Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Tesztkörnyezetek generálása és monitorozása autonóm robotok teszteléséhez

Készítette
Hajdu Csaba

Konzulens
Vörös András, Dr. Majzik István

HALLGATÓI NYILATKOZAT

Alulírott Hajdu Csaba, hallgató kijelentem, hogy ezt a dolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2017. 10. 29.

.....

Hajdu Csaba

Table of Contents

Absztrakt

Napjainkban kiemelt figyelmet érdemelnek a robotok, mint jellegzetes kiber-fizikai rendszerek. Korábban főleg az ipari és kutatási területeken voltak jelen, ahol jól meghatározott és könnyen megfigyelhető környezetben működtek. Ahogy felhasználási módjuk szélesedik, úgy az emberek és autonóm robotok interakciója, illetve kooperációja is megjelenik. A robotok az emberekkel és környezettel való interakció során egyre biztonságkritikusabb szituációkba kerülhetnek. A robotok működésének biztonságosságát így valamilyen módon ellenőrizni, tesztelni kell.

A kiber-fizikai rendszerek, következtetésképpen a robotok tesztelése is bonyolult feladat. Ezek a rendszerek komplex, környezetfüggő funkcionalitással rendelkeznek, amelyet szükségszerűen le kell tesztelni üzembe helyezés előtt. További nehézséget jelent, hogy a működési környezetük, a fizikai valóság alapvetően dinamikus és sokszor kooperatív. Így az elterjedt komponens- és integráció tesztelési módszerek nem megfelelőek a környezetfüggő biztonságos működés vizsgálatához. A valós környezetben történő, prototípusokon végzett tesztelés viszont költséges és időigényes feladat.

Dolgozatom célja egy olyan módszert adni, amivel lehetségessé válik a valós környezet egy szimulációjában lefuttatni az egyes teszt forgatókönyveket (robot missziókat). Ezzel a valós tesztkörnyezet berendezése és a fizikai prototípus megvalósítása nélkül elvégezhető a tesztelés. Továbbá a szimuláció megfigyelhetővé tételével az eredményeket és a tesztelés során létrejött eseményeket rögzíteni lehet, így a teszt lefutása precízen kiértékelhető. Továbbá a rögzített adatok felhasználhatók további tesztkörnyezetek és forgatókönyvek előállítására.

Dolgozatomban bemutatok egy olyan, általam fejlesztett keretrendszert, amely alkalmas robotok vezérlő komponenseinek tesztelésére. Az új eredmény elsősorban a robot számára megfelelő környezet legenerálása és az ebben futó forgatókönyv megfigyeléséhez szükséges monitorozó komponensek kialakítása. Egy olyan rendszert kellett megvalósítani, amelyben szétcsatolható a monitor a szimulációtól, ezek egymástól függetlenül fejleszthetők. Alkalmas fejlett modellgenerátorok felhasználásával különböző teszt-környezet elrendezések (pl. terem berendezések) szisztematikus előállítása lehetséges. Emellett egy olyan megfigyelő alrendszert készítettem, amely a bejövő adatokból egy megfigyelési modellt épít fel (EMF technológiával), ezáltal detektálhatóvá válik egy esetleges hiba a robot vezérlőrendszerében. Ezen a modellen futásidőben futtathatók modell lekérdezések, amelyek alapján azonosíthatók hibaesemények és ezekre végrehajtható szabályok definiálhatók (a VIATRA-CEP technológiával). A monitor által előállított jelzések felhasználhatók a teszt kiértékeléshez. A teszt rendszer tehát magába foglalja a környezet és robotkomponensek szimulációját, a megfigyelési alrendszert és a kiértékelő motort is. A keretrendszerben számos meglévő technológiát felhasználtam (pl. ROS). A feladat kidolgozásához a motivációt az ARTEMIS R5-COP európai projekt adta.

A rendszer segítségével fizikai implementáció nélkül is végrehajthatók és megfigyelhetők a robotok különböző tesztforgatókönyvei. Ezáltal már a fejlesztés korai fázisában detektálhatók olyan hibák, amik máskülönben a fizikai megvalósítás után derülnének ki. Így

összességében olcsóbban és gyorsabban fejleszthetők a robotok, ezek biztonságos működése pedig alaposabban ellenőrizhető.

Abstract

Nowadays robotic systems deserve special attention as typical Cyber-Physical systems. Until now robots were more present in special industrial and scientific applications where they operated in well-controlled and fully observable environments. As their use extends the human-robot interactions (possibly their cooperation) becomes more frequent. During their interactions with humans and environments, robots may cause safety-critical situations. Therefore, the safe operation of the robots must be verified.

The testing of cyber-physical systems, thus also robotic systems, is a complex task. These systems are attributed with complex context-dependent functionality which must be tested before deployment. Robots operate in the physical reality which is nondeterministic and often cooperative. Therefore, the widely used methods of component and integration testing cannot be used. On the other hand, executing realistic testing scenarios on robot prototypes is costly and time-consuming.

The goal of my work is offering a method which allows the execution of realistic testing scenarios in a simulation of the physical environment. With this method, each robot task (mission as a test case) can be executed without the setup of a real test environment. Besides that, enabling monitoring on the simulation the results and events can be identified and stored to evaluate the execution of the test case. Moreover, the acquired data can be used to generate new testing environments and scenarios.

In my work I present a framework developed by myself which enables testing of the control components of robotic systems. The main contribution is the development of components to generate appropriate test environments and to monitor the test case under execution. The monitor had to be loosely coupled with the simulator. With the use of appropriate model generators diverse test rooms and environments can be generated systematically. Besides, I developed a monitoring subsystem which converts the incoming data to an observation model (EMF based) which enables to detect runtime errors in the robot control. Model queries can be executed runtime which can be the basis of monitoring events and executable rules can be defined (using VIATRA-CEP technology). The signals generated by this component can be used for evaluation of the test. In summary, the simulation-based test framework encapsulates the simulation of robotic components and environment, the monitoring subsystem and the evaluation engine. In the framework I used available technologies (e.g., ROS). The motivation for developing this solution came from the European R&D project ARTEMIS R5-COP.

With the help of the system, context-dependent missions of autonomous robots can be tested and observed in the early development phase even without physical implementation. This enables the detection of errors which would normally be detected after physical implementation. Consequently, development of robots can be more efficient regarding time and efforts, and their safe operation can be checked more precisely.

1. Introduction

In this section I introduce the context and the goal of my work.

1.1. General introduction

Nowadays robotic systems started to gain special attention as typical autonomous Cyber-Physical systems. Autonomous agent systems are such systems which operate without or with very limited external control, only based on their own perceptions and beliefs. Cyber-physical Systems are defined by NIST [1] as “smart systems that include engineered interacting networks of physical and computational components”. Until now robots were more present in special scientific and industrial applications and operated in limited, well-controlled and fully observable environments. Even their autonomy was not a requirement, teleoperation was sufficient enough. Besides, mobile robots existed to operate in a hostile environment through teleoperation (space missions, volcano/submarine exploration, nuclear meltdown sites).

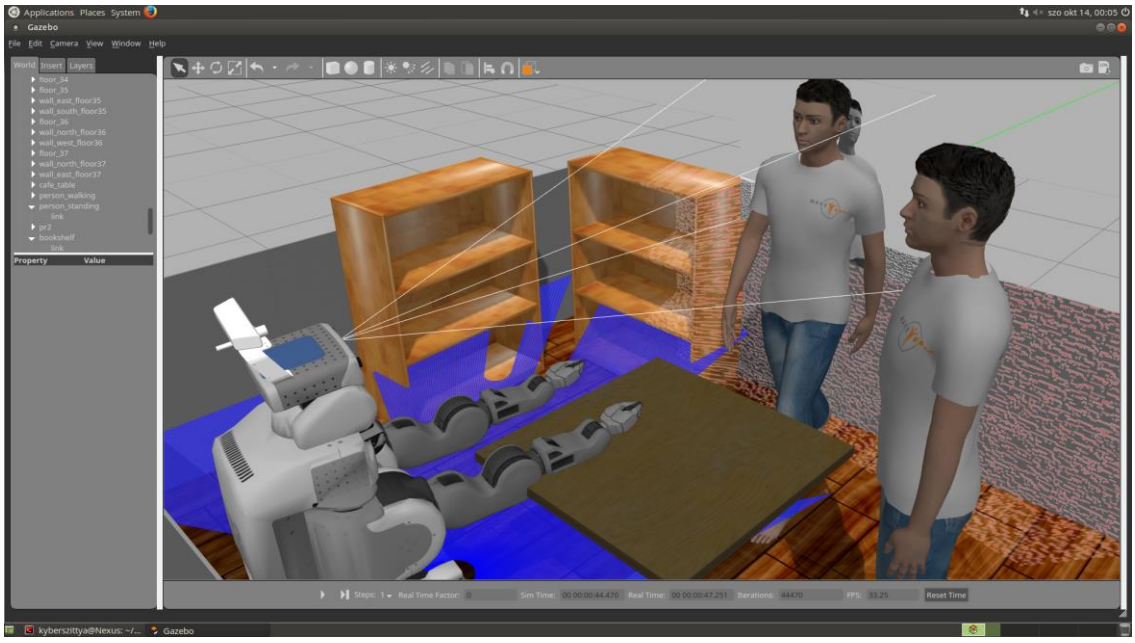

Figure 1. A complex robotic test scenario setup involving human and a complex robot (PR2[1])

The endowment of robots with autonomy seems to be an obvious choice. Currently there is a growing demand for full automation in the industrial sectors, consequently existing industrial manipulators shall gain some degree of autonomy. Additionally, autonomous vehicles (which can be viewed as a special case of autonomous mobile robots) and other new applications (nursery robots, airport assistants) started to gain attention. During their operation, robots may participate in safety-critical situation, thus these autonomous systems become critical. Therefore, the safe operation of the autonomous systems must be

[1] PR2 is a registered trademark of Willow Garage

ensured. The widely used traditional test methods shall be improved to test autonomous robotic systems, thus enabling the development of safer robots in shorter time.

1.2. Problem definition

Various problems arise from the autonomous operation of robots. Robots are autonomous mechatronic systems with complex functionality which operate in the real physical world. These functionalities and their outcomes must be exhaustively tested, as these use cases of autonomous robots are usually related to safety-critical operation, as faults can lead to catastrophic consequences (injury, damage to property). Also, autonomous systems should be aware of their environment and handle unexpected hazardous situations. Autonomous agents are usually does not exist alone, there are other agents (such as robots), with similar or different goals. Their interaction or cooperation should also be tested. Generally speaking, the physical world can be described as a difficult environment from the viewpoint of rational agents.

The naïve method of component- and integration tests are usually considered inefficient. The conventional way of testing of basic robot functionality essentially contains the following steps:

- Setting up a test environment and
- Observe the test execution by human inspector(s) who also evaluates the test execution.

This way of testing is a time consuming, expensive and human resource exhaustive process. Also, it is difficult to automatize the test execution process, even by using automated observation or execution methods. Moreover, these test cases cannot always guarantee the correct behavior of the system in unforeseen situations.

1.3. Goals

The goal of my work is to enhance the testing process by introducing a testing and monitoring framework. The main purpose of this framework is to create a simulator-based testbed for robotic systems whose behavior can be monitored and evaluated by the components of the proposed framework. The testbed should enable the automation of text execution process. These steps include the setup of the test environment, configuration for the automatic execution of robot tasks and automatic movement of environment objects. The monitoring framework should access all information regarding the run of test execution through software components and analyze events occurring at runtime.

To make the environment fully observable to the monitor, the physical environment can be simulated. Using a physical simulator, virtual mockups of the physical objects appearing in real situations can be populated in simulation providing this way the test setup. Nowadays physical simulators are widely available which can be configured with an external description. After the setup of the simulation and starting the robot mission, the current state of the simulation must be regularly extracted and forwarded to the monitor components.

After the data has been received by the monitor, it should process the data and generate significant events based on the observed data. The monitoring framework also should construct a runtime model capturing the important aspects of the system. This model can be serialized at any time during test execution and used for further evaluation.

As it was mentioned, the framework should support other tasks like robot mission execution. To construct a robot mission, sequence of tasks can be composed, which might be executed at a certain point of time during test execution. The purpose of the framework is to interpret the high-level descriptions of tasks and execute the tasks in the simulated world. Also there are objects in the simulated environment which are in movement, the framework should allow the definition of different movement scenarios.

In summary, the ultimate goal of this framework is not only to make the test execution process less exhaustive but to support the development of safer robotic systems and components even before the construction of the prototype.

The motivational background of this work is based on the results of the European ARTEMIS project R5-COP [2].

1.3.1. Contribution

In this work, I introduce a methodology to support the verification of autonomous robotic systems. The methodology is implemented in a framework to support the developers analyzing the behavior of robotic systems at the design phase of the development.

I propose to combine simulation-based verification with runtime monitoring to evaluate the behavior of autonomous robot systems.

- I developed a method to transform the abstract test cases into concrete test scenarios that provide a dynamic environment for the tested robot to perform its mission. These concrete scenarios are defined in the input language of a simulation tool. Note that this method is built upon an existing solution for generating the abstract test cases using models of the robot context and models of the requirements. The novelty of my method is that the test scenarios allow the verification of the robot behavior in case of a robot mission in dynamic environment with moving objects and obstacles.
- I developed a model-based approach for the monitoring and evaluating the behavior of the simulated robotic system in the dynamic test environment. The novelty of my approach is the use of runtime models and high-level graph-queries and temporal specifications for evaluation purposes, avoiding this way the implementation of low-level event processing and test evaluation functions.
- I have implemented the transformations that support the automatic generation of concrete test scenarios from the input abstract test description, and also implemented the monitoring infrastructure to support the test engineers.

2. Background

This section describes the conceptual and technological background of this framework.

2.1. Autonomous systems

The motivating example of this framework was to test autonomous mobile robot systems. These systems can be considered as a subset of autonomous agent systems with special requirements and complex environments. This document uses the terms established in [3].

Autonomous robotic systems operate in the real physical world that is typically characterized as a *partially observable, nondeterministic, sequential, dynamic, continuous* and *multi-agent (adversarial)* environment. Also, faults of operation even with the most basic functionalities robot actuation may cause serious damage to property and/or threaten human health. There exists standards currently, which describes the safety requirements for industrial robotic applications [4] [5]. Moreover, when a robotic system is mobile, more complex functionalities arise to enable robust mobility and the purpose of operation (see [6]). Mobility and complex functionalities yield new hazards during robot operation. Safety requirements of mobile robots are also addressed in recent standards [7].

Consequently, the driving motivation of this framework is to support the development of mobile robots which operate safely and with an ability to cooperate with humans and each other. The need for such a system arises as the use-cases of autonomous mobile robots extends to the daily life. The appearance of (semi-)autonomous vehicles is a prime example of the daily usage of autonomous mobile robots. Safety features are still in an immature stage [8] and the risk of a traffic incident is still significant. An automatized testing framework could check the safe operation even for more complex evading scenarios in the context of autonomous vehicles.

2.1.1. Robot software components

The following subsection identifies the main robotic components to be considered throughout the test of robotic systems. My work uses the terminology introduced in [6].

Autonomous mobile robot systems can be modeled through an abstract control scheme. The robot is embedded into an environment (real or simulated), which is *percepted* throughout the use of **sensors**. From the raw data generated by the **sensors,** information is *interpreted and extracted* by **filters**. This information can be used to construct the *environment model* of the robot, sometimes a *local map* can be gained. The *environment model* and the *local map* is used by the **localization** and **map building** components of the robot, which then extracts a *believed position* and a *global map* of the environment. These components might be augmented by an external **knowledge base**. The **cognition** (**path planning**) components additionally receives *mission commands* to generate a *path* (which might be a series of low-level commands other than a trajectory of mechanical

movement). This *path* is executed by the **path execution** component, which extracts direct *actuator commands* to the **actuators** of the system. The actuators are inherently changes the state of the system.

2.1.2. Definition of a mission

This section proposes the definition of a *mission* and a *task* in the context of autonomous systems. An autonomous system is expected to perform numerous tasks, or collection of tasks with different constraints, which form a mission. For the full functionality of a test framework, to define a mission it has to forward a series of input actions to the cognitive components of the autonomous system.

Task is a concrete command, which the system executes with optional parameters defined in the description. While they are usually composing a high-level action, they are described in a concrete format understandable for the autonomous system. All actions must be valid within the scope of the autonomous system. A task is, for example, a navigation goal which tells the robot to move from its current position to another in the environment. The task is usually parameterized with some constraints regarding the execution of the system. The task succeeds if the state is changed to the desired state and no global or local constraints have been violated. Otherwise, the task fails.

Mission is a sequence of tasks with predefined starting and ending state. A mission describes from what starting state should reach an ending state through a series of tasks to be completed. The mission usually describes the system constraints which should not be violated. The outcome of a mission depends on some tolerance on the completed mission goals. The mission **succeeds** if all tasks have been completed. The mission **fails** if important tasks could not be completed, or any global constraint has been violated.

2.2. Testing

The following section introduces the definition used for testing throughout this document.

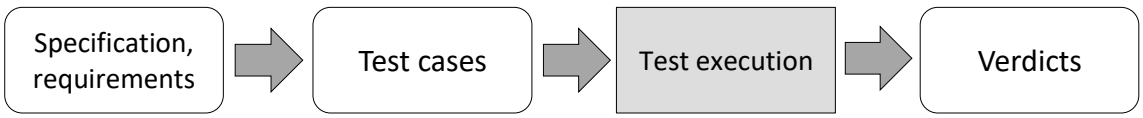

Figure 2. Brief description of the testing process

Generally speaking, testing is an essential but complex and resource-consuming task to verify system operation against requirements [9]. IEEE defines this process as an activity of the target system or one of its components. The target is executed under specified conditions and the results are observed or recorded, and an evaluation is made regarding the test target [10]. The draft description of the process (Figure 2) identifies the basic elements of a typical test scenario. Test cases are created from the specification of the system, which are executed and verdicts are assigned describing the outcome of the test case (GO/NO GO).

More specifically, going into the details of the whole testing process, additional test artifacts must be identified to precisely evaluate the high-level stakeholder requirements. The specification of the system (*test requirements)* is derived that define the system behavior in a certain situation (should/should not). The *Test approach* collects how and when the testing should be conducted, e.g. what processes, techniques, test levels and tools should be used. *Test purposes* describe what part or functionality of the system should be covered by testing. *Test case specifications* are created in which the inputs, predicted results and set of execution conditions are specified. The expected output for a given input is obtained from a *test oracle*. The test cases are implemented with the help of *test adapters* and executed in a *test execution environment* (which contains the *system under test* and potentially some test doubles). Possible verdicts are usually **pass, fail** or **error** during execution. All of these concepts are combined into a *test framework*.

This document presents a test framework for robotic systems, focusing on test adapters, test oracles, test trace generation and test execution. The document follows the terminology described in [10] and [11] onwards.

2.3. Monitoring

The following section introduces the definition of monitoring used throughout this document. As this document presents a test framework along with a test case observer (monitor), the definition of monitoring is required. Regarding monitoring this document uses the taxonomy described in [12], but changing the context from software systems to mobile robots.

Runtime software monitoring has been used for software fault-detection, diagnosis and recovery. Fault detection provides evidence that program behavior complies or does not comply. The intention of runtime software-fault monitoring is to determine whether the current execution preserves specified properties (e.g. the robot does not collide). Thus monitoring can be used to support testing by exposing state information. *Execution failure* can be defined as a deviation between the observed behavior and the required behavior of a task. A *fault* occurs during execution of the task and results in an incorrect state that may or may not lead to a failure. An error is a mistake made by human that leads to fault that may result in a failure. Software requirements are implementation-independent descriptions of the external behavior of computation, and can be used to construct system properties for the monitor. *System Properties* are relations within and among state of the execution (set of sequences of states).

The *monitor* can be defined, as a system that observes the behavior of another system and determines if it behaves consistently with a given specification. A monitor takes a system execution scenario and a specification of system properties, in order to check that the properties hold for the given execution. The current work proposes a monitor that observes the *system properties* and analyzes these system properties. The monitor uses this analysis to capture and process events and delegates the event data to other components of the system. These events are composed into a structured *runtime model*, which contains

the current snapshot of the environment. The *pattern matcher* part of the monitor queries this model and tries to match the extracted events with different patterns.

