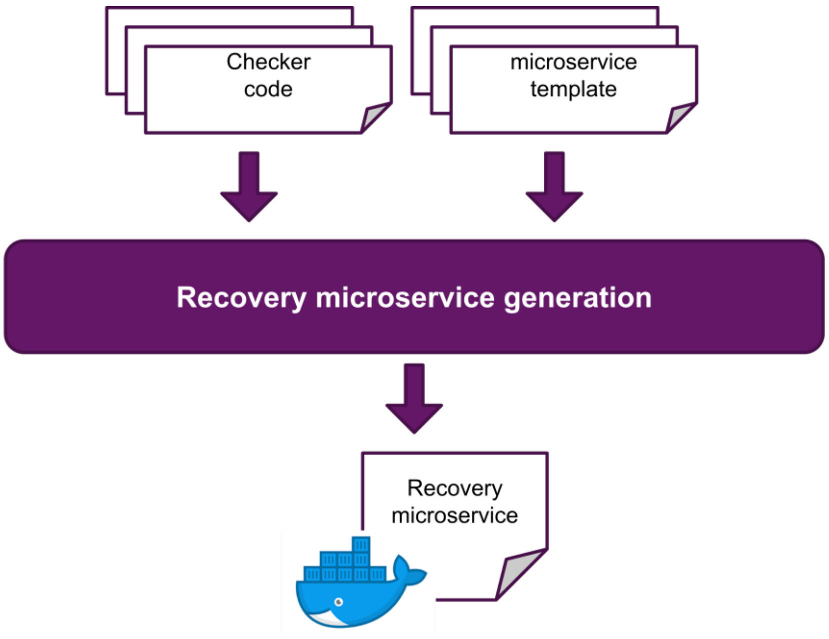


Fig. 7: Recovery microservice integration process

At this time, we have the use-case-specific checker generated and integrated in the microservice and we are able to detect uncertain or undesirable situations.

4.4 Recovery MS interactions at run-time

As explained in the previous section, the Recovery MS is able to detect undesired or unexpected situations but to do that it must be connected to another ADEPTNESS MS that are also activated at runtime.

On the one hand, this MS needs inputs from the Oracles that are publishing verdicts of the correctness of the controls that they are checking. In addition, Monitor MSs are also monitoring the runtime systems and providing this information at runtime. Finally, we can also obtain some information from Uncertainty MS. It is true that at this moment, there are discussions about if this last MS has to be integrated with the Oracles or be working as an independent MS.

On the other hand, Recovery MS is going to publish recovery actions when something uncertain or undesirable happens. This is why it also needs to be in communication with the MSs that need to make or perform these recovery actions.

At this moment, Oracles integration has been developed and the Recovery MS is checking verdicts that are published (MQTT) by oracles. We have validated that the all the steps are working properly and the Recovery MS is able to publish the Recovery actions to adapt the system to a safe state. So, once a failed or inconsistent verdict is detected the Recovery MS launches automatically monitoring plans or deployment plans (depends on the rule that was not fulfilled).

In addition, it publishes the recovery actions (the ones launched in the previous step) by MQTT and in the Figure 8 we can see the Sequence Diagram of the Recovery MS.

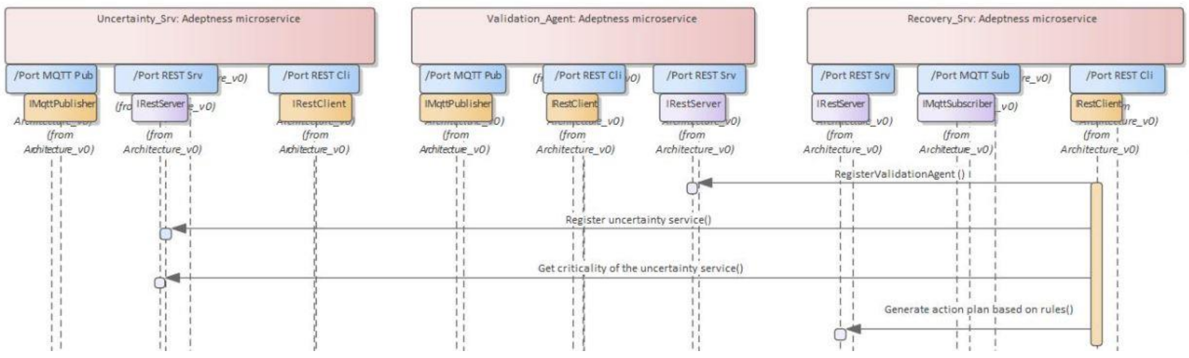


Fig. 8: Interaction with recovery microservice

5 VALIDATION OF THE PROPOSED APPROACH

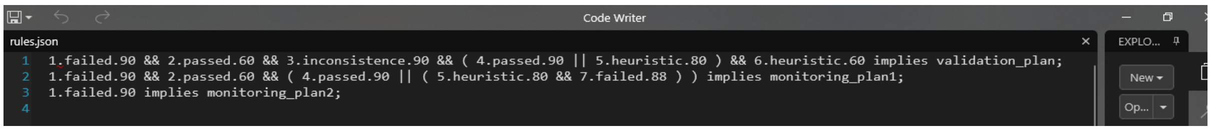
In the previous section, we have presented the main architecture and the way that the MS has been developed. In this section, we want to show how we have implemented and validated it using a specific use case.

This use case consists of 7 input entities:

- 1, 2, 3, 4: oracles. Each oracle publishes a verdict (failed, passed, inconsistency) and a confidence level.
- 5, 6, 7: uncertainty modules: Each uncertainty module publishes a heuristic measure.

Regarding the output entities, this use case will publish information that will have effect on validation agent (other plan have to be activated) and on the monitoring plan (depending on the not fulfilled rule, one or other monitoring plan will be activated)

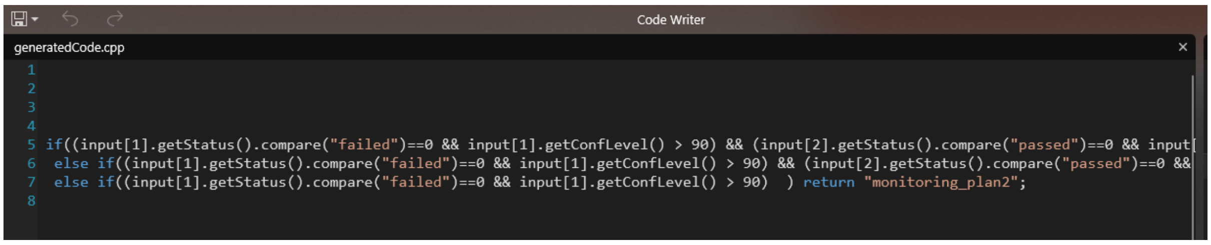
The proposed approach has been validated in a laboratory environment and we have defined the next rules:



```
1.failed.90 && 2.passed.60 && 3.inconsistence.90 && ( 4.passed.90 || 5.heuristic.80 ) && 6.heuristic.60 implies validation_plan;
1.failed.90 && 2.passed.60 && ( 4.passed.90 || ( 5.heuristic.80 && 7.failed.88 ) ) implies monitoring_plan1;
1.failed.90 implies monitoring_plan2;
```

Fig. 9: Recovery MS: Rules.json

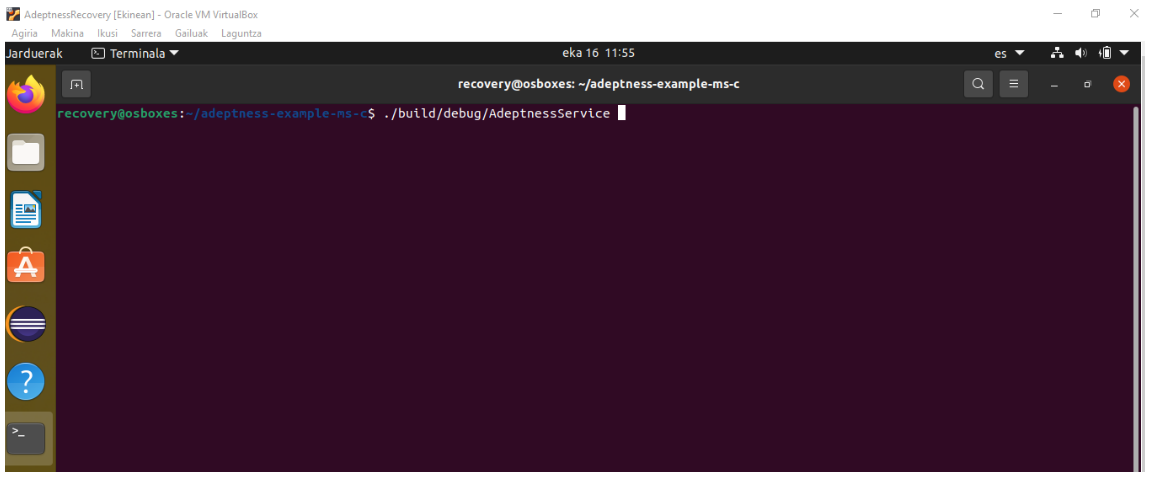
The next step in the process is to generate the code automatically. To do that, we have to execute the checker generator C application generated to do that and, in this case, this was the result:



```




if((input[1].getStatus().compare("failed")==0 && input[1].getConfLevel() > 90) && (input[2].getStatus().compare("passed")==0 && input[
 else if((input[1].getStatus().compare("failed")==0 && input[1].getConfLevel() > 90) && (input[2].getStatus().compare("passed")==0 &&
 else if((input[1].getStatus().compare("failed")==0 && input[1].getConfLevel() > 90)  ) return "monitoring_plan2";

```

Fig. 10: Recovery MS: generated cpp code for the use case

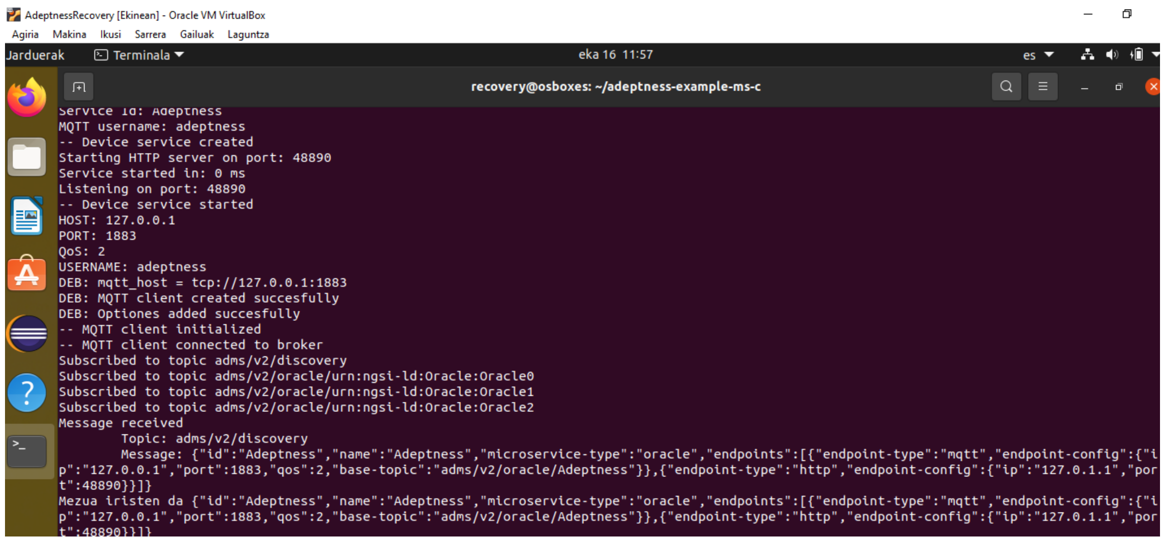
This generated code is integrated in the Recovery MS code (template). There is a specific C++ function where we have to copy this code. At this moment, this process is manual but our objective is to automate this process in order to avoid any human error. Once the specific code for the use case is integrated, we have to build the Recovery MS entity.



```
recovery@osboxes:~/adeptness-example-ms-c$ ./build/debug/AdeptnessService
```

Fig. 11: Recovery MS building

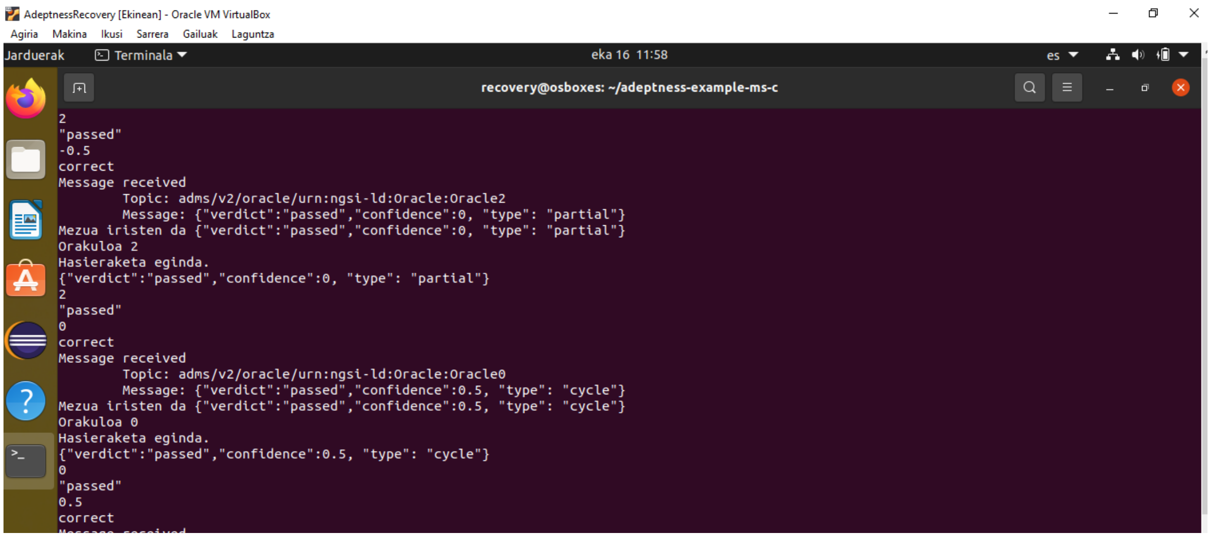
The last point in this process is to check if this Recovery MS works as expected at runtime. For that, we execute the MS. Before executing this MS, we also need to put working the input entities (7 entities, oracles and virtual uncertainty modules). Thus, when Recovery MS starts working, it receives the information published by them. In Figure 12 we can see how the Recovery MS is starting.



```
Service id: Adeptness
MQTT username: adeptness
-- Device service created
Starting HTTP server on port: 48890
Service started in: 0 ms
Listening on port: 48890
-- Device service started
HOST: 127.0.0.1
PORT: 1883
QoS: 2
USERNAME: adeptness
DEB: mqtt_host = tcp://127.0.0.1:1883
DEB: MQTT client created succesfully
DEB: Optiones added succesfully
-- MQTT client initialized
-- MQTT client connected to broker
Subscribed to topic adms/v2/discovery
Subscribed to topic adms/v2/oracle/urn:ngsi-ld:Oracle:Oracle0
Subscribed to topic adms/v2/oracle/urn:ngsi-ld:Oracle:Oracle1
Subscribed to topic adms/v2/oracle/urn:ngsi-ld:Oracle:Oracle2
Message received
        Topic: adms/v2/discovery
        Message: {"id":"Adeptness","name":"Adeptness","microservice-type":"oracle","endpoints":[{"endpoint-type":"mqtt","endpoint-config":{"i
p":"127.0.0.1","port":1883,"qos":2,"base-topic":"adms/v2/oracle/Adeptness"}},{"endpoint-type":"http","endpoint-config":{"ip":"127.0.1.1","por
t":48890}}]}
Mezua iristen da {"id":"Adeptness","name":"Adeptness","microservice-type":"oracle","endpoints":[{"endpoint-type":"mqtt","endpoint-config":{"i
p":"127.0.0.1","port":1883,"qos":2,"base-topic":"adms/v2/oracle/Adeptness"}},{"endpoint-type":"http","endpoint-config":{"ip":"127.0.1.1","por
t":48890}}]}
```

Fig. 12: Starting the Recovery MS

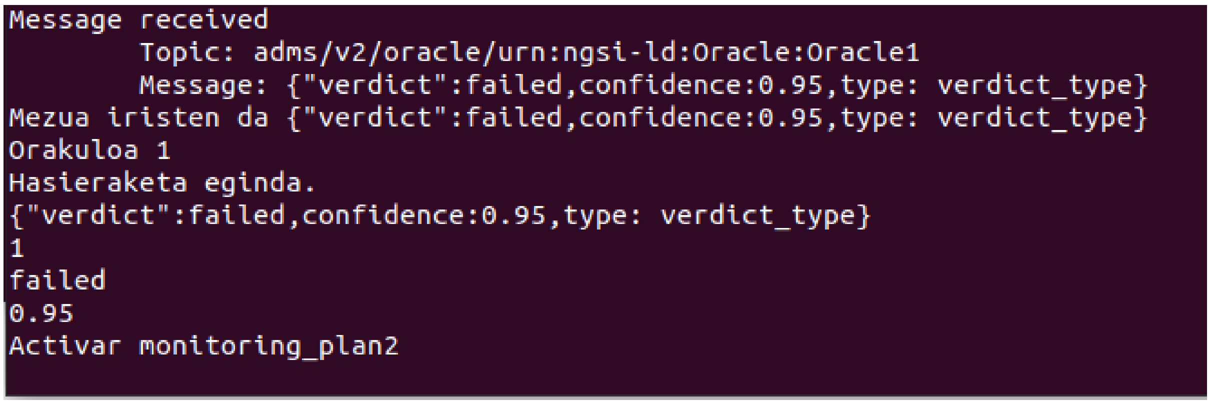
Lastly, we verify if the checker is making the expected decisions. Figure 13 shows how the Recovery MS is receiving information from the Oracles and also which has been the decision after checking the rules.


AdeptnessRecovery [Ekinean] - Oracle VM VirtualBox
Agiria Makina Ikusi Sarrera Gailuak Laguntza
Jarduerak
Terminala
eka 16 11:58
es
recovery@osboxes: ~/adeptness-example-ms-c

```
2
"passed"
-0.5
correct
Message received
        Topic: adms/v2/oracle/urn:ngsi-ld:Oracle:Oracle2
        Message: {"verdict":"passed","confidence":0, "type": "partial"}
Mezua iristen da {"verdict":"passed","confidence":0, "type": "partial"}
Orakuloa 2
Hasieraketa eginda.
{"verdict":"passed","confidence":0, "type": "partial"}
2
"passed"
0
correct
Message received
        Topic: adms/v2/oracle/urn:ngsi-ld:Oracle:Oracle0
        Message: {"verdict":"passed","confidence":0.5, "type": "cycle"}
Mezua iristen da {"verdict":"passed","confidence":0.5, "type": "cycle"}
Orakuloa 0
Hasieraketa eginda.
{"verdict":"passed","confidence":0.5, "type": "cycle"}
0
"passed"
0.5
correct
```

Fig. 13: Recovery MS: Correct messages, no need to activate recovery actions

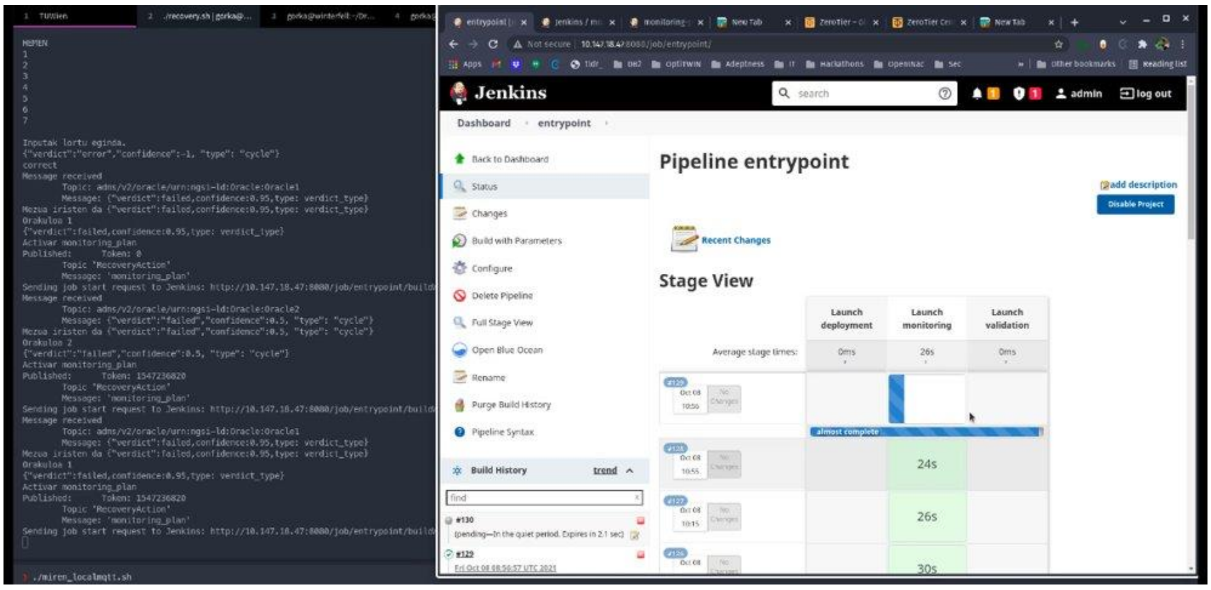
When inserting faults in the system, the Recovery MS was able to detect those circumstances and start the recovery actions. As can be seen in Figure 14, it successfully publishes the corresponding recovery actions using the MQTT interface. In this case, the monitoring_plan2 action was published.



```
Message received
        Topic: adms/v2/oracle/urn:ngsi-ld:Oracle:Oracle1
        Message: {"verdict":failed,confidence:0.95,type: verdict_type}
Mezua iristen da {"verdict":failed,confidence:0.95,type: verdict_type}
Orakuloa 1
Hasieraketa eginda.
{"verdict":failed,confidence:0.95,type: verdict_type}
1
failed
0.95
Activar monitoring_plan2
```

Fig. 14: Recovery MS: Fault detected

Figure 15 shows how Jenkins has received this published action and the corresponding monitoring plan has been activated.



```
MEMEN
1
2
3
4
5
6
7

Inputak lortu eginda.
{"verdict":"error","confidence":-1, "type": "cycle"}
correct
Message received
        Topic: adns/v2/oracle/urn:ngsi-ld:Oracle:Oracle1
        Message: {"verdict":failed,confidence:0.95,type: verdict_type}
Mezua iristen da {"verdict":failed,confidence:0.95,type: verdict_type}
Orakuloa 1
{"verdict":failed,confidence:0.95,type: verdict_type}
Activar monitoring_plan
Published:      Token: 0
        Topic 'RecoveryAction'
        Message: 'monitoring_plan'
Sending job start request to Jenkins: http://10.147.18.47:8080/job/entrypoint/build
Message received
        Topic: adns/v2/oracle/urn:ngsi-ld:Oracle:Oracle2
        Message: {"verdict":"failed","confidence":0.5, "type": "cycle"}
Mezua iristen da {"verdict":"failed","confidence":0.5, "type": "cycle"}
Orakuloa 2
{"verdict":"failed","confidence":0.5, "type": "cycle"}
Activar monitoring_plan
Published:      Token: 1547236820
        Topic 'RecoveryAction'
        Message: 'monitoring_plan'
Sending job start request to Jenkins: http://10.147.18.47:8080/job/entrypoint/build
Message received
        Topic: adns/v2/oracle/urn:ngsi-ld:Oracle:Oracle1
        Message: {"verdict":failed,confidence:0.95,type: verdict_type}
Mezua iristen da {"verdict":failed,confidence:0.95,type: verdict_type}
Orakuloa 1
{"verdict":failed,confidence:0.95,type: verdict_type}
Activar monitoring_plan
Published:      Token: 1547236820
        Topic 'RecoveryAction'
        Message: 'monitoring_plan'
Sending job start request to Jenkins: http://10.147.18.47:8080/job/entrypoint/build

❯ ./miren_localmqtt.sh
```

Fig. 15: Recovery MS: Jenkins activates the Monitoring plan

As a conclusion, we can say that all the process has been validated in a laboratory environment and we can state that the current version of the Recovery MS is able to perform its objective of adding safety to the systems when uncertain and undesirable circumstances are detected.

6 SUMMARY

This Deliverable reports the work carried out in WP3 on Task 3.3 to provide recovery mechanisms for CPSoS and overall resilience improvement. Based on the previous research, we propose and validate a recovery system that can be applied on the operational time of CPSoS. For that aim, we developed a rule definition language that is able to express multiple hazardous situations. We have also developed a tool for the automatic transformation of the rules into C++ code, which can be integrated into the ADEPTNESS microservice template. Moreover, the microservice generation process has been automatized by means of a Gitlab pipeline. The resulting microservice successfully interacts with oracles as input entities and publishes the recovery actions through MQTT. We have also shown this process working in a laboratory environment.

Nevertheless, it is important to highlight that the current version of the Recovery microservice still requires some human intervention to generate the microservice. In addition, the rule engine is not able to infer the messages coming from the Monitoring and Uncertainty input entities. At this stage, this module is a prototype that is ready for being integrated with the rest of the ADEPTNESS ecosystem.

7 RISK REGISTER

Major risks were not identified.

8 QUALITY ASSURANCE

The executive board is the body for quality assurance. The procedure for review and approval of deliverable is described in Deliverable Report D8.1 – "Project handbook". The quality will be ensured by checks and approvals by WP Leaders as part of the executive board (see front pages of all deliverables).

9 ACKNOLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no. 871319.

10 REFERENCES

[1] Long, F., Sidiroglou-Douskos, S., & Rinard, M. (2014). Automatic runtime error repair and containment via recovery shepherding. ACM SIGPLAN Notices, 49(6), 227-238.

[2] Frtunikj, J., Rupanov, V., Camek, A., Buckl, C., & Knoll, A. (2014). A safety aware run-time environment for adaptive automotive control systems. In Embedded real-time software and systems (ERTS2).

[3] Hicks, M., Sturton, C., King, S. T., & Smith, J. M. (2015, March). Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs. In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 517-529).

[4] Gu, T., Sun, C., Ma, X., Lü, J., & Su, Z. (2016, September). Automatic runtime recovery via error handler synthesis. In 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 684-695). IEEE.

[5] Müller, H., & Villegas, N. (2014). Runtime evolution of highly dynamic software. In Evolving Software Systems (pp. 229-264). Springer, Berlin, Heidelberg.

[6] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. fai Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard, "Automatically patching errors in deployed software," in Symposium on Operating Systems Principles, 2009, pp. 87–102.

[7] Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., & Keromytis, A. D. (2009). Assure: automatic software self-healing using rescue points. ACM SIGARCH Computer Architecture News, 37(1), 37-48.

[8] Illarramendi, M., Etxeberria, L., Elkorobarrutia, X., & Sagardui, G. (2019, April). Runtime observable and adaptable UML state machines: models@ run. time approach. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (pp. 1818-1827).

[9] Gomaa, H., & Hashimoto, K. (2012, June). Dynamic self-adaptation for distributed service-oriented transactions. In 2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) (pp. 11-20). IEEE.

Disclaimer

Mondragon Unibertsitatea

Goi Eskola Politeknikoa
Escuela Politécnica Superior