Ref. Ares(2018)4998109 - 29/09/2018
JU GA# 692529

SAFE COOPERATING CYBER-PHYSICAL SYSTEMS USING WIRELESS COMMUNICATION

D4.2 - SAFECOP Prototype runtime mechanisms

Reporttype	**Deliverable D4.2**
Reportname	SafeCop Prototype runtime mechanisms
Dissemination level	CO
Report status:	Final
Version number:	1.0
Date of preparation:	2018/09/27

Revision chart and history log		
Version	Date	Reason
0.1	2018/09/23	First draft
0.2	2018/09/24	First review
0.3	2018/09/25	Updated Draft
0.4	2018/09/26	Final review
1.0	2018/09/27	Final release

Authors

David Pereira, Ricardo Severino, Luigi Pomante, Walter Tiberti, Maurizio Mongelli, Marco Muselli, Enrico Ferrari, Karl Meinke, Matteo Grotto, Carlo Brandolese, Luciano Bozzi, Fabio Del Forno, Leonardo Napoletani,

Table of contents

SAFE COOPERATING CYBER-PHYSICAL SYSTEMS USING WIRELESS COMMUNICATION 1

ReviRevision chart and history log 2

Authors 3

Table of contents 4

Abbreviations **5**

1 Introduction **6**

2 Prototype Runtime Mechanisms **7**

2.1 Run time manager function (CNR-IEIIT, Impara) *7*

2.2 Agilla2 (UNIVAQ) *9*

2.3 TinyWIDS (UNIVAQ) *11*

2.4 Run-time safety monitor for CO-CPS based on linear temporal logic (KTH) *13*

2.5 Safety integrity mechanisms to support run-time management (IBT Solutions) *16*

2.6 A Run Time Manager prototype in C++ language (ROT) *18*

2.7 RMTLD3Synth Runtime Verification Framework (ISEP) *22*

2.8 ROS based Runtime Monitoring Architecure (ISEP) *31*

2.9 Runtime Monitoring Architecture for Safety-Critical Systems (ISEP) *34*

ABBREVIATIONS

Abbreviation	Description
CO-CPS	Cooperative Cyber-Physical System
WSN	Wireless Sensor Networks
MAMW	Mobile Agent Middleware
IDS	Intrusion Detection System
LCU	Local Central Unit
RTM	Run-time Manager

1 INTRODUCTION

This deliverable is a "demonstrator" deliverable from WP4 and is a compendium of prototype runtime mechanisms targeting the support for the Runtime Manager of SafeCOP. These have been developed by different partners, and their usage varies depending on the target use cases, as can be seen in each of the prototypes description (some of which are described in more detail in D4.1).

The deliverable is structured as follows.

- Each runtime mechanism prototype is presented via an "Item Form" which collects the main details about the prototype;
- The actual prototype is available in binary executable format and/or as source code at the repository specified in the Item Form;
- The Item Form contains also information (references) about the method and/or tool, and additional documentation may also be uploaded in the repository.

The text below presents eight items that has been developed by the following seven partners: CNR-IEIIT, Impara, UNIVAQ, KTH, IBT SOLUTIONS, ROT, and ISEP

2 PROTOTYPE RUNTIME MECHANISMS

2.1 RUN TIME MANAGER FUNCTION (CNR-IEIIT, IMPARA)

Item Type

Run time manager function.

Item Name

Run time manager function coming from intelligible machine learning for safety/performance prediction of CO-CPS.

Reference Links

Paper [1].

Purpose of Item

Run time manager function for collision prediction in vehicle platooning.

Description of Item

Once machine learning has been applied at design time, a function is derived to be applicable at run time to infer safety conditions. In this case, the function applies to vehicle platooning and shows how collision prediction is provided according to system conditions monitored at run time. The function based on a set of Boolean rules may be easily implemented on the field. [Mon18] shows how the set of run time functions provide forecast with statistical error very close to zero, while keeping the largest set of working conditions.

The software related with the item form is as follows.

- SAFECOP_D42_CNR&Impara_RTMA model.c: the run time manager function.
- SAFECOP_D42_CNR&Impara_RTMA model with application.c: the run time manager function applied in a simulation context. Nothing prevents to adopt the run time manager function in a real device.
- SAFECOP_D42_CNR&Impara_RTMA model with application output.txt: the output provided by the application; it gives an example about the prediction of collision before it may happen.

SAFECOP exploitation and extensions

N/A

TRL

- Current TRL – 4 : Technology validated in lab.

- Target TRL – 4: Technology validated in lab.

Relation to processes and platforms

The prediction model (resulting from the analysis developed through machine learning) may be used as a unit of the run time manager to predict unsafe behaviors at run time.

Use Cases Interactions

- UC3: collision/instability prediction of vehicle platoon.
- UC5: performance prediction of vehicle traffic scenarios.

Contact Persons

- maurizio.mongelli@ieiit.cnr.it
- marco.muselli@ieiit.cnr.it
- enrico.ferrari@rulex-inc.com

Bibliography

- [1] E. Ferrari. A. Fermi, M. Mongelli, M. Muselli, *"Identification of Safety Regions in Vehicle Platooning via Machine Learning,"* 14th IEEE Internat. Work. on Fact. Commun. Sys. WFCS 2018.

2.2 AGILLA2 (UNIVAQ)

Item Type

Co-CPS Platform

Item Name

Agilla2

Reference Links

- D4.1_COCPS
- Source code: https://github.com/SafeCOP/WP4

Purpose of Item

Agilla2 provides an agent-based environment in which application and software components are distributed in the form of agents.

Description of Item

Agilla2 is a mobile agent middleware (MAMW) for Wireless Sensor Networks (WSN). It is the evolution of the Agilla middleware, a mobile-agent platform based on TinyOS and written in the nesC language.

TRL

- Current TRL - 4: Agilla2 has been developed and tested in a laboratory scenario.
- Target TRL – 5: Agilla2 will be exploited in the UC5 context, inside the RSU-SN component.

SAFECOP exploitation and extensions

In SAFECOP, Agilla2 is exploited in the WSN platforms in order to provide a flexible, mobile-agent based approach in distributing applications.

Relation to processes and platforms

In the context of the UC5, Agilla2 will be installed in the WSN nodes which are part of the RSU-SN component.

Tool usage process

Agilla2 allow applications to be written in the form of agents. Agents are written in a simple Agilla2-specific programming languages. Once the agents are written, they can be deployed (i.e. injected) in the WSN dynamically, without requiring normal WSN node board programming.

Inputs/Outputs

Agilla2 requires the TinyOS platform to be installed in the nodes. The inputs of Agilla2 are the software application, written in the Agilla2 specific programming language. As outputs, Agilla2 simplify the application development and distribution.

Use Cases Interactions

- Use Case 5 (UC5): Agilla2 is used to provide a dynamic mobile agent-based middleware for the WSN present in the RSU-SN component.

Contact Persons

- Luigi Pomante, luigi.pomante@univaq.it
- Walter Tiberti, walter.tiberti@graduate.univaq.it

Bibliography

- [1] L. Pomante, L. Corradetti, D. Gregori, S. Marchesani, M. Santic, W. Tiberti. *"A Renovated Mobile Agents Middleware for WSN - Porting of Agilla to the TinyOS 2.x Platform."* IEEE RTSI 2016.

2.3 TinyWIDS (UNIVAQ)

Item Type

Mechanism

Item Name

TinyWIDS

Reference Links

- D4.1_RTMO
- Source code: https://github.com/SafeCOP/WP4

Purpose of Item

TinyWIDS is an Intrusion Detection System (IDS) for Wireless Sensor Networks (WSN). It allows WSN nodes to detect a set of common malicious attacks and to notify the control center

Description of Item

TinyWIDS is the implementation of a intrusion detection system designed for the WSN environment. It is based on WIDS, a reference IDS for WSN which exploits the Weak-Process model (WPM) in order to achieve a misuse-based intrusion detection. TinyWIDS contains an internal database of known malicious attacks (stored as WPM representations) which is exploited to determine whether the WSN is under a malicious attack. Once TinyWIDS detect such a situation, it sends notifications to the controlling application, which in turn, can notify the control center or perform application-defined reactions.

TRL

- Current TRL: 4 - TinyWIDS has been developed and tested in a laboratory scenario against a set of simple attacks
- Target TRL: 5 - TinyWIDS implementation will be exploited in the UC5 context, inside the RSU-SN component

SAFECOP exploitation and extensions

In SAFECOP, we provide TinyWIDS as first WIDS implementation written in the nesC language and based on the TinyOS platform. It is exploited thanks to its ability to perform intrusion detection operation using only the scarce resources available in the WSN nodes.

Relation to processes and platforms

In the context of the UC5, TinyWIDS will be installed in the WSN nodes which are part of the RSU-SN component.

Tool usage process

TinyWIDS has to be installed in the WSN nodes in the form of a set of software modules. Once it is added to the application program, TinyWIDS automatically perform its tasks.

Inputs/Outputs

TinyWIDS inputs are the WSN node state information and the database of known malicious attacks. TinyWIDS elaborates those inputs in order to produce, upon the detection of malicious attacks, alarms and notifications exploitable by both the application program running on the WSN node and the LCU.

Use Cases Interactions

Use Case 5 (UC5): TinyWIDS is used to provide an intrusion detection system in the RSU-SN component. Each WSN node inside the RSU-SN exploits TinyWIDS for monitoring malicious attack attempts, eventually notifying the local central unit (LCU).

Contact Persons

- Luigi Pomante, luigi.pomante@univaq.it
- Walter Tiberti, walter.tiberti@graduate.univaq.it

Bibliography

- [1] L. Pomante, W. Tiberti, M. Santic, F. Santucci, M. Pugliese, L. Di Giuseppe, L. Bozzi. TinyWIDS: a "*WPM-based Intrusion Detection System for TinyOS2.x/802.15.4 Wireless Sensor Networks*". Fifth Workshop on Cryptography and Security in Computing Systems (CS2 2018).
- [2] L. Pomante, M. Pugliese, S. Marchesani, F. Santucci. *"WINSOME: A Middleware Platform for the Provision of Secure Monitoring Services over Wireless Sensor Networks"*. IWCMC 2013.
- [3] L. Pomante, S. Marchesani, M. Pugliese, F. Santucci. *"A Middleware Approach to Provide Security in IEEE 802.15.4 Wireless Sensor Networks"*. Mobilware 2013.

2.4 RUN-TIME SAFETY MONITOR FOR CO-CPS BASED ON LINEAR TEMPORAL LOGIC (KTH)

Item Type

Method (built within a simulation)

Item Name

Run-time safety monitor for CO-CPS based on linear temporal logic.

Reference Links

Paper [1].

Purpose of Item

The purpose of this item is to support run-time safety monitoring by continuously evaluating safety conditions that have been derived from a hazard analysis. Such safety conditions are expected to have been formalized in linear temporal logic (LTL) and tested using the quantitative safety analysis (QSA) framework developed in Task 4.3. The QSA approach will identify quantitative safety thresholds on system and environment parameters.

The derived formulas and boundaries from QSA will be continuously assessed within the run-time monitor, and as soon as they are violated then error handling mechanisms will be called that return the system to a safe state.

An example would be to monitor the maximum safe packet loss in wireless communication (an environmental parameter) between platoon vehicles (SafeCOP UC 5.6). The maximum safe value for this parameter will be established by QSA. The packet loss value will then be monitored at run-time, and where it rises above the maximum safe value, error handling will increase the distance between vehicles to restore safety.

Description of Item

The item is a run-time monitor that observes key system and environment parameters, on an iterative sense-evaluate-actuate cycle. The cycle time needs to be short to ensure that CO-CPS system safety is never compromised.

During the evaluate phase, temporal logic safety formulas are evaluated. When a safety formula becomes false, the actuate phase invokes a safety response from appropriate control algorithms to smoothly restore system safety.

TRL

- Current TRL - 1
- Target TRL – 3

SAFECOP exploitation and extensions

The QSA methodology will be defined and exemplified within research conducted within SAFECOP Task 4.3.

The methodology will be developed and evaluated using an extension of an already existing platooning simulator for UC5.6 developed at KTH.

Some software features need to be added to the background technology (platooning simulator) to support this methodology.

The intended use is to provide run-time safety support which complements design-time safety analysis through safety requirements testing.

Relation to processes and platforms

Our approach will conform as an instance of the SafeCOP reference platform for run-time management.

Methodology usage process

The run-time monitoring process is an iterative cycle as follows:

1. Obtain updated copies of system sensor values. If necessary filter such values for noise. If necessary estimate state space values, e.g. by Kalman filtering.
2. Evaluate a set of temporal logic safety formulas, based on current and previous sensor values.
3. If a temporal logic safety formula becomes false, activate the appropriate error handling mechanism. If the formulas was immediately previously false, check the current status of the error handling mechanism. If no response, escalate the error.
4. Return to Step 1.

Inputs/Outputs

1. *Input:* one or more system and/or environment parameters
2. *Output:* an activation call to an emergency response mechanism

Use Cases Interactions

The run-time methodology will be exemplified in UC6 (platooning) to show that it is a generic methodology that can be applied to the other CO-CPS use cases. Different safety requirements will be formulated for different uses cases on a platooning simulator (e.g. emergency braking). Different system parameters will be identified (e.g. time headway, distance headway), and a QSA will be conducted to identify system and environment parameter boundaries. The resulting parameter values will be taken as threshold values for monitoring within the runtime manager.

Contact Persons

Professor Karl Meinke,
School of Computer Science and Communications karlm@kth.se
mobile: 076 223 8679
skype: karl_meinke

Bibliography

- Karl Meinke: Learning-Based Testing of Cyber-Physical Systems-of-Systems: A Platooning Study. EPEW 2017: 135-151 https://link.springer.com/chapter/10.1007/978-3-319-66583-2_9

2.5 Safety integrity mechanisms to support run-time management (IBT Solutions)

Item Type

Method

Item Name

Safety integrity mechanisms to support run-time management.

Reference Links

WPs/WP4/Working documents/D4.1/SAFECOP_D41_RTME.docx

Purpose of Item

The purpose of the item is to provide specific guidelines for the implementation of safety integrity mechanisms necessary to support the run-time management.

Description of Item

The item collects a list of indications and guidelines to implement diagnostic and integrity mechanisms in the context of resource-constrained embedded systems. Particular focus is posed on data integrity, both in the acquisition phase and in the storage phase. Most of the techniques described in the item are specific realizations of the methods proposed in Functional Safety Standards such as IEC61508 and ISO26262.

TRL

- Target TRL – 5: From the software point of view, the target TRL to be reached by the end of the project for the main techniques is 5. It shall be noted, though, that some of these techniques require a hardware counterpart, which will not be designed/redesigned due to the limited effort available. Prototypical demonstrators will be used instead.

SAFECOP exploitation and extensions

The method provides techniques to prevent, diagnose and react to possible hazardous conditions. All these three portions of the item will be used, possibly to different extents. While most of the techniques are based on general principles, their specific implementation is strongly dependent on the specific application safety requirements and, most importantly, on the resource availability, in terms or processing and memory. In addition, since several techniques are conceived for generic sensor data, they have been customized and specialized for the specific sensors (Accelerometer, Gyroscope, GPS and other) used by the SafeCOP application, in particular for UC5.

The main goal of such techniques is twofold: on one hand they are aimed at guaranteeing that the data (position, speed, crash event alarms, …) collected by the OBU to be forwarded to the infrastructure is reliable to the desired level of integrity (ASIL B or ASIL C); on the other hand, some techniques are adopted to prevent unwanted activation of potentially hazardous safety-related functions such as vehicle brakes actuation.

Relation to processes and platforms

The methods and algorithms constituting this item are part of the monitoring infrastructure of the run-time manager.

Tool usage process

Being the item a collection of methods and techniques, the actual usage in the use-case application is strongly dependent on the specific context. The guidelines provide suggestions on which specific techniques shall be used for diagnostic purposes, and, in some cases, for the implementation of reactions bringing the system to a safe-state.

In general a custom implementation is required to introduce the desired diagnostic or integrity measure into the functional portion of the application. To this general situation, some exception are generic techniques such as double-inverted variable protection, CRC/Checksum data protections, program control flow checking and a few other. Such techniques are provided in the form of a C library but still a clear understanding of the operating principles and manual coding is necessary to properly integrate them into the application.

Inputs/Outputs

The methods are very varied and can hardly be generalized to identify common inputs. On the other hand, outputs are standardized. Two different types of outputs can be identified:

For the diagnostic techniques, the output is a Boolean variable indicating whether integrity is violated or not. The specific form of the output - from the programming point of view - can vary from technique to technique.

For system integrity technique, the output is actually constituted by a change of state of the system, which is brought to either a degrade operating mode or a safe state, typically with the support of specific hardware components.

Use Cases Interactions

The diagnostic and integrity methods constituting this item (both in the form of models and in the form of specific C libraries) will mainly be employed in the V2I use-case (UC5). They will primarily be used on the OBU (on-board unit) to guarantee reliability of the data generated by the device and shared with the infrastructure and to prevent system faults to lead to hazardous situations and potential injuries.

Contact Persons

- Matteo Grotto, matteo.grotto@ibtsolutions.it
- Carlo Brandolese, ibt@ibtsolutions.it

Bibliography

- [1] IEC 61508 - Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems.
- [2] ISO 26262 - Road vehicles – Functional safety

2.6 A Run Time Manager prototype in C++ language (ROT)

Item Type

Co-CPS function/RT Manager-Monitor (relating to SafeCOP - D1.1 - Requirements and Evaluation Metrics Baseline V2).

Item Name

A Run Time Manager prototype in C++ language

Reference Links

See bibliography below.

Purpose of Item

The main scope of this item is to provide a general-purpose design and an implementation that could be reasonably adopted by every UC, following the SafeCOP Generic Architecture as in [1].

Description of Item

A Run Time Manager (RTM) prototype that takes into account results and approaches as in [2], [3], [4] and [5].

The main idea is to let users extend and customize this RTM implementation to fit specific UC needs, without losing the main objective of guaranteeing safety at Runtime.

Basically, the RTM can be viewed as a Finite State Machine with 3 operative states:

- Idle: Initial state
- Running: after an initialization phase, the RTM enter its "nominal" state where messages fetched by queues are monitored and controlled with respect to Safety Contracts;
- FailSafe: the RTM reacts, possibly with an actuation, depending upon violations raised when verifying message data against Safety Contracts.

The proposed architecture is composed by 3 main modules:

CO-CPS Application interface: As already discussed, CO-CPS applications contains cooperative functions under development by each UC. All the requirements based on cooperative functions being developed under each demonstrator will be linked here. We will define an interface which allows these applications to interact with the safety manager.

Safety Manager/Safety Monitor: these modules reflect the need of having two separate handlers for the safety functions; the Safety Manager gathers the inputs from the CO-CPS module and ensures that the safety contracts for the cooperative functions are not violated thus to provide the correct actuation choices for the vehicle within the platoon. Complementary to this behavior, the Safety Monitor will ensure the quality of the data gathered by the on-board sensing and will provide the correct actuations for the vehicle itself based on the safety contracts.

Either if coming from the cooperative functions or from the on-board sensing, the gathered inputs are evaluated based on a FSM logic and are used to implement the action/reaction functionalities.

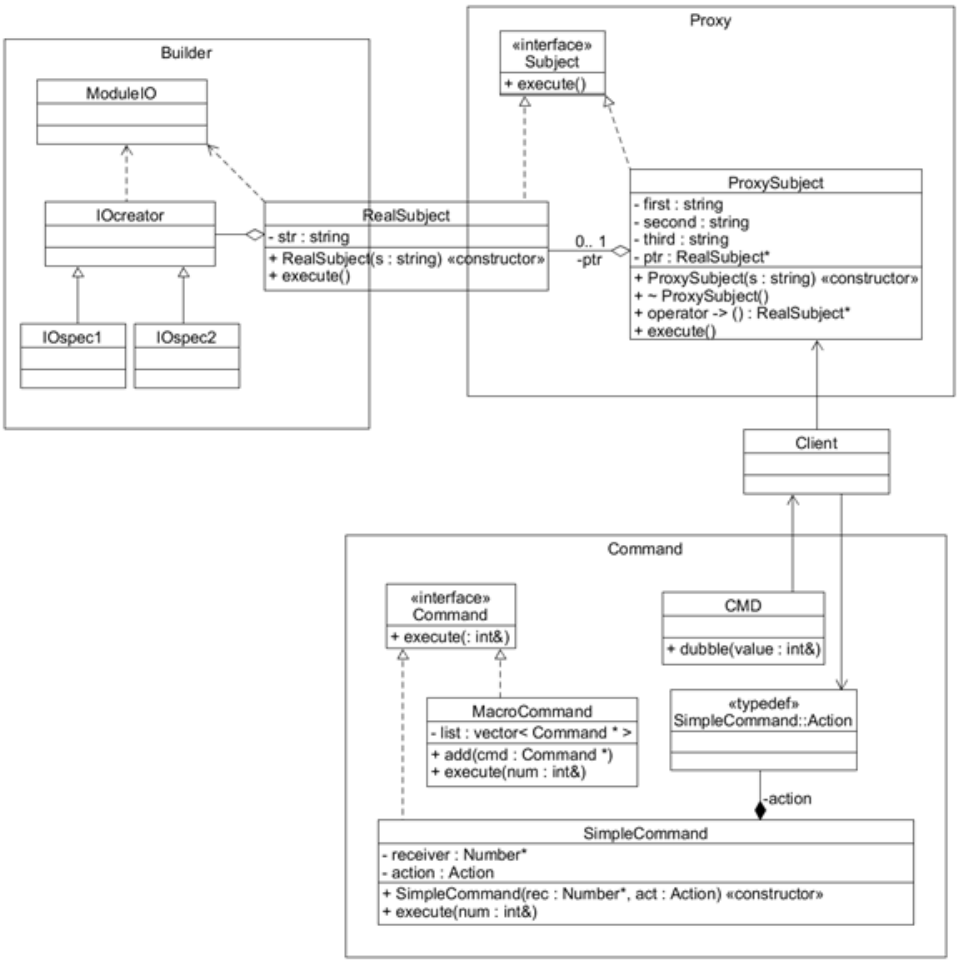

Figure 1. Safety Manager and Monitor architecture.

As shown in Figure 1, both the Safety Manager and the Safety Monitor are composed by 4 main modules:

- **Builder**: It manages the primitives for reading from a communication channel (Module I/O) and implements the functions defined in the proxy interface (*RealSubject*). It will contain for example message parsing of 802.15.4 data rather than velocity/odometer/accelerations data.
- **Proxy**: It exposes the standard interface to the Client (i.e. FSM) to use the concrete builder methods and at the same time it hides from client the knowledge of above methods.
- **Client**: It defines the logic of data reading and writing and it sends command execution r equests to the Command. It also contains the application data buffer. It will

actually contain the Runtime Manager FSM and mechanisms to implement isolation (I.e. message queues).

- **Command**: Entity that, by receiving commands, applies the logic defined by the Safety Contracts (*SimpleCommand*), and performs the action in case of violations (it could be a message to a task, a driver directly invoked on the O.S., an invocation of an API function, etc.), informing the Client of the resulted action (violation or not), thus managing FSM state transitions.

TRL

At the current level of design, the prototype is intended to be an experimental proof of concept that needs to be reviewed and approved by all the partners involved. Current TRL is 3 and the target TRL will be 4 by the end of the project.

Describe the Intended use of the Item in SAFECOP

Provide a general-purpose design and an implementation of the RTM that could be reasonably adopted by every UC, depending upon their specific needs.

Inputs/Outputs

- *Input:* CPS-Data(eg. sensing, alarm, ecc..)
- *Output:* Monitoring data and control actions on the different systems controllers to comply with safety contracts.

Use Cases Interactions

- WP2: the safety contracts coming from the Hazard Analysis of each UC will be used to create the algorithms inside the Safety Manager to validate the cooperative functions (basically a set of parameters that will be monitored at Runtime to ensure Safety between the CO-CPS).
- WP3: everything that relates to communication protocols is essential to guarantee the correct communication between the CO-CPS. The cooperation with the WP 4 is crucial in order to be able to ensure the right choices for the RTM in the different environments.
- WP1: The UCs and RTM requirements are used for the design phase of the RTM
- WP5: The feedback loop with the different UCs will serve as a validation and verification tool for the RTM implementation in the different scenarios.

Contact Persons

- Luciano Bozzi, luciano.bozzi@rotechnology.it
- Fabio Del Forno, fabio.delforno@rotechnology.it
- Leonardo Napoletani, leonardo.napoletani@rotechnology.it

Bibliography

- [1] SafeCOP - D1.1 - Requirements and Evaluation Metrics Baseline V2

- [2] A. Casimiro, J. Kaiser, E. M. Schiller, P. Costa, J. Parizi, R. Johansson, R. Librino - *The KARYON Project: Predictable and Safe Coordination in Cooperative Vehicular Systems*, WPs/WP4/Working documents/SOA_D4.1/KARYON.pdf
- [3] CISTER – *Runtime Verification of safety Critical Systems*, Documents/Meetings/Runtime Manager Workshop, Rome, Nov. 28-29, 2017/presentations/RuntimeVerification.pdf
- [4] C. Brandolese (Polimi/IBTS), L. Pomante (UNIVAQ) - SAFECOP_RT_v17_checkKARYON.ppt
- [5] S. Puri (INT) - *About modelling contracts and their possible usage at runtime*, Documents/Meetings/Runtime Manager Workshop, Rome, Nov. 28-29, 2017/presentations/Workshop RTMA - INT.pptx

2.7 RMTLD3SYNTH RUNTIME VERIFICATION FRAMEWORK (ISEP)

Item Type

Monitor Generation Framework

Item Name

RMTLD3Synth Runtime Verification Framework

Reference Links

Source code: https://github.com/SafeCOP/WP4

Purpose of Item

The purpose of this item is to serve as a framework to specify and generate code of runtime monitors based on the RMTLD3 formal timed temporal logic. This framework has been designed to be general enough to be adopted virtually in any use case of the project.

Description of Item

The rmtld3synth is a framework of the expressiveness and potential of generating monitors of RMTLD-3, presented already as one of the formal languages that was designed exactly to suit the needs of verifying timing constraints of complex systems, such as those addressed in SafeCOP. This framework is able to automatically generate C++11 or OCaml programs that are implementations of monitors specified using the RMTLD-3.

For instance, to specify a very simple condition stating the following, assuming as the set of events under observation the events a, b, and c. The condition can be specified as follows: "if the event a holds now, then either a or b will hold until the event c holds, but within the limit of 10 units of time and, furthermore, the overall duration in time of c cannot take more than 4 units in the same maximum of 10 units of time". This property can be specified in RMTLD-3 as follows:

$$(a \rightarrow ((a \vee b)\ U_{<10}\ c)) \wedge \int^{10} c < 4.$$

Now, in order to generate the code for the monitor that is capable of verifying this property, we have to input this formula in the text box of the rmtld3synth web interface, as depicted in Figure 2. The tool takes as input the specification written in LaTex in its current status of development, but better input facilities are expected to be developed during the project. The actual specification in LaTex is

"(a \rightarrow ((a \lor b) \until_{<10} c)) \land \int^{10} c < 4".

We can see, right after the text box where this specification is inserted, a part of the code that the tool generates, in this case, OCaml code which was the language that we have opted for exemplifying the tool. Nevertheless, generation of semantically equivalent C++ code is also straightforward to obtain, being enough to change the option "--synth-ocaml" by the alternative option "--synth-cpp11", as shown in Figure 3.

rmtld3synth web demonstrator

Monitor Synthesis ▾ SMT Synthesis ▾

Type a command

```
./rmtld3synth --synth-ocaml --input-latexeq "(a \rightarrow ((a \lor b) \until_{<10} c)) \land \int^{10} c < 4" --verbose 2
```

Please see the results

```
./rmtld3synth
--synth-ocaml
--input-latexeq
(a \rightarrow ((a \lor b) \until_{<10} c)) \land \int^{10} c < 4
--verbose
2
-----------------------------------------------------------------------------

v0.3-alpha1-11-g92f74fd (92f74fd1412921bd4f1101f441485f53b84ce2ca)
x86_64 Cygwin 2018-02-19 21:35

Latex Eq parsing enabled.
Latexeq input: (a \rightarrow ((a \lor b) \until_{<10} c)) \land \int^{10} c < 4

(Fland
 ((FImplies (FVar a)
   (ULess (POp (Less ()) ((TVal 10))) (Flor ((FVar a) (FVar b))) (FVar c)))
  (FIneq
   (((FTerm ((TInt (TVal 10) (FVar c)))) (Less ()))
    ((FTerm ((TVal 4))) (N ()))))))

Synthesis for Ocaml language
-----------------------------------------------------------------------------

cluster_name: mon1

open List
open Rmtld3

module type Trace = sig val trc : trace end
(* one trace :: module OneTrace : Trace = struct let trc = [("a",(1.,2.))] end *)

module Mon0_( T : Trace_ ) = struct
```

Figure 2. We interface of the rmtld3synth tool.

rmtld3synth web demonstrator

Monitor Synthesis SMT Synthesis

Type a command

```
./rmtld3synth --synth-cpp11 --input-latexeq "(a \rightarrow ((a \lor b) \until_{<10} c)) \land \int^{10} c < 4"
```

Please see the results

```
./rmtld3synth
--synth-cpp11
--input-latexeq
(a \rightarrow ((a \lor b) \until_{<10} c)) \land \int^{10} c < 4

--------------------------------------------------------------------------------



 #ifndef _MON0_COMPUTE_H_
 #define _MON0_COMPUTE_H_

 #include "rmtld3.h"

 auto _mon0_compute = [](struct Environment &env, timespan t) mutable -> three_valued_type { auto sf = [](struct Environment &env, timespan t)
mutable -> three_valued_type { auto sf1 = [](struct Environment &env, timespan t) mutable -> three_valued_type { auto sf = [](struct Environment
&env, timespan t) mutable -> three_valued_type { auto sf1 = [](struct Environment &env, timespan t) mutable -> three_valued_type { auto sf = []
(struct Environment &env, timespan t) mutable -> three_valued_type { return env.evaluate(env, 4, t); }(env,t); return b3_not (sf); }(env,t); auto sf2 = []
(struct Environment env, timespan t) -> three_valued_type
 {
  auto eval_fold = []( struct Environment env, timespan t, TraceIterator< int > iter) -> four_valued_type
  {

   // eval_b lambda function
   auto eval_b = []( struct Environment env, timespan t, four_valued_type v ) -> four_valued_type
   {
    // eval_i lambda function
    auto eval_i = [](three_valued_type b1, three_valued_type b2) -> four_valued_type
    {
     return (b2 != T_FALSE) ? b3_to_b4(b2) : ( (b1 != T_TRUE && b2 == T_FALSE) ? b3_to_b4(b1) : FV_SYMBOL );
    };

    // change this (trying to get the maximum complexity)
```

Figure 3. Generation of C++11 code from the same specification.

TRL

At the current level of design, the tool is intended to be an experimental proof of concept that needs to be validated in a laboratorial use-case setting. The level of TRL is 3 and the aim is to reach TRL level 4.

SAFECOP exploitation and extensions

In SafeCOP the plan is to validate the tool in terms of its effectiveness to formally specify properties that are relevant in the automotive domain, namely via its integration in UC3.

Relation to processes and platforms

The proposed framework can be used by any partner, as it has been designed for specify general properties about real-time timing properties. It will be experimented and validated in UC3.

Tool usage process

The rmtld3synth synthesis tool is able to automatically generate monitors based on the formal specifications written in RMTLD3. Polynomial inequalities are supported by this formalism as well as the most common operators of temporal logics. Furthermore, quantification is also considered in the language of RMTLD3 as a means to facilitate the decomposition of the quantified formulas into several monitoring conditions.

We will now present an overview of the typical process for generating monitors for OCaml and C++11 languages using this tool, together with a running example of a simple monitoring case generation. We begin by the running example, present the generated monitors, and show how to configure the RV monitoring model to couple with the system.

Let us consider the following formula

$$(a \rightarrow ((a \vee b)\, \mathrm{U}_{<10}\, c)) \wedge \int^{10} c < 4$$

that intuitively describes that given an event a, b occurs until c and, at the same time, the duration of b shall be less than four-time units over the next 10-time units. For instance, a trace that satisfies this formula is $(a, 2), (b, 2), (a, 1), (c, 3), (a, 3), (c, 10)$.

From rmtld3synth2ocaml tool, we synthesized the formula above into the code presented in **Error! Reference source not found.** using the following command line:

```
./rmtld3synth --synth-ocaml --input-latexeq "(a \rightarrow ((a \lor
                b) \until_{10} c)) \land \int^{c} < 4"
```

Next, we can also generate C++11 monitors by replacing `--synth-ocaml` with `--synth-cpp11`. The outcome is the monitor code is presented in Figure 5**Error! Reference source not found.** and Figure 6**Error! Reference source not found.**, the latter being the header code that connects monitors to the corresponding runtime monitoring architecture, and that we will describe in more detail latter in this section.

To use the first monitor (generated for the OCaml language), we need to define a trace for OCaml reference as follows, which consists in defining a concrete trace to be analyzed and then instantiate the code of the monitor with it (using the OCaml module system).

```
module OneTrace : Trace = struct let trc = [("a",(0.,2.));("b
    ",(2.,4.));("a",(4.,5.));("c",(5.,8.));("a",(8.,11.));("c
    ",(11.,21.))] end;;

module MonA = Mon0(OneTrace);;
```

Now, we describe the settings for constructing the RV monitoring model. he settings for rmtld3synth tool are defined using the syntax

```
(<setting_id> <bool_type | integer_type | string_type>)
```

where the symbol "|" distinguishes between the supported types of arguments such as Boolean, integer or string, and `setting_id` is a string containing the name of the setting to which values are assigned. An example of a set of possible settings for the tool is given in the first five lines of

Error! Reference source not found.. We now briefly describe the purpose of each of the setting entries:

- `gen_tests` sets the automatic generations of test cases;
- `gen_concurrency_tests` constructs tests for testing lock- and wait-free monitors executing concurrently;
- `gen_unit_tests` constructs tests for C++11 synthesis using the Ocaml source code as an oracle;
- `buffer_size` sets the static size of the buffer to be used (rmtld3synth tool can change it if required by some constraints);
- `minimum_inter_arrival_time` establishes the minimum inter-arrival time that the events can have. It is a very pessimistic setting but provides some information for static checking;
- `maximum_period` sets the maximum interval between two consecutive releases of a task's job. It has a correlation between the periodic monitor and the minimum inter-arrival time. It provides static checks according to the size of time-stamps of events;
- `event_type` provides the type for dealing with events (commonly is a class parameter);
- `event_subtype` provides the type for the event data. In that case, it is an identifier that can distinct 255 events. However, if more events are required, the type should be modified to `uint32_t` or greater. The number of different events versus the available size for the identifier is also statically checked;
- `cluster_name` identifies the set of monitors. It acts as a label for grouping monitor specifications.

The formulas `m_simple` and `m_morecomplex` are to RMTLD3 formulas that follow the theoretical definition of the rule to form formulas inductively, and they are represented in OCaml by the type of data presented in Figure 4 .

```
open List
open Rmtld3

module type Trace = sig val trc : trace end
module Mon0  ( T : Trace  ) = struct
let compute_uless gamma f1 f2 k u t =
  let m = (k,u,t) in
  let eval_i b1 b2 =
    if b2 <> False then b3_to_b4 b2 else if b1 <> True && b2 = False then b3_to_b4 b1 else
        Symbol
  in

  let eval_b (k,u,t) f1 f2 v =
    if v <> Symbol then v else eval_i (f1 k u t) (f2 k u t)
  in

  let eval_fold (k,u,t) f1 f2 x =
    fst (fold_left (fun (v,t') (prop,(ii1,ii2)) > (eval_b (k, u, t') f1 f2 v, ii2)) (Symbol,t)
        x)
  in

  if not (gamma >= 0.) then
    raise  (Failure "Gamma␣of␣U␣operator␣is␣a␣non negative␣value")
  else
  begin
    let k,_,t = m in
    let subk = sub_k m gamma in
    let eval_c = eval_fold m f1 f2 subk in
    if eval_c = Symbol then
      if k.duration_of_trace <= (t +. gamma) then Unknown else ( False ) else b4_to_b3 eval_c
  end


let compute_tm_duration tm fm k u t =
  let dt = (t,tm k u t) in

  let indicator_function (k,u) t phi = if fm k u t = True then 1. else 0. in
  let riemann_sum m dt (i,i') phi =
    (* dt=(t,t') and t in ]i,i'] or t' in ]i,i'] *)
    count_duration := !count_duration + 1 ;
    let t,t' = dt in
    if i <= t && t < i' then
      (* lower bound *)
      (i'.t) *. (indicator_function m t phi)
    else (
      if i <= t' && t' < i' then
        (* upper bound *)
        (t'.i) *. (indicator_function m t' phi)
      else
        (i'.i) *. (indicator_function m i phi)
    ) in
  let eval_eta m dt phi x = fold_left (fun s (prop,(i,t')) > (riemann_sum
  m dt (i,t') phi) +. s) 0. x in
  let t,t' = dt in
  eval_eta (k,u) dt fm (sub_k (k,u,t) t')

  let env = environment T.trc
  let lg_env = logical_environment
  let t = 0.
  let mon = (fun k s t > b3_not ((fun k s t > b3_or ((fun k s t > b3_not ((fun k s t > b3_or
       ((fun k s t > b3_not ((fun k s t > k.evaluate k.trace "a" t) k s t)) k s t) ((
      compute_uless 10. (fun k s t > b3_or ((fun k s t > k.evaluate k.trace "a" t) k s t) ((
      fun k s t > k.evaluate k.trace "b" t) k s t)) (fun k s t > k.evaluate k.trace "c" t)) k
      s t)) k s t)) k s t) ((fun k s t > b3_not ((fun k s t > b3_lessthan ((
      compute_tm_duration (fun k s t > 10.) (fun k s t > b3_or ((fun k s t > k.evaluate k.
      trace "c" t) k s t) ((fun k s t > k.evaluate k.trace "d" t) k s t))) k s t) ((fun k s t
      > 4.) k s t)) k s t)) k s t)) k s t)) env lg_env t
end
```

Figure 4. Generated OCaml monitor.

```
#ifndef _MON0_COMPUTE_H_
#define _MON0_COMPUTE_H_
#include "rmtld3.h"

auto _mon0_compute = [](struct Environment &env, timespan t) mutable -> three_valued_type {
    return [](struct Environment env, timespan t) -> three_valued_type { auto tr1 = [](struct
    Environment env, timespan t) -> duration {

  auto eval_eta = [](struct Environment env, timespan t, timespan t_upper, TraceIterator< int >
      iter) -> duration
  {
    auto indicator_function = [](struct Environment env, timespan t) -> duration {
      auto formula = [](struct Environment &env, timespan t) mutable -> three_valued_type { auto
          sf1 = [](struct Environment &env, timespan t) mutable -> three_valued_type { return
          env.evaluate(env, 2, t); }(env,t); auto sf2 = [](struct Environment &env, timespan t)
          mutable -> three_valued_type { return env.evaluate(env, 1, t); }(env,t); return b3_or
          (sf1, sf2); }(env, t);

      return (formula == T_TRUE)? std::make_pair (1,false) : ( (formula == T_FALSE)? std::
          make_pair (0,false) : std::make_pair (0,true)) ;

    };

    auto lower = iter.getLowerAbsoluteTime();
    auto upper = iter.getUpperAbsoluteTime();
    timespan val1 = ( t == lower )? 0 : t - lower;
    timespan val2 = ( t_upper == upper )? 0 : t_upper - upper;
    auto cum = lower;

    return std::accumulate(
      iter.begin(),
      iter.end(),
      std::make_pair (make_duration (0, false), (timespan)lower), (duration starts at 0)
      [&env, val1, val2, &cum, t, t_upper, indicator_function]( const std::pair<duration,
          timespan> p, Event< int > e )
      {
        auto d = p.first;
        auto t_begin = cum;
        auto t_end = t_begin + e.getTime();
        cum = t_end;
        auto cond1 = t_begin <= t && t < t_end;
        auto cond2 = t_begin <= t_upper && t_upper < t_end;
        auto valx = ((cond1)? val1 : 0 ) + ((cond2)? val2 : 0);
        auto x = indicator_function(env, p.second);

        return std::make_pair (make_duration (d.first + (x.first * ( e.getTime() - valx )), d.
            second || x.second), p.second + e.getTime());
      }
    ).first;
  };

  auto sub_k = []( struct Environment env, timespan t, timespan t_upper) -> TraceIterator< int >
  {
    TraceIterator< int > iter = TraceIterator< int > ( env.trace, env.state.first, 0, env.state.
        first, env.state.second, 0, env.state.second );
    // to use the iterator for both searches we use one reference
    TraceIterator< int > &it = iter;

    ASSERT_RMTLD3( t == iter.getLowerAbsoluteTime() );
    auto lower = env.trace->searchIndexForwardUntil( it, t);
    auto upper = env.trace->searchIndexForwardUntil( it, t_upper - 1 );
    it.setBound(lower, upper);

    return it;
  };

  auto t_upper = t + make_duration(10.,false).first;

  return eval_eta(env, t, t_upper, sub_k(env, t, t_upper));

  }(env,t);

  auto tr2 = make_duration(4.,false);
  return b3_lessthan (tr1, tr2);
}(env,t);};
#endif //_MON0_COMPUTE_H_
```

Figure 5. Generated C++11 monitor.

```
#ifndef MONITOR_MON0_H
#define MONITOR_MON0_H
#include "Rmtld3_reader.h"
#include "RTML_monitor.h"
#include "mon0_compute.h"
#include "mon1.h"

class Mon0 : public RTML_monitor {
private:
  RMTLD3_reader< int > trace = RMTLD3_reader< int >( __buffer_mon1.getBuffer(), 0. );
  struct Environment env;

protected:
  void run(){
    three_valued_type _out = _mon0_compute(env,0);
    DEBUG_RTEMLD3("Veredict:%d\n", _out);
  }

public:
  Mon0(useconds_t p): RTML_monitor(p,SCHED_FIFO,50), env(std::make_pair (0, 0), &trace,
      __observation) {}
};
#endif //MONITOR_MON0_H
```

Figure 6. Generated C++11 monitor header.

```
(gen_tests true)
(minimum_inter_arrival_time 102)
(maximum_period 2000000)
(event_subtype uint_8)
(cluster_name monitor_set1)

(m_simple 1000000 (Or (Until 200000 (Prop A) (Prop C)) (Prop B)))
(m_morecomplex 500000 (Or (Until 200000 (Prop set_off) (Or (Until 200 (Prop A) (Prop C)) (Prop B))) (
    Prop B)))
```

Figure 7. The default configuration file.

```
type var_id = string with sexp
type prop = string with sexp
type time = float with sexp
type value = float with sexp

type formula =
        True of unit
      | Prop of prop
      | Not of formula
      | Or of formula * formula
      | Until of time * formula * formula
      | Exists of var_id * formula
      | LessThan of term * term
and term =
        Constant of value
      | Variable of var_id
      | FPlus of term * term
      | FTimes of term * term
      | Duration of term * formula
with sexp

type rmtld3_fm = formula with sexp
type rmtld3_tm = term with sexp
type tm = rmtld3_tm with sexp
type fm = rmtld3_fm with sexp
```

Figure 8. Inductive type for the rmtld3synth formulas and terms.

Inputs/Outputs

The inputs for the tool are well-formed formulas written in the RMTLD3 logic [], described in D4.1. The output of the tool is the code for the monitor (either in C++ or OCaml, for now) that is synthesized for the formula given as input.

Use Cases Interactions

The tool with be validated in the context of UC3.

Contact Persons

- David Pereira, dmrpe@isep.ipp.pt
- Ricardo Severino, rarss@isep.ipp.pt

Bibliography

- [1] André Pedro, "Dynamic contracts for verification and enforcement of real-time systems properties", PhD Thesis. 10, Apr, 2018.

2.8 ROS based Runtime Monitoring Architecure (ISEP)

Item Type

Runtime Monitoring Architecture

Item Name

ROS based runtime monitoring architecture.

Reference Links

Source code: https://github.com/SafeCOP/WP4

Purpose of Item

The purpose of this item is to provide a generic ROS based infrastructure that allows for the runtime monitoring via monitors specified using a provided language and/or via the usage of the rmtld3synth tool described in the previous section.

Description of Item

The runtime manager based on ROS that we propose here follows a different approach, providing a YAML based specification language which allows to abstract ROS Nodes to event identifiers that can then be read and analyzed by monitors that can either be generated by the runtime manager via an internal language, or via plugging external monitor generation frameworks. The general view of this reference runtime manager architecture is depicted in Figure 9.

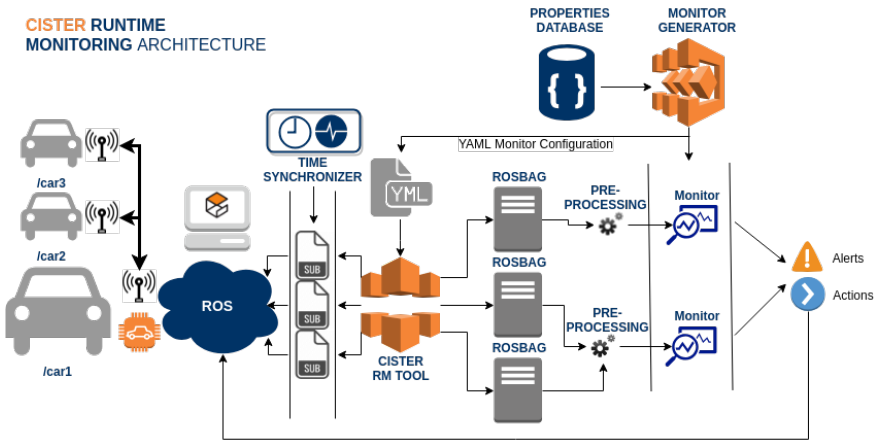

Figure 9. Architecture of ROS based reference runtime manager.

In the proposed architecture, components can subscribe to any topic and extract its content, or can read data from any bag. Bags are typically created by a tool like rosbag, which subscribes to one or more ROS topics, and stores the serialized message data in a file as it is received. These bag files can be played back in ROS to the same topics they were recorded from (to identify runtime witnesses of unspecified behaviors), but most importantly they can be remapped to new topics, from which monitors can read and trigger the SRM to react.

The runtime manager architecture we are describing use a synchronizer filter that synchronizes incoming channels by the timestamps contained in their headers of the ROS Messages, and outputs them in the form of a single callback that takes the same number of channels. The C++ implementation can synchronize up to 9 channels. The synchronizer filter is templated on a policy that determines how to synchronize the channels. There are currently two policies: ExactTime and ApproximateTime. The ExactTime policy requires messages to have exactly the same timestamp in order to match. The ApproximateTime policy uses an adaptive algorithm to match messages based on their timestamp.

When generating the runtime manager from configuration files (described below), a preprocessing expression evaluator is used to generate simple monitors, that follow a typical programming language, ad hoc manner of writing monitors. This is sometimes very useful when focusing on particular aspects of the Co-CPS for which a safety condition to be written in a formal language could reveal to be too much time consuming. This preprocessing expression evaluator supports the following fundamental arithmetic operations, and imperative language style types and statements:

- *Types:* Scalar, Vector, String
- *Basic operators:* +, -, *, /, %, ^
- *Assignment:* :=, +=, -=, *=, /=, %=
- *Equalities & Inequalities:* =, ==, <>, !=, <, <=, >, >=
- *Logic operators:* and, mand, mor, nand, nor, not, or, shl, shr, xnor, xor, true, false
- *Functions:* abs, avg, ceil, clamp, equal, erf, erfc, exp, expm1, floor, frac, log, log10, log1p, log2, logn, max, min, mul, ncdf, nequal, root, round, roundn, sgn, sqrt, sum, swap, trunc
- *Trigonometry:* acos, acosh, asin, asinh, atan, atanh, atan2, cos, cosh, cot, csc, sec, sin, sinc, sinh, tan, tanh, hypot, rad2deg, deg2grad, deg2rad, grad2deg
- *Control structures:* if-then-else, ternary conditional, switch-case, return-statement
- *Loop statements:* while, for, repeat-until, break, continue
- *String processing:* in, like, ilike, concatenation
- *Optimizations:* constant-folding, simple strength reduction and dead code elimination
- *Calculus:* numerical integration and differentiation

The current status of this language consists in a YAML file that allows to specify topics to which data to be monitored is being published, define meta-variables that gather data from the subscribed topics, the selection of a time synchronization policy, and a section in which one can write safety specifications either using the simple procedural-like language described in the above paragraphs, or by specifying and RMTLD3Synth compliant formal specification. In the case of the latter, the RMTLD3Synth tools is externally called to generated the monitor (this is still early work, but that is going to be continued during SafeCOP to allow for a general way of plugging-in other external monitor generation frameworks). An example is presented in D4.1.

TRL

- Current TRL: 2;
- Target TRL: 3;

SAFECOP exploitation and extensions

This framework will be exploited in SAFECOP for validation purposes, i.e., we will validated its capabilities of properly monitoring several automotive scenarios (within the same use-case) and from there, understand how to generalize it so that it becomes a framework with cross-domain applicability.

Relation to processes and platforms

The tool with be validated in the context of UC3 and the usage of the RTMLD3Synth as a back-end to generated monitors is going to be experimented.

Tool usage process

This tool requires that a set of ROS topics is mapped to particular variables that are declared in a YAML file. Monitors are also to be specified in the same file. Then, a compilation process takes place and generates the target complete ROS infrastructure (nodes, topics, services, etc.) and the code for the monitors.

Inputs/Outputs

Takes as input an YAML specification file according to the structure defined above and, as output, generates the corresponding ROS infrastructure (including the monitors).

Use Cases Interactions

The tool with be validated in the context of UC3.

Contact Persons

- David Pereira, dmrpe@isep.ipp.pt
- Ricardo Severino, rarss@isep.ipp.pt

Bibliography

- André Pedro, “Dynamic contracts for verification and enforcement of real-time systems properties”, PhD Thesis. 10, Apr, 2018.

2.9 Runtime Monitoring Architecture for Safety-Critical Systems (ISEP)

Item Type

Runtime Monitoring Architecture

Item Name

Runtime Monitoring Architecture For Safety-Critical Systems

Reference Links

Source code: https://github.com/SafeCOP/WP4

Purpose of Item

This work refers to a reference architecture for the runtime monitoring that is suited to safety critical as well as non-critical applications, and therefore can be adapted by UCs for their specific demands. This reference architecture could then be implemented in any language, as long as it respects the prescriptions discussed below. The proposed run-time monitoring framework is depicted in Figure 10.

Description of Item

The architecture presented in Figure 10 is composed of five main components, namely, the *monitored application*, the monitors, the *buffers* through which events are transmitted, the *buffer writers* for writing events in a specific buffer, and the *buffer readers* used by the monitors to read events stored in a buffer. This architecture focuses on time and space isolation guarantees between applications (seen here as tasks) and the monitors. In the context of a Co-CPS, distributed components can adopt this architecture for managing the interaction between its inner applications and monitors, whereas safe distributed monitoring can be implemented via the guidelines already introduced and described in Section **Error! Reference source not found.** of D4.1.

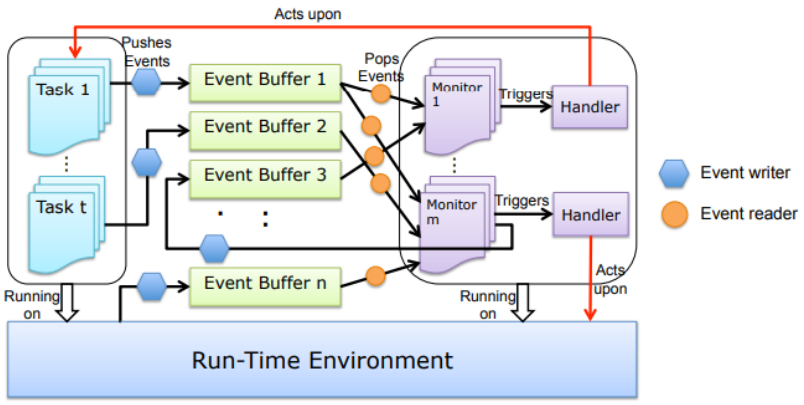

Figure 10. Reference runtime monitoring architecture.

TRL

At the current level of design, the tool is intended to be an experimental proof of concept that needs to be validated in a laboratorial use-case setting. The level of TRL is temporary fixed at 3, but the aim is to reach TRL level 4.

SAFECOP exploitation and extensions

In SafeCOP, the plan is to extend the current architecture with distributed characteristics, ideally mapping the several design guidelines described in D4.1. and recommended for the various implementations of the runtime manager in the different use cases.

Relation to processes and platforms

The architecture was developed so that it is generic enough to be used in any of the UC, and it will be used in the implementation of UC3.

Tool usage process

Documentation on the usage process is available in the source code repository. The tool consists in a set of libraries that have to be used to wrap monitors and define event buffers to allow the communication between application and monitors.

Inputs/Outputs

The inputs for this library is the code intended to implement the monitors, and the code for the tasks that are defined to be observed by those monitors.

Use Cases Interactions

Extensions to this library are expected to be validated in the context of UC3.

Contact Persons

- David Pereira, dmrpe@isep.ipp.pt
- Ricardo Severino, rarss@isep.ipp.pt

Bibliography

- [1] Nelissen, G., Pereira, D., Pinho, L., *"A Novel Run-Time Monitoring Architecture for Safe and Efficient Inline Monitoring"*, 20th International Conference on Reliable Software Technologies - Ada-Europe 2015 (Ada-Europe 2015). 22 to 26, Jun, 2015. Madrid, Spain.