2.4. Case study

The following subsection describes a concrete use case of robotic testing, which serves as a motivating example used also in the later sections.

Consider a scenario, in which robots operate in an industrial environment. The purpose of these robots is to deliver packets from a given section of the plant to another. During their operation they may encounter humans and other robots as well. Otherwise, the environment elements are considered static or unreachable by the robot. The robots used in this scenario are predominantly wheeled robots (differential- or omnidirectional robots).

It is obvious that we want to test critical interactions of the robot with its environment in this scenario. The robot should reach from its (arbitrary) starting location a goal location in the plant. The satisfaction of this requirement would ensure that the robot is able to deliver packets. During reaching its goal, the robot must not collide with any other mobile or static target since collision often causes serious health injury and damage to property. Besides, the robot should reach its goal with minimal manoeuvring and in the possible shortest time. The observations also verify that the sensors are working properly and feed the robot with correct data.

With the available graphical model of the industrial plant and the robot itself, the environment can be simulated and the behavior of the robot can be evaluated in this simulated environment - instead of building the robot and trying it in the physical world. Also, there is no need to construct a physical mockup or prototype to run the test cases.

2.5. Used technologies

Many technologies had to be integrated together in the framework. This section describes the most important properties of these frameworks and the main reason behind their selection for use.

2.5.1. Robot Operating System (ROS)

Robot Operating System (ROS) is a robotic middleware embedded into a general operating system [13]. Its purpose is to support robotic software development by providing an extensible and versatile communication framework for the internal communication of software components. It also provides a huge selection of predefined data structures and libraries to simplify software development.

From the monitoring point of view, the most important property of ROS is its communication model. ROS provides both remote procedure calls (RPC) and Publisher/Subscriber model. An application can use both depending on the use case. All components are connected through a master process. Publisher/Subscriber model is useful for monitoring purposes, as it provides data continuously.

Currently, ROS is a popular general robotic framework used for a large set of tasks. It is not only supported by the closely related applications, but more general applications such as MATLAB and even Android wrappers do exist. One of the ROS design principles were the reusability of components. ROS also provides a huge library with solutions of the most frequent robotic problems, or at least an interface to develop the control of components.

The only drawback of ROS is that it does not currently provide real-time communication between components. This is addressed by other frameworks such as OROCOS [14], RT-middleware implementations [15], which focuses more on robotic control rather than the interoperation of software components and can be easily interfaced with ROS. ROS is somewhat resource consuming, so it cannot be functionally installed on an embedded device with limited resource. ROS currently does not address the security issues.

The upcoming successor, ROS 2 tries to resolve the issues regarding security, real-time operation and embedded installation [16].

2.5.2. Gazebo

Gazebo is a physical simulation application with close connection to ROS, currently developed by Open Robotics Software Foundation (OSRF) [17].

The main purpose of Gazebo is to provide mechanical simulation between mesh models. The simulated mesh models use visual mesh for appearance and collision model (which is usually coarser than the visual mesh). Meshes can be connected through links and joint, to allow forward/inverse kinematic calculations. These chains can be used to populate the simulation with simple objects or robot physical descriptions. Additionally, Gazebo enables the simulation of sensors such as cameras, proximity sensors and GPS localization.

The object properties are described in a semi-structural file with well-defined format called Simulation Description Format (SDF) [18]. To interface with the simulation, the input file must comply with this format. Given a model of the test environment, a description shall be generated fitting into this specification.

One defining advantage of Gazebo is, that it is able to broadcast Pub/Sub channels and RPC services to ROS, enabling middleware applications to read geometric and kinematic states, the simulated sensor reads. This data can be used to compare monitored robot state supplied by the simulator (i.e. actual state) and robot belief state. Moreover, objects can actuate geometric state, enabling the control of robots and animate objects with different scenarios.

2.5.3. WAMP

WAMP is a relatively new communication protocol enabling (soft) real-time communication between application endpoints in a loosely coupled distributed system [19]. This protocol is used to forward ROS based communication to platform-independent remote applications, such as the monitor. Onwards the WAMP-based communication network of the system will be referred as *hive*. The choice was to use this protocol, while other alternatives were considered such as MQTT.

There are some very significant advantages of WAMP over MQTT and competent protocols [20] [21]. WAMP enables the usage of both RCP calls and the Pub/Sub model. This can be significant from a viewpoint of performance, as there are many use cases in this system, when some information is needed by a node at a single time point but not continuously (e.g.: configuration of the monitor). Also this model is the most similar to ROS which also enables the usage of both communication models, which can effectively mean that the delegation of information flow can be done simply in further applications. The reason of the construction of this whole system is given by the main disadvantages of ROS. ROS does not handle security issues well, as it was stated by many authors. Moreover, ROS is not platform-independent, Microsoft Windows and some Linux distributions are not supported. Most of the libraries are written in C++, with Python wrappers, sometimes Java wrappers. Considering this, a platform independent interconnection is reasonable.

WAMP also focuses heavily on security. Further work can lead to a secure interconnection between components. There are many possible workarounds for this, such as WAMP-CRA, WAMP-TLS, etc. WAMP is also a web native language, which means that any system, which is authorized to connect to the WAMP network and uses Websocket protocol, is able to access the data feed provided by the components. MQTT uses a broker to connect other devices in a P2P manner, this can be useful on a large sensor network, the WAMP implementation is more reasonable in this case. WAMP uses a delegated soft-router to handle connections between components. This can eventually lead faster and more reliable communication between components. Drawbacks arises from the fault tolerance measures of the soft-router, as this single component may has to be redundant which has to be addressed during operation.

2.5.4. EMF

Eclipse Modeling Framework (EMF) is a modeling framework and code generation facility first appeared alongside with Eclipse [22]. The framework allows the construction of tools and runtime support to produce a set of Java classes for the model along with a set of adapter classes (which enables viewing and command-based editing of the model), and a basic reflective editor. The model specification is described in XMI.

Our framework relies heavily on EMF. Many tools, extensions and derivation had been created on top of EMF. A common use of EMF is to create tools based on Eclipse. Our framework employs model based text editors (Xtext[2] based) and graphical editors (Sirius[3] based) to create configurations and descriptions. Besides creating the tools to create new artifacts, the EMF allows the use of models runtime. The observation model is constructed during observation.

[2] Xtext is developed by Itemis AG.
[3] Sirius is an Eclipse plugin developed by Obeo.

2.5.5. VIATRA Query

VIATRA Query is the successor of EMF-IncQuery[4]. It is mainly used to query EMF instance models to extract information matching to predefined rules or to transform instance models into another model.

This framework employs both typical use cases of this tool. During the construction of the test room environment description, the well-formed constraints are checked by this tool. When creating ROS launch files, the instance model of the test environment room is transformed into the launch file domain using model transformation.

2.5.6. VIATRA-CEP

VIATRA-CEP is an experimental framework developed as a plugin to VIATRA Query project [23]. This framework proposes complex event processing (CEP) on EMF based instance models. The pattern matching is tightly integrated with VIATRA Query engine. As the basic query engine is able to match queries on dynamically changing models (incremental query), the changes in the model can be evaluated efficiently.

Besides providing a relatively fast processing engine, a considerable advantage of VIATRA-CEP is the ability of the description of event patterns at a high-level with a provided tool (VIATRA Event Processing Language – VEPL). This language is based on a special event algebra which relies on temporal logic and automaton theory.

The framework I introduce in this paper employs this tool in the monitor component to realize the analysis and event handling components of the monitor described in 2.3.

2.5.7. MATLAB

MATLAB [5]is a widely used application framework frequently used for prototyping and research. With its extension, SIMULINK, control systems can be designed fast.

During the development of this framework, there were many cases, when a simple controller had to be designed and deployed to verify the correct operation of the monitor (Section 7.2). With the help of MATLAB and its Robotics toolbox these controllers can be easily designed [24]. Such simple controllers are realizing forward-backward moving, simple teleoperation and circular movement.

2.5.8. Acceleo

Acceleo [25] is a Model-To-Text (M2T) framework for Eclipse. Its main purpose is to generate text from arbitrary model realizing a metamodel.

Acceleo is the pragmatic implementation of the Object Management Group (OMG) MOF Model to Text Language (MTL) standard. It can be used to write generators without special knowledge of code generators.

[4] VIATRA Query is developed by IncQuery labs.
[5] MATLAB is a trademark of Mathworks Inc.

This framework employs generated codes from metamodels, mostly configuration read stubs and nested class initializers. Each use case is solved through the use of Acceleo.

3. Approach for the analysis of autonomous systems

The following section proposes our approach used to construct a framework to analyze autonomous systems.

3.1. Overview of the approach

This subsection provides information of the design approach and the corresponding architectural choices.

The conceptual process, i.e., operation flow of using the whole framework is depicted on Figure 3. Rounded rectangles depict artifacts, while simple rectangles depict processes.

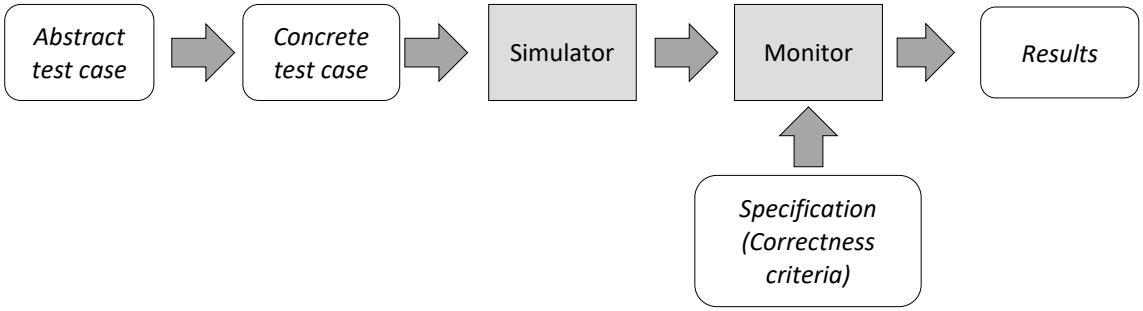

Figure 3. Overview of the process

The whole process begins with generating an environment description – possibly with the help of a tool. The result is the Abstract test case. Next, this description should be used to populate the simulated environment with objects and the robot(s) and embed the tested robot components into the environment. This is done by mapping the elements of the abstract test case into a concrete test case description.

The abstract test case description is also used to animate some subset of objects periodically, this way providing a dynamic environment for the tested robot. The mission contains the related tasks limited to the space of the generated environment.

The monitor observes and analyzes the data incoming from the environment simulation and the robot control components. The monitor should construct the observation model from the intercepted data, then run queries to analyze and extract events. The extracted events should be matched with the runtime model (knowledge base), which is represented by the graph model constructed runtime.

After the events are matched, the event patterns should be evaluated. The evaluation process should be able to calculate mission scores based on the incoming events (each event should be associated with an arbitrarily defined utility value). The historical data may be stored for further analysis of the monitored data.

3.1.1. Use case description

The main functionalities described in previous sections can be translated to high-level use-cases (Figure 4). This is useful to overview the expected functionalities of the framework. The context in this setup is the simulated environment and the computational

backend running the simulator. The actors are those who somehow interacts with the simulated environment. The **User (Test engineer)** ultimately wants to access information the belief state and control model of the robot (which is part of the **Simulator**) and its components embedded into the simulation. All geometric information is provided by the simulator itself.

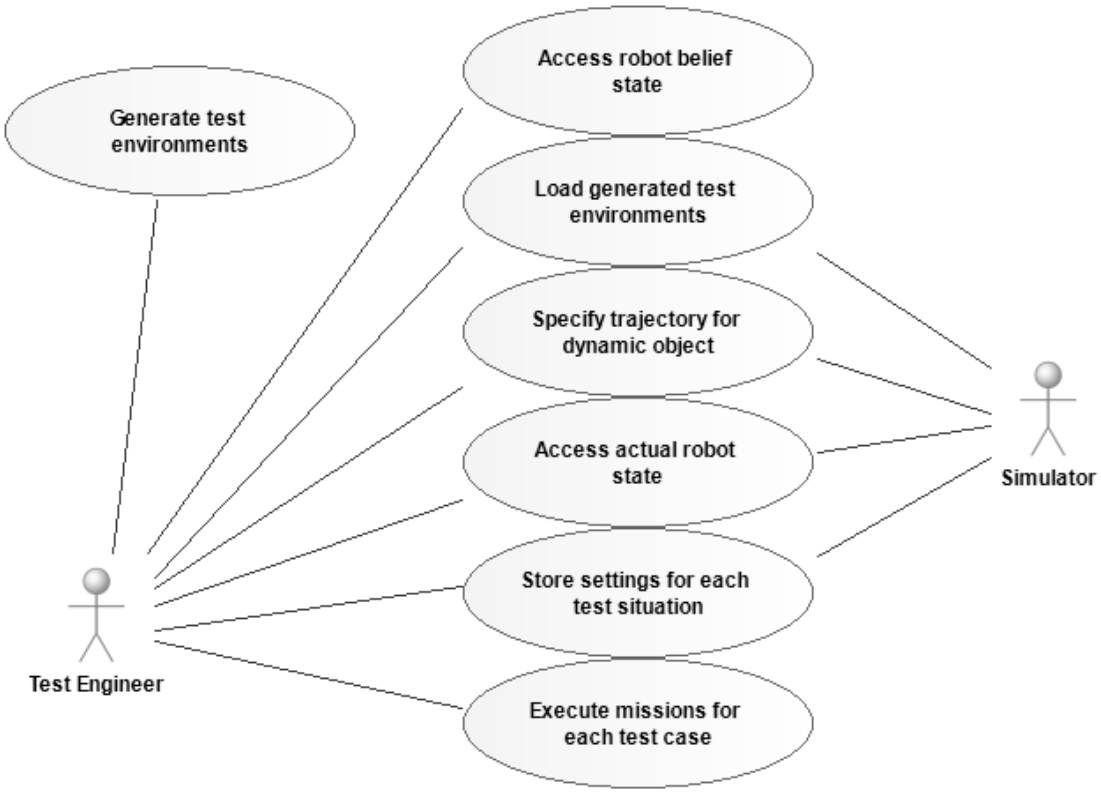

Figure 4. Overview of basic use-cases

There are some use cases which comes obviously from the high-level proposal of the solution of the testing problem. First the test engineer *generates test environments* with the help of a tool. As the description of the environment is now available, the user *loads it to the* simulator by transforming it to a concrete description. The user may specify the *trajectory of dynamic object*. These steps should setup the simulated environment and the robot itself. As the simulator is up and running, along with the robot components embedded to it, the user wants to *monitor the actual state of the robot* and *access its belief state*, gaining information about its operation continuously. The user may associate the currently running test case with a sequence of tasks, and *execute them* at some point of time.

3.2. Architectural concept

This section provides information about the framework architecture, identifies the components and gives the detailed conceptual description of each component.

3.2.1. Conceptual architecture

Given the basic use cases of this framework, a high-level description of the architecture can be described. The main purpose of this framework is to connect to a robotic system, with limited access to internal information to outside. The *Simulator* provides the environment and the robot components. *Data Collector* collects all data of interest from the simulator and forwards them to the *Monitor* component for further analysis. On a wider perspective, other components might be connected to these mandatory components. One very useful component is an evaluator component. This subscribes to monitoring events and uses some additional information of the robotic system, such as navigation component settings and initial room properties. Onwards, the framework shall be called Robotic Event Interceptor (short: REI[6]). Figure 5 summarizes these ideas and shows the components in the context of the system.

Figure 5. High-level architecture

Another useful component might be a filter component. This reduces the computational load of the monitor component by calculating some required data for the monitor component.

A different aspect can be also used to extend the testing environment with some functionalities. The simulation is defined with a test room description. The room description contains information about the setup of the room. This includes the static construction of a room, dynamic objects initial positions and the used robot in the simulation. The test room description is tightly attached to those components which initializes the test environment and actuates some of its components (6.2.2).

3.2.2. Detailed architecture

This subsection describes in detail the identified components and artifacts which they use and their interaction with each other.

3.2.2.1. Artifacts

In our definition, the *abstract test case* is the main input of the framework. It contains all information required to construct a simulation of an environment and feeds the framework with actions.

[6] A reference to the common Japanese given name (Rei).

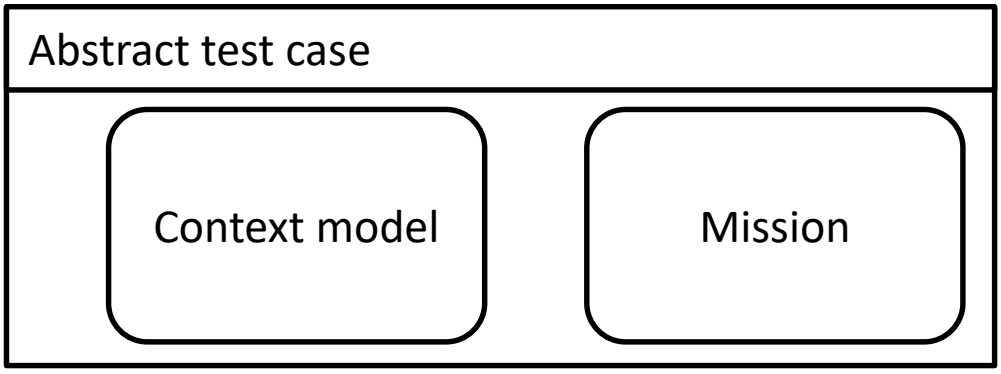

Figure 6. Abstract test data parts

It consists two parts (Figure 6). The *context model* describes the structure of the test environment, and the *mission* describes the test execution tasks in the context. The abstract test data is used to generate input for components described in 3.2.2.

- **Environment description (concrete model):** this artifact describes the environment simulation interpretable by the simulator. This description is also used by some other components of the framework. For example the animator component (Section 6.2.2) uses this description to animate a subset of objects that exists in the simulation. This description must be generated by an external tool.
- **Monitor configuration:** The configuration contains the connection information to the data source. This defines the data and targets to be collected and shares the information among other components. The data collector provides this configuration to other components.
- **Mission description:** this artifact describes the mission to be performed by the robot itself. It is mapped from the mission defined in the abstract test case embedded into its context model. It contains a series of actions and might contain the global constraints of the mission. While it is essentially used by the mission executor, it might be used by the evaluator component as well.
- **Robot component launch:** this artifact starts up the software components of the robot used in the abstract test data. This can also be generated, from the abstract test case description.

3.2.2.2. Components

In the previous section we identified the high-level components needed to realize the functionality of the test framework. Now high-level components are resolved to more components, as it is depicted on Figure 7. Each component might be configured through a configuration and there are some shared artifacts between components.

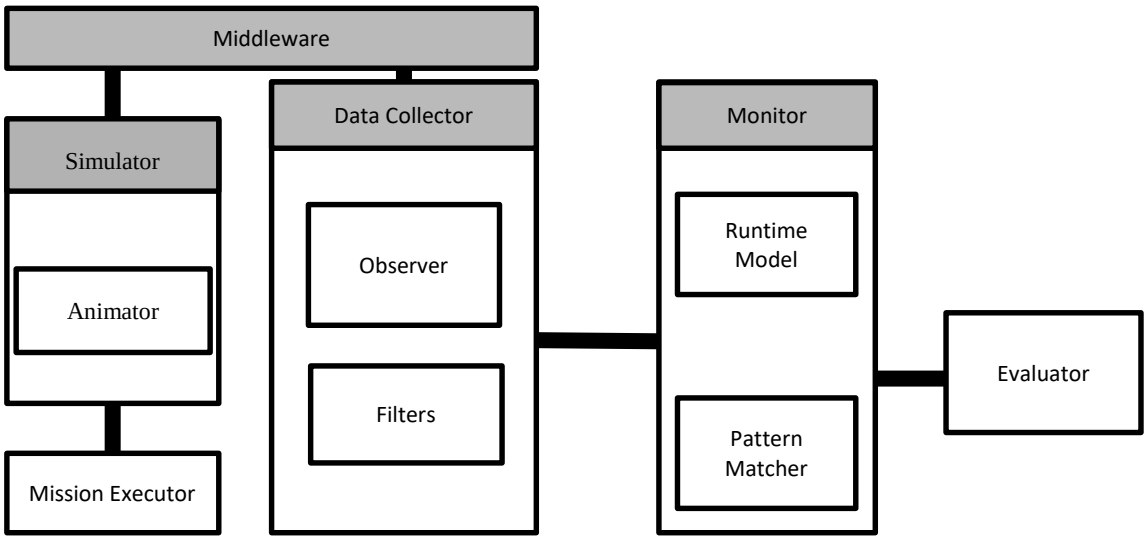

Figure 7. Detailed architecture of high-level components in depth

The framework is composed of the following components:

- **Simulator:** the simulator is a mandatory component of the framework. Every data is based on the actual state of the simulation besides the obvious geometric state (including sensor data and indirectly the robotic components). Also basic external control (like animation of objects) is exerted on simulated objects provided by the simulation. The simulator uses
    - **Animator:** this component exerts external control based on a spline trajectory on a subset of objects in the simulation. This models real situations, where the movement of physical objects can be modeled through a trajectory – which is stored as a part of the abstract test case. This component is closely attached to the simulation, and there is no purpose of running this component without an appropriate simulation.
- **Mission Executor:** this component reads a mission and executes it by following each task. This component fetches new tasks to the action execution component of the robot, so it is closely related to robotic components – especially to robotic planning components. It can be interpreted as the component which feeds the cognition component of the robot (section 2.1.1) with mission commands.
- **Middleware:** the middleware provides a common communication interface between components and external applications. The main function is, therefore, to connect closely connected external components. The monitoring components should connect the simulator through a middleware.
- **Data Collector:** a mandatory component which connects to the simulated environment using the middleware and establishes interconnection between other loosely attached components. The main and only purpose of the component is to forward and optionally filter data collected from the whole simulation. This component reads the monitor configuration and shares it between other components.
    - **Observer:** the observer component is closely attached to the middleware, as this directly reads data from middleware interfaces.

    - **Filters:** an optional component of the system and closely attached to the functionality of the data collector. It filters data generated by the data collector and forwards this resulting data to the monitor component. Its responsibility should be limited to simple tasks (such as filtering laser data based on a filter model). It is not depicted, but a filter component might be loosely detached and distributed in the WAMP hive (2.5.3). This component uses the *monitor configuration* shared by the data collector.
- **Monitor:** a mandatory component which subscribes to data forwarded by the data collector and constructs an observation model. The observation model can be used to match events interactively. The
    - **Runtime Model:** as part of the monitor, this component analyzes incoming data. It processes messages and constructs the runtime observation model of the simulation. The component should be designed in an extensible manner to enable the extension of the processing model with new robot component events and environment data.
    - **Pattern Matcher:** part of the monitor, this component provides essential functionality. As the observation model is constructed incrementally (as new data is fetched, parts of the model is added, changed or removed by this data), new events are generated as a result of query matches. A pattern match should occur, if the newly generated events appear in specified order. The event handler should also send signal messages about the occurrence of new events externally.
- **Evaluator:** also part of the monitor. This should fetch the signals generated by the event handler. The purpose of this component is to evaluate the execution of the test case based on the occurring events. This is solved through a simple utility model, as each signal has a corresponding utility value.

4. Testing based on simulation

This subsection describes the simulation process and the operation of components realizing this functionality of the framework.

4.1. Testing functionality

This segment describes the conceptual ideas and some details behind the testbed functionality of the framework.

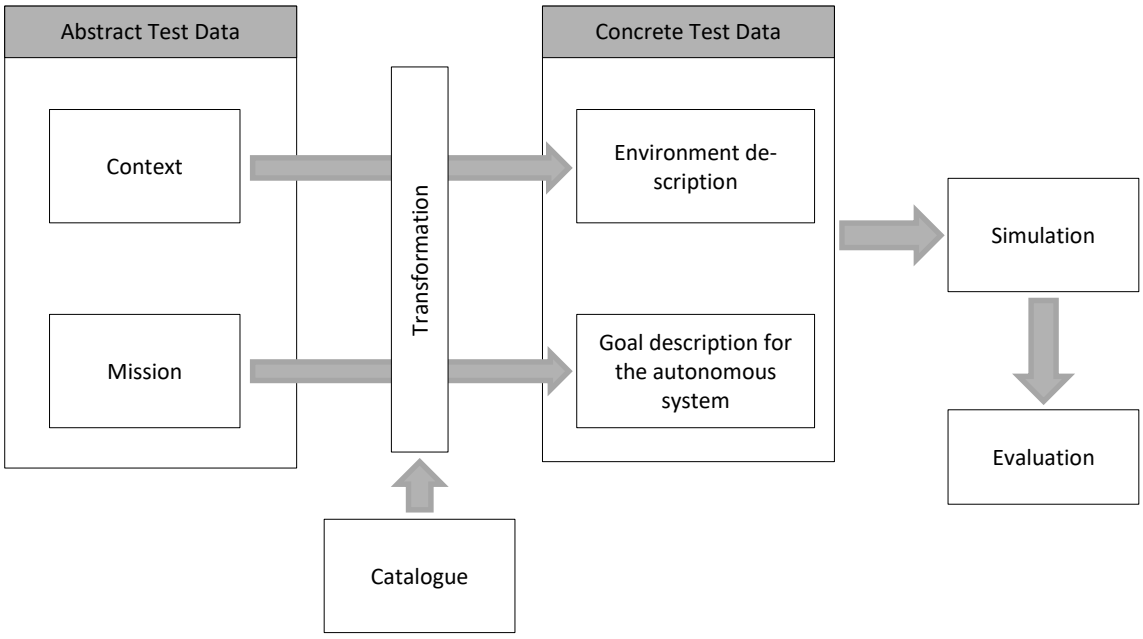

Figure 8. High-level description of the functionality

Figure 8 describes the high-level functionality of testing. The abstract test case and its parts are transformed into configurations compatible with other parts of the framework, the concrete test data (Section 3.2.2.1). The generation is supported by a catalogue of object mappings (Section 4.1.3). The artifacts are processed by the simulated components and evaluated with the help of the monitor.

4.1.1. Process

As the definition of testing was presented in section 2.2, the terms used in the definition has to be identified and assigned to a component.

- **Test Framework:** the whole system, *Robotic Events Interceptor*.
- **Test Oracle:** the evaluator component, which evaluates constraints and correctness criteria.
- **Test execution environment:** the whole simulator environment, including the robotic middleware.
- **System Under Test:** the simulated robot and its components embedded into the physical simulator.

The overview of the whole testing and simulation process is summarized on Figure 9. The main functionalities of the testing process are identified on this diagram. After the *successful startup of the simulation* the test case must be processed to *populate the simulation with objects from an environment description*. As the environment is ready, the *robot components must be initialized*, to test their functionality in the embedded environment.

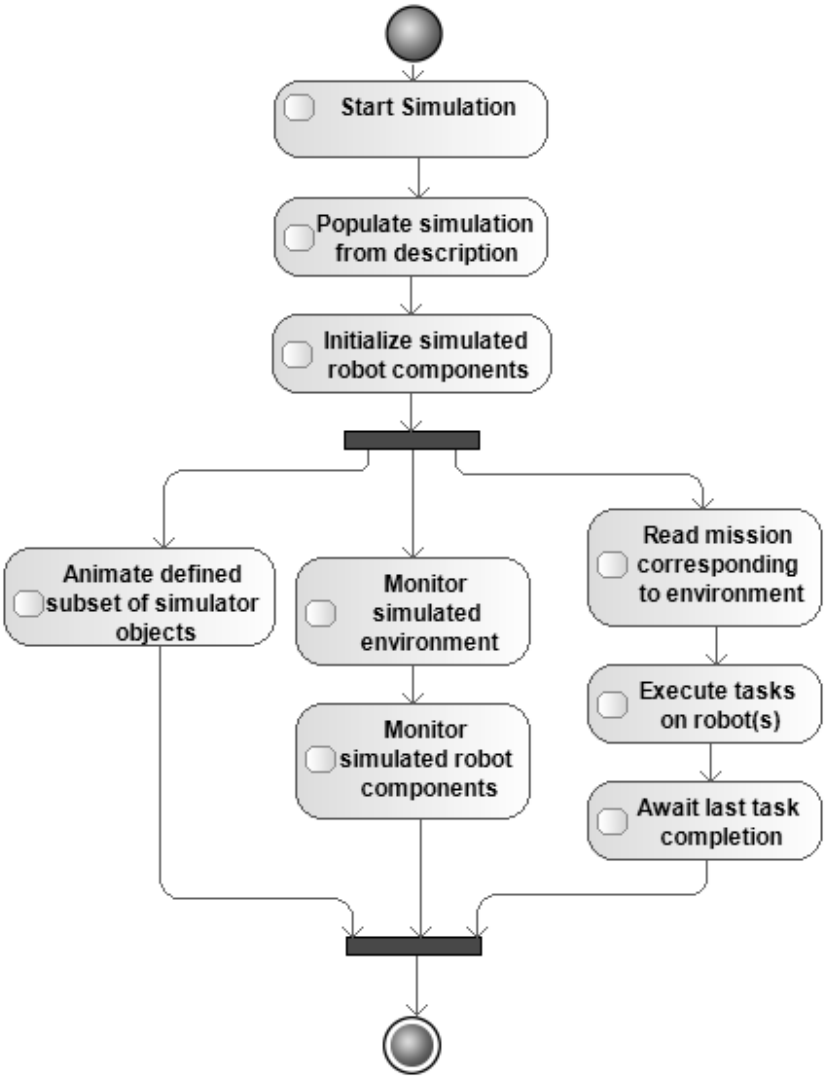

Figure 9. Overview of testing and simulation

As all components are initialized, some objects in the simulation should be animated to provide dynamic test contexts.

The test input is fed through mission execution, which is only consistent in the environment loaded to the simulation. Each mission contains a collection of tasks, which are *executed one-by-one*. The execution of the test shall be *monitored* in parallel of the previously described processes. More details on the process of monitoring are described in section 4.2.

4.1.2. Test environment description

In realistic scenarios, the test environment is composed of realistic inanimate objects, such as furniture, decoration objects, etc. The test environment is also constructed from building elements, such as floor, wall. The terrain is modeled through geometric models. For simplicity, the scope of this work is limited to buildings. The structure of the abstract test case is described through a metamodel (Figure 10).

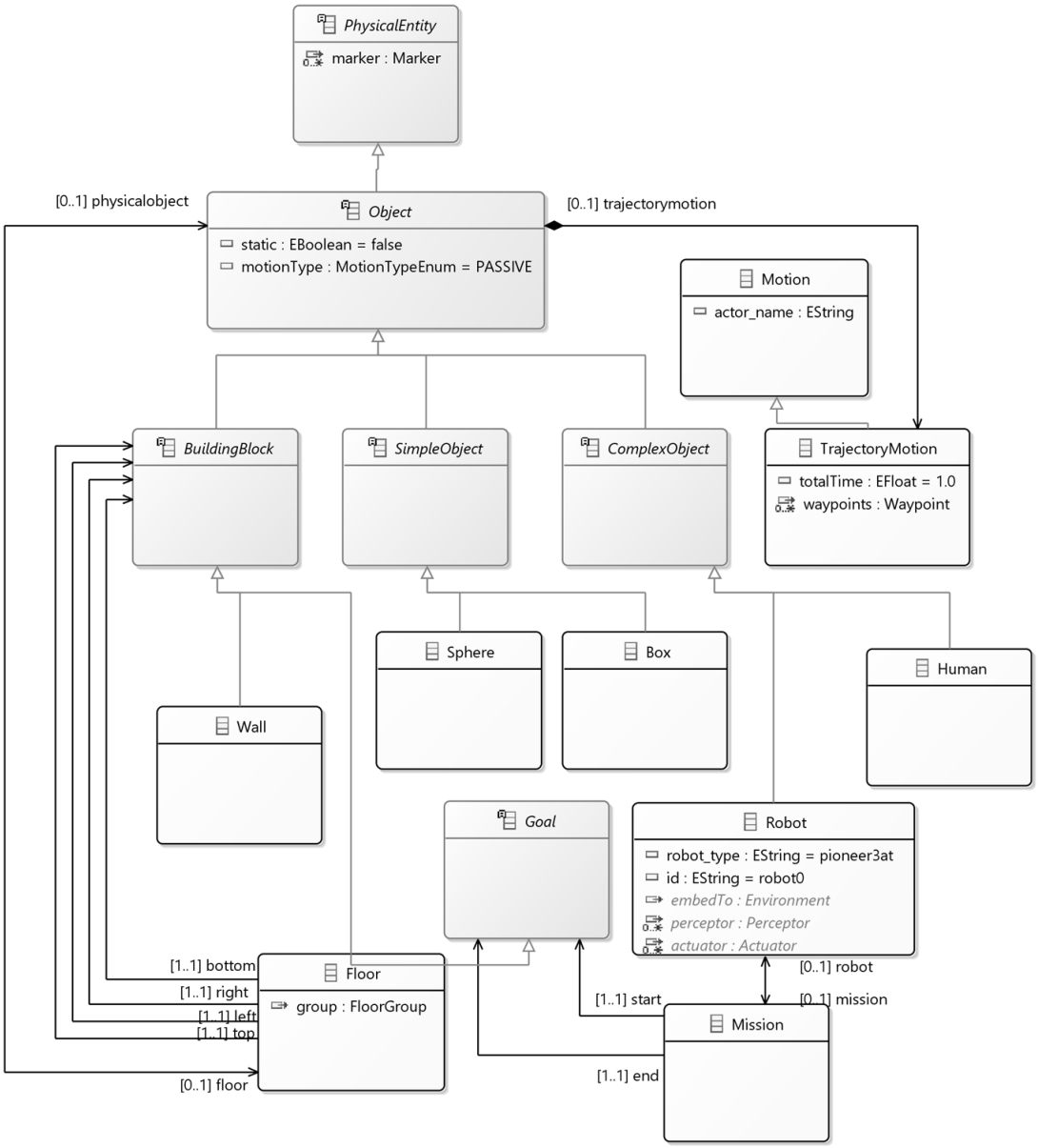

Figure 10. Metamodel description of the abstract test case

The environment is therefore composed of the following types of objects.

- **Building elements (BuildingBlock):** the elements which are used to construct the environment. Examples include wall- and floor block elements. These elements are modeled as mechanically static objects. Therefore, these objects may interact

with other objects, but their state is not changed through collisions. These type of elements are referred as building blocks onwards.

  - **Floor:** floor elements are special building elements The floor is modeled as a rectangle each side is connecting to other building element (from directions: **TOP, RIGHT, BOTTOM, LEFT**). The floor is associated with a non-building element which is placed on the planar face center of the floor. Also floors can be the control vertex of an animation trajectory of a dynamic object (see below). A floor might also be the target of a navigation task.
  - **Wall:** the wall element limits the environment and might be connected to the floor element as their neighbor element. It has perpendicular extent to the floors.

- **Non-building objects:** these type of objects should not be used to construct the environment. They can be generalized as primitive objects (visual appearance can be easily generated by series of mathematical functions) or mesh objects (their visual appearance is modeled by external tools). These type of objects are referred onwards as physical objects. From the viewpoint of testing and monitoring, their movement classification is more informative:
  - **Static objects:** the elements which populates the environment. Objects of this type do not exert movement by themselves, only through interaction with other mechanical objects. Examples of static objects include furniture.
  - **Dynamic objects:** the elements which populates the environment. Object of this type exert movement by themselves. Also they should interact with other objects, at least by exerting force on interacting objects.
    - **Human:** in this context, the human is modeled as a dynamic object. The movement of humans might be modeled as a trajectory of predefined goal points or random models of long paths.
- **Robot:** the robots are special populating objects. They have no predefined movement scenario but perform mechanical interaction based on its own perception-actuator model. A robot should also interact with other objects.
  - **Mission:** the robot is associated with a mission (moving tasks), with starting and goal points (or more point between them). In this context, each goal endpoint may be represented with a floor element. This is required to generate mission scenarios.

The environment is therefore composed of building elements and depending on the scope the initial list of building elements While environments can be constructed with the help of a tool these might be also generated. Some constraints must be maintained to help the creation of valid test environments.

- Each object must have a unique name.
- Any objects other than floor blocks must be associated with a floor.
  - If the object is a building block, it must be the neighbor of the floor.
  - If not, it must be placed in the center a floor block.

- The trajectory control vertex must be placed on a floor block.
- The robot must be placed on a floor block element.
- The environment must have at least two goals (start and final place).

4.1.3. Mapping abstract test environment description to the input of the simulator

In 4.1.2, we introduced the abstract model of the test environment. This subsection describes the solution of loading the abstract model to a running simulation.

The first problem is that this abstract description does not contain any geometric information. All simulators await some information about the geometric pose of objects to populate the simulation. This problem can be resolved by using graph traversal in the abstract model. The nodes of this graph are the floors and the edges are the neighboring connections. So the floor elements are mapped to geometric coordinates, when all floors are mapped to a corresponding coordinate. Other objects are added incrementally.

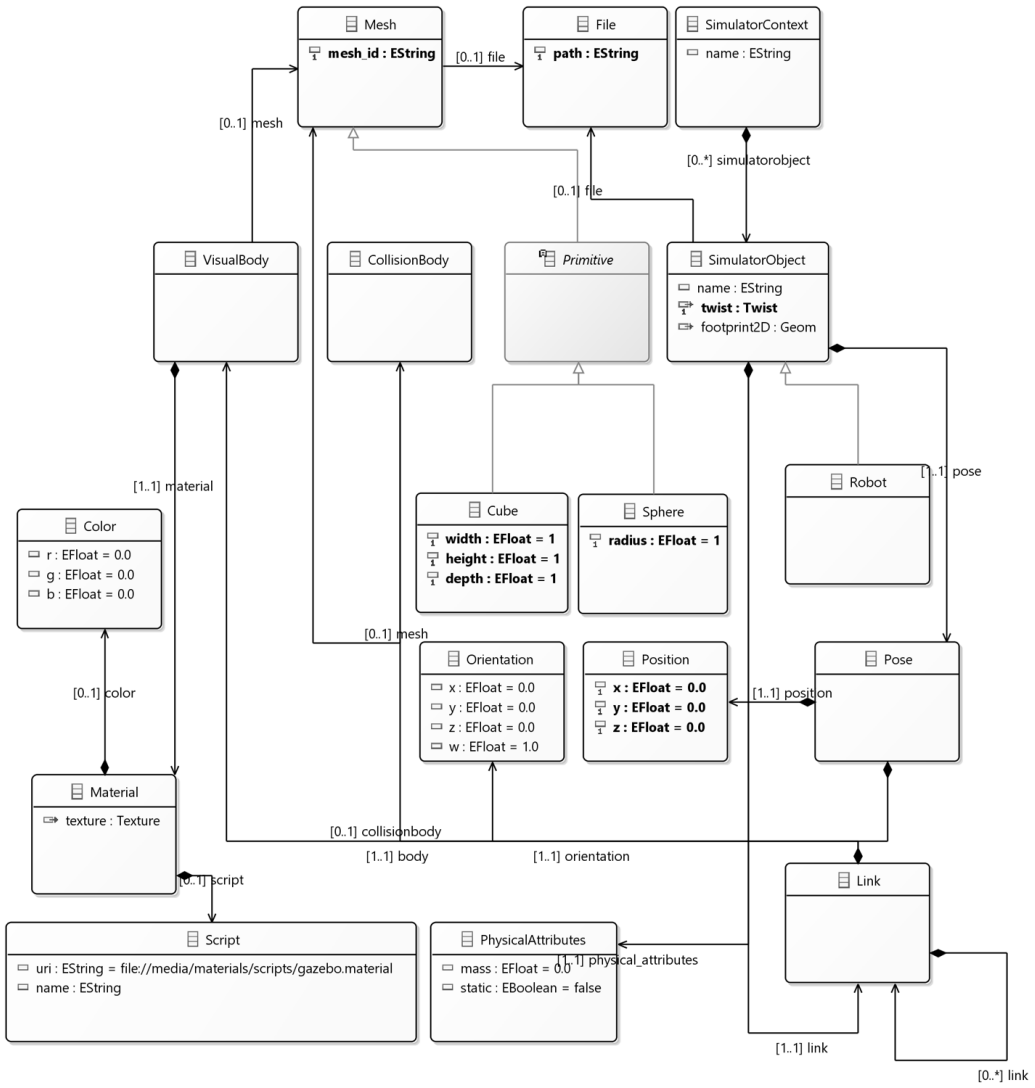

Figure 11. Catalogue object representation

Another problem arises from the fact, that this abstract model does not describe the visual appearance and the mechanical properties of the object. Both are mandatory for a valid simulation. This can be resolved by constructing a catalogue of geometric objects which contains additional information about the mapping of objects into the simulation scope. Each object stored in this catalogue contains the visual appearance as the coarse collision mesh description and mechanical properties (such as mass, inertia, etc.). Abstract description of the environment must be mapped to the elements of this catalogue. From this mapping, a description compatible with the simulation can be generated. Each object is identified in this catalogue by the object type.

The structure of a mapped object is summarized on Figure 11. The following subsections describe the tool using this mapping method and a brief introduction to all artifacts generated from abstract test data.

4.1.3.1. Environment description tool

An editor tool was created to support the description of abstract test environments. This tool is used by the test engineer and outputs the abstract description of the abstract test environment using the basic set of elements defined in 4.1.2.

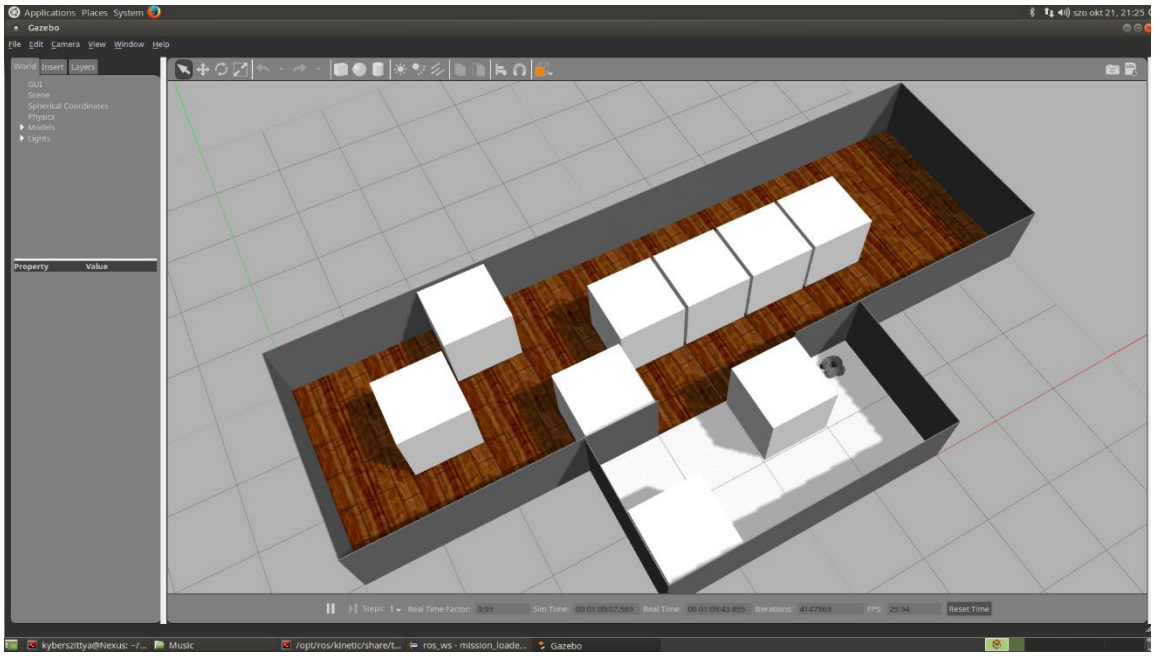

Figure 12. Example of generated and mapped environment

Besides creating the abstract model of a test environment, the tool must transform the abstract model into the concrete description of the environment (the format interpretable by the simulator). The editor needs a catalogue of objects as input to perform this operation. It then outputs a format readable by the simulator, resulting of simulation with elements (Figure 12).

4.1.3.2. Artifacts generated from environment description

The environment description may contain additional information for other processes of the testing and simulation, which can also be generated with the help of the tool. The artifacts that the tool might generate are depicted on Figure 13. Gray rectangles indicate artifacts that are generated from the abstract test data. Other rectangles indicate artifacts created by the user. This diagram does not depict processes rather the resulting artifacts from the generation process.

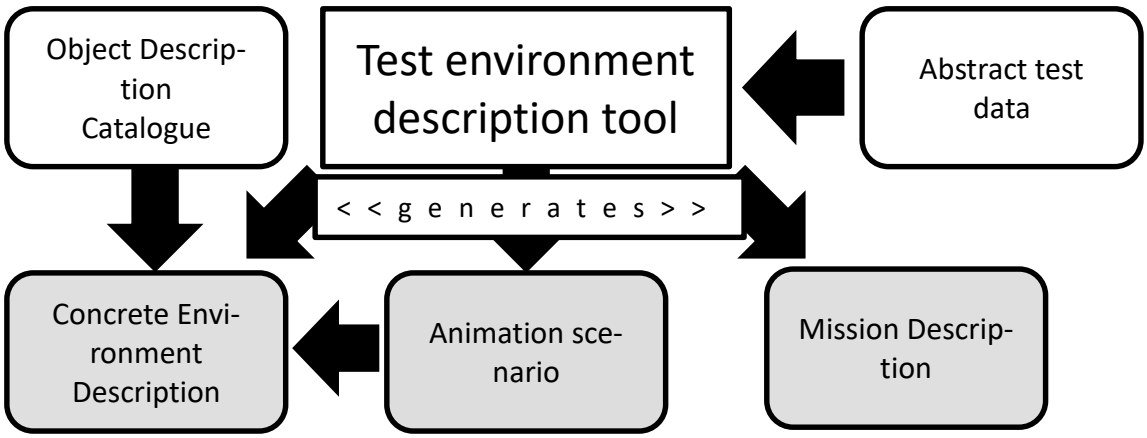

Figure 13. Artifacts generated by the Test environment description tool

Animation scenarios may be assigned to some objects of the simulation. Eventually, the animation scenario should not be detached from the environment description. The animation scenario is described in detail in Section 4.2.

To generate mission, another intermediate model is required whose structure is described in 4.3.1.

4.2. Simulating moving objects

This section describes the application which animates simulator objects of running simulation. This enables the creation of realistic simulations of a dynamic and adversarial context.

4.2.1. Mathematical background

The most effective method to construct a trajectory based on given set of geometric points is to interpolate a Catmull-Rom spline (Hermite cubic spline). This type of spline is widely used in the field of computer graphics and robotic applications due its simplicity (therefore low computational overhead) and its relatively low interpolation error. The Catmull-Rom spline can be easily constructed with the method commonly used in CG animation [26]. As a reminder, the Catmull-Rom spline is defined by Hermite interpolation, with using a sequence of geometric points as control points, with the following coefficients (p_i are control points). The coefficients used to calculate the spline is summarized in Table 1.

$a_i = \frac{v_{i+1} + v_i}{(t_{i+1} - t_i)^2} - \frac{2(p_{i+1} - p_i)}{(t_{i+1} - t_i)^3}$	$b_i = \frac{3(p_{i+1} - p_i)}{(t_{i+1} - t_i)^2} - \frac{v_{i+1} + 2v_i}{t_{i+1} - t_i}$
$c_i = v_i$	$d_i = p_i$

Table 1. Catmull-Rom spline coefficients

As a Hermite interpolation, the velocity component has to be defined. The simplest way to calculate the velocity of one control point is to consider the two neighboring points, using the following velocity function,

$$v_i = \frac{1}{2}\left(\frac{p_i - p_{t-1}}{t_i - t_{i-1}} + \frac{p_{i+1} - p_i}{t_{i+1} - t_i}\right)$$

At arbitrary point of time, the current position on the $[t_0; t_n]$ range on the spline is defined as:

$$r(t) = a_i(t - t_i)^3 + b_i(t - t_i)^2 + c_i(t - t_i) + d_i, \qquad t_i \leq t \leq t_{i+1}$$

In computer animation, the time parameter of each control point defines the time in secs which is set by the animator arbitrarily. There are some cases, when each point should be interpolated with a spline, but only the final time is known.

4.2.2. Approaches to movement

This subsection summarizes the concepts of resolving the movement of simulator objects given a Catmull-Rom spline assigned to an object.

Given the mathematical background, two approaches possible for the movement of simulator objects. The position is defined for any given t parameter. With calculating the desired position and storing the previous position, both the required position and velocity can be calculated.

$$\vec{v} = \frac{\vec{r}(t_0) - \vec{r}(t_1)}{dt}, \qquad dt = t_1 - t_0, \qquad t_0, t_1 \in \mathbb{R}^+$$

The velocity can be used for multiple purpose. A very straightforward approach is to exert force on the target object at given point of time (using Newton's second law $\vec{F} = \frac{d\vec{v}}{dt}m$, dt being the duration between the current and previous call). The force exerted on this object would result in movement of the object to the desired position and would be completely affected by interactions of other physical objects. While this is working well on objects with simple inertia (e.g. a sphere), with objects of complex inertia or inertia of stiff objects (e.g. cuboid) it can result with undesired results.

The other approach is to set the velocity of the object at the given time according to the trajectory of the object. This would allow limited interaction with other objects, but as the mass and the velocity is defined, the inertia can be calculated ($\sum mv$), allowing the simulation to run correctly.

4.3. Mission execution

This section provides information about the mission execution related to the generated test environment.

For the functionality of the framework this component plays a major role. It feeds the path planning component of the robot – and therefore the test case itself - with new goals.

4.3.1. Mission description

This subsection describes the structure of a mission.

As it was defined in section 2.1.2, the mission is a list of tasks. The tasks can be defined multiple ways depending on the operating context of the robot. In the context of the test framework, a mission provides test input for the SUT. My work focuses on defining geometric goals and forwarding them to the navigation component of the simulated robot. The structure of the goal description is depicted on Figure 14. A mission is usually identified by a name and the environment in which the target performs action.

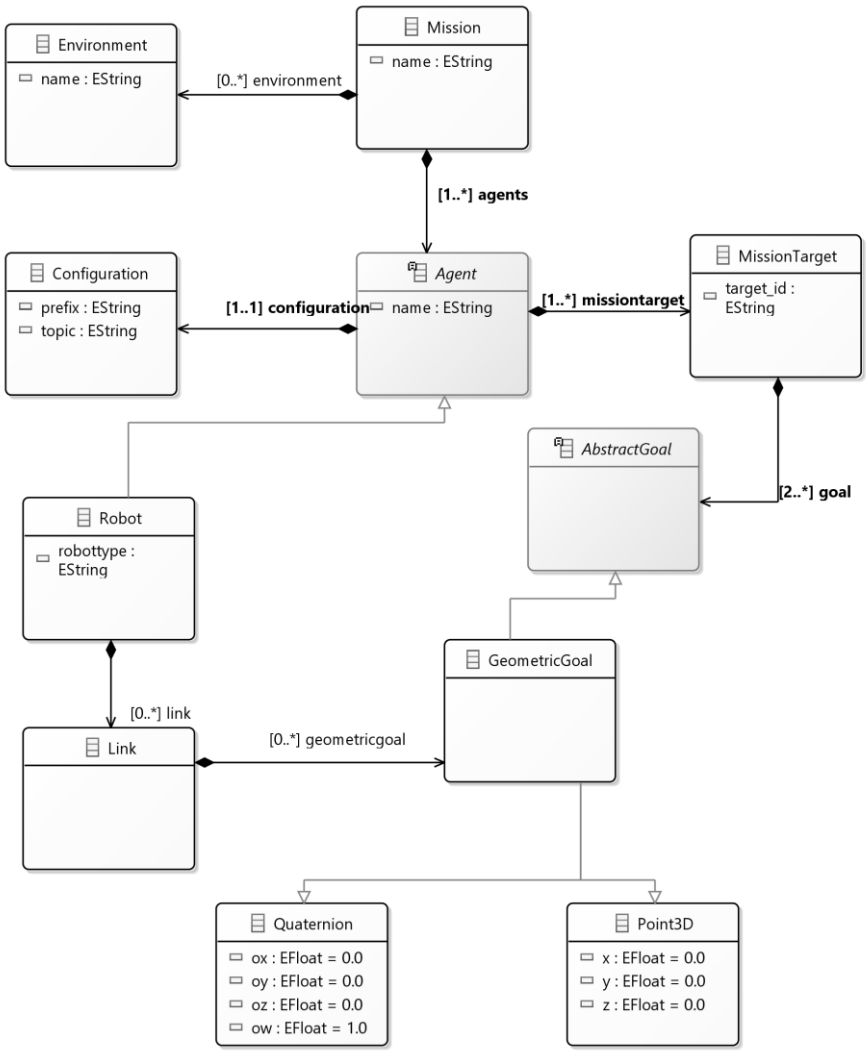

Figure 14. Description of a mission

Most mobile robots implement some navigation capabilities (section 2.1.1). Robots incrementally reconstruct their environment based on sensory data, which is transformed into a local map. Based on previous local map snapshots a global map may be constructed, if it is not available statically. Widely spread navigation algorithms use graph-based approaches to identify and extract landmarks from the global map (e.g. SLAM). As the path planning component of the robot only awaits a goal in a predefined format (navigation is usually done in the geometric space, so a geometric message is awaited, as in **GeometricGoal**), the mission description must be compatible with that high-level description of the action format (5.1.6.4). The mission target may vary for each scenario (or might be a kinematic link of the robot), or a robot might have multiple targets throughout a mission. Each target might have multiple goals (at least two), always starting with the starting state and the final state. In a robotic sense, the target is usually a mechanic frame. The target of a navigation problem is the global (world) frame.

A versatile mission description tool handles different controller (action) formats and not strictly revolves around geometric description. High-level commands may be issued to a controller component (e.g. find unique object/marker, transport some crates). The definition of the goal as an abstract object addresses this problem, to allow the extension of the mission metamodel with additional goal structures.

Also this mission description might be useful to define tasks for other autonomous systems, not only robotic systems. The robot is an agent in some sense (also can be described with a sensor-actuation cycle)

This framework handles mission description in the generator component as part of the abstract test case. For experimental purposes a textual editor (Xtext-based) had been created to define and generate goal descriptions. From an existing abstract test case, the mission is generated in a similar approach, like the simulation description was generated. The mission endpoints are mapped to existing floor elements of the simulation (assume that the geometric coordinates are available for each floor element, consequently for each goal endpoint), and their centric geometric coordinates are used as goal coordinates. Furthermore, the abstract test case is executed on a robot and the target frame is the world frame.

4.3.2. Mission execution component

This following subsection describes the operation of the mission executor itself. The mission executor component itself should be simple, given a mission description extracted from the abstract test case.

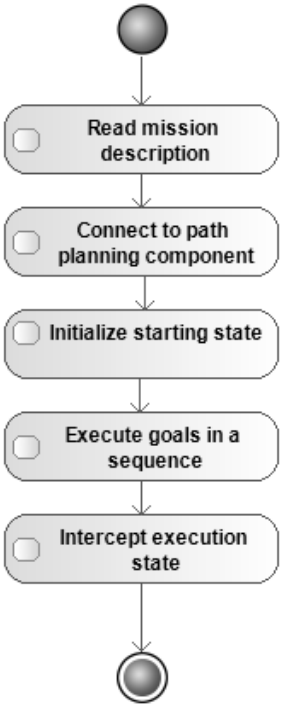

Figure 15. The high-level process of mission execution

This component iterates through every task and feeds them to the target navigation component. During every iteration, the component waits for the result of the task and then moves to the next task. The target is an operating and well-defined component of the simulated robot. The overview of the process is depicted on Figure 15.

5. Monitoring

5.1. Overview

This section describes the monitoring with a focus on the monitor and data collector components. The high-level process of the monitoring is depicted on Figure 16. Data is received from the simulator, which is used to construct a context model by reverse transforming the concrete elements, with the support of the same catalogue used to transform concrete description from abstract test data (4.1.3). The context model is then used to create a runtime model of the mission, which is used to match event patterns according to the input specification (5.1.1). The pattern matcher (5.1.6) then detects events and forwards them to the evaluator component.

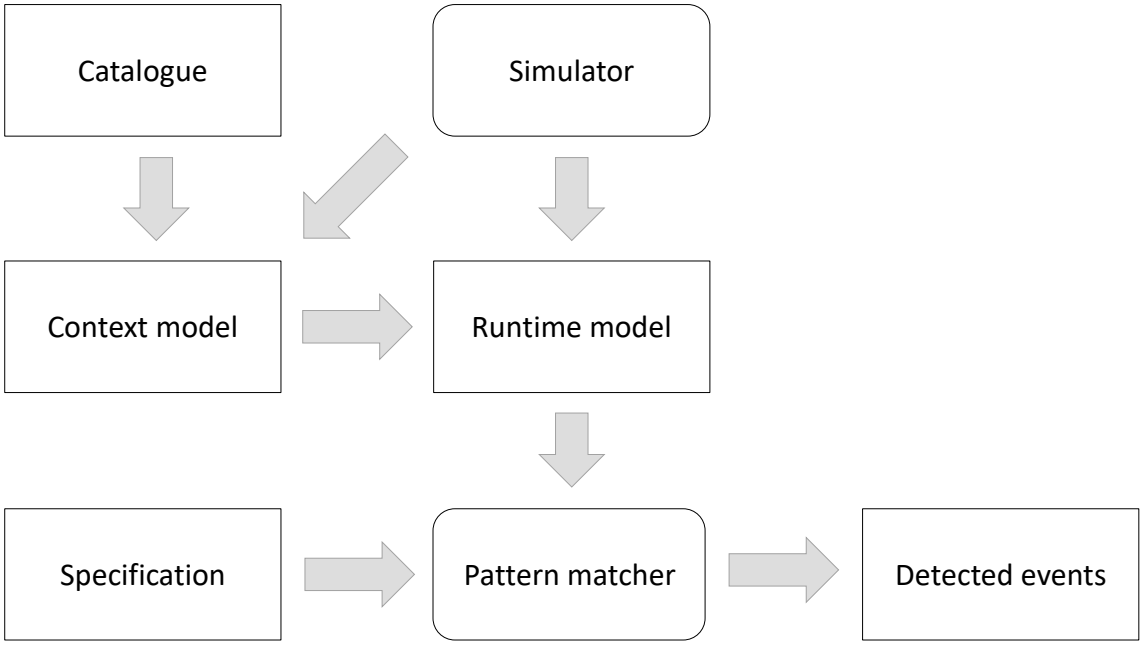

Figure 16. High-level description of monitoring process

5.1.1. Concepts

This subsection describes the concept of the monitoring process.

The monitoring itself is based on a specification, which defines the runtime requirements to be satisfied (Figure 17). The specification defines in *what context* the monitoring is done, and it inherently specifies the spatial constraints of monitoring. It also defines queries to check whether *target properties of the SUT* are satisfied. *Temporal patterns* can be defined to specify the temporality of sequential event generation.

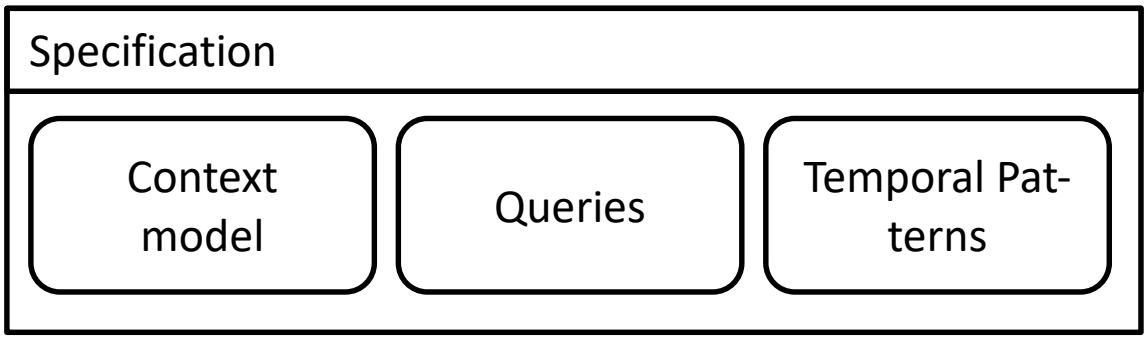

Figure 17. Components of specification

5.1.2. High-level requirements

The specification is also closely related to the high-level requirements of the system. This subsection defines some high-level requirements regarding the motivational example of this work (Section 2.4).

As the target robots are operating in an industrial environment, there might be places with huge sparse space with locally distributed objects (a hall or shed). People are moving around in the environment.

The robots should move in this environment from one point to another. The robot is able to move in this environment (by wheels or bipedal motion). During this movement, the robot should not collide with another objects, especially not with humans. The robot should have all the capabilities, to sense objects and their movement in such an environment (proximity sensor). The high-level requirements stated against a robot in this case study are summarized in Table 4.

Requirement identifier	Short description	Test scenario
GEOM_CLOSE_LASER_CORRECT	The robot correctly measures the close event	GIVEN the simulation and the robot WHEN the robot closes THEN the proximity sensors measure closeness
GEOM_COLLISION_LASER_CORRECT	The robot correctly measures the collision event	GIVEN the simulation and the robot WHEN the robot collides THEN the proximity sensors measure collision
MISSION_SIMPLE	The robot moves to the neighboring floor without collision	GIVEN the simulation and the robot and a pair of goals (g_1, g_2) WHEN the robot executes $g_1 -> g_2$ THEN collision never occurs

MISSION_SEQUENCE	The robot feeds back information about all goals	GIVEN the simulation and the robot and a sequence of goals $m = (g_1, g_2, \dots, g_n)$ WHEN the robot executes for all g_i in m THEN $send(finalstate(g_1))$
MISSION_SEQUENCE_CYCLE	The robot never collides in a mission where it moves back and forth	GIVEN the simulation and the robot and a sequence of goals $m = (g_1, g_2)^*$ WHEN the robot executes for all g_i in m THEN collision never occurs
MISSION_WITH_HUMAN_TRAJECTORY	The robot avoids collision with a human following a trajectory (number of collisions is limited)	GIVEN the simulation and a r robot and a sequence of goals $m = (g_1, g_2, \dots, g_n)$ and a h human on CR trajectory WHEN the robot executes for all $g_i \in m$ THEN $count(collision(h, r)) < N_{COLLISION}$

Table 1. Table of high-level requirements

5.1.3. Process

Figure 18 summarizesone run of the monitoring process, this process is run continuously during the monitoring. Assume the simulation is set up and running as described in 4.1. *Data is intercepted* during every step of simulation. This data is used to *update runtime observation model* (about the structure of the model, see 5.1.5.1). After every update, *events are matched* on results model queries. *Monitoring results can be evaluated* afterwards. The data might be *stored as a historical data*, for further evaluation or prediction used in machine learning methods.

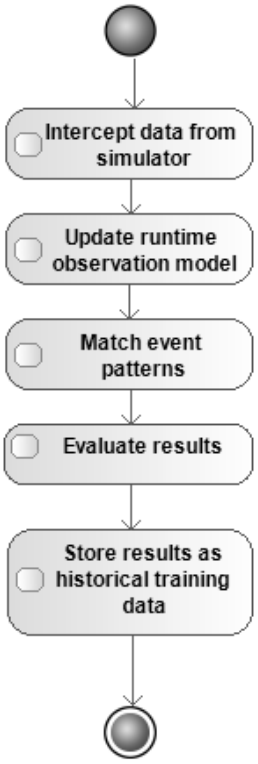

Figure 18. High-level workflow identifying the main activities of monitoring

Our framework performs basic runtime monitoring with the purpose to detect and diagnose the dangerous situation. The overview of the process Recovery may be solved through a safety logic connected to REI. The compartments of the monitor follows the terminology in [12] and 2.3.

- **Observer:** the data adapter attached to the ROS communication model and patching data to the hive. Filters might be used to transform incoming data (*Filter*).
- **Runtime model (~Analyzer):** the communication adapter interfacing the hive and subscribing the data stream. provided by the Observer component (*Monitor/Adapter*).
- **Pattern Matcher:** the monitor subcomponent, which executes Complex Event Processing (CEP) on the analyzed data and broadcasts generated events to the hive (*Monitor/CEP*).
- **Executing system:** the simulation of the entire physical environment and robotic components (~SUT).

The data collector component collects data from the middleware and patches through a more general network protocol usable even by platform-independent applications. The monitor connects to the collector through the protocol and processes complex events. In summary, both components as a whole connect the middleware to the evaluator component.

It is required to attach each component as loosely as it is possible. Each task has considerable computational overhead.

5.1.4. Observer component

This component collects data from the simulator and forwards to the analyzer component. The data might be filtered through special components.

The observer essentially fetches the geometric state of the environment. Without available geometric data, the analyzer cannot determine whether critical events occur (e.g. collision) independently of robotic belief state. The data should contain information of all objects present in the simulation.

The observer component should also provide information on the belief state of the robot. This enables the remote verification of the robot cognitive operation..

The subset of collected data to be collected can be selected in a configuration file. The configuration data should also contain technical data mainly regarding the communication with other components.

Technically, the observer is as closely attached to the simulator as it is possible. Usually, the simulator provides interfaces on a middleware layer, so the simplest way to solve the data fetch is through using these interfaces. This component is required to provide the data reliably to remote endpoints. This component forwards the data to interfaces of the selected communication channel, probably in a continuously available and refreshing manner. Also, the observer should provide the configuration of this component.

5.1.5. Runtime model component

This subsection describes the runtime model (analyzer) and its purpose. It subscribes to the data provided by the observer component, based on the configuration also provided by that component.

The main purpose is to construct runtime model based on data fetched from the simulator and share this model among other high-level components. This reconstructed data can be used for further operations such as event pattern matching.

Technically, the analyzer component is loosely attached to the simulator, it can be in a remote location – the observer already provides the simulator data remotely.

5.1.5.1. Runtime model structure

This subsection describes the runtime model update during the fetch of data provided by the observer component. This subcomponent is analogous to the analyzer component of general monitoring systems.

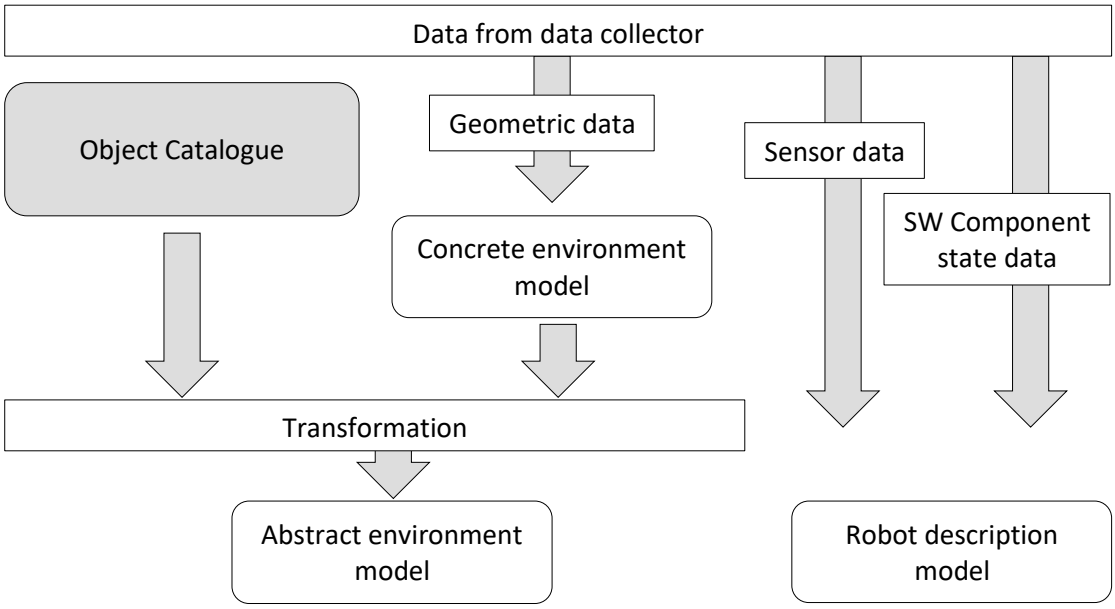

Figure 19. Runtime model reconstruction based on incoming data

The basic process of monitoring includes the analysis of incoming data from the data collector component. Technically this means to extract data from raw messages and construct a structured model from the data of interest runtime.

The runtime model reconstructed from incoming data can be divided into three main parts as it is depicted on Figure 19.

- **Concrete environment model:** it is constructed from the observed geometric state of present objects in the simulated environment. Constructing a concrete environment model is straightforward.
- **Abstract environment model:** the quasi-original abstract model of the simulated environment. It is constructed as the inverse process of transforming a user-defined model to concrete format 4.1.3, but uses the concrete environment model as an input
- **Robot description model:** this model is constructed from the belief state and static attributes of the robot. This model contains instant sensory reads (odometry, laser) or filtered data (RGB camera) and the state of selected software components (e.g. state of the navigation component).

5.1.6. Pattern matcher

This subsection provides information on the pattern matcher subcomponent of the monitor component. The high-level process is described on Figure 20.

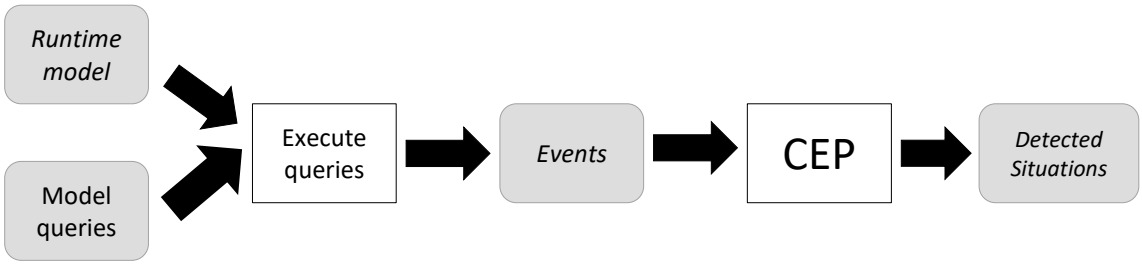

Figure 20. The process of pattern matching

The monitor uses pattern matching on events extracted from the runtime model. Model based queries are executed after every update. These queries are extracting information mostly based on available requirements. For example, if the target (robot) moves to close to an object, a corresponding query should return with a match.

Matches itself contain information about the state of the environment. While this is useful enough, the transitions from one state to another is more meaningful in this scenario. An example is a transition when the robot collides with objects but it was previously "close" to the object.

The evaluator component connected to the pattern matcher component – thus to the monitor itself – should assign scoring mostly to transitions.

5.1.6.1. Correctness criteria based on requirements

Previously the background has been discussed to construct a framework capable of running tests and intercepting incoming events from the test execution. With the pattern matching functionality, the framework is capable of verifying requirements defined on the available data.

5.1.6.2. Geometric regions

The motivational example of my work revolves around mobile robots operating in an industrial environment. The most important thing to consider and therefore to monitor is the current geometric state of the robot. It should be indicated if the robot closes to another object or building element (onwards in this section the unified set is used for all objects). The critical event is if the robot collides with another object.

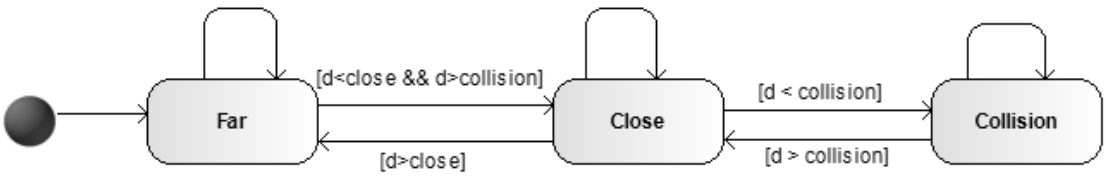

Figure 21. Robotic state in the context of geometric monitoring

The state transitions are depicted on Figure 21. The explanation of the defined states follows:

- **Far:** the robot is not close to any object.
- **Close:** the robot is close to an object, but not in collision range (i.e. the distance between the robot and the object is lower than a defined threshold ε_{close}, but larger than $\varepsilon_{collision}$). This state is part of the danger zone.
- **Collision:** the robot collides with an object (i.e. the distance between the robot and the object is lower than the defined threshold $\varepsilon_{collision}$. This state is considered as part of the danger zone.

In this context, the robot is considered to transit from one state to another, if it is closer to an object than a given threshold (Table 5). Consider, that the runtime model is available from the analyzer component. The transitions of each state are summarized below. Each $\varepsilon \in \mathbb{R}^+$ threshold is available, an o_i object is selected from the set of objects O, and the robot r is given in this context (which is also $r \in O$), with the position function of $p(o_i)$, and a distance function $d(o_i, o_j)$ of two arbitrary objects.

Transition	**Guard condition**
$FAR \rightarrow CLOSE$	$\exists o_i (d(o_i, r) < \varepsilon_{close} \wedge d(o_i, r) > \varepsilon_{collision}),$ $o_i \in O$
$CLOSE \rightarrow FAR$	$\nexists o_i (d(o_i, r) < \varepsilon_{close}),$ $o_i \in O$
$CLOSE \rightarrow COLLISION$	$\exists o_i (d(o_i, r) < \varepsilon_{collision}),$ $o_i \in O$
$COLLISION \rightarrow CLOSE$	$\nexists o_i (d(o_i, r) < \varepsilon_{collision}),$ $o_i \in O$

Table 2. Transition table of the geometric states

The complex event processing should raise events on transitions and forward event identifiers to the evaluator component.

It should be noted, that it is computationally complex to check that a condition does not hold at instant (like in case of $CLOSE \rightarrow FAR$).

The VEPL implementation of these patterns is described in 9.2.2.

5.1.6.3. Sensory states

To verify the correct operation of the robot, sensors have to be checked online.

All industrial robots have some sort of proximity checking capability, whether being mobile or fixed manipulators. This ensures that the robot is (theoretically) capable of detecting objects. In case of proximity sensors, the same regions are defined as for geometric distances.

When using a single source of sensor, the query of sensor data is relatively simple. Only the previously measured value is checked against region containment. But proximity sensors are usually array sources with multiple values (usually around 500), refreshing very frequently. Running runtime checks after their model update is as the queries are running incrementally on data change. To resolve this problem and reduce the computational

overhead of the sensor monitoring, filters can be employed to determine the minima of the measured proximity and only this value is used for further analysis.

Moreover, multidimensional sensors, like cameras provide information whose raw values are usually inefficient to store in a model structure. Mostly extracted values are useful from a camera image, i.e. the position of a landmark, line position, etc. In this motivational example, camera images are not extensively used, as industrial robots are usually solely equipped with laser proximity sensors of high quality.

5.1.6.4. Navigation states

During robotic operation the control components of the robot must be also monitored.

Navigation can be abstractly defined as an abstract action that a robot implements. An action can be defined with a goal (in sense of navigation), with feedback of the execution process and the result of the execution (fail/success). The behavior of action execution can be modeled as a simple state machine (Figure 22). The action itself is realized by an embedded (software) component compatible with this description. Robotic frameworks like ROS allows the description of high-level actions, and their implementation as action-servers.

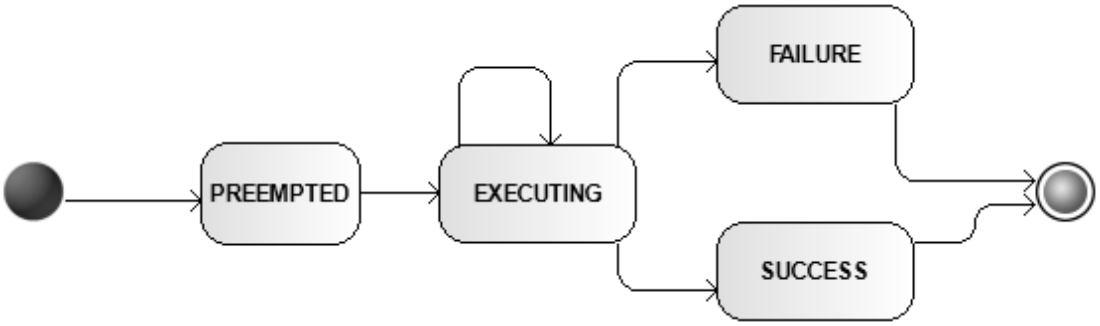

Figure 22. Simplified state machine of action execution

When concretely the navigation components are monitored the information of interest is the current state of the execution, which is actually indicates how far the robot traveled on the planned trajectory (i.e. feedback). The result on completion, and the geometric goal itself holds interest. Also, the current state of planner may return a local and global plan, which can help to verify the correct operation of navigation.

Navigation can result in additional events (e.g. re-planning, evasion scenario execution, block recovery). These events may or may not be observed, depending on the test expectations.

During monitoring the changes are not essentially updated into the robot description model, rather than analyzed runtime as part of atomic operations or by filter components resulting with extracted properties. While the state and feedback is a structured and somewhat object-oriented information, the path and the map constructed during the SLAM process is a complex multi-array data which may not be stored as part of a structured

model. Practically, some subset of information might be gained from this component and the data extracted might be stored in the model.

5.1.6.5. Queries

The following subsection describes the queries related to high-level requirements (Section 5.1.2) and employed by the pattern matcher (as part of the specification Figure 17).

Assume that the structured model of the target environment is available, alongside with some runtime observation of robot sensors. Extracted from the requirements, the following queries must be matched after any update of the artifacts (assume r is the robot, o_i is any object from the set of objects):

- Geometric queries
  - Close to object: match if the robot is in the $\varepsilon_{collision} < d(r, o_i) < \varepsilon_{close}$ range.
  - Collision with object: match if the robot is in the $\varepsilon_{collision} > d(r, o_i)$ range.
- Sensor queries
  - Laser sensor senses closeness: match if the minimal proximity measured by the laser sensor is in the $\varepsilon_{collision} < d(r, o_i) < \varepsilon_{close}$ range.
  - Laser sensor senses collision: match if the minimal proximity measured by the laser sensor is in the $\varepsilon_{collision} > d(r, o_i)$.

With the help of defined queries, temporal patterns can be matched on the runtime model created during monitoring.

5.1.6.6. Temporal patterns

This following subsection describes the concept of defining temporal patterns composed of queries (as part of the specification, Figure 17). The monitor awaits the specification of the monitoring subject, which defines the events to process for the evaluator component (Figure 23).

Figure 23. Temporal patterns

It was described in section 5.1.6.2, that in this motivational example, the transition from one state to another holds more interest. The temporal patterns are therefore intuitively defined as the match of one state after another according to abstractly defined state machines. The queries define the current state of the monitored robot.

Therefore, to define a new event pattern, analogous patterns to the transitions defined in Table 2 can be used. On the exact implementation of temporal patterns the implementation part describes the details (6.1.3.3). The temporal patterns realizing the geometric state detection (5.1.6.2) are described in the appendix.

A large contribution of this framework that it enables the use of external temporal specification editors (e.g. VEPL, part of the VIATRA-CEP project). The pattern matching rules generated from this description can be used to detect events as described in a high-level specification.

5.2. Evaluating the events

This section describes the evaluation component. The detected events shall be forwarded to an evaluator component, which logs them and executes basic analysis operations.

The evaluator logs each event of interest. It is a good practice to operate the logging in a distributed way, which is inherently enabled by the communication protocol used by the framework.

The simple operation of the evaluator component can be summarized as follows:

- The evaluator component connects to the monitor component and receives complex detected events.
- Analysis of events
- Each mission is logged to a different file.
- The evaluator essentially logs all intercepted events.

The evaluator maps each event to a corresponding utility value and at the end of mission returns with a total utility. To do this, the evaluator associates each event type with a corresponding utility value. If all observed events are mapped to their corresponding utility values, then at the end of the test case execution, the utility of the test case can be defined as the weighted sum of event utilities.

$$U_{testcase} = \sum_{i=0}^{n_{events}} U(events[i]), \qquad U(event) \in \mathbb{R}$$

The utility values can be useful in this phase to indicate which test environment is considerably hard for the target robot. Furthermore, utility values might be used for more complex evaluation components.

5.2.1. Utility assignment

Each event can be mapped to a corresponding utility value. Utility values are useful to give an objective measurable result of the mission execution supporting the evaluation process. Utility values can also be used for further machine learning algorithms as an objective to learn.

Choosing the correct utility value is usually a difficult task. This subsection uses the requirements defined in section 5.1.2and presents some ideas behind the assignment of a useful value.

The events (state transitions) that can be associated with a negative utility value – i.e. penalty - are considered as safety critical events or events that may be a presage to such critical events. These include:

- Robot collides with an object. This event is associated with considerable penalty as usually results with damage.
- Robot closes to an object. This is a recoverable event, and therefore a moderate penalty is assigned.
- The proximity sensor senses transition to collision from closing range. This might be a recoverable event, as the sensor might fail.
- While it is not considered as a safety critical event, the failed task execution might be highly penalized. Executing tasks is a functional requirement and failing to do it reliably, is violating the main functionality of the system.

Contrary, the events (state transition) that can be associated with a positive utility value – i.e. reward – are those which indicate that the robot tries to recover from a safety critical situation. These include:

- Robot leaves the danger zone, or tries to recover from collision.

The following functional requirements are rewarded:

- Robot starts a new task.
- Robot completes a task. This is a functional requirement. The reward of the task execution is not necessarily equal to the penalty of the failure of the task.

The initial setup of utility values is summarized in Table 3. The explanation of each event is below the table (note that some transitions are based on the state-machine depicted on Figure 21):

Event identifier	**Utility value**
SIG_MISSION_BEGIN	20
SIG_MISSION_END	20
SIG_MISSION_TASK_START	5
SIG_MISSION_TASK_FAIL	-100
SIG_MISSION_TASK_SUCCESS	10
SIG_GEOM_FAR_TRANSIT_CLOSE	-2
SIG_GEOM_CLOSE_TRANSIT_FAR	1
SIG_GEOM_CLOSE_TRANSIT_COLLISION	-10
SIG_GEOM_COLLISION_TRANSIT_CLOSE	1
SIG_LASER_CLOSE_TRANSIT_COLLISION	-5
SIG_LASER_COLLISION_TRANSIT_CLOSE	1

Table 3. Assignment of events with utility value

- **SIG_MISSION_BEGIN:** indicates the beginning of mission.
- **SIG_MISSION_END:** indicates the end of mission.
- **SIG_MISSION_TASK_START:** task of the mission started.
- **SIG_MISSION_TASK_FAIL:** task of the mission ended and resulted in failure of some cause (e.g. goal is blocked), or the planner reports success when it is not actually reaching its goal.
- **SIG_MISSION_TASK_SUCCESS:** task of the mission ended and resulted in success (the robot reached its task according to the planner object) and it has actually reached its goal.
- **SIG_GEOM_FAR_TRANSIT_CLOSE:** this signal is raised when the robot closes an object.
- **SIG_GEOM_CLOSE_TRANSIT_FAR:** this signal is raised when the robot moves away from the object, from previous closing movement.
- **SIG_GEOM_CLOSE_TRANSIT_COLLISION:** this signal is raised when the robot collides with an object from previous close state.
- **SIG_GEOM_COLLISION_TRANSIT_CLOSE:** this signal is raised when the robot recovers from collision, but moves to close state.
- **SIG_LASER_CLOSE_TRANSIT_COLLISION:** this event is raised, when the robot senses collision from previous far sense.
- **SIG_LASER_COLLISION_TRANSIT_CLOSE:** this event is raised, when the robot senses evasion from collision.

6. Implementation

The following section describes the considerations of implementation and some of its details. The components were identified in Section 3.2.2.2, the system overview is depicted on Figure 24. This is different from the high-level architecture description in a sense, that only the contributed elements are depicted. The environment simulation is an external component (Gazebo), which connects the middleware thus the framework components to access environment state.

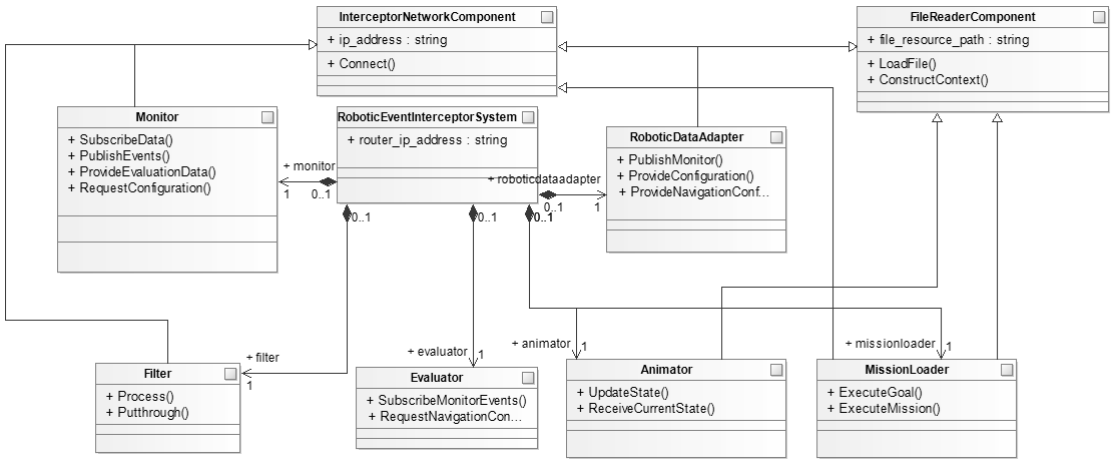

Figure 24. Component setup of the system

The following sections describe the components from two perspectives, from a network-interconnection perspective and artifact usage perspective.

6.1. Interconnection of components

The following section describes the components from the viewpoint of network connectivity. This describes, how the components are connected to each other over the protocol of choice, and what interfaces are provided to each other.

6.1.1. Hive implementation

To connect each loosely coupled component, the framework uses a WAMP hive (section 2.5.3). As it was described, WAMP uses a dedicated router to connect components to each other. WAMP uses Websocket as transport protocol, this allows to develop general applications connecting to the system (even JavaScript-based frontends).

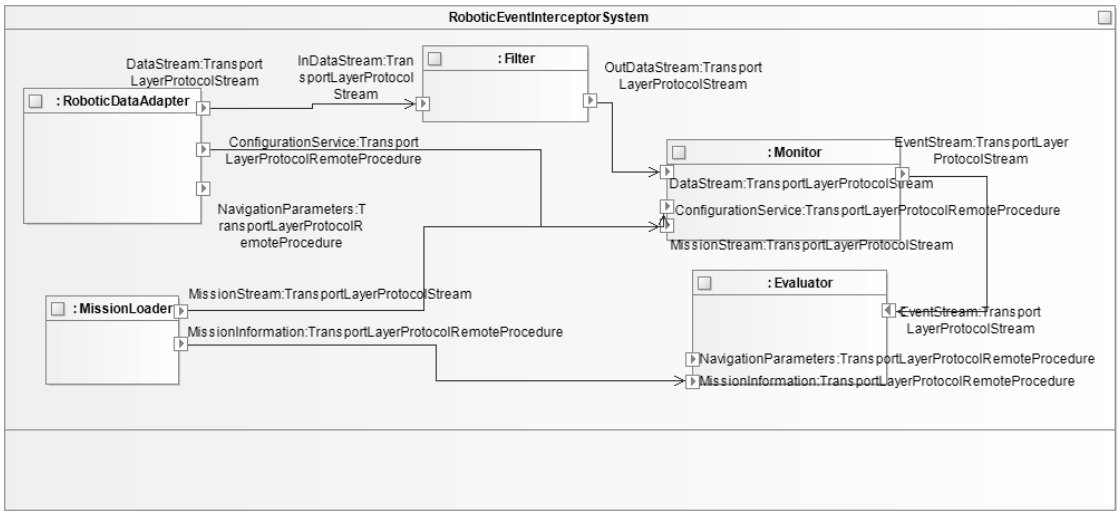

Figure 25. Network interconnection of components

This router fulfills the broker (Publisher/Subscriber) and dealer (RPC) roles simultaneously. Technically this means that a WAMP application can employ both approaches at the same time without using two different 3rd party libraries – and inherently two different network settings. Also, the router defines the AAA (Authentication/Authorization/Accounting) tasks of the network. This means, that ACLs can be easily defined to filter connections that are not supposed to connect in the router configuration. This is really useful as it is relatively easy to write an application to connect to the WAMP backend.

There are numerous implementations of WAMP based routers, or allows the implementation of routers by a program. This framework uses Crossbar.io[7] router.

The network interconnection of the implemented components is depicted on Figure 25. The router is not depicted there, it is connected to each component and helps to establish connection between components.

6.1.2. Data collector

The main purpose of this component is to supply data collected from the robotic middleware (and simulation) and patches through the communication channel in use. This is the main performance bottleneck of the system. Other components synchronize to its publish rate, therefore this component defines the processing sample time of the whole system. The data adapter limits the target of monitoring as it provides the data stream, which is configured before the beginning of the operation (through file).

The component shall be configured to set the network connectivity settings of the monitor, the identifiers of the published topics and to which subset of the data fetched from the environment should be forwarded.

[7] Crossbar.io is a registered trademark of Crossbar.io GmBH

The data adapter is basically an ROS node, subscribing to ROS topics and services. But it is also composed of a WAMP node part, to forward data to the WAMP hive. The data adapter provides a WAMP topic to allow other WAMP-based applications to subscribe to this data stream.

For transmission, WAMP uses MsgPack format. MsgPack serializes only primitive types and structures by default. Therefore, all ROS messages must be serialized to this format. Fortunately, all ROS messages have strict structures and they can be easily mapped to a corresponding C++ built-in type (with the help of generators).

Also the data adapter reads a semi-structured format of configuration with essential information to connect the WAMP hive. This configuration is shared to the monitor component, to help with the initialization of the monitoring process.

6.1.3. Observer

This component subscribes to the data stream generated by the data adapter and analyses its content. This component unifies the monitor and event handler component of the typical monitoring structure (identified in 2.3).

To initialize the monitoring context, the monitor uses the configuration received from the data collector. The monitor connects to the data collector component through the use of a WAMP client (jawampa). The data collector is a native application which supports raw socket format, but the Java implementation only supports web-based transport. WAMP allows the use of mixed transports between two endpoints, but each transport must be broadcasted on different port.

The monitor employs an EMF instance model of the abstract and concrete test case description. On each message update, the runtime model is updated with new state of every object. The queries are executed runtime on the model and the matches are used to generate events with the pattern matcher generated from VEPL files by the VIATRA-CEP framework.

The detected complex events are forwarded on a WAMP topic to the evaluator component.

The following subsections specifies some details of the monitoring operation.

6.1.3.1. Distance calculation

The general idea behind the collision detection is to determine the distance of all objects to the robot. In this context yet only the footprint of all the objects is needed.

The footprint definition of primitive objects is straightforward, as they are easily defined mathematically. Also, the distance calculation of primitive geometric forms to a point requires only high-school level mathematics. Most of the building blocks are modeled as cuboids primitives.

On the other hand, the simulation employs complex mesh objects and the calculation of footprint of such objects is computationally difficult in runtime under the usual require-

ments of robotic systems. Fortunately, the simplified modeling of footprint of some complex objects with primitive objects is applicable and greatly decreases the computational overhead.

For example, the robot might be modeled with its bounding box (the collision model is usually constructed from primitive objects anyway) which simplifies the footprint calculation. Calculation of distance of a rectangle to any other primitives is just multiple calculation of vertices to the target primitive.

This simplification might be useful when other complex models, like a human is used, in the early phases of the verification process. The human can be modeled with a bounding box, and if the robot moves close to this box, the same distance calculations can be performed.

6.1.3.2. Implementation of queries

The queries defined in 5.1.6.5 can be easily implemented with the help of VIATRA queries. VIATRA provides an intuitive language to define queries on EMF instances, and the monitor employs EMF metamodels at runtime.

VIATRA queries are running relatively fast on graph based models with large number of nodes. As the defined queries are implemented with the help of the language, the execution speed can be further enhanced with some implementation-level considerations. The robot will be most likely in danger zone if it is inside the circular range of other objects. The radius of this range should be greater than the maximum possible length of the building objects. With these conditions, the query input of distance estimation can be significantly reduced. VIATRA allows the use of external function calls. The distance function can be included in a query this way. This allows using different distance functions in special scenarios.

An example of defining a query is described below. This query reduces the objects to match into a smaller radius. This definition also shows the possible use of external Java functions.

```
pattern closeToObjectRadiusFilter(t: Robot, w: SimulatorObject, p0: Posi-
tion, p1: Position){
	find uniqueObjects(t,w,_,_);
	Robot.pose(t, pos0);
	Pose.position(pos0, p0);
	Position.x(p0, x0);
	Position.y(p0, y0);
	SimulatorObject.pose(w,pos1);
	Pose.position(pos1, p1);
	Position.x(p1, x1);
	Position.y(p1, y1);
	check (
		hu.bme.mit.inf.robotics.util.geometry.GeometricFunctions.eu-
clideanDistance(x0, y0, x1, y1) <= 1.0
	);
}
```

First, it must be ensured, that only unique objects are selected. After defining the fields of interest, each field can be used to check against a condition (in this case, against the result of an external function).

6.1.3.3. Implementation of timed patterns

The temporal patterns defined in 5.1.6.6 and implemented queries defined in 5.1.6.5 can be easily implemented with the help of VIATRA-CEP. VIATRA-CEP also provides an editor to define temporal patterns based on existing VIATRA queries.

From the pattern description, VIATRA-CEP generates Java classes, which can be used by external applications, in this case by the monitor itself. The VEPL pattern description employs the implementation of temporal patterns in 5.1.6.6. The only difference from temporal patterns defined there is that new atomic events are defined, indicating that a new message update has been received (for all input topics). All patterns are executed soon after that. Without this mechanism, the queries might run on all changes of the EMF instance, and therefore the re-execution of model queries can be controlled this way.

VIATRA-CEP allows two main type of event context, immediate and chronicle. In the early development phase, chronicle event context was used. This is to ensure that the framework is capable of detecting any event. The main problem is with the chronicle event context is, that it tends to store way too many snapshots of the runtime model. In further versions, immediate event context is used which operates in a narrower context.

Also, intuitively the temporal patterns would be directly implemented as analogues of state machine transitions. If strict event context is used, the temporality is realized if one match is lost (the left side of the transition expression). An example of the implementation of a temporal pattern is described in the code example below.

```
complexEvent closeTransitCollision(){
   as(notAtomicCloseToObject -> atomicCollisionToObject){+}[1000]
   context immediate
}
```

This describes the transition of the robot from close zone to a collision course. The query events are previously defined (close matches *atomicCloseToObject*, collision matches *atomicCollisionToObject*, loss of match is with *not* prefix). The match of object closeness is lost, and immediately after collision with object is matched. This pattern defines some temporality, as there is some delay between losing and gaining a match. Additional

6.1.4. Mission executor component

The mission executor provides information about the started or finished task. The information provided is the goal setup, the name of the current mission.

The mission executor supplies task state on a topic to the monitor. The monitor can further evaluate, whether the goal execution satisfies results.

The evaluator component uses the service provided by the mission executor component. This service provides basic information about the mission itself (e.g. name of the mission, targets, etc.).

6.1.5. Evaluation component

The evaluation component which subscribes to the events generated by the monitor (defined in 5.2).

The evaluation component intercepts the events detected by the monitor component. The evaluation process can vary in complexity. At its most basic, this component creates a log for every mission. Also the evaluation component connects to the mission executor component, to receive mission information at the beginning of each test scenario.

The log file contains the time and identifier of each event and their associated utility along with the current total utility. The log is stored in simple file format (CSV) to allow extensive use by general data analysis frameworks.

6.2. Abstract test case artifacts perspective

The components listed in this section use one of the artifacts generated from the abstract test case. The following subsections specify the

6.2.1. Test room generation tool

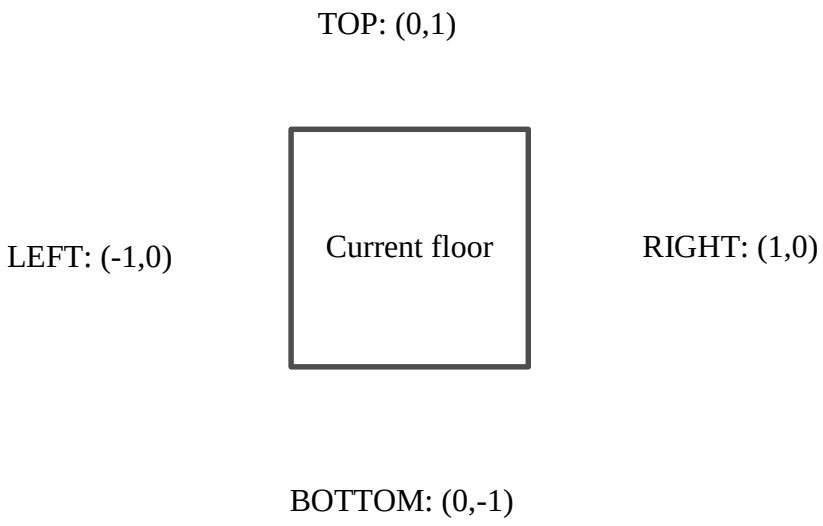

Figure 26. Coordinate mapping during graph traverse

The Test room generation tool inputs a metamodel description and outputs the artifacts defined in 4.1.3.2. This component plays a mandatory role regarding the whole operation of the system while it is not performance critical.

The test room generation tool is an Eclipse-based AST tool, generated from the Ecore description of the room (4.1.2) and employs generator libraries using the models. The instance model creation is implemented by straightforward Ecore library methods.

The abstract test data does not contain geometric data. The idea behind mapping abstract test case objects into geometric coordinates is to select a starting floor – which is most likely the starting position of the goal. Then a graph traversal can be started from this dedicated floor, using the four neighbors of the floor. Each node in the traversal tree is

mapped relative to its parent (Figure 26). The branch stops on a wall element neighbor. The whole algorithm stops, when all floor elements are traversed through its neighbors. The coordinates are essentially unit length. This algorithm essentially works with a single room at a time.

As the floor mapping is available, the dynamic objects can be placed according to the available coordinate mappings. If all objects have been added, the simulation description can be created according to the specific format. This is actually an XML serialization process. The animation scenarios are injected into each object description, and therefore into the environment description itself. A mission description can be created, with the available coordinate mappings of floors (as the goal start and end points are floors in the abstract test case).

The robot is usually available through a separate description in ROS manifesting not only the physical description of the robot, but also the simulated software components. Therefore, a launch file should be only generated to an existing robot setup.

6.2.2. Animator component

This component implements the object moving functionality of the framework. The component inputs a room description which contains the room description and animation scenario of target objects (i.e. a subset of environment objects). The animator component is not connected to the network, but closely connected to the simulator and the simulator description format itself.

From an implementation perspective, it is easier to use a simple velocity controller interface and forward control messages. To use a velocity controller, the concrete model must include the controller setup description.

The animator component sets the velocity of the target object corresponding to the trajectory function defined in 4.2.2.

6.2.3. Mission executor component

The mission executor component loads a list of goals from a mission description file (. As mission is described in a semi-structured manner (XML), the component employs an XML deserialization subroutine.

The mission loader forwards all read goals to the path planner component of the robot. The mission loader forwards a goal to the navigation action server defined in the target description. The topic of the controller of the corresponding target awaits new goals from series of tasks (**move_base/goal**), the program forwards each compact description to this topic. The mission loader component receives feedback on each goal completion from the controller. The state of completion is then forwarded to its dedicated WAMP topic. When no more tasks are available, the program terminates.

7. Evaluation

This section describes the evaluation of the framework.

7.1. Evaluation context

The evaluation is performed on two sets of data. One is a motivational example for the whole context which is evaluated in more depth. The other data set consists of numerous environments generated by an external tool. Also, consider a differential driver robot (Turtlebot3[8] waffle) as the target of the simulation.

7.2. Simple test scenarios

The following scenarios are basic test cases mostly demonstrating the capabilities of the current framework. The robot is controlled through Simulink components rather than using the navigation capabilities of the robot.

7.2.1. Toolset

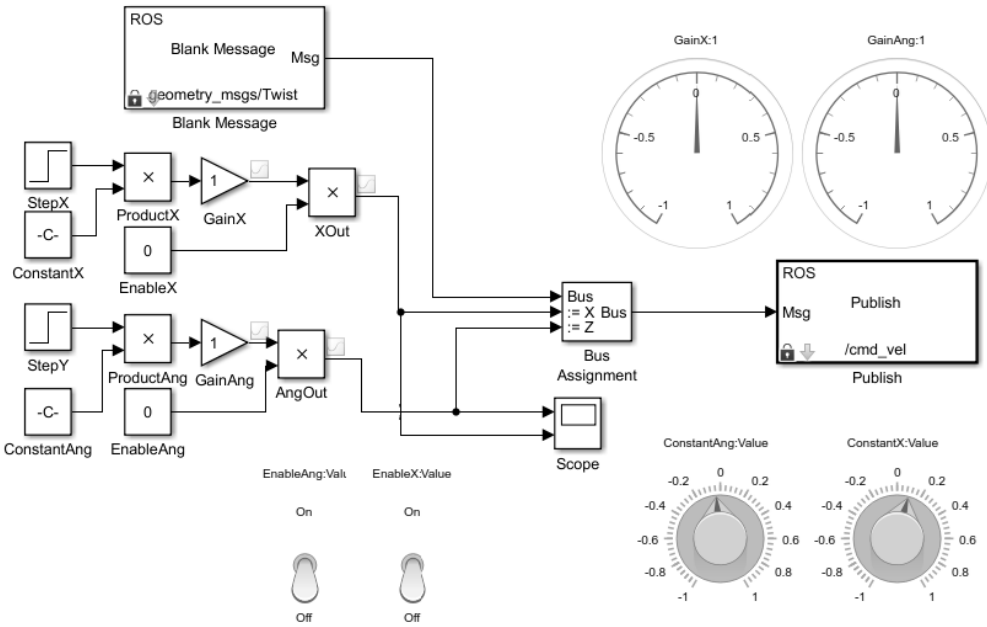

Figure 27. Dashboard to teleoperate ROS-based robot in simple scenarios

Throughout the simple test scenarios, a MATLAB/Simulink tool was developed to perform basic movements using the low-level controller of the robot. The simulation is monitored with the help of the framework.

The tool created for the simple test is basically a dashboard, allowing to change the angular and forward linear speed of a differential drive robot. The tool connects directly to an ROS network and publishes control commands to the drive control of the robot

[8] Turtlebot is a trademark of Robotis Inc.

(**cmd_vel** topic in general). The movement can be stopped at any time (the speed is set to zero).

7.2.2. Simple translation scenario

This is a simple scenario demonstrating the basic functionality of the framework on a very basic room. This scenario showed that events are raised correctly, when the robot moves closely to the wall or any other static object.

During execution of this test, the robot moved forwards and backwards. The monitor detected the events, when the robot really collided with a wall or an object.

7.2.3. Circular movement scenario

This is another simple scenario demonstrating the basic functionality of the framework throughout a robot moving in circles. With this, it was demonstrated that the monitor can indeed monitor the events detected when the robot collides the object from any side.

7.3. Simple navigation scenario

The following scenario selects a goal of the robot, and leads it to one part of the room. This scenario is fully supervised and each crucial step is described. The goal is set through the mission executor component. The room is relatively small.

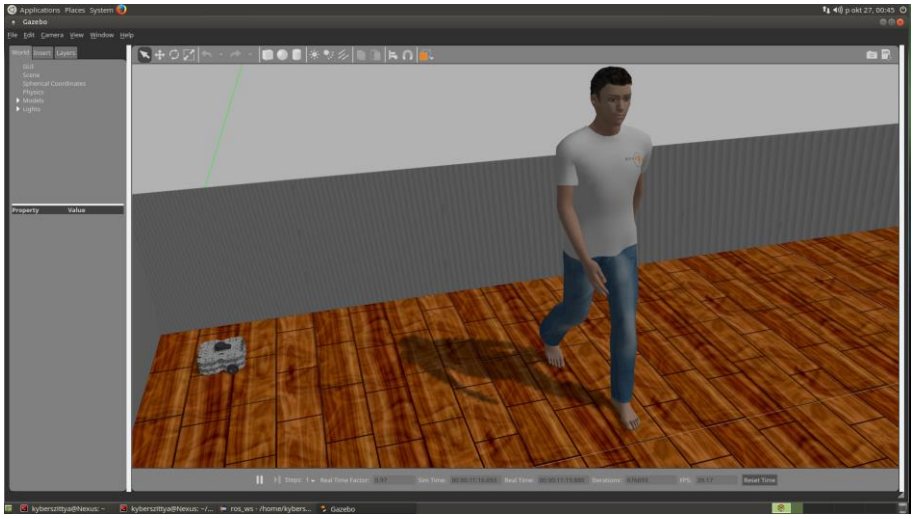

Figure 28. Initial state of the room

Assume that the **rosmaster** component and the Crossbar.io router is running – also the framework is properly installed. The following steps must be performed:

1. Generate the abstract test case with the help of the editor tool.
    a. This can be done with the help of the graphical editor (Sirius-based) or textual editor (Xtext based), also by the default reflective editor.
    b. Generate artifacts from the abstract test case, by saving the text description or by right-click in the graphical editor and click on “Generate Artifacts”.
2. Start the simulation and load the environment (be sure that the ROS instrumented simulator is launched).

```
$> rosrun gazebo_ros gzserver solution_10_0.mircontext.sdf
```

3. Start the robotic components of use. This can be done by launching the generated software setup. This will place the robot inside the simulation and starts its software components.

```
$> roslaunch solution_10_0.mircontext.launch
```

4. Start the animator node, with the parameter of the simulator context description. After startup it will move the human inside the simulated environment on a predefined trajectory.

```
$> rosrun object_animator object_animator_node solution_10_0.mircontext.sdf
```

5. Start the data collector on the ROS side. Provide the configuration file which describes the data of interest. After this, WAMP applications can access to the data provided by the simulator.

```
$> rosrun ros_test_bridge ros_test_bridge_node example.xml
```

6. Start the monitor. This is a Java application. It outputs the events detected and visualizes the position of the robot. Provide the IP address of the Crossbar.io router.
7. Start the evaluator component. This is a Python application. The log file is created on the hard disk. Provide the IP address of the crossbar router.
8. Start the mission executor component on ROS side. After start, the navigation component of the robot receives the start position of the mission. After the goal execution stopped (with the result of success or fail), the end state of the mission is send to the navigation component.

```
$> rosrun mission_loader mission_loader_node solution_10_0.mircontext.xml
```

The robot has to arrive into the “elevator” part of the room (Figure 31). During execution, the robot shall not collide with anything, particularly not with humans.

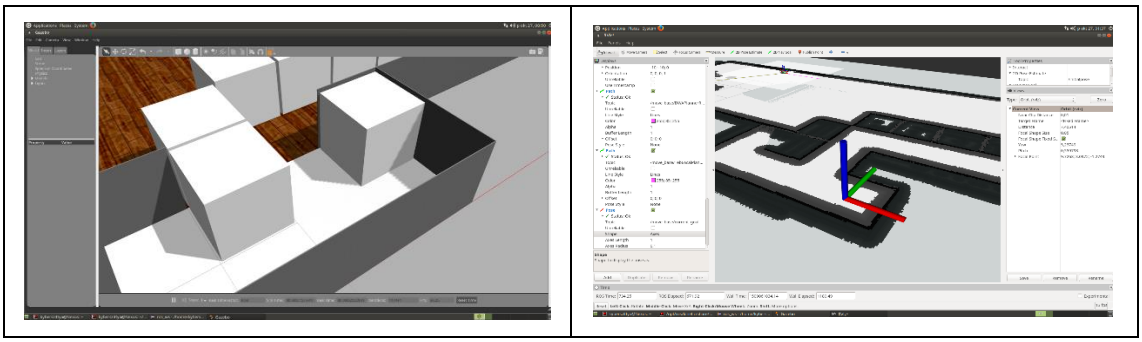

Figure 29. Goal of the robot

At startup, it can be noticed that the robot percepts the human trajectory as a moving obstacle. The monitor outputs the current observed state of the robot (FAR).

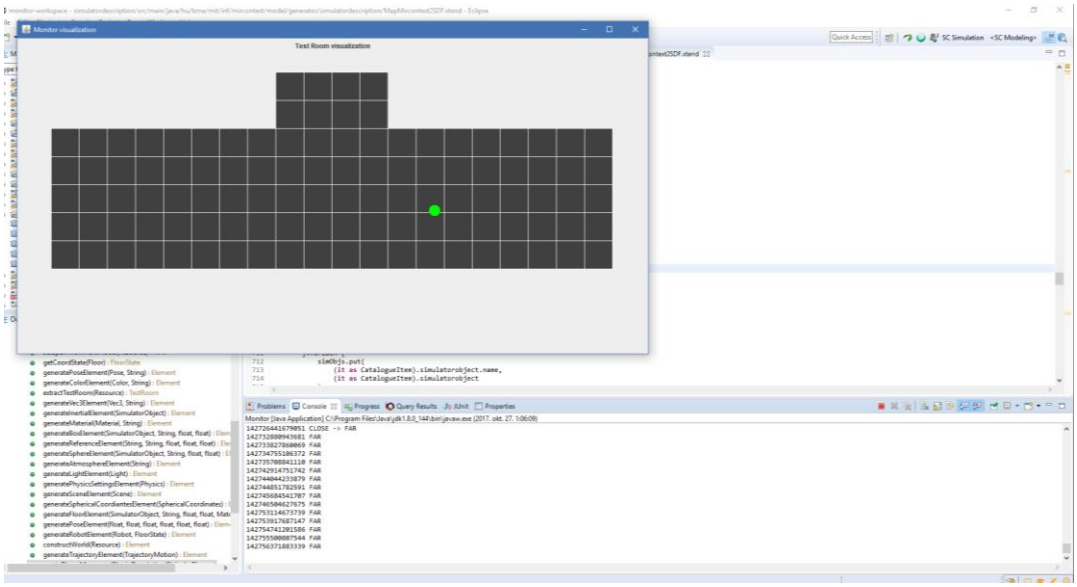

Figure 30. Observing the mission with the monitor

As the robot progresses to its goal, different events and the exact position of the robot might be observed through the observer component. We can observe on the monitor side, that the robot frequently goes into danger zone (Figure 31), but tends to avoid collision. The robot even manages to avoid contact with the moving person.

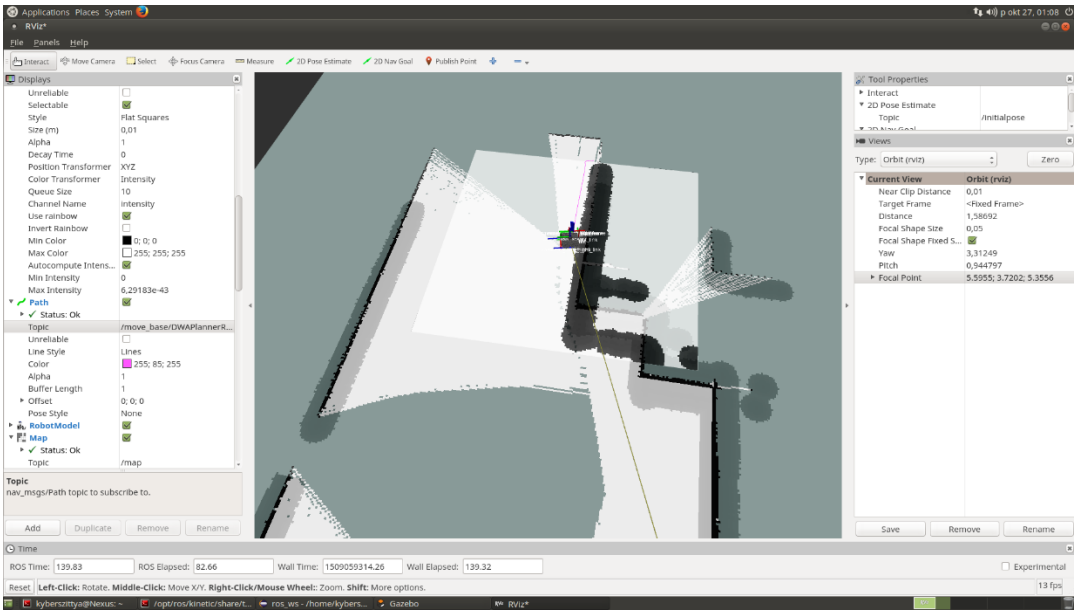

Figure 31. Robot in danger zone

As the mission progresses we may notice that the robot has difficulties to evade objects and some map construction errors appear. The navigation settings could be fine-tuned or the SLAM algorithm implementation must be revisited.

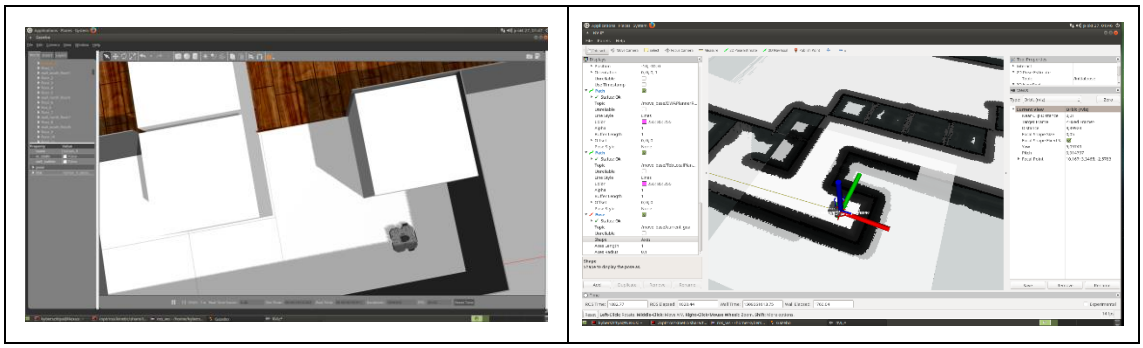

Figure 32. Final state of the mission

Finally, the robot arrives to its goal location (Figure 32). The evaluator collected a log of files, which can be further analyzed. It indicates that the mission ran for 381 seconds. Overall, this mission ran particularly well, with a total utility value of 80.

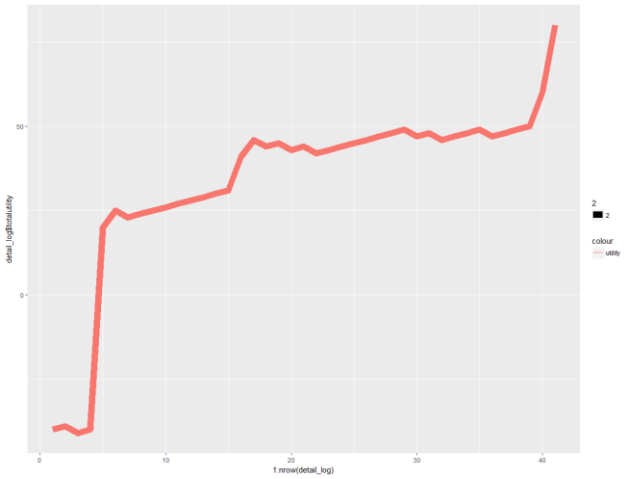

Figure 33. Change of utility as the mission progress

The events occurred during this scenario are summarized in Table 4. No collisions occurred, the robot successfully evaded the human.

Event identifier	Occurrence
`SIG_GEOM_CLOSE_TRANSIT_FAR`	26
`SIG_GEOM_FAR_TRANSIT_CLOSE`	9
`SIG_MISSION_BEGIN`	1
`SIG_MISSION_END`	1
`SIG_MISSION_TASK_START`	2
`SIG_MISSION_TASK_SUCCESS`	2

Table 4. Occurrence of events in the detailed scenario

7.4. Generated test scenarios

The following scenarios are using the mission executor component to perform missions using on the robot. For each selected test environment, two test are executed one with and one without animated objects.

7.4.1. Scenario template

The execution of each complex test scenario follows the same template. The *test environment is loaded into the simulation*. If there were animated objects defined, the animator component starts to animate the selected objects. When the simulation is ready, the *mission is executed*. During execution, the simulation is monitored and *events are extracted and logged*.

7.4.2. Toolset

These test cases use only the components provided by the framework. The logged results were analyzed offline with the help of the R framework.

7.4.3. Expected outcome

We expect to intercept events on mission completion, geometric transition events (into and from danger zones), mission execution time and the utility.

7.4.4. Small and dense rooms

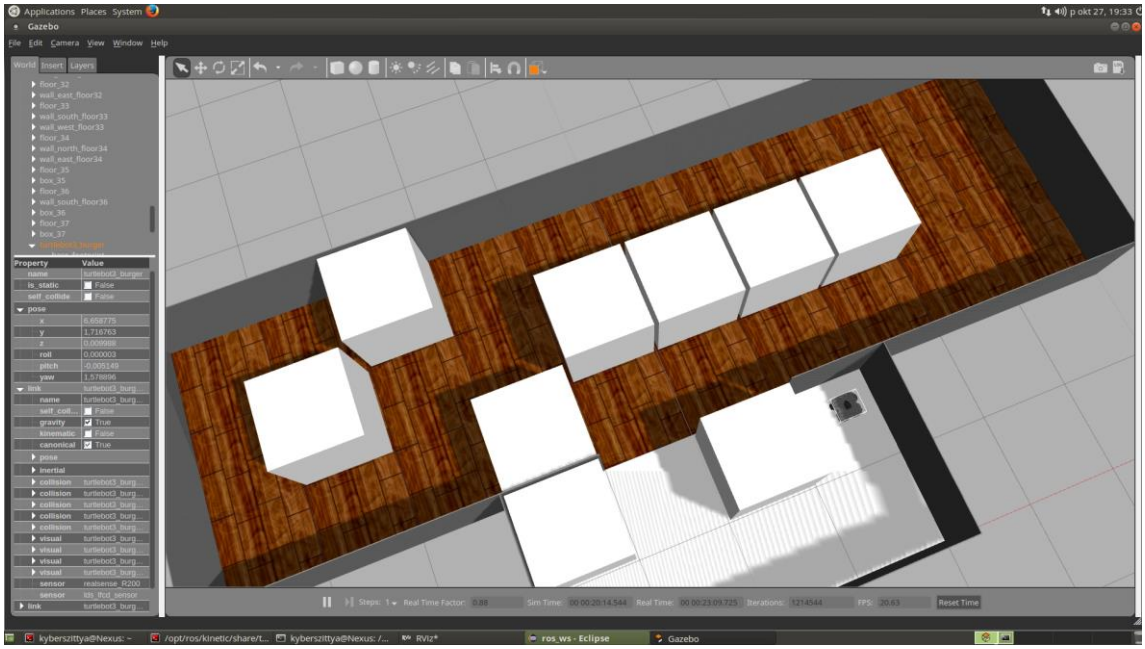

Figure 34. Example of generated small room

The following scenario executes a mission on the robot in a relatively small room, with t densely placed objects (total count of objects is 10) and no animated objects. The robot starts from one side of the room to reach the another.

Test room name	Exe-cu-tions	Mean execu-tion time	Median execu-tion time	Mean utility	Median utility	Max utility	Min utility
Small_1	4	16.0224	16.755	-4.5	23.5	109	-174
Small_2	5	1.1359	1.24003	-87.6	-92.0	-8	-184
Small_3	5	26.999	27.3056	-215.0	-198.0	-61	-477
Total	14	14.6261	16.7549	-109.4	-105.5	109	-477

Table 5. Execution summary of small and dense rooms

The results indicate that the robot went close to the walls or even failed to perform some tasks. Indeed, the signal logs show (Table 6) that the robot moved close to objects often and some tasks could not be completed. Also, long execution time may indicate, that the robot was usually blocked (**solution_7_0**). This did not last for an infinite time, as the robot controller is implemented in a way to cancel a task if the robot is only oscillating back and forth. The navigation parameters may need to be adjusted.

	SMALL_1	SMALL_2	SMALL_3
SIG_GEOM_FAR_TRANSIT_CLOSE	231	402	132
SIG_GEOM_CLOSE_TRANSIT_COLLISION	1	0	0
SIG_MISSION_TASK_SUCCESS	5	5	8
SIG_MISSION_TASK_FAIL	3	5	1
SIG_MISSION_TASK_START	8	10	9

Table 6. Events captured during monitoring of small rooms

7.4.5. Medium rooms

The following scenario executes a mission by the robot in a medium-sized room (Figure 35). The objects are sparsely placed, and there is an additional moving human. The missions are composed of the same tasks as previously. An example of such room is depicted on Figure 35.

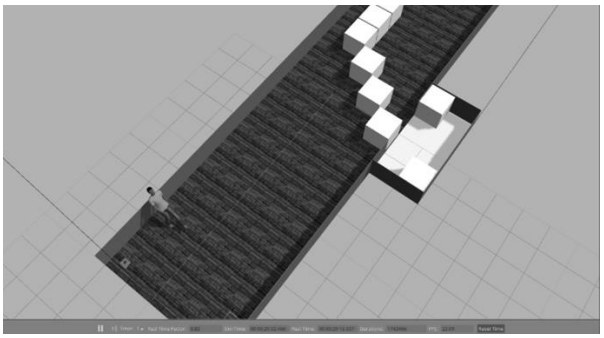

Figure 35. Example of a medium room

The summary of the execution (Table 7) indicates overall long execution time with some rooms performing fairly good compared to small room scenarios.

Test room name	Executions	Mean execution time	Median execution time	Mean utility	Median utility	Max utility	Min utility
MEDIUM_1	2	42.762	42.762	-40	-40	-40	-40
MEDIUM_2	2	51.663	51.663	-267	-267	-202	-332
MEDIUM_3	4	47.792	48.3	-16	-37.5	51	-40
Total	8	47.502	48.3	-84.75	-40	51	-332

Table 7. Summary of mission executions of medium rooms

In some scenarios, the robot was still blocked or wrongly believed that it was blocked (**solution_5_9**). The test results indicate that the navigation parameters should be further adjusted. The captured events indicate (Table 8) that on collision, the robot did not proceed to execute the final goal of the mission. This issue might be addressed by implementing new controllers.

	MEDIUM_1	MEDIUM_2	MEDIUM_3
SIG_GEOM_FAR_TRANSIT_CLOSE	175	0	46
SIG_GEOM_CLOSE_TRANSIT_COLLISION	0	0	9
SIG_MISSION_TASK_SUCCESS	2	2	7
SIG_MISSION_TASK_FAIL	2	2	1
SIG_MISSION_TASK_START	4	4	8

Table 8. Events captured during monitoring of medium-sized rooms

7.4.6. Large rooms

One test scenario was executed on a fairly large room with default robot settings. The test was unsuccessful. The robot could not find a path to the selected goal. The problem could be resolved by changing navigation parameters. This way the testing indicated that further improvement (configuration) of the tested robot is necessary.

7.4.7. Summary

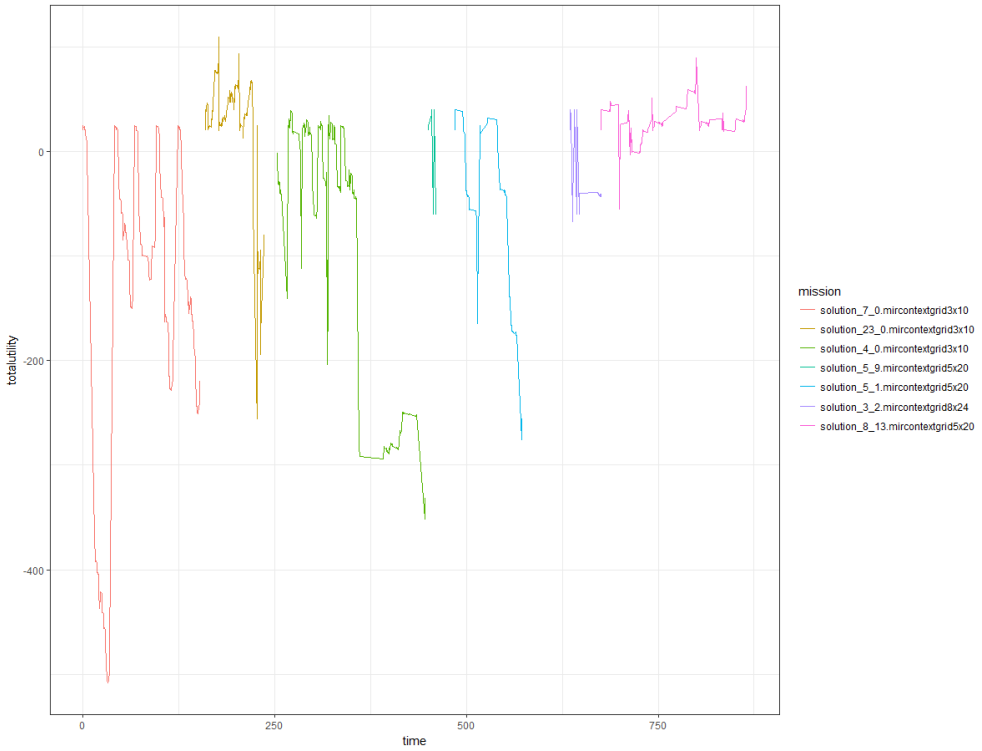

Figure 36. Total utility value change as missions execute

The following subsection summarizes the results of each test case. The change of utility for each mission (Figure 36) indicates and the final utility of each mission was usually negative – as expected. The missions described ran for 779 seconds.

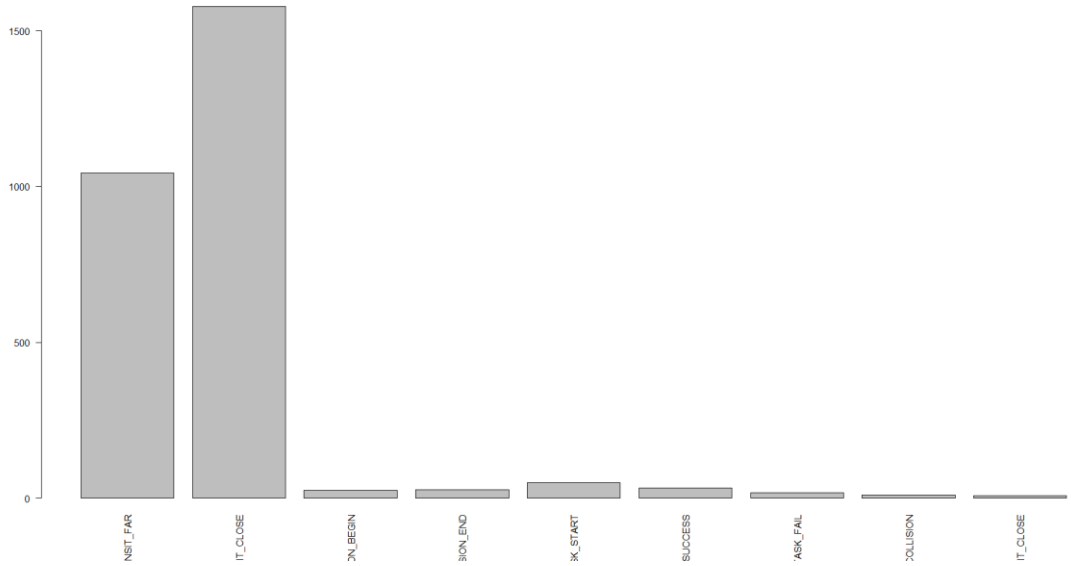

Figure 37. Distribution of signals

The distribution of signals shows (Figure 37), that the robot moved close to objects in many cases. Collisions occurred less frequently. There were actually more cases of far to close transitions than otherwise. This should be closely equal. The temporal pattern implementation shall be reviewed to address this issue.

7.5. Testing error prone proximity sensor

The following scenario is based on injecting error periodically to the laser proximity sensor of the robot. The setup is familiar to the previous scenarios, an arbitrary room is selected and the corresponding mission is executed. In this case, a small room can be used as the purpose of this scenario is to verify that the monitor is able to detect the events.

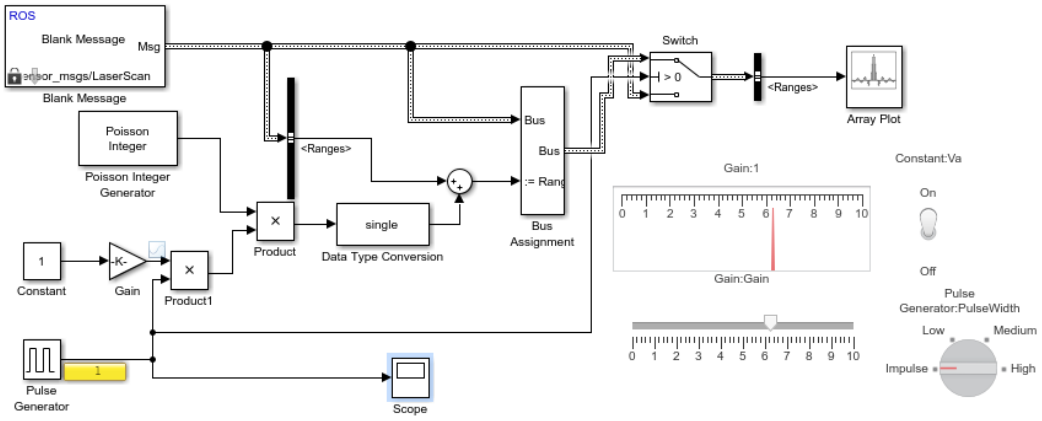

Figure 38. Error injecting logic

For injecting error, the sensor output must be overridden by an external ROS node. The external error injecting node can be created fast and simple by a Simulink module (Figure

38). This module uses a Poisson distribution to model the error occurring in laser sensors (Figure 39).

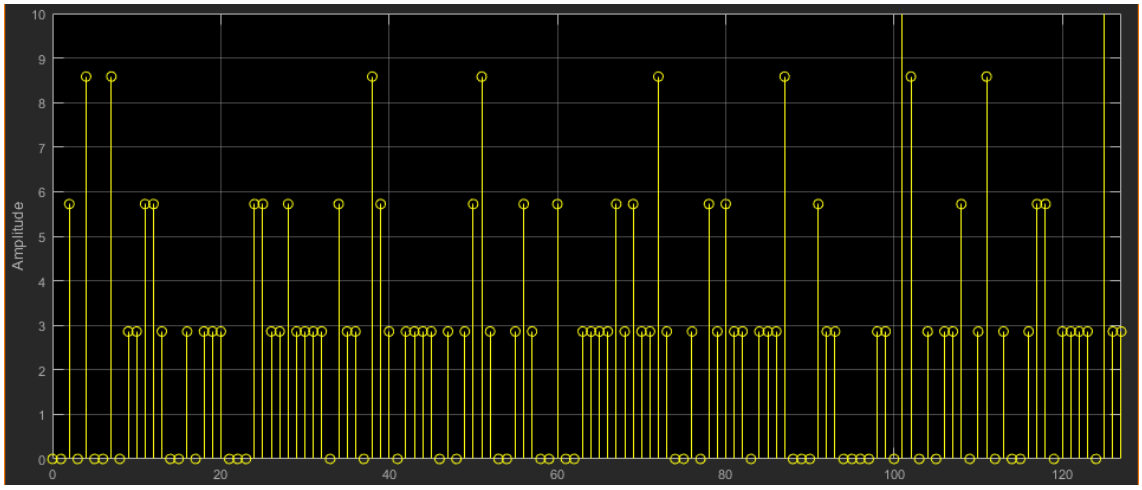

Figure 39. Snapshot of output noise produced by the error module

The noise is additive and added to the previous read of the laser scanner. The gain of this error can be adjusted during mission execution. The error injection is controlled by an impulse which may have variable pulse width.

7.5.1. Low error

The following scenarios were executed with a 1% of pulse width . The map is constructed close to a laser scanner with no error at all. On the other hand, the localization encountered problems derived from estimation error and sometimes failed to localize the robot correctly. Even the final state had a ∼10% error. The failures can be contributed to the local map construction (Figure 40) which might be resolved with adjusting the planner parameters or revisiting the controller implementation and evasion strategies.

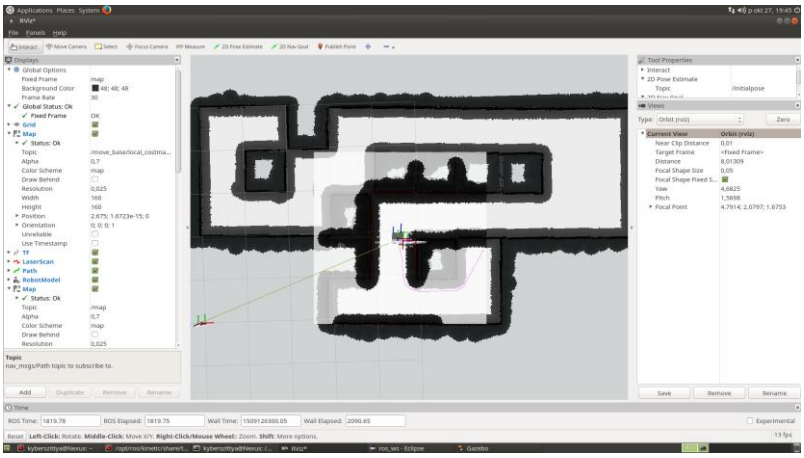

Figure 40. Map constructed with low sensor error rate

Investigating the intercepted signals (Table 9), no collision was detected. Indeed, the robot did manage to avoid collision with static objects of the simulation.

Signal id	Count
SIG_GEOM_CLOSE_TRANSIT_FAR	251
SIG_GEOM_FAR_TRANSIT_CLOSE	391
SIG_MISSION_BEGIN	4
SIG_MISSION_TASK_FAIL	2
SIG_MISSION_TASK_SUCCESS	6
SIG_MISSION_END	4

Table 9. Signal captured during monitoring

7.5.2. Effect of high error

The following scenarios were investigated with high error injected (10%). This would mean, that the proximity sensor is malfunctioning, but the error is injected more frequently at the same rate.

One significant anomaly is, that the robot fails to construct a precise map. On the other hand, SLAM is a relatively reliable algorithm of high tolerance. The robot manages to find its goal and avoids collision.

On the other hand, with error of large amplitude, the robot fails to achieve its goal and constructs a wrong map (Figure 41) with falsely presumed blockades.

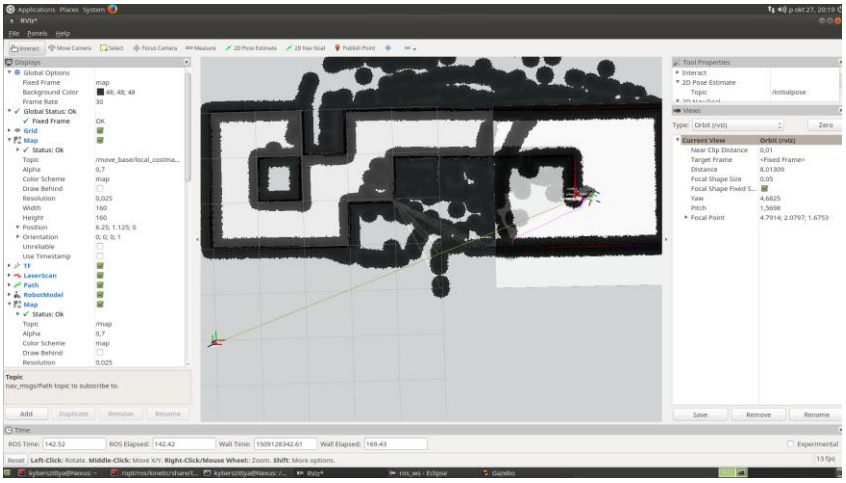

Figure 41. Effect of high amplitude error on SLAM algorithm

7.5.3. Low error in dynamic environment

The following scenario is done with low sensor error (~1%) in a dynamic environment. Otherwise than the localization error previously seen in the static environment, the SLAM process resulted in some visible curvature in the map (Figure 42). This is significant,

considering, that the environment is built from rectangle blocks. Also, the robot mistakenly places some objects on its constructed map, the whole map seems to be a little distorted. The robot managed to avoid the static building blocks, but did collided with the person few times.

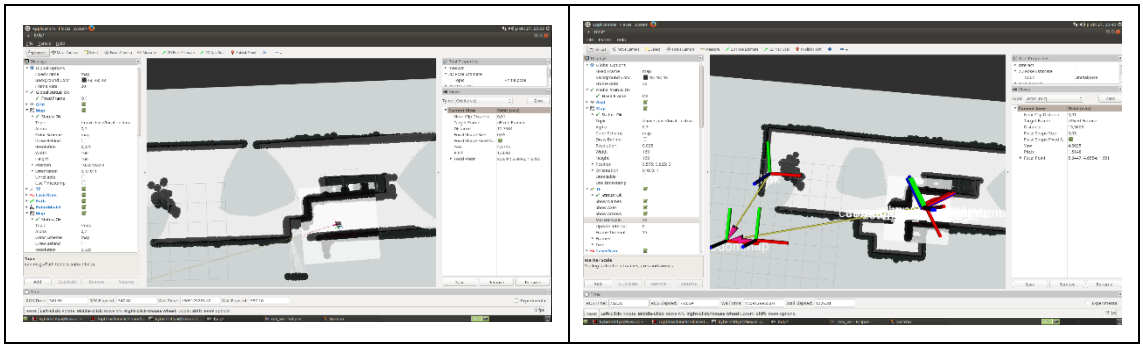

Figure 42. SLAM map reconstruction during mission execution

The event summary is listed in Table 10.

Signal id	**Count**
SIG_GEOM_CLOSE_TRANSIT_FAR	20
SIG_GEOM_FAR_TRANSIT_CLOSE	38
SIG_MISSION_BEGIN	3
SIG_MISSION_END	3
SIG_MISSION_TASK_FAIL	1
SIG_MISSION_TASK_START	6
SIG_MISSION_TASK_SUCCESS	5
SIG_GEOM_CLOSE_TRANSIT_COLLISION	1

Table 10. Summary of events on a dynamic environment with low error rate

8. Summary

8.1. Brief summary

In my work I presented a model-based approach to test and monitor robotic systems. The characteristics of the framework can be summarized as follows:

- The framework supports the testing of robots implemented using a widely spread robotic middleware (ROS).
- The framework populates a test environment for the tested robot based on an abstract test case
- The framework feeds a running simulation with tasks from a mission based on an abstract test case.
- The framework also extends the existing simulator with the ability to move target objects, having this way a dynamic environment
- The monitor component intercepts data incoming from the simulation and matches events based on user-defined high-level patterns. The evaluator component logs the detected events and calculates utility values.
- In its current state, the monitor is able to check the simulation of differential drive robots.

8.2. Future work

Having the basic functionalities of the framework successfully completed, there is still possibility to extend it with additional functionalities.

The framework is not yet ready for humanoid or pedal robots in general. There might be use cases where specifically humanoid robots might be used (e.g. nursery) or quadrupedal robots (e.g. military or disaster recovery applications). The framework would be ready for testing way more complex robots with somewhat large amount of extensions. At least the distance calculation used by this framework must be improved to allow the monitoring of humanoid robots precisely.

The possible test environment can be also extended to arbitrary terrains not only interiors of buildings in general. Indeed, some complex robotic applications are set in natural environments. Examples include disaster recovery (test whether the robot is capable to navