FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Framework for the Runtime Monitoring and Verification of Distributed Automated Cyber-Physical Systems using ROS

Pedro José Santos Oliveira

DISSERTAÇÃO



Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Doutor Armando Jorge Sousa

External Supervisor: Prof. Doutor David Pereira

February 28, 2020

Abstract

The Robot Operating System (ROS) is becoming more and more adopted as a reference framework for the development of Cyber-Physical Systems (CPS), as a well-featured distributed system that facilitates and regulates communication between distributed applications through a publish-subscribe architecture.

Given the critical levels of many CPS applications, it becomes fundamental that we have proper means to observe and verify the system during its operation. One way is to follow an approach based on runtime monitoring and verification, but these approaches require strict management of the events observed in the system. ROS lacks such means of enable monitoring and verification, and the results of this thesis aim at filling that gap.

In this thesis we present a runtime monitoring and verification framework that implements an instrumentation technique that we use to monitor inferred events from ROS topics, detect ROS nodes intrusion, and estimate traces of the states of the system to enable the coupling of monitors specified using formal language with our monitoring framework.

We have validated our proposal in two different simulators, in an automotive scenario. We also performed some initial tests that indicate that the overhead introduced could be acceptable to some classes of CPS, at least in simulations, in order to increase their accuracy.

Keywords: Chronograf, Cyber-Physical Systems, Distributed Systems, InfluxDB, Kapacitor, Robot Operating Systems, Intrusion Detection, Monitoring, rmtld3synth, Runtime, Verification

Resumo

O ROS (Robot Operating System) está a ser cada vez mais adotado como uma estrutura de referência para o desenvolvimento de sistemas ciber-físicos (CPS), como um sistema distribuído com recursos avançados que facilita e regula a comunicação entre aplicações distribuídas através de uma arquitetura publish-subscribe. Dado os níveis críticos de muitas aplicações CPS, torna-se fundamental que tenhamos meios adequados para observar e verificar o sistema durante operação. Uma maneira é seguir uma abordagem baseada em runtime monitoring e runtime verification, mas essas abordagens exigem uma gestão rigorosa dos eventos observados no sistema. O ROS carece de tais meios para permitir a monitorização e a verificação, e os resultados desta tese visam preencher essa lacuna. Nesta tese, apresentamos uma estrutura de monitorização e verificação em tempo de execução que implementa uma técnica de instrumentação usada para monitorizar eventos inferidos de tópicos de ROS, detectar intrusão de nós ROS e estimar traços de estados do sistema para permitir o acoplamento de monitores especificados usando linguagem formal com a nossa estrutura de monotorização. Validámos a proposta em dois simuladores diferentes, ambos num cenário automóvel. Também realizámos alguns testes iniciais que indicam que a sobrecarga introduzida pode ser aceitável para algumas classes de CPS, pelo menos em simulações, a fim de aumentar sua precisão.

Keywords: Chronograf, Sistema Ciber-Físico, Sistemas distribuídos, InfluxDB, Kapacitor, Robot Operating Systems, Deteção de Intrusão, Monitorização, rmtld3synth, Tempo de execução, Verificação

Acknowledgements

With this thesis conclusion I would like to thank all those made it possible, to everyone that believed in me from the beginning. Firstly, I would like to thank my supervisors, Prof. Doutor David Pereira and Prof. Doutor Armando Jorge Sousa, that gave me an important support and incentives without which this would not become reality, I will be eternally grateful.

I would also give a special thanks to Maria Teresa Chaves, that always has been by my side, making me believe that I could finish this journey.

Thanks to my mom that have always carried me on their arms and have never let me fall, without whom this would never be possible.

A big thanks to all my friends.

Pedro Santos

"You always pass failure
on your way to success."

Mickey Rooney

Contents

1 Introduction **1**
- 1.1 Context . . . 1
  - 1.1.1 The ENABLE-S3 project . . . 1
  - 1.1.2 The REASSURE project . . . 2
- 1.2 Motivation . . . 2
- 1.3 Objectives . . . 3
- 1.4 Contributions . . . 3
- 1.5 Technology and Tools Used . . . 3
- 1.6 Report structure . . . 4

2 Theoretical Background and Related Work **5**
- 2.1 Cyber-Physical Systems . . . 5
  - 2.1.1 Highly Automated Cyber-Physical Systems . . . 5
  - 2.1.2 Distributed Cyber-Physical Systems . . . 6
  - 2.1.3 Robot Operating System . . . 6
- 2.2 Runtime Monitoring . . . 9
  - 2.2.1 Drawbacks of Classic Runtime Monitoring Architectures . . . 9
  - 2.2.2 Independent monitors and buffered communications . . . 10
  - 2.2.3 Distributed Runtime Monitoring . . . 11
- 2.3 Formal Verification . . . 13
  - 2.3.1 Three-Valued Restricted Temporal Logic with Durations (RMTLD-3) . . . 14
- 2.4 Runtime Verification . . . 14
  - 2.4.1 Correct-by-Construction Monitor Generation . . . 15
- 2.5 Related Work . . . 16
  - 2.5.1 ROSRV . . . 17
- 2.6 Summary . . . 18

3 The RMVCPS Framework **19**
- 3.1 Problem Formulation . . . 19
  - 3.1.1 Monitoring distributed systems . . . 20
- 3.2 Proposed Solution . . . 21
- 3.3 Simulation Environments . . . 22
  - 3.3.1 Simulation in Gazebo . . . 22
  - 3.3.2 Simulation in CARLA . . . 25
- 3.4 ROSTOPICMON . . . 26
  - 3.4.1 Subscribing ROS Topics (Collecting data) . . . 26
  - 3.4.2 Storage . . . 29
  - 3.4.3 Event Extraction . . . 30

3.4.4 Dashboard Visualizer . . . 32
3.4.5 Real-Time Streaming Data Processing Engine . . . 32
3.4.6 Rostopicmon monitoring environment configuration . . . 34
3.4.7 Summary Diagram of the rostopicmon enviroment . . . 38
3.5 ROSTOPICVER . . . 38
3.5.1 Estimating States From Events . . . 39
3.5.2 Verification Monitor Generation . . . 40
3.6 ROSTOPICINTRUSION . . . 43
3.6.1 ROS Computation Graph . . . 43
3.6.2 ROS Master . . . 43
3.6.3 Running ROS Topic intrusion monitor . . . 44

4 Experiments **45**
4.1 Results . . . 45
4.1.1 Hardware Specifications . . . 45
4.1.2 Software Specifications . . . 45
4.1.3 Test Specifications . . . 46
4.1.4 Dashboard . . . 47
4.1.5 Delay measurements . . . 47
4.1.6 CPU and Memory Overhead . . . 48

5 Conclusions and Future Work **49**
5.1 Further Work . . . 49

A RMVCPS Swimlane Diagram **51**

References **53**

List of Figures

1.1 ENABLE-S3's methodology approach for verification & validation of CPS. . . . 2

2.1 An example of Highly Automated CPS in an ENABLE-S3 event. 6
2.2 Robot Operating System (ROS) Simplified Architectural Design 7
2.3 A classic approach to runtime monitoring architectures 9
2.4 Bus-Monitor Architectural Design 12
2.5 Single Process-Monitor Architectural Design. 13
2.6 Distributed Process-Monitor Architectural Design 13
2.7 High-level view of the ROSRV architecture. 17

3.1 The problem addressed by our RMVCPS Framework 19
3.2 Division in different layers of abstraction. 20
3.3 General overview of a possible generic runtime monitoring framework. 21
3.4 Division in RMVCPS Framework. 22
3.5 Simulation with three independent controlled Prius vehicles in the same Gazebo simulation. 23
3.6 RVIZ view of ROS sensors included in Gazebo simulation. 25
3.7 Simulation created in CARLA. 26
3.8 Example of a ROS Message. 27
3.9 Implementation of intropection in ROS Topics 28
3.10 Flow Diagram of deserialize ROS messages from ROS Topics 28
3.11 Diagram of the visualization dashboard 32
3.12 Diagram of Event extraction. 33
3.13 Example of complete YAML monitor configuration file. 38
3.14 General overview of proposal solution for the rostopicmon. 39
3.15 Flow Diagram of rostopicver. 42
3.16 rostopicver example output. 43
3.17 Flowchart of rostopicintrusion Node. 44
3.18 rostopicintrusion example output. 44

4.1 Dashboard of CARLA Simulation Environment. 47
4.2 Time delay between reception by the rostopicmon and the storage in InfluxDB. . 47
4.3 CPU and memory overhead of used tools. 48

A.1 Swimlane Diagram with relation between rostopicmon, rostopicver and rostopicintrusion 51

Chapter 1

Introduction

1.1 Context

Increasing automation of cyber-physical systems is a huge contribution to overcome some of many challenges in society, caused by a changing world with an ageing population that lives more and more in city enviroments [19]. To improve the quality of life, there is a need to increase the automation of these systems, helping the preservation of natural resources, air quality, clean and efficient transportation, and many more.

1.1.1 The ENABLE-S3 project

The European project ENABLE-S3 - *European Initiative to Enable Validation for Highly Automated Safe and Secure Systems* is an industry-driven project, framed within the ECSEL Join Undertaking, that aims to replace today's cost-intensive verification and validation efforts with another one newer, with more advanced and efficient methods that will ultimately lead to the commercialization of highly automated CPS. The approach followed in order to achieve the objectives of the project is depicted in the Figure 1.1.

The work presented in this thesis will have direct impact in the participation of CISTER / ISEP in developments within ENABLE-S3. In particular, this work will contribute to one of the use cases in the automotive domain considered within the project, namely use case 4 - *Traffic Jam Pilot With V2X Communication* which is lead by GMV Skysoft. The contribution is a runtime monitoring architecture for ROS that allows simulations to be observable and verified while being executed. Furthermore, the design and implementation of the monitoring architecture (described in more detail further ahead in this thesis) is expected to be performed in such a way that allows the integration of external tools, where the code of the monitors is automatically generated from formal specifications.

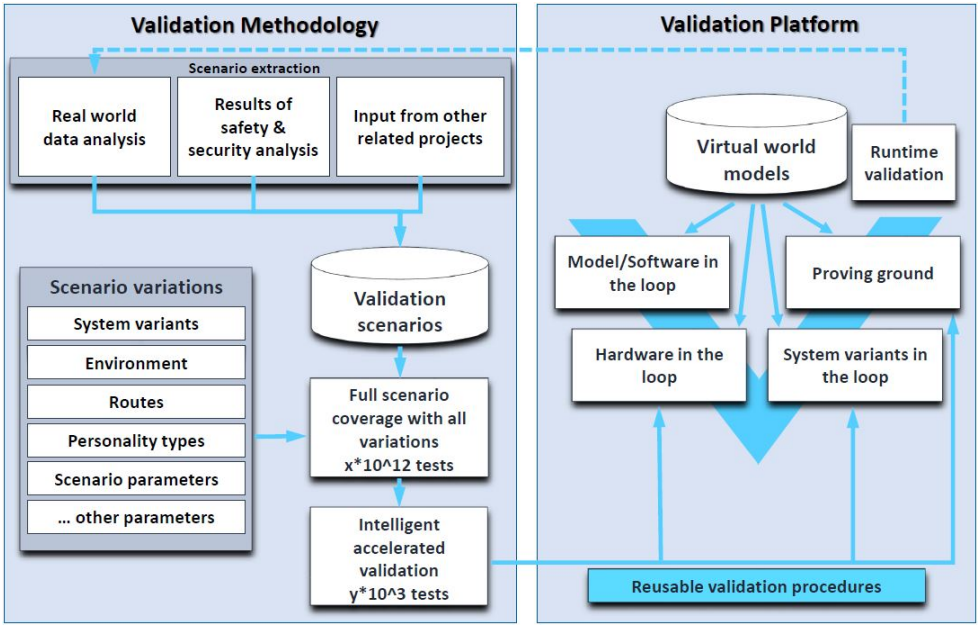


Figure 1.1: ENABLE-S3's methodology approach for verification & validation of CPS.

1.1.2 The REASSURE project

The goal of REASSURE is to improve over the state-of-the-art Runtime Verification (RV) approaches for Real-Time Embedded Systems by developing a new framework that: (1) extends existing runtime monitoring infrastructures capturing system properties (with focus on extra functional properties) with features to guarantee safety and ensuring that monitored data is kept secure without revealing information about the system (no security leak); (2) provides a domain specific language (DSL) and the tools to specify the requirements that must be verified at runtime; (3) automatically generate monitors and deploy them on energy and computing power constrained platforms. The framework developed in REASSURE will be validated within an industrial context. The work presented in this thesis has contributed to this project in what respects the development of distributed runtime monitoring architectures which enable their usage in CPS applications.

1.2 Motivation

Although several technology demonstrators for highly automated systems already exist, however there is not only a severe lack of cost-effective, commonly accepted verification methods, but also the need of tools supporting these methods. Winner in [41] predict that more than 100 million km of road driving would be required to statistically prove that an automated vehicle is as safe as a manually driven one, which implies that a proven-in-use certification by performing physical tests on the road only is simply not feasible any more. Similiar statements hold for other applications as well.

1.3 Objectives

The main objective of this thesis is to design and develop a novel runtime monitoring and verification infrastructure that suits the requirements of Distributed Cyber-Physical Systems. The envisioned solution must provide guarantees that the operational data of interest are properly observed and analyzed during runtime and must be able to trigger responses to unexpected or unsafe situations. For that, the runtime monitoring architecture developed should consider a simple domain specification language that allows identifying which data should be observed and also allows specifying the properties that monitors need to verify on that data. Such specifications should be performed using rigorous specification techniques based on formal languages, which will allow to generate code for the data collection mechanisms and also to synthesize monitors in a way that ensures that the resulting monitors will execute according to what was specified.

1.4 Contributions

- Gazebo simulation with 3 Toyota Prius vehicles and with communication between them. Code is being submitted to the Gazebo community.
- rostopicmon [7], ROS Node to perform runtime monitoring. Code is being submitted to the ROS community.
- rostopicver [8], ROS Node to perform runtime verification.
- rostopicinstrusion [6], ROS Node to detect other nodes intrusion in subscribing or publish to a ROS Topic. Code is also being submitted to the ROS community.

1.5 Technology and Tools Used

The tools used in this thesis are:

- Robot Operating System (ROS): As Distributed Cyber Physical System Operating System.
- Gazebo Robot Simulator: As Environment Simulation.
- CARLA: As a more realistic Environment Simulation.
- InfluxDB: As Time-series database to store variables and events.
- Kapacitor: As real-time streaming data processing engine.
- Chronograf: To build a dynamic dashboard.
- The RMTLD3synth Runtime Verification Framework: To generate correct-by-construction monitors.

1.6 Report structure

Besides this introduction, that includes the thesis context, motivation, goals and contributions of this thesis there are four additional chapters.

In chapter 2, we present a theoretical background overview regarding all subjects of the thesis, providing an introductory knowledge about the field, and also presents the state of the art and the work directly or indirectly related to the thesis.

Section 3 contains a description of the problem the thesis wants to help solving, as well as the description of all the proposed solutions, and the process used for that.

Chapter 4 presents the tests with of the implemented solution and an analyse through the obtained results.

Finally, chapter 5 concludes about the findings and deliberates about future work.

Chapter 2

Theoretical Background and Related Work

2.1 Cyber-Physical Systems

Cyber-Physical Systems (CPS) refers to the interconnection between computational systems and the physical world [29], and this diversified connection makes this class of systems difficult to analyze or even to design.

The physical side of a CPS monitors and controls the physical world through sensors and actuators, respectively. On the cyber side, CPSs usually consist of embedded systems that perform computations, based on physical data collected. Depending on the CPS application, you may still have involved elements of communication or storage.

A robot is an example of a CPS, as it is a machine that uses sensors, actuators and computing power to resemble a human being and is able to automatically replicate certain human movements and functions in the physical world. A self-driving vehicle can be also seen as CPS whose smart sensors and networking systems enable them to monitor its operation, plan trajectory and change the actuators parameters while coordinating with other traffic elements (cars, roadworks alerts, etc). (See example in figure 2.1)

2.1.1 Highly Automated Cyber-Physical Systems

A CPS, by definition, already requires to be automated, but in a Highly Automated Cyber-Physical System (HACPS), the role of the human element is largely restricted to provide higher level (strategic) control inputs and supervise the operation of the machine (See Figure 2.1). In its role of controller, the human element is heavily supported by functions provided by the "physical" system. The role may be important such as defining the overall goals and performance objectives or supervising the system when the state of the system veers towards a critical failure. [22]

The research literature is rich in the analysis and design of highly automated CPS, focuses on intelligent vehicles (cars or aircraft), removing some of the "mundane" tasks of keeping the vehicle on the road and allow the driver to focus on in-vehicle tasks, such as entertainment.



Figure 2.1: An example of Highly Automated CPS in an ENABLE-S3 event.

2.1.2 Distributed Cyber-Physical Systems

Distributed Cyber-Physical Systems (DCPS) are systems that allows no assumptions about where the node runs, and thus allows a runtime reallocation to match the resources available in the environment network [42].

DCPS can be the next major technology field and challenge enabling a huge number of new applications, business models and major commercial opportunities for European high-tech companies in many essential industry sectors, ranging from the Automotive industry to Aerospace and Military applications, among others. Indeed, DCPS offer solutions to many of today's grand societal challenges. For example, automated driving functions will increase traffic safety, reduce traffic jams, increase passenger comfort, and also enable disabled or elderly people to use transportation systems in a more suitable, comfortable, and secure manner [32].

2.1.3 Robot Operating System

Robot Operating System (ROS) is an open-source, middleware system for robots, widely used in diferent CPSs. It provides the services would be expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. ROS can be consider as a distributed system since ROS runtime "graph" potentially be a peer-to-peer network of processes distributed across machines that are loosely coupled using the ROS communication infrastructure [39]. A simplified architectural design overview can be seen in Figure 2.2.

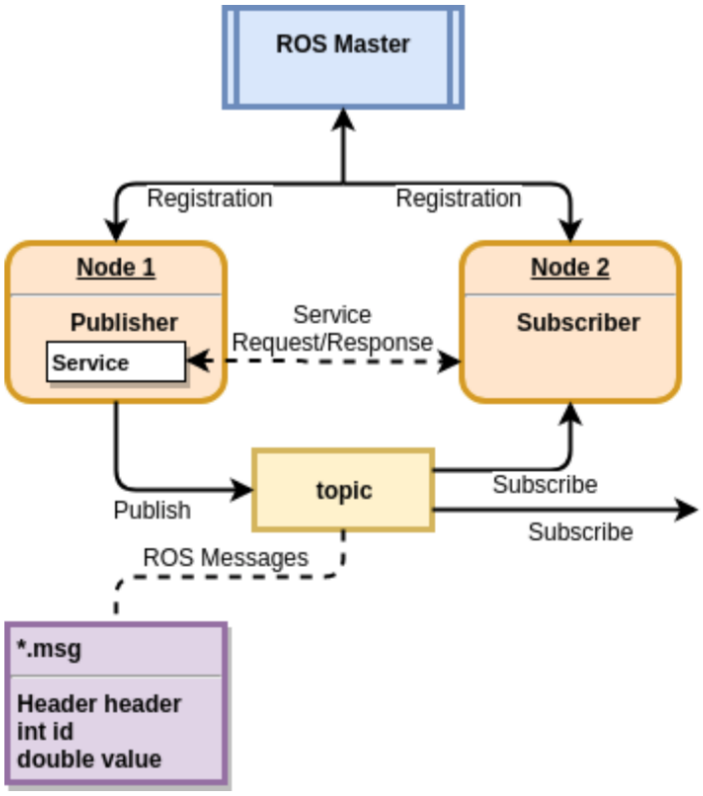


Figure 2.2: Robot Operating System (ROS) Simplified Architectural Design

2.1.3.1 ROS Master

The ROS Master [3] provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer.

2.1.3.2 ROS Node

A ROS Node [5] is a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server. These nodes are meant to operate at a fine-grained scale; a robot control system will usually comprise many nodes. For example, one node controls a laser range-finder, one Node controls the robot's wheel motors, one node performs localization, one node performs path planning, one node provide a graphical view of the system, and so on.

2.1.3.3 ROS Topic

Topics [12] are named channels over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption.

In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic.

2.1.3.4 ROS Service

The publish / subscribe model is a very flexible communication paradigm, but its many-to-many one-way transport is not appropriate for RPC request / reply interactions, which are often required in a distributed system. Request / reply is done via a Service [10], which is defined by a pair of messages: one for the request and one for the reply. A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply. Client libraries usually present this interaction to the programmer as if it were a remote procedure call.

2.1.3.5 ROS Messages

ROS Nodes [4] communicate with each other by publishing messages to topics. A ROS message is a simple data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs). Nodes can also exchange a request and response message as part of a ROS service call. These request and response messages are defined in srv files.

2.1.3.6 ROS Time

There are two types of temporal notions in ROS, wall-time or sim-time [2]. Usually, the ROS client libraries will use the computer's system clock as a time source, also known as the "wall-clock" or "wall-time". But when running a simulation or playing back logged data, however, it is often desirable to, instead of having the system clock, use a simulated clock so that is possible to accelerate, slow down, or step control over the system's perceived time. For example, if sensor data is played back into main system, will be good having the running time correspond to the timestamps of the sensor data. To support this, the ROS client libraries can listen to the /clock topic that is used to publish "simulation time".

2.1.3.7 Synchronization of ROS Nodes in different machines

When using distributed system capabilities, and ROS runs on different machines, it is necessary to ensure a temporal synchronization between them. ROS does not provide this functionality, but there are already quite a few well-established methods (e.g. chrony using ntp protocol) that assure that. If the wall-clocks are not synchronized in the distributed machines, they will not agree on temporal calculations like those used in tf, and can collapse the system or invalidate all the monitoring and verification of the system.

2.2 Runtime Monitoring

Runtime Monitoring, by definition, consists in adding pieces of code, called monitors, to run applications. The monitors scrutinize the system behavior and also can extract events during the execution of the application and that information can be propagated back to the system designer so that functions that bring the system into a safe state are activated [35].

A runtime monitoring architecture serves as a layer that gathers information from the monitored application and transmits that data to a set of monitors.

Therefore, without an efficient and safe architecture, any adopted monitoring system becomes useless because its inputs may be flawed and do not represent the actual traces of events occurring in the system being observed, and therefore its outputs can not be trusted.

This fact is particularly challenging when addressing distributed and complex systems like the ones being addressed in the context of this thesis which are highly complex and naturally generate large amounts of data that may be relevant for establishing their safety and security.

In [35], the authors highlight four core requirements that should be met when developing a runtime monitoring library for high-assurance systems:

- Independent and composable development
- Time and space partitioning
- Simplicity
- Efficiency and responsiveness

2.2.1 Drawbacks of Classic Runtime Monitoring Architectures

The most common implementation of monitors in the state-of-the-art runtime monitoring and verification frameworks consists of injecting the monitoring code directly in the application code, whose general architecture Figure 2.3 illustrates such an approach with three tasks. Two tasks write data that are read by the third task. A monitor named Monitor 1 is called after each write and before each read to check properties on the data exchange. In such implementation, the monitoring procedure is executed as part of the three tasks.

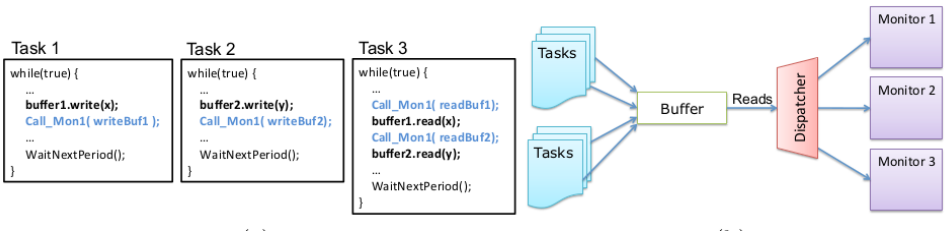


Figure 2.3: A classic approach to runtime monitoring architectures [35].

Most architectures and frameworks are based on Aspect-Oriented Programming or similar instrumentation mechanisms. The monitoring procedures are directly called when the monitored

application passes by prespecified positions in the source or compiled code. Whilst simple to implement, this kind of monitoring architecture exhibits multiple drawbacks as discussed below.

Whereas space partitioning could still be enforced with such monitoring architecture, the existing implementations of those frameworks simply do not consider it.

This lack of isolation is the cause of two major issues: Since the monitored application shares the same address space as the monitors, a faulty application generating memory errors due, for example, to stack overflows or wrong pointer manipulations, may corrupt the data manipulated by the monitor and alter its correct behavior.

The monitor would become unreliable, incapable to detect the fault or even worse, outputting a wrong diagnosis, which may lead to the triggering of counter-productive measures, thereby worsening the situation. Monitors have direct reading and writing access to globally defined entities of the monitored application (for example, Buffer1 and Buffer2 in the example of Figure 2.3). A faulty monitor (i.e. wrongly implemented) could modify and corrupt the content of the application variable and hence impact its execution. This generates safety and security threats for the overall system.

This would be unacceptable in safety-critical applications. Such monitoring architecture does not allow any kind of timing isolation between monitors and monitored applications. Since the monitoring procedures are directly included in the application code, the timing properties such as the application worst-case execution time (WCET), are irremediably modified after the addition of the monitoring code. [35]

2.2.2 Independent monitors and buffered communications

The alternative to the monitoring architecture described on section 2.2.1 is to implement monitors as independent components that receive events information through buffers. The most straightforward implementation of this solution would assume that all events are written in a shared buffer. In order to avoid the need for every monitor to read all events stored in the buffer — including those that are of no interest — an event dispatcher is usually attached to the output of the buffer. The dispatcher reads the events pushed in the buffer by the monitored application and redistributes them to the monitors requiring them. Unfortunately, the use of a unique global shared buffer causes a bottleneck in the flow of information. Indeed, only one event instance can be written in the buffer at a time. This requires the use of a synchronization mechanism among different threads that would try to access the buffer concurrently. This may lead to unwanted blocking of the tasks of the monitored application, thereby impacting their timing properties and thus going against any sort of timing isolation as their timing behaviors now become dependent on the number of threads pushing events in the buffer. Moreover, forbidding parallel event writing slows down the transfer of information to the monitors and hence their capacity to detect anomalies in an acceptable amount of time (i.e., responsiveness).

The alternative implementation would be to provide a private buffer for each monitor. As a consequence, there is no need for an event dispatcher anymore. Although diminishing the number

of potential concurrent accesses to the same buffer, a synchronization mechanism is still necessary when events generated by multiple threads are needed by the same monitor. Furthermore, additional issues must now be considered:

When an event is useful for multiple monitors, the monitored application must be instrumented multiple times in order to push the event in the buffers of each relevant monitor. This means that the instrumentation code is redundant. Moreover, the same event instance might be copied in multiple buffers, thereby increasing the memory footprint of the monitoring architecture.

The monitors using the events must be known when instrumenting the application. Adding, removing or replacing monitors require re-instrumenting the code, even if they all use the same events and data. This impacts the extensibility of the system as well as the independency between monitors and monitored applications.

2.2.3 Distributed Runtime Monitoring

Although the guidelines described in the previous section should be adhered to by a system implementing a runtime monitoring infrastructure, these guidelines say very few when it comes to concrete aspects of a runtime monitoring infrastructure, namely when the focus are distributed systems as the Highly Automated Cyber Physical Systems being addressed in this thesis.

Indeed, even today, most research in runtime monitoring is focused on monolithic software and not to distributed systems, and less even regarding safety critical or hard-real time distributed systems. In this section, we review the state of the art on runtime monitoring of distributed systems, following along the lines of Goodloe and Pike, who present a thorough survey of monitoring distributed real-time systems [23]. Notably, they present a set of monitor architecture constraints and propose three abstract monitor architectures in the context of monitoring these types of systems. Starting with the base constraints, we have that a monitoring architecture should exhibit:

- **Functionality**, that is, the monitoring solution should not change the original functionality of the system being monitored;
- **Schedulability**, meaning that the monitoring solution shall never impact the real-time constraints of the system;
- **Reliability**, which imposes that the system under observations must be equally or become more reliable that in the case it was not coupled with the monitoring solution; and, finally,
- **Certifiability** which imposes that the monitoring architecture does not require modification of the code of the target system, unless for those that are predicted by the designers and contained in some way by ensuring the fulfillment of the previous constraints.

2.2.3.1 Bus-Monitor architecture

The monitor silently observes the data passing through the system bus as if it was any other element of the system, although it performs its function of checking for potential failures. If a failure is recognized, the monitor reacts and sends messages via the bus to the other elements.

Despite being very simple and requiring few hardware, it is not a purely software-based architecture, which in turn has its consequences in scaling up to systems with larger and more complex dimensions and the design and implementation must ensure that the monitor does not enter in a failure mode (it would become useless), and that it does not consume excessive bandwidth, which could compromise the operation of the system being observed. This architecture is essentially motivated for usage on component-off-the-shelf platforms that do not provide fault-tolerance mechanisms, and it is depicted in the Figure 2.4 below.

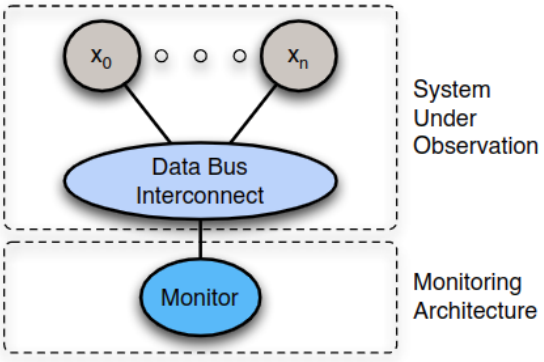


Figure 2.4: Bus-Monitor Architectural Design [23].

2.2.3.2 Single Process-Monitor architecture

The objective of this architecture is to address the constraints of the bus-monitor architecture, namely in what concerns the presence of a single bus linking all the elements of the observed system and the monitor. The single process-monitor architecture suggests that each element of the system is instrumented so that they send the information onto the monitor dedicated bus and therefore, due to the usage of a separate bus for receiving monitoring messages, these messages are not passed on the observed system's interconnects, which decreases the possibilities of the monitor to compromise the system's timeliness guarantees and that potential faults occurring in the data bus become independent of the monitor's bus. The Single Process-Monitor architecture is depicted in Figure 2.5.

2.2.3.3 Distributed Process-Monitor architecture

In this approach, there exists one monitor per process of the system being observed, such that each monitor can be either implemented in the same hardware as the process being monitored, or alternatively in an isolated, fault-containment unit. Furthermore, the monitors must communicate between each other in order to reach a consensual agreement regarding the status of the system being observed. When compared to the two previously described architectures, this one has the advantage that even if one or more monitors fails, it will not compromise the complete monitoring activities of the system since they live in a distributed environment. Also, in this architecture, each

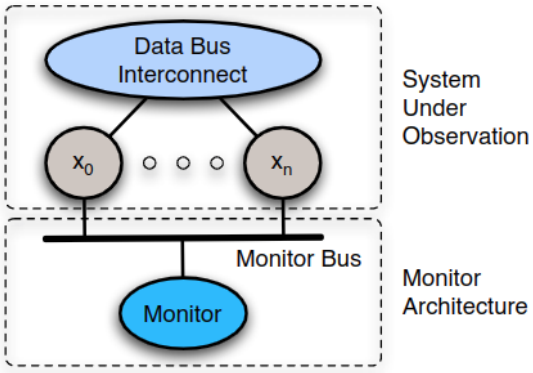


Figure 2.5: Single Process-Monitor Architectural Design. [23]

monitor can be solely responsible for controlling its corresponding process of the system, whereas in the single monitor architectural approach the contamination of one component can prevent other components from sending messages to the monitor. However, this architecture is also challenging since it may become as costly to implement as the system under observation. The architecture is depicted in Figure 2.6.

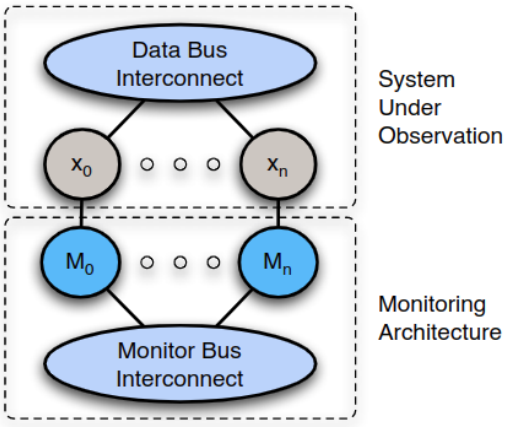


Figure 2.6: Distributed Process-Monitor Architectural Design [23].

2.3 Formal Verification

By definition [18], formal verification is a rigorous mathematical proof at the code level of the correctness of a system with respect to a certain formal specification or property, using formal methods of mathematics.

The growth of this area in the past decade has lead to the development of several formal languages, some of which have been specifically designed to allow to specify real-time timing safety

specifications and that are valuable to ensure proper runtime assurance mechanisms to complex systems such as DCPS. This section provides an overview of the usual kind of formal languages that are suited to the specification of safety properties. Most of the reviewed formal languages have their roots in model checking (since Runtime Verification, the scientific area from which they have been conceived has also its roots in Model Checking), in general the adopted formal languages derived from Linear Temporal Logic (LTL) [38]. Some examples of formal verification languages:

- Linear Time Logic (LTL)
- Metric Temporal Logic (MTL)
- Metric Temporal Logic with Durations (MTLD)

2.3.1 Three-Valued Restricted Temporal Logic with Durations (RMTLD-3)

This logic is a three-valued extension over a restricted fragment of MTLD that allows us to reason about explicit time and temporal order of durations. The syntactic restrictions over MTLD include the use of bounded formulas, of a single relation < over the real numbers, the restriction of the n-ary function terms to use one of the $+$ or $\times$ operators, and a restriction of α constants to the set of rational numbers. The semantic restrictions include the conversion of the continuous semantics of MTLD into an interval-based semantics, where models are timed interval sequences and formulas are evaluated in a given logical environment at a time t from the real number larger or equal to 0. We start by defining this restriction in the next chapter before addressing the 3-valued logic extension that is RMTLD-3 in order to allow the specification and generation of monitors that incrementally are able to gain awareness of the system but that until that happens are not able to respond with a definitive true or false evaluation of the trace they are processing (thus, answering with an unknown value).

2.4 Runtime Verification

Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property [31].

Runtime verification has it's origins in model checking and, to some extent, the main issue of monitor generation is comparable to the generation of automata in model checking. Model checking, however, also has important distinctions, while model checking typically considers infinite traces, runtime verification deals with finite executions, and while in model checking a complete model is given allowing to consider arbitrary positions of a trace, in runtime verification, especially when dealing with online monitoring, its only considered finite executions of increasing size. For this, a monitor should be designed to consider executions in an incremental fashion.

2.4.1 Correct-by-Construction Monitor Generation

In this section, we provide an overview of the state-of-the-art of frameworks that support the automatic generation of monitors from formal specifications. Most of the work described here has been carried out in the area of Runtime Verification and are typically called Runtime Verification Frameworks. Here we focus on works that have characteristics relevant for CPSs. Most of the frameworks reviewed in this section are also valuable as references for the implementation of various use cases, particularly those designed to fit the requirements of DCPS and also of ROS (Robot Operating System), and other publish-subscribe architectures (e.g., ZeroMQ).

2.4.1.1 The RMTLD3Synth Framework

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. This framework is able to automatically generate C++11 or Ocaml programs that are implementations of monitors specified using the RMTLD-3. [36]

Supported by this formalism are polynomial inequalities (using the less relation <) and the common modal operators of temporal logics U (until) and S (since). The existential quantification over these formulas is also possible by adopting the cylindrical algebraic decomposition (CAD) method. This method is suitable to convert quantified formulas into several decomposed conditions without these quantifiers.

For formula satisfiability checking, the tool is ready to synthesize the logic fragment into the input language accepted by the Z3 SMT solver. It can be used to discard several constraints involving duration and temporal order of propositions, in an offline and before execution fashion. For instance, schedulability analysis of hard real-time systems is possible by specifying the complete problem in RMTLD3, first using RMTLD3Synth to synthesize the problem in SMT-LIB and then using Z3 to solve it. The idea is to know if there exists a trace for which the RMTLD3 problem is satisfiable, or whether the SMT gives us an unsatisfiable answer meaning that is impossible to schedule such configuration. The latter enforces the refinement by drawing a counter-example. However, this facility provided by RMTLD3Synth is not the target of this section. In what follows, we will briefly show how to use RMTLD3Synth's web interface to obtain the code for a simple property involving a limit in time for which a certain property has to hold, and also imposing constraints on the duration of it.

2.4.1.2 Others Frameworks

- **CoPilot**: The Copilot platform [37]. This platform is composed of: (i) a strongly-typed, synchronous stream-based domain specific language (DSL) embedded in the Haskell functional programming language [40]; (ii) an interpreter; (iii) and also a compiler that compiles specifications into small constant-time and constant-space C monitors, which periodically sample a set of global variables synchronized through a global abstract clock.

- **Statemate**: In [13] the authors present an approach for the dynamic analysis of reactive systems via RV of code generated from Statechart [24] models and verified by the Statemate approach. The approach is based on the automatic synthesis of monitoring statecharts from formulas that specify the system's temporal and real-time properties in a proposed assertion language. The promising advantage of this approach is in its ability to analyze real-world models (with attributes reflecting the various design decisions) in the system's realistic environment. This capability is beyond the scope of model checking tools.

- **Temporal Rover**: This framework [21] is appropriate for monitoring of hard real-time systems due to the temporal constraints being specified in MTL in spite of the monitoring software being closed, therefore we are not able to understand how it is designed. Temporal Rover is a commercial RV tool based on future time metric temporal logic. It allows programmers to insert formal specification in programs via annotations, from which monitors are generated. An Automatic Test Generation (ATG) component is also provided to generate test sequences from logic specifications. Temporal Rover and its successor, DB Rover, support both inline and online monitoring. However, they also have their specification formalisms hardwired and are tightly bound to Java.

- **RT-MaC**: RT-MaC [30], an extension of MaC [27]. MaC provides two languages: The Primitive Event Definition Languages (PEDL) and the Meta-Event Definition Language (MEDL). The former is implementation dependent and allows to express qualitative properties (based on LTL); the latter is independent of particular implementations, allows to define which information is extracted from the application, and how it is transformed into events and conditions. RT-MaC supports the specification of real-time and probabilistic properties.

- **Runtime Verification for LTL and TLTL**: Bauer et al. have developed an algorithm for generating efficient monitors from TLTL for real-time systems [15]. The authors introduce the notion of TLTL with three truth values, denoted TLTL3. This basic notion is interesting and adequate for RV, since the complete set of traces is not available and the RV requires that the specification is evaluated increasingly. This approach employs so-called event-clock automata (ECA) for monitoring of TLTL3 formulas. Moreover, Bauer et al. introduce the symbolic timed runs and show their benefits for checking specifications efficiently, avoiding a possible but generally expensive translation of ECA to predicting-free timed automata. Yet, without considering counting time explicitly.

2.5 Related Work

There are many Runtime Verification frameworks available, but in general, they were not designed with all the particularities of ROS, or even CPS, in mind. Notorious examples are MOP [17], RuleR [14], RMOR [25], but also [16]. The MOP and RuleR have the flexibility to allow users to define RV specifications using different input formal languages, which makes these platforms

very flexible depending on the target system they intend to monitor. Their major drawback though is their intrusiveness. Most of those frameworks add calls to verification procedures into the application code. This modification of the target application may be problematic in safety-critical systems if it is not planned from the very beginning of the system design. Furthermore, they do not ensure isolation between the monitors and the monitored application. Hence, a failure of the monitored application may very well impact the capability of the monitor to detect that failure. But there is a framework that is directly related to this thesis, which is ROSRV, that we'll describe in the next subsection.

2.5.1 ROSRV

The ROSRV framework was designed with ROS in mind, namely in what concerns its low-level mechanisms which provide a message passing interface for inter-process communication including publish/subscribe messages. This framework was designed such that it is able to intercept, observe and potentially modify commands and messages that are passing through the system, as well as performing actions when events of interest are identified based on the characteristics of the messages being passed through the system. ROSRV provides a specification language for safety properties from which monitors are automatically generated, and a language for security which allows to establish access control policies and enforces them at runtime. The high-level architecture of ROSRV is depicted in Figure 2.7.

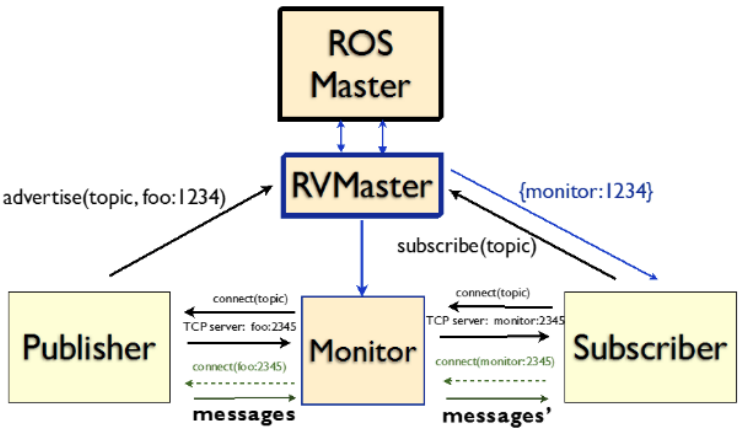


Figure 2.7: High-level view of the ROSRV architecture.

As can be easily noticed in Figure 2.7, there is a new component named RVMaster that intercepts all requests made by nodes to the ROS Master, thus acting as a security layer by protecting the ROS Master and also acting as a functional layer for protecting the safety of applications. Monitors are generated by the RVMaster from safety specifications and implemented as ordinary ROS nodes that can subscribe and publish messages. However, RVMaster keeps track of all the communication requests by the other nodes in the system and manipulates the point-to-point

communication addresses so that the generated monitors act as men-in-the-middle. An important property of ROSRV is that no changes are necessary to be made into ROS, since the only requirement is to configure the RVMaster to listen at the standard port and the ROS Master to listen at a hidden port visible only to RVMaster and was implemented using a firewall to block access to the ROS master port and therefore all the ordinary nodes in the system remain as they were and are not even aware that they are being monitored. Safety properties are specifiable via reference to events and actions based on event sequences, while security properties are specifiable via sections which refer to the categories of nodes, publishers, subscribers, and commands. Further details about how these specifications can be achieved are described in [33].

Currently limitations of the ROSRV:

- It relies only on IP and on network routing to guarantee security and that is not sufficient to protect against attackers who can run processes on the same (virtual) machines as trusted nodes, or spoof packets on physical network segments carrying unencrypted traffic;
- ROSRV is a centralized system, and therefore all the monitor nodes live in the same multithreaded process, and all communication in the system is monitored, meaning that the solution can suffer from scalability problems with a large number of nodes.

2.6 Summary

In this chapter, we have provided a state-of-the-art of formal languages that are being used for specifying safety properties, and we have shown that most of them are based on temporal logic, namely having as basis the LTL logic. We have also presented other formal specification languages that can be used in the future in this thesis as reference for the derivation of safety properties. These languages, respectively RMTLD3, have a focus on timing aspects, including the time intervals, durations, jitter of events caused by the operation of applications and the events these generate. These languages are not exclusive, as they can be used to specify different monitors if the test scenario so requires. We have also presented an overview of the state-of-the-art of monitor generation frameworks and described the RMTLD3Synth framework that generates monitors that focus on real-time safety specifications, namely on formally establishing the time until when some sequence/pattern of events has to take place, as well as inspecting the duration of particular events of sequences/patterns of events; the framework generates both C++ or OCaml code for the monitors specified in its input language.

Chapter 3

The RMVCPS Framework

3.1 Problem Formulation

The Robot Operating System (ROS) is becoming more and more commonly used in the development of CPS, not necessarily with robotics requirements, but essentially as a well-featured distributed system that facilitates and regulates communication between distributed applications via a publish-subscribe architecture. When comes to some sensitive applications, ROS's safety and security are essential, and it is therefore necessary to add to the ROS infrastructure a set of monitors that can produce not only performance enhancement as dynamic system configuration, dynamic program tuning, or on-line steering, but also to offer runtime correctness checking of ROS Node's to ensure consistency, preferably via formal specifications. Solving this issue, will possible not only allow the detection of runtime errors, but also to detect some security violation such as illegal actions in the ROS network. Figure 3.1 shows an overview of the problem addressed by RMVCPS.

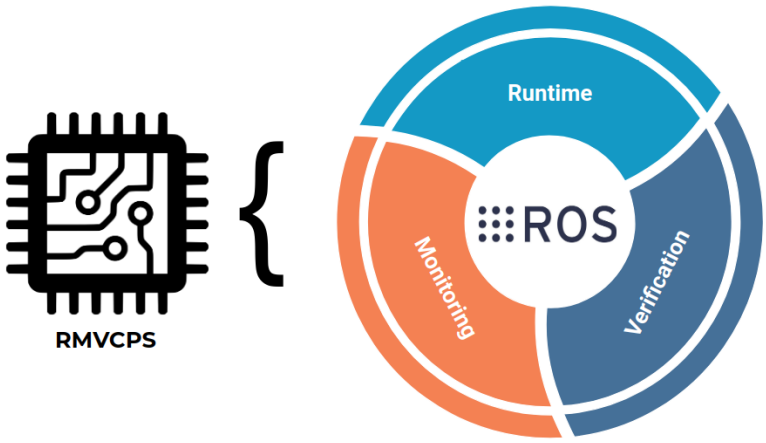


Figure 3.1: The problem addressed by our RMVCPS Framework

In a general context we can consider that the physical world is treated as continuous in relation to time, and the sensors make the discretization of that time by sampling the data at a given

frequency, which we call Variables. In turn, based on a specification, and through one or more variables we can, through relations derived from logical and / or algebraic calculations between them, obtain what we call Events. An Event communicates a significant change that happened. With an analysis of an Events stream, we can deduce the current state of the system, and from there create a State stream. (See Figure 3.2)

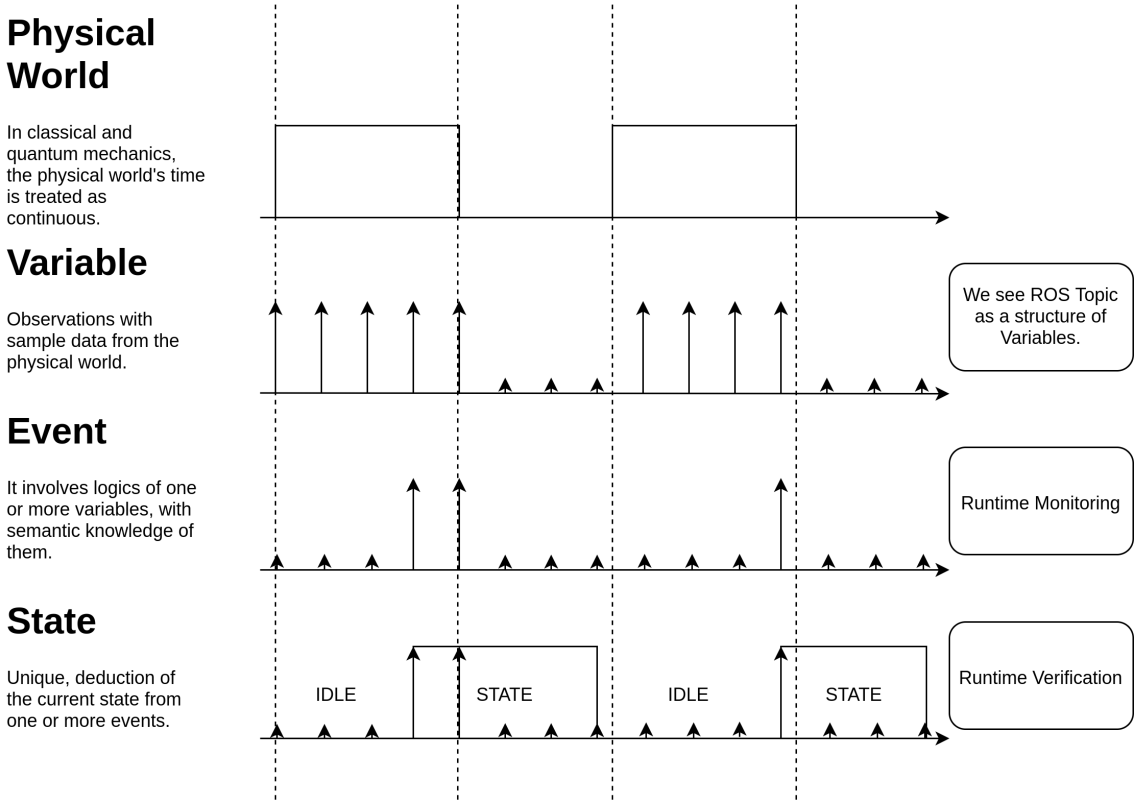


Figure 3.2: Division in different layers of abstraction.

It is important to clarify the following: we consider runtime monitoring to be different from runtime verification. By runtime monitoring, we consider a systematic process of collecting, filtering, down-sampling, extracting and analysing Events from time-series data (Variables) at runtime. In Figure 3.3 a possible generic runtime monitoring framework is exemplified. By runtime verification, we consider checking the correctness and providing evidence that the sequences of states meet the needs and requirements (of the corresponding specification) at runtime.

3.1.1 Monitoring distributed systems

Monitoring distributed systems brings with it a set of problems. The main issues in monitoring distributed systems are the following:

- Delays in transferring information, meaning that the information may become out of date.
- Variable delays in transferring information may result in events arriving out of order.

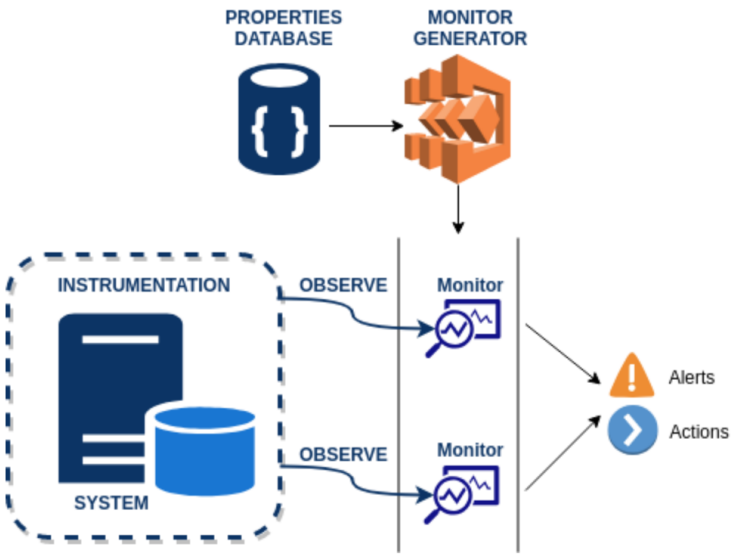


Figure 3.3: General overview of a possible generic runtime monitoring framework.

- The number of objects generating monitoring information in a large system can easily swamp monitors.
- In the likely situation that the distributed system is heterogeneous, a canonical form is needed to encode messages passed between heterogeneous machines.

3.2 Proposed Solution

While designing and debugging robotics software, it often becomes necessary to observe some state while the system is running. Although printf is a familiar technique for debugging programs on a single machine, this technique can be difficult to extend to large-scale distributed systems, and can become unwieldly for general-purpose monitoring. Instead, ROS can exploit the dynamic nature of the connectivity graph to "tap into" any message stream on system. Furthermore, the decoupling between publishers and subscribers allows for the creation of general-purpose visualizers.

Simple programs can be written which subscribe to a particular topic name and plot a particular type of data, such as laser scans or images, or a visualization program which uses a plugin architecture, like rviz program, which is distributed with ROS.

However, a more powerful concept would be to allow runtime monitoring or even runtime verification of ROS topics, with the possibility to create some boundaries between what is expected to happen and what is not.

So, we define RMVCPS (Runtime Monitoring and Verification of Cyber-Physical Systems) a set of 3 ROS Nodes that allow to apply monitoring, verification and intrusion detection in real-time to any ROS environment.

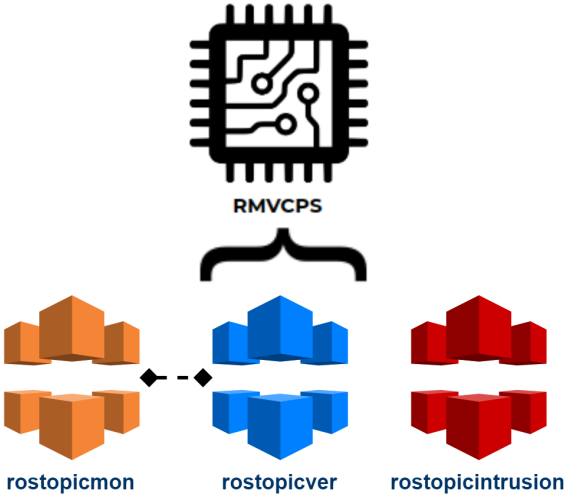


Figure 3.4: Division in RMVCPS Framework.

The first application, rostopicmon, is responsible for subscribing to ROS topics, storing the Variables in the database, as well as creating continuous queries that will allow extraction of events from the Variables, and also perform runtime monitoring of these Events. The second application, rostopicver, is responsible for estimating the current state from the events and using formal state sequence validation methods. The third application, rostopicintrusion, is responsible for detecting ROS Node's intrusions in ROS Topics.

3.3 Simulation Environments

To test the proposal solution in a way that allows for its refinement and incremental improvement, a simulation is an essential tool, with that it is possible to test and validate the application in different scenarios, preferably scenarios close to reality, as well as perform regression testing.

3.3.1 Simulation in Gazebo

Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. Gazebo have robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces. Best of all, Gazebo is free and open-source.

Gazebo is integrated with ROS, and provide the necessary interfaces to simulate a robot in Gazebo using ROS messages, services and dynamic reconfigure.

There is a simulation in Gazebo available, developed by Open Robotics (OR), where is possible to find a Toyota Prius with sensor data being published through ROS Topics. The car's throttle, brake, steering, and gear shifting are controlled by publishing a ROS message. A ROS node allows driving with a gamepad or joystick.

To increase complexity of the scenario we upgraded that version creating a set of three cooperative autonomous vehicles (Prius), so we can validate the RMVCPS framework usability in a more realistic simulated scenario. (See Figure 3.5)

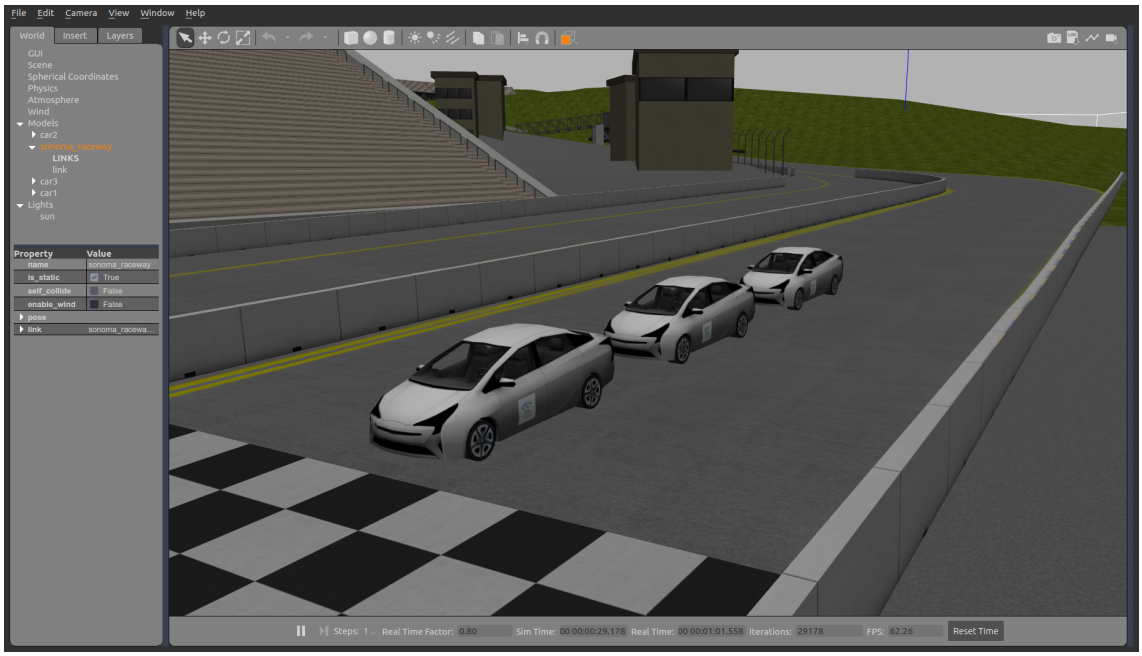


Figure 3.5: Simulation with three independent controlled Prius vehicles in the same Gazebo simulation.

It was a challenging upgrade, as it was necessary to change the entire plugin (PriusHybridPlugin.cc) responsible for controlling the car, since it was needed to place each car within a specific namespace, and configure the plugin to control each car within the associated namespace.

In the Load of the plugin, we added a parameter to load the car namespace from gazebo launch action.

```
    this->dataPtr->robot_namespace_ = "";
    if (_sdf->HasElement("robotNamespace"))
        this->dataPtr->robot_namespace_ =
            _sdf->GetElement("robotNamespace")->Get<std::string>();
```

This namespace is previously configured in the simulation ".launch" file.

```
<param name="robot_description"
    command="$(find xacro)/xacro --inorder
    '$(find prius_description)/urdf/prius.urdf'
        namespace:=$(arg car_name) tf_prefix:=$(arg car_name)_tf" />
```

In addition, we took the opportunity to develop a form of communication between the cars, sending a structure with information regarding the current values of the car variables. To enable this we setup a ROS NodeHandle with the associated namespace, so every topic advertised could be inside the namespace.

```
ros::NodeHandle nh(this->dataPtr->robot_namespace_);
```

After that, in the PriusHybridPlugin::Update() function, a code was added in order to periodically send a message to the XNetwork ROS topic, which is a topic that simulates the broadcast environment network.

```
if (curTime - this->dataPtr->lastCAM_MsgSentTime > CAM_PERIOD)
{
  this->dataPtr->lastCAM_MsgSentTime = curTime;
  ros_its_msgs::CAM_simplified CAM_msg;
  ignition::math::Pose3d pose = dPtr->chassisLink->WorldPose();

  CAM_msg.car_name = this->dataPtr->robot_namespace_;
  CAM_msg.latitude = pose.Pos().X();
  CAM_msg.longitude = pose.Pos().Y();
  CAM_msg.altitude_altitudeValue = pose.Pos().Z();
  CAM_msg.heading_headingValue = pose.Rot().Yaw();
  CAM_msg.speed_speedValue =
      this->dataPtr->chassisLinearVelocity.Length();
  //forward(0), backward(1), unavailable(2)
  CAM_msg.driveDirection =
      this->dataPtr->directionState;
  CAM_msg.steeringWheelAngle_steeringWheelAngleValue =
      (this->dataPtr->flSteeringAngle +
      this->dataPtr->frSteeringAngle) / 2;
  CAM_msg.gasPedalPercent_Value =
      this->dataPtr->gasPedalPercent;
  CAM_msg.brakePedalPercent_Value =
      this->dataPtr->brakePedalPercent;

  this->dataPtr->XNetwork_Pub.publish(CAM_msg);
}
```

However, all cars will receive data regarding the current values of all cars, which means that they will also receive their own data. To avoid this, and only receive data from the remaining

cars, we have implemented a filter that publishes for ROS topic RXNetwork only data from the remaining cars.

```
void PriusHybridPlugin::OnNetworkRX(
const ros_its_msgs::CAM_simplified::ConstPtr& msg){
    if(msg->car_name != this->dataPtr->robot_namespace_)
        this->dataPtr->RXNetwork_Pub.publish(msg);
}
```

This implementation uses the environmental data obtained by the sensor network (see Figure 3.6) as well as vehicle-to-vehicle communications at a high automation level. This action implies to execute control in the direction, acceleration or deceleration, as well as monitoring the driving environment and also the fallback performance of the dynamic driving task.

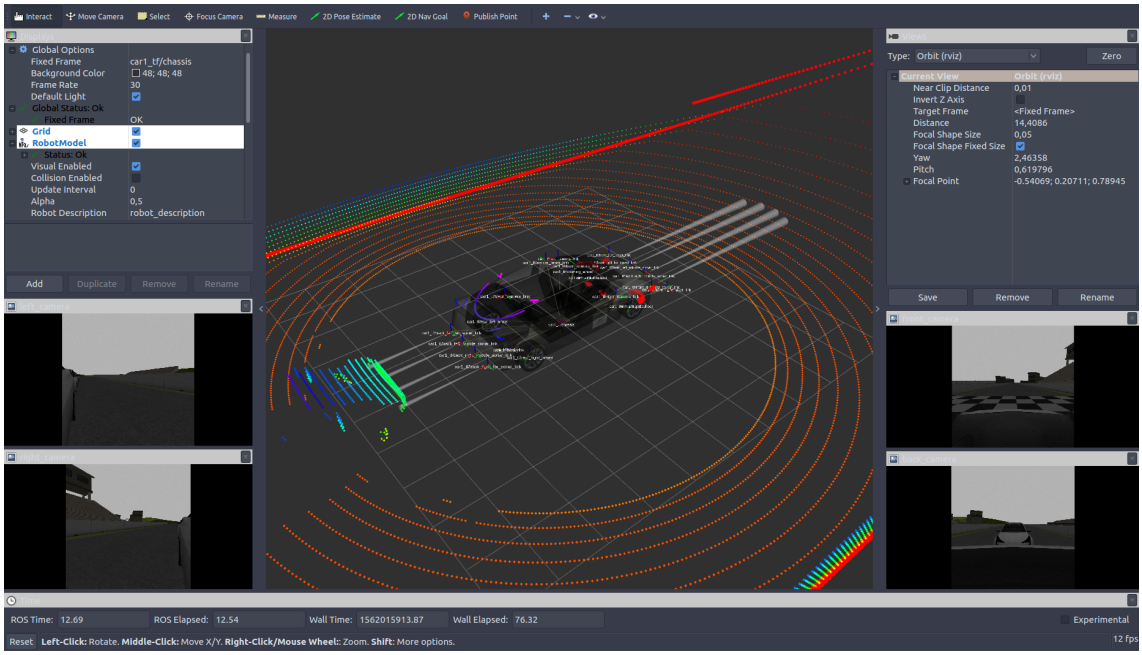


Figure 3.6: RVIZ view of ROS sensors included in Gazebo simulation.

3.3.2 Simulation in CARLA

To improve the monitoring and verification of physical proprieties of the car, we use a more powerful simulator, a free, open-source simulator powered by Unreal Engine that has been designed from day one to support the development of autonomous driving systems, CARLA.

CARLA simulation [1] environment consists of complex urban layouts, buildings and vehicles rendered in high quality, and compared to Gazebo, allow for a much better realistic representation of real world scenarios. (See Figure 3.7)

CARLA also provides a CARLA-ROS bridge that enables data acquisition from the simulation in native ROS message formats, where the data can be recorded, stored, and processed by the same code running on the actual vehicle.



Figure 3.7: Simulation created in CARLA.

3.4 ROSTOPICMON

Rostopicmon is responsible for subscribing to ROS topics, store the Variables in the database, as well as creating continuous queries that will allow extraction of events from the Variables or others Events, and also configure the runtime monitoring of these Events. (See Swimlane diagram in Figure A.1)

3.4.1 Subscribing ROS Topics (Collecting data)

ROS supports a variety of variables data types (see Figure 3.8) and also supports custom messages which are user-defined and that can be used to extend the set of message types currently supported by default.

For that reason, following good practices, the collection of data by the framework of Runtime and Verification of Cyber-Physical Systems (RMVCPS) should be as dynamic as possible, in a type-agnostic way, to support different types and structures.

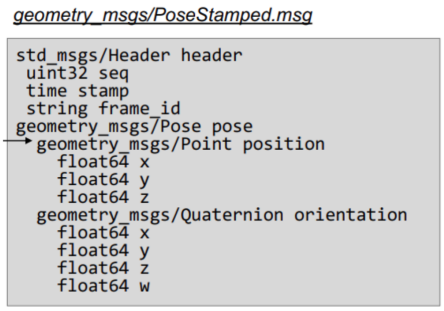


Figure 3.8: Example of a ROS Message.

3.4.1.1 ROS Introspection

Normally, during compilation time the publisher or subscriber object are instantiated with the message type's class. The rostopicmon only works as a subscriber, and, at compilation time, it does not know what type of message the publisher instantiated, even if it is indicated in a specification file, will only work with ROS standard message types (std_msgs), and not with custom message types.

To enable subscribing ROS Topics, in a type-agnostic way, a method is needed to bind publishers and subscribers to a message type during runtime.

The common way is to use an interpreted language, like Python, that is able to load new data types during runtime and generate objects based on the name of the message structure type. However, this reduces performance: interpreted languages are generically much slower compared to compiled programs.

Another way is to use a code generation tool to make a large switch structure, which combines the name of a message type, to the instantiation of an actual object. Using this type of code generation results in a large code file and program, since all possible messages are coded inside it, and will be only possible to code messages that are standard or known. Also, using code generation adds another step in the design process, since each time message definitions change in ROS, the code generation need to be rerun.

A better approach is to use the ShapeShifter class in ROS that provides a method to subscribe data without a predefined message type. (See Figure 3.9). It requires however a custom implementation of the serialization and deserialization of the message's variables, and the checksums and message definition need to be set by the user: these are normally specified in the generated header files of the message type. Two classes were derived, performing these actions during runtime. (See Figure 3.10)

As arguments of the Topic Callback is passed the message (msg) in a generic ShapeShifter class, the topic class associated with the message, and an instantiating of the introspection parser.

```
void topicCallback(const topic_tools::ShapeShifter::ConstPtr& msg,
const Topic &topic_element, RosIntrospection::Parser& parser);
```

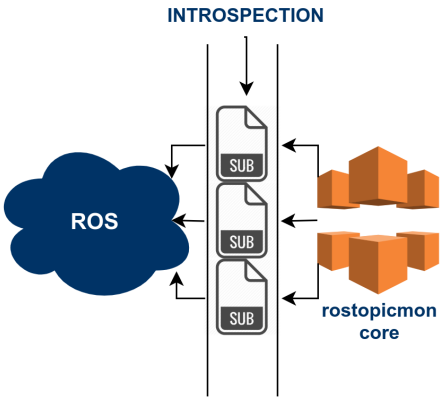


Figure 3.9: Implementation of intropection in ROS Topics

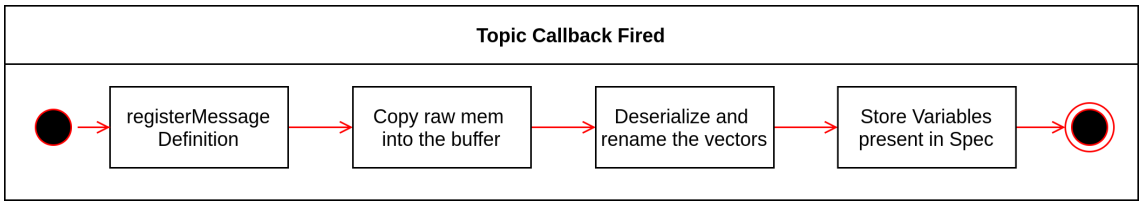


Figure 3.10: Flow Diagram of deserialize ROS messages from ROS Topics

---

In the Topic callback, first we get the message datatype and definition, and register to the introspection parser. We don't need to worry about doing this more than once, because already registered messages are not overwritten.

```
const std::string&  datatype   =  msg->getDataType();
const std::string&  definition =  msg->getMessageDefinition();

parser.registerMessageDefinition( topic_element.get_name(),
RosIntrospection::ROSType(datatype), definition );
```

Next, we instantiated the buffer to store raw data from message, the flat_containers map and renamed_vectors map. This maps will be used to parse the message in double or string datatypes according with the message structure datatype. After, we copied the raw memory to the buffer.

```
static std::vector<uint8_t> buffer;
static std::map<std::string,RosIntrospection::FlatMessage>
    flat_containers;
static std::map<std::string,RosIntrospection::RenamedValues>
```

```
    renamed_vectors;

RosIntrospection::FlatMessage&   flat_container =
    flat_containers[topic_element.get_name()];
RosIntrospection::RenamedValues& renamed_values =
    renamed_vectors[topic_element.get_name()];

// copy raw memory into the buffer
buffer.resize( msg->size() );
ros::serialization::OStream stream(buffer.data(), buffer.size());
msg->write(stream);
```

Next, we can deserialize, parsing the raw data in the buffer, into flat_container map. In this flat_container we already have the structures values where the datatypes are strings. After we apply a name transform to convert the non-string datatypes in double.

```
parser.deserializeIntoFlatContainer( topic_element.get_name(),
    RosIntrospection::Span<uint8_t>(buffer), &flat_container, 100);
parser.applyNameTransform( topic_element.get_name(), flat_container,
    &renamed_values );
```

In the end we only need to find in flat_container map or in renamed_values the variables that are included in specification to store in the database.

3.4.2 Storage

To make comparisons with elements of the past, a database is needed. As we are dealing with how a given variable changes over time, so called time-series, the best to use will be a time-series database (TSDB).

There are several options for a database type which is optimized for time series or time-stamped data, but choosing the right time-series database between more than 15 time series vendors [11] in the industry can be a headache.

According with [34], there are 4 commonly used technologies for time-series data: InfluxDB, Cassandra, OpenTSDB and Elasticsearch.

Also, in [34], we found a benchmark comparing the performance of InfluxDB against the 3 other commonly used time-series databases with respect to the data ingest performance, on-disk storage requirements and mean query response time. When comparing data ingestion performance, InfluxDB performed better than Cassandra by 5.3x, was also proven to be 8 times better than Elasticsearch and outperformed OpenTSDB by 5.0x. When evaluating on-disk compression,

InfluxDB outperformed Cassandra by 9.3x, outperformed Elasticsearch by by 4x to 16x, depending on the configuration used, and outperformed OpenTSDB by 16.5x. In query performance (server side aggregations), InfluxDB is up to 168x faster than Cassandra, outperformed Elasticsearch by proving to be 4x to 10x better query performance depending on how large is the data-set, and was proven to be 4.0x faster than OpenTSDB.

So, based on this benchmark, without a doubt, the best database to use is InfluxDB.

3.4.2.1 InfluxQL

InfluxQL is a SQL-like query language for interacting with InfluxDB and providing features specific to store and analyzing time series data. Example of a simple SELECT query:

```
1 SELECT "level description"::field,"location"::tag,"water_level"::field FROM "
    h2o_feet"
```

The query selects the level description field, the location tag, and the water_level field from the h2o_feet measurement.

3.4.2.2 Downsampling and Retention Policy

Working with that much data over a long period of time can create storage concerns. A natural solution is to downsample the data; keep the high precision raw data for only a limited time.

Retention Policy (RP) describes for how long RMVCPS keeps data, comparing local server's timestamp to the timestamps on data and deletes data that are older than the RP's duration.

In InfluxDB to create a RP we just need to run the following command:

```
1 CREATE RETENTION POLICY <retention_policy_name> ON <database_name> DURATION <
    duration> REPLICATION <n>
```

Where the DURATION clause determines how long InfluxDB keeps the data and the REPLICATION clause determines how many independent copies of each point are stored in the cluster. By default, the replication factor n usually equals the number of data nodes. However, if you have four or more data nodes, the default replication factor n is 3.

3.4.3 Event Extraction

As mention in Chapter 3.1, to do runtime monitoring analysis we have to extract Events from the Variables previously stored. As we are dealing with an SQL-Like database, queries returning aggregate, summary, and computed data are frequently used. These summary queries are generally expensive to compute since they have to process large amounts of data, and running them over and over again just wouldn't scale. But with InfluxDB, we can pre-compute and store the aggregates query results so that they are ready when we need them, and it significantly speed up summary

queries, without overloading the database. In InfluxDB this feature is called Continuous Queries. Continuous queries (CQ) are just InfluxQL queries that run automatically and periodically on realtime data and store query results in a specified measurement.

3.4.3.1 Processing traces

To extract events from Variables we used the Continuous Queries (CQ) feature. Besides that, taking measurements at irregular intervals is common in distributed CPS, so time series may have missing observations or may have multiple time-series with different frequencies: it can be useful to make some type of regression (e.g., linear regression) to fill the missing observations or match multiple time-series with different frequencies. For the missing observation completion InfluxDB supports some "fill" operations:

- "none": (Default) Reports no timestamp and no value for time intervals with no data
- "linear": Reports the results of linear interpolation for time intervals with no data.
- "null": Reports null for time intervals with no data but returns a timestamp.
- "previous": Reports the value from the previous time interval for time intervals with no data.
- "(*any value*)": Reports the given numerical value for time intervals with no data. (e.g. "0")

3.4.3.2 Supported operations

InfluxDB also provides tools to support algebraic operations that are useful to extract complex events. The operations can be performed in one or multiple Variables or even Events. It can make the following operations:

- Aggregations: count, distinct, integral, mean, median, mode, spread, stddev, sum
- Selectors: bottom, first, last, max, min, percentile, sample, top
- Transformations: abs, acos, asin, atan, atan2, ceil, cos, cumulative_sum, derivative, difference, elapsed, exp, floor, histogram, ln, log, log2, log10, moving_average, non_negative_derivative, non_negative_difference, pow, round, sin, sqrt, tan
- Mathematical Operators: addition, subtraction, multiplication, division, modulo, bitwise AND, bitwise OR, bitwise Exclusive-OR
- Equalities and Inequalities: =,!=,<,>,<=,>=,<>

3.4.4 Dashboard Visualizer

To view the data collected, whether from Variables or Events, we created a dashboard (See Figure 4.1 in Chapter 4), using Chronograf technology, which allows us to graphically visualize all the data present in the Influx database. (See Figure 4.1). The dashboard is very dynamic and we can adapt it to different situations, adding or removing modules that are useful or unnecessary to visually analyze.

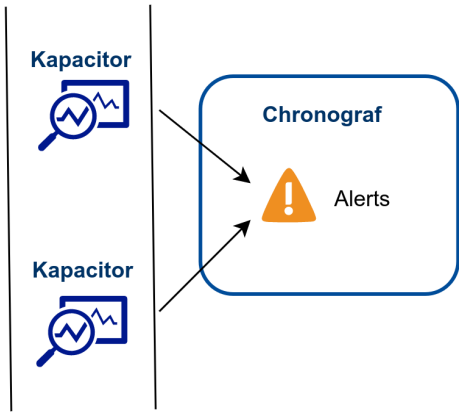


Figure 3.11: Diagram of the visualization dashboard

3.4.5 Real-Time Streaming Data Processing Engine

We used Kapacitor as runtime monitoring engine, because it is a native data processing engine for InfluxDB. Kapacitor is designed to process streaming data in real-time. It can process both stream and batch data from InfluxDB. It can be deployed across the infrastructure as both a pre-processor to downsample and perform advanced analytics. (See Figure 3.12).

Like ROS, kapacitor's alerting system follows a publish-subscribe design pattern. Alerts are published to topics and handlers subscribe to a topic. This pub/sub model and the ability for these to call User Defined Functions make Kapacitor very flexible to act as the control plane in any environment, performing tasks based on runtime monitoring.

3.4.5.1 TICKscript Domain Specific Language

Kapacitor uses a Domain Specific Language (DSL) named TICKscript to define tasks involving the extraction, transformation and loading of data and involving, moreover, the tracking of arbitrary changes and the detection of events within data. One common task is defining alerts.

The TICKscript DSL is used to define a monitor specification in a .tick file, chaining together the invocation of data processing operations defined in nodes.

Each script has a flat scope and each variable in the scope can reference a literal value, such as a string, an integer or a float value, or a node instance with methods that can then be called.

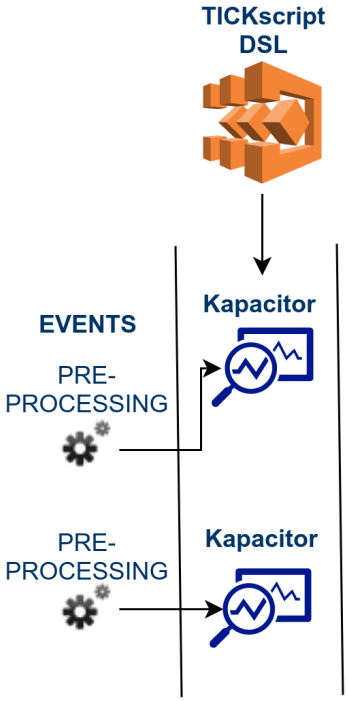


Figure 3.12: Diagram of Event extraction.

It allows the scripting to be done using lambda expressions to define transformations on data points as well as define boolean conditions that act as the filter.

3.4.5.2 Handlers, Actions and Alerts

The alert node triggers an event of varying severity levels and passes the event to event handlers. There are 3 different alert levels, based on severity:

- INFO
- WARNING
- CRITICAL

The criteria for triggering an alert is specified via a lambda expression present in .tick file. Example of a simple TICKscript file with alert with different levels of severity.

```
stream
  |from()
    .measurement('EderivativeVelocity')
    .where(lambda: "sensor" == 'mainsensor')
  |alert()
    .info(lambda: "value" > 60)
```

```
    .infoReset(lambda: "value" < 50)
    .warn(lambda: "value" > 70)
    .warnReset(lambda: "value" < 60)
    .crit(lambda: "value" > 80)
    .critReset(lambda: "value" < 70)
```

After some detected violation it is necessary not only to proceed to the alert but also to do some specific action. For this it is possible to add some actions to the clause alert, such as:

```
    .post("http://example.com/api/alert")
    .post("http://another.example.com/api/alert")
    .tcp("exampleendpoint.com:5678")
    .email('oncall@example.com')
```

Natively ROS is not supported, however it is possible to run a shell script, with:

```
    .exec('/home/ps/Projects/rostopicmon/kapacitor_rospub.sh')
```

So we created a simple script to allow the publication of alerts for the ROS network.

```
#!/bin/bash

var=$(head -1)

source /opt/ros/kinetic/setup.bash &&
    /opt/ros/kinetic/bin/rostopic pub --once alert_topic
    std_msgs/String $(printf '%q' $var) &
```

3.4.6 Rostopicmon monitoring environment configuration

The configuration of the monitoring environment begins with specification of which ROS topics should be subscribed, also the data structure intended to be stored in the database (Variables) and the definition of events extraction. All of user definition is based on a YAML, a human-readable data serialization language. YAML is commonly used for configuration files. The standard YAML in-line key-values pair is used for definition of hierarchically related data structures. In the YAML structure, the specification can also contain collections, enumerations, and hooks for the procedures to initialize the data in the structures on a by-variable basis.

3.4.6.1 Database configuration

To be able to access the InfluxDB database API, we need to configure the following parameters:

- ip: IP of InfluxDB (can be local or remote).
- port: Port used by influxdb.
- db_name: Database name used to store time-series values.
- epoch: Granularity of time desired, n, nanoscond, u, microsecond, ms, milisecond, s, second, m, minute and h, hour.
- username: Username to access database. (if exists)
- password: Password to access database. (if exists)

Example of YAML database configuration:

```
database:
  - ip: "127.0.0.1"
  - port: "8086"
  - db_name: "rmvcps_db"
  - epoch: "u" #ns, u, ms, s, m, h
  - username: "rmvcps"
  - password: "rmvcps"
```

3.4.6.2 ROS Topics configuration

Next, we have to configure the topics that rostopicmon will be listening to. For that, we simply have to configure the following parameters:

- name: The desired name for the topic. It has nothing to do with ROS, but it is just an abbreviation of the topic's complete path to make it easier to be identified. At the beginning we should include an anchor '&' to define a chunk of configuration to be reference by variables.
- topic: ROS topic full path.

Example of YAML ROS topic configuration:

```
topics:
  - name:           &car1_info "car1_info"
    topic:          "/carla/hero/vehicle_status"
```

3.4.6.3 Variables configuration

After ROS topics are configured, we need to configure the variables from these topics that will be stored in the database. For that, we need to configure the following parameters:

- name: The desired name for the variable. It has nothing to do with ROS, but it is just an abbreviation of the variable's complete path to facilitate identification.
- topic: We should refer to the anchor previous created with the symbol '*' to define the topic the variable is associated to.
- topic_var_path: ROS variable structure path inside ROS Message.

```
variables:
  - name:           "Velocity"
    topic:          *car1_info
    topic_var_path: "velocity"
  - name:           "BrakePedal"
    topic:          *car1_info
    topic_var_path: "control/brake"
```

3.4.6.4 Events

When we have the Variables configured we can configure the Events obtained from Variables. For that, we need to configure the following parameters:

- event_name: The desired name for the Event. It has nothing to do with ROS, this name will be used in the database.
- event_resample: Describes how long the database resample the data (duration) with the clause "FOR" (eg. "FOR 1h") or the period interval of the extraction procedure with the clause "EVERY" (eg. "EVERY 1h"). We can include both (eg. "FOR 10s EVERY 2h"). If empty will process in runtime stream with the default retention policy.
- event_select: Function with aggregations, selectors, mathematical Operator or transformations as specification for the event extractor. We can make calculations including Variables or including others Events. See the functions supported in Section 3.4.3.2.
- event_where: Function with condition to run the event extractor, for example, we may want to extract events only when a Variable's value exceeds a certain limit. (eg. "Velocity > 10"). If empty will be always running.
- event_groupby: Time range for the Event extraction.

- event_fill: Function that changes the value reported for time intervals that have no data. See in Section 3.4.3.1 the supported fill functions.
- event_orderby: Used to sort the result-set in ascending or descending order withing "groupby" time range. Supports ASC or DESC. The default is ASC, ascending order, from the oldest to the newest. The clause "DESC" reverses the order. If empty, the default clause is chosen.
- event_limit_size: Limit the number of points stored in Event stream. (e.g. "1", will only have the last value)

```
events:
  - event_name: "EVelocityinKMH"
    event_resample: ""
    event_select: "mean(Velocity) * 3.6"
    event_where: ""
    event_groupby: "time(100ms)"
    event_fill: ""
    event_orderby: ""
    event_limit_size: ""
  - event_name: "EderivativeVelocity"
    event_resample: ""
    event_select: "derivative(mean(Velocity), 1s)"
    event_where: ""
    event_groupby: "time(100ms)"
    event_fill: ""
    event_orderby: ""
    event_limit_size: ""
```

3.4.6.5 Monitors

For the monitor we first have to write the TICKScript specification and store it in some folder inside the rostopicmon project, and configure the following parameters in the YAML file:

- monitor_name: The desired name for the Monitor.
- monitor_gen: Monitor generator framework. Right now, only supports "kapacitor".
- monitor_specification: Path to the TICKscript specification file of the monitor.

```
monitors:
  - monitor_name:       "monitor1"
```

```
monitor_gen:          "kapacitor"
monitor_specification:  "tick_monitors/loss_of_acceleration.tick"
```

All the configuration above should be included in a YAML file (config.yaml) included in the project rostopicmon. (See Figure 3.13)

```yaml
database:
  - ip: "127.0.0.1"
  - port: "8086"
  - db_name: "rmvcps_db"
  - epoch: "u" #ns, u, ms, s, m, h
  - username: "rmvcps"
  - password: "rmvcps"

topics:
  - name:           &carl_info "carl_info"
    topic:          "/carla/hero/vehicle_status"

variables:
  - name:           "Velocity"
    topic:          *carl_info
    topic_var_path: "velocity"
  - name:           "BrakePedal"
    topic:          *carl_info
    topic_var_path: "control/brake"
  - name:           "HandBrake"
    topic:          *carl_info
    topic_var_path: "control/hand_brake"
  - name:           "GasPedal"
    topic:          *carl_info
    topic_var_path: "control/throttle"

events:
  - event_name: "EVelocityinKMH"
    event_resample: "1m"
    event_select: "mean(Velocity) * 3.6"
    event_where: ""
    event_range: "time >= now() - 1s AND time < now()"
    event_groupby: "time(100ms)"
    event_fill: ""
    event_orderby: ""
    event_limit_size: ""
  - event_name: "EderivativeVelocity"
    event_resample: "1m"
    event_select: "derivative(mean(Velocity), 1s)"
    event_where: ""
    event_range: "time >= now() - 1s AND time < now()"
    event_groupby: "time(100ms)"
    event_fill: ""
    event_orderby: ""
    event_limit_size: ""


monitors:
  - monitor_name:       "monitor1"
    monitor_gen:        "kapacitor"
    monitor_trigger:    "true"
    monitor_rate:       "10" #Hz
    monitor_specification:  "tick_monitors/maxvelocity.tick"
```

Figure 3.13: Example of complete YAML monitor configuration file.

3.4.7 Summary Diagram of the rostopicmon enviroment

It is possible to summarize the entire rostopicmon operating environment in the diagram of Figure 3.14. Starting from left to right it is possible to cover ROS, which does not have to be limited to just one platform but can also be monitoring topics connected to the same ROS network, but running on a different platform. Any simulated platform, such as the Gazebo and CARLA simulators, is also supported. The subscription of these ROS topics is done in a type-agnostic way, so that even with customized messages it is possible to read the Variables present in the structures of the ROS messages quickly and without the need for manual configuration of the ROS message data types followed by compilation. Then, all data is then stored in the InfluxDB database, which will then be pre-processed for event extraction. In the end, these Events are monitored and in case of any anomaly, in addition to show an alert, it is also possible to take actions directly on the ROS network.

3.5 ROSTOPICVER

As mentioned in Section 3.1, rostopicver is responsible for estimating the current state of the system being observed from the events. With a sequence of state changes, we are able to use formal monitors that can validate the correctness of those sequences of states.

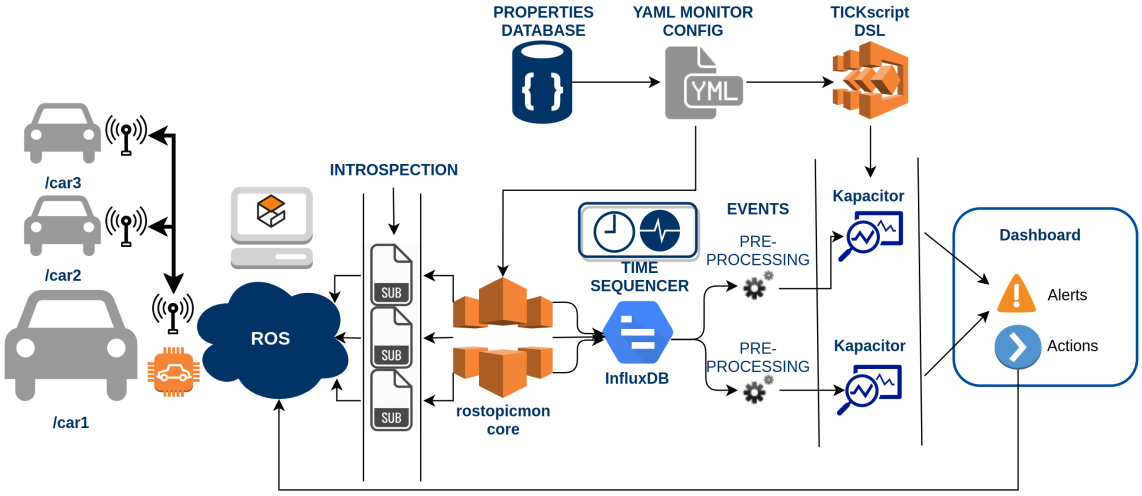


Figure 3.14: General overview of proposal solution for the rostopicmon.

3.5.1 Estimating States From Events

Our goal is to perform runtime verification from runtime data collected in the form of events. However, runtime verification frameworks are typically based on temporal logic, and the semantics of that logic considers states rather than events. Since we don't have the current state of the environment, to enable such verification monitors, we need to estimate it. In the literature of reactive application development [20] the authors explain a similar concept to state estimation from events, named Event Sourcing. By definition, Event Sourcing is a way to atomically update state, since it uses a radically different, event-centric approach to persistence. An object is persisted by storing a sequence of state changing events. Whenever an object's state changes, a new event is appended to the sequence of events. The current state is reconstructed by replaying its events. Since we already have the stream of Events stored in the database, we only need to estimate the state change by analysing its events. As mentioned in the reactive application development literature, we call the estimation of states, projection, views on state or the state history, created and updated by the projector. The projector runs database queries using the specification of states included in YAML configuration file, and updates the projection. The state specification has to be mapped in a Finite-state transducer machine [28], that consists of a finite number of states which are linked by transitions labeled with an input/output pair. To configure the specification of the states we need to configure the following parameters in the rostopicver YAML file:

- states: List of states.
- initial: Initial state.
- transitions: List of changes from one state to another.
- source: Source state of transition.

- dest: Destination state of transition.
- inputtapeEvent: EventA
- outputtapeEvent: EventB

```
states:
  - A
  - B
  - C

initial:
  - A

transitions:
  - source: A
    dest: B
    inputtapeEvent: EventA
    outputtapeEvent: EventD
  - source: B
    dest: C
    inputtapeEvent: EventD
    outputtapeEvent: EventZ
```

3.5.2 Verification Monitor Generation

To build monitors we used the rmtld3synth toolchain which is able to automatically generate cpp11/ocaml monitors from a fragment of metric temporal logic with durations, so can be integrated with ROS C++. This logic is a three-valued logic that extends a restricted fragment of metric temporal logic with durations that allows us to reason about explicit time and temporal order of durations, which is sufficient for the test cases proposed.

Rmtld3synth formulas can be syntactically connected using some temporal operators and the relation '<' for terms. Consider the formula:

$$(HardBraking \rightarrow ((AntiLockBrake \vee HardBraking)\, U_{<10}\, Stopped)) \wedge \int^{10} HardBraking < 4 \tag{3.1}$$

Formula 3.1 describes that if the system is in the state HardBraking, then it will reach state AntiLockBrake or state HardBraking until the state Stopped is reached in less than 10 time units; and, at the same time, the duration of the system in state HardBraking shall be less than four time units during the next 10 time units. This is just a simple safety condition in the automotive domain

that can be established by imposing strict time bounds in limits in the duration of these systems. Other types of conditions can be monitored by coupling monitors specified in rmtld3synth, e.g, scheduling of operating system tasks, communication properties, among many others, as long as we are able to capture the relevant events and be able to estimate the corresponding states.

To generate the formal verification monitor we need to configure the following parameters in the rostopicver YAML file:

- monitor_name: The desired name for the Verification Monitor.
- monitor_gen: Monitor generator framework. Right now, in verification mode only supports "rmtld3synth".
- monitor_trigger: Trigger to start the monitor. If true, always on.
- monitor_rate: Frequency rate of the monitor.
- expression_string: Verification monitor expression in LaTex format.
- monitor_range: Time range of monitor. Should be configured according with monitor expression.

```
monitors:
  - monitor_name:       "ABSCheck"
    monitor_gen:        "rtmld3synth"
    monitor_trigger:    "true"
    monitor_rate:       "10" #Hz
    expression_string:  "(HardBraking \\rightarrow ((AntiLockBrake
        \\lor HardBraking) \\until{<10} Stopped))
        \\land \\int^{10} HardBraking < 4"
    monitor_range:      "time >= now() - 4s AND time < now()"
```

The parsing of the monitor expression needs to be done before compilation, as the rmtld3synth toolchain generates C ++ libraries and monitor code that needs also to be included before rostopicver compilation. Then, for the compilation of the monitors in C ++ code we created a script that allows to parse the YAML configuration file and compile the expression:

```
#!/usr/bin/env bash
# shellcheck disable=SC1003

parse_yaml() {
  (..)
}
```

```
create_variables() {
    local yaml_file="$1"
    local prefix="$2"
    eval "$(parse_yaml "$yaml_file" "$prefix")"
}

create_variables config.yaml

for i in "${!monitors__monitor_gen[@]}"
do
  if [ ${monitors__monitor_gen[i]} = "\"rtmld3synth\"" ]; then
    s=${monitors__expression_string[i]#*'"'}; s=${s%'"'*}
    mkdir -p mon1
    rm -f mon1/*
    rmtld3synth --synth-cpp11 --input-latexeq "$s" --out-src="mon1"
        --verbose 2 --config-file default.config &&
        cp mon1/mon0_compute.h src/rmvcps/ros_cps/ &&
        echo "[OK] Monitor [${monitors__monitor_gen[i]}] compiled!"
   else
    echo "Monitor type [${monitors__monitor_gen[i]}] not supported!"
  fi
done
catkin_make
source devel/setup.sh
echo "[OK] RMVCPS compiled!"
```

The formal verification monitor runs periodically at a frequency described in the configuration file. The flow diagram for each iteration can be seen in Figure 3.15.

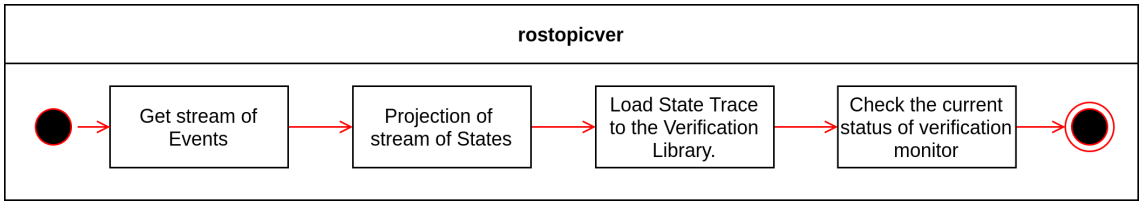


Figure 3.15: Flow Diagram of rostopicver.

In Figure 3.16 it is possible to see an example of the rostopicver output running in a ROS environment.

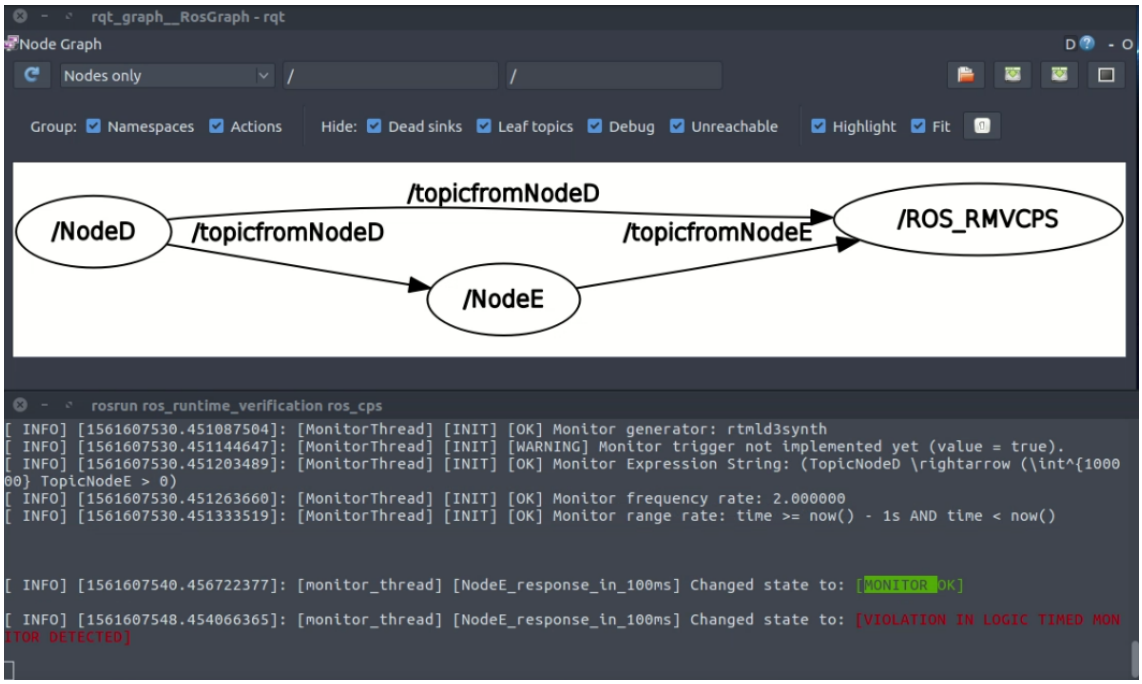


Figure 3.16: rostopicver example output.

3.6 ROSTOPICINTRUSION

3.6.1 ROS Computation Graph

In ROS, as mention in Section 2.1.3.2, nodes are responsible for doing computational processing. The network of ROS nodes is called computation graph. This graph also includes the ROS Master, Parameter server, Messages, Topics, Services, and Bags, running in ROS environment.

3.6.2 ROS Master

ROS Master is much like a DNS server [26]. When any ROS node starts in the system, it will start looking for ROS Master and register the name of the node in it. So ROS Master has the details of all nodes currently running on the ROS system. When any details of the nodes change, it will generate a call-back and update with the latest details. These node details are useful for connecting with each node.

The same happens when a node starts publishing or subscribing a topic, the node will give the details of the topic such as name and data type to ROS Master. ROS Master will check whether any other nodes are subscribed to the same topic. If any nodes are subscribed to the same topic, ROS Master will share the node details of the publisher to the subscriber node. After getting the node details, these two nodes will interconnect using the TCPROS protocol, which is based on TCP/IP sockets. After connecting to the two nodes, ROS Master has no role in controlling them.

3.6.3 Running ROS Topic intrusion monitor

The ROS communication related packages including core client libraries such as roscpp and rospython and the implementation of concepts such as topics, nodes, parameters, and services are included in a stack called ros_comm [9].

This stack in addition to include tools such as rostopic, rosparam, rosservice, and rosnode to introspect the preceding concepts, it also contains the ROS communication middleware packages and these packages are collectively called ROS Graph layer.

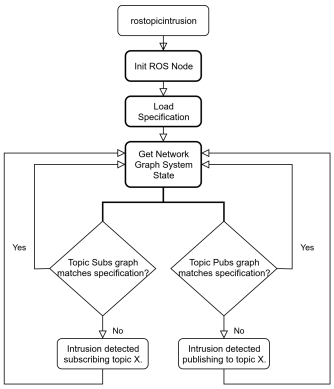


Figure 3.17: Flowchart of rostopicintrusion Node.

We created a ROS Node named rostopicintrusion, that uses the package rosgraph (ROS Graph layer) to have access to ROS Master logs to be able to monitor the links to the topics, either from the publishing or subscription, comparing with specification to check if the topics are subscribed or published from nodes present in the specification.

In Figure 3.18 it is possible to see an example of the rostopicintrusion output running in a ROS environment.

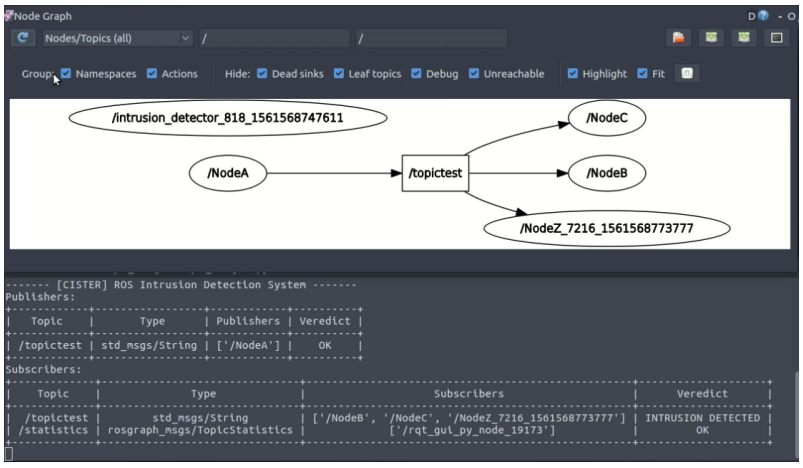


Figure 3.18: rostopicintrusion example output.

Chapter 4

Experiments

In this chapter, we demonstrate the configuration of the runtime monitor and fault detection. We also analyze the performance and impact on the system.

4.1 Results

4.1.1 Hardware Specifications

The benchmarking was executed using a generic notebook, Asus X402CA laptop, with the following specifications:

- OS Name: Linux Ubuntu 16.04
- System Type: x64-based PC
- Processor: Intel ®Celeron ®1007U dual core, 1.5 GHz
- Installed Physical Memory (RAM): 4GB DDR3
- L2 Cache Size: 512 KB
- L3 Cache Size: 2048 KB

4.1.2 Software Specifications

The benchmarking was executed using the following software specifications:

- ROS: kinetic
- InfluxDB: 1.7.5
- Kapacitor: 1.5.2
- Chronograf: 1.7.9

4.1.3 Test Specifications

The test was executed using the following test specifications in ROS Node Publisher (CARLA Simulator):

- Duration: 100s
- Data transmited size: 6.4MB
- Number of messages: 18391
- Compression: none
- Number of ROS topics in ROS Network: 13 (See List of all published topics in Table 4.1)

In rostopicmon, the test was carried out with the following configuration:

- Topics subscribed: 1
- Number of different Variables stored simultaneously: 20
- Total number of Variables stored: 36400
- Number of Events extractors: 4
- Number of Monitors: 1

ROS Topic	**messages (#)**	**ROS Type**
/carla/actor_list	1	carla_msgs/CarlaActorList
/carla/hero/gnss/front/fix	2006	sensor_msgs/NavSatFix
/carla/hero/objects	1820	derived_object_msgs/ObjectArray
/carla/hero/odometry	1820	nav_msgs/Odometry
/carla/hero/vehicle_info	1	carla_msgs/CarlaEgoVehicleInfo
/carla/hero/vehicle_status	1820	carla_msgs/CarlaEgoVehicleStatus
/carla/marker	3640	visualization_msgs/Marker
/carla/objects	1818	derived_object_msgs/ObjectArray
/carla/status	1820	carla_msgs/CarlaStatus
/carla/world_info	1	carla_msgs/CarlaWorldInfo
/clock	1821	rosgraph_msgs/Clock
/rosout	5	rosgraph_msgs/Log
/tf	1818	tf2_msgs/TFMessage

Table 4.1: Number of message published by each topic for 100 seconds.

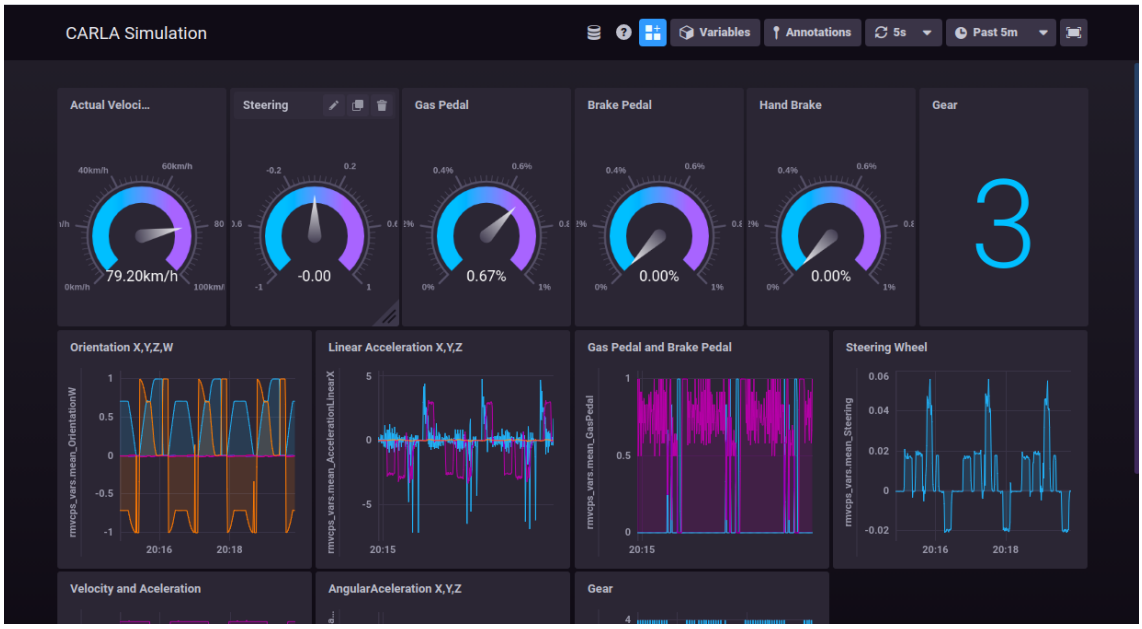


Figure 4.1: Dashboard of CARLA Simulation Environment.

4.1.4 Dashboard

As mention in Section 3.4.4 in Chapter 3, we create a dynamic dashboard to monitor visually all the Variables and Events that are stored in the database. (See Figure 4.1)

4.1.5 Delay measurements

We focus our delay measurements in the time between reception of ROS message by the rostopic-mon callback and the storage in InfluxDB. (See Figure 4.2)

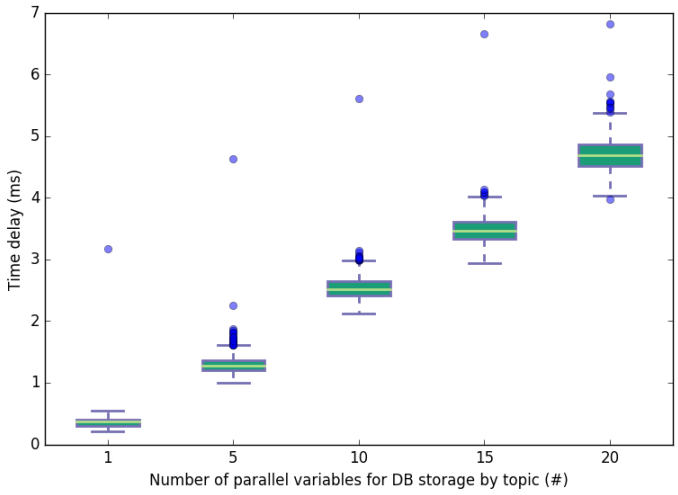
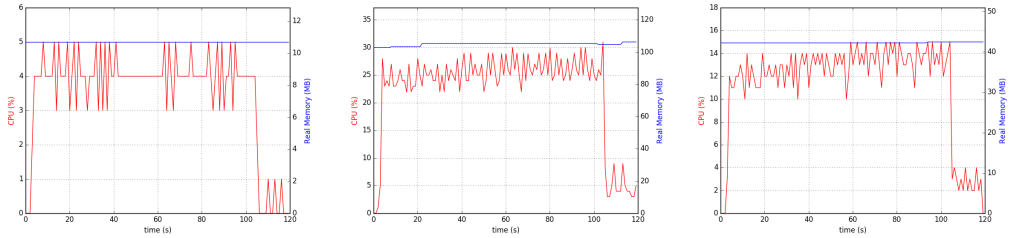


Figure 4.2: Time delay between reception by the rostopicmon and the storage in InfluxDB.



(a) rostopicmon. (b) InfluxDB. (c) Kapacitor.

Figure 4.3: CPU and memory overhead of used tools.

4.1.6 CPU and Memory Overhead

To record the CPU and memory activity of an existing processes and make a plot graph, we used the psrecord tool, a small utility that uses the psutil library, a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python. (See Figure 4.3)

The obtained results indicate that the overhead introduced could be acceptable to some classes of CPS, at least in simulations in order to increase their accuracy, but for very high demand application we may need another approach.

Chapter 5

Conclusions and Future Work

In these thesis, we have described an approach to perform runtime monitoring and verification to Cyber-Physical Systems running the Robot Operating System. The essential test showed that the principle worked as intended. It is possible to subscribe any type of ROS topic without knowing beforehand the structure of the ROS message included in that topic. It is also now possible to check if nodes that publish or subscribe to ROS topics are authorized and, if they are not, detect the intrusion. It also was possible to make a projection from the events traces recorded onto states of the system and enable formal verification via the coupling of monitors generated using adequate frameworks, such as the one we have adopted.

We validated our approach by pluging it into scenarios implemented in two different simulators, with different bridges to ROS and different topics for their simulation engine.

We also make the code available as open source to contribute to the community. [7] [8] [6]

5.1 Further Work

The three frameworks created, rostopicmon, rostopicver and rostopicintrusion can be improved in various directions. Firstly is on its usability, since each of these framework have different and independent configuration files, and so a natural solution is to design a domain specific language that would allows to create a unique specification file for the monitoring system. Such specification language would also open the path to construct or plug static verification tools that could pre-validate the specification before starting the code generation process.

In terms of computational impact in the system being monitored it is possible to verify in the experiments and tests carried out that any of the frameworks have impact on the system. Hence, an approach to explore will be to use isolation mechanisms, e.g., based on hypervisors, in order to guarantee the computational isolation between the monitors and and the system to be monitored, sharing only communication.

In addition, we are preparing to submit a paper on the subjects covered in this thesis for a conference.

Appendix A

RMVCPS Swimlane Diagram

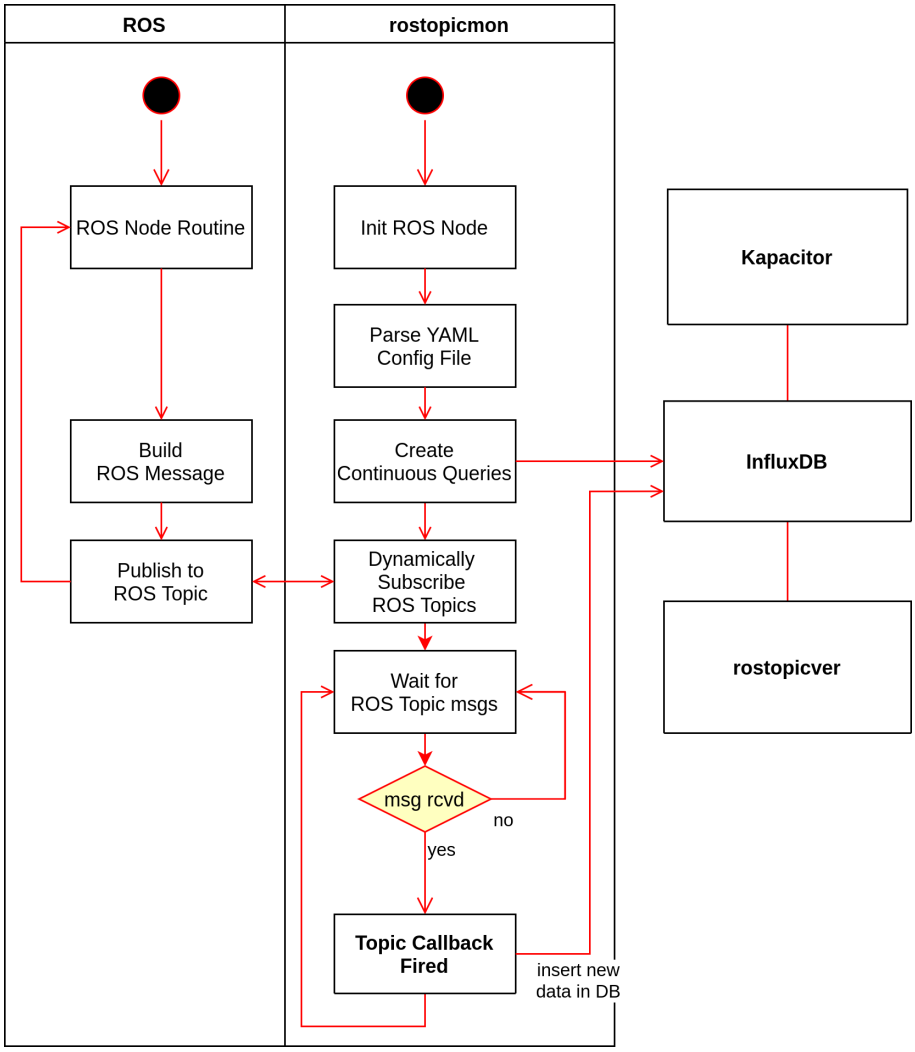


Figure A.1: Swimlane Diagram with relation between rostopicmon, rostopicver and rostopicintrusion

References

[1] Carla simulator. http://carla.org/. (Accessed on 01/20/2020).

[2] Clock - ros wiki. http://wiki.ros.org/Clock. (Accessed on 01/20/2020).

[3] Master - ros wiki. http://wiki.ros.org/Master. (Accessed on 01/20/2020).

[4] Messages - ros wiki. http://wiki.ros.org/Messages. (Accessed on 01/20/2020).

[5] Nodes - ros wiki. http://wiki.ros.org/Nodes. (Accessed on 01/20/2020).

[6] pedrosantospt/rostopicintrusion. https://github.com/pedrosantospt/rostopicintrusion. (Accessed on 01/20/2020).

[7] pedrosantospt/rostopicmon. https://github.com/pedrosantospt/rostopicmon. (Accessed on 01/20/2020).

[8] pedrosantospt/rostopicver. https://github.com/pedrosantospt/rostopicver. (Accessed on 01/20/2020).

[9] ros_comm - ros wiki. http://wiki.ros.org/ros_comm. (Accessed on 01/20/2020).

[10] Services - ros wiki. http://wiki.ros.org/Services. (Accessed on 01/20/2020).

[11] The state of the time series database market – alt + e s v. https://redmonk.com/rstephens/2018/04/03/the-state-of-the-time-series-database-market/. (Accessed on 01/12/2020).

[12] Topics - ros wiki. http://wiki.ros.org/Topics. (Accessed on 01/20/2020).

[13] Mikhail Auguston and Mark Trakhtenbrot. Synthesis of monitors for real-time analysis of reactive systems. In *Pillars of computer science*, pages 72–86. Springer, 2008.

[14] Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from eagle to ruler. *Journal of Logic and Computation*, 20(3):675–706, 2008.

[15] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

[16] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Time-triggered runtime verification. *Formal Methods in System Design*, 43(1):29–60, 2013.

[17] Feng Chen, Traian Serbanuta, and Grigore Rosu. Jpredictor. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 221–230. IEEE, 2008.

[18] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[19] Enable-S3 Consortium et al. Enable-s3 european project. *available at: www. enable-s3. eu/(accessed 22 January 2018).[Google Scholar]*, 2016.

[20] Duncan DeVore, Sean Walsh, and Brian Hanafee. *Reactive Application Development*. Manning Publications Co., 2018.

[21] Doron Drusinsky. The temporal rover and the atg rover. In *International SPIN Workshop on Model Checking of Software*, pages 323–330. Springer, 2000.

[22] Maximilian M Etschmaier and Gordon Lee. Defining the paradigm of a highly automated system that protects against human failures and terrorist acts and application to aircraft systems. *Int J Comput Appl*, 23:4–11, 2016.

[23] Alwyn E Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. 2010.

[24] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[25] Klaus Havelund. Runtime verification of c programs. In *Testing of Software and Communicating Systems*, pages 7–22. Springer, 2008.

[26] Lentin Joseph and Jonathan Cacace. *Mastering ROS for Robotics Programming: Design, build, and simulate complex robots using the Robot Operating System*. Packt Publishing Ltd, 2018.

[27] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Javamac: A run-time assurance approach for java programs. *Formal methods in system design*, 24(2):129–155, 2004.

[28] András Kornai. Extended finite state models of language. *Natural Language Engineering*, 2(4):287–290, 1996.

[29] Edward A Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[30] Insup Lee, Oleg Sokolsky, John Regehr, et al. Statistical runtime checking of probabilistic properties. In *International Workshop on Runtime Verification*, pages 164–175. Springer, 2007.

[31] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[32] Rupak Majumdar, Richard M Murray, and Pavithra Prabhakar. Verification of cyber-physical systems (dagstuhl seminar 14122). In *Dagstuhl Reports*, volume 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[33] Patrick Meredith and Grigore Roşu. Runtime verification with the rv system. In *International Conference on Runtime Verification*, pages 136–152. Springer, 2010.

[34] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. Time series databases and influxdb. *Studienarbeit, Université Libre de Bruxelles*, 2017.

[35] Geoffrey Nelissen, David Pereira, and Luis Miguel Pinho. A novel run-time monitoring architecture for safe and efficient inline monitoring. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 66–82. Springer, 2015.

[36] A Pedro, JS Pinto, D Pereira, and LM Pinho. Rmtld3synth: Runtime verification toolchain for generation of monitors based on the restricted metric temporal logic with durations. *online: https://github. com/cistergit/rmtld3synth, last accessed*, 2, 2018.

[37] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: a hard real-time runtime monitor. In *International Conference on Runtime Verification*, pages 345–359. Springer, 2010.

[38] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[39] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[40] The Haskell Development Team. Haskell, An advanced, purely functional programming language. https://www.haskell.org/.

[41] Hermann Winner and Walther Wachenfeld. Absicherung automatischen fahrens, 6. *FAS-Tagung München, Munich*, 9, 2013.

[42] Yuanfang Zhang, Christopher Gill, and Chenyang Lu. Reconfigurable real-time middleware for distributed cyber-physical systems with aperiodic events. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 581–588. IEEE, 2008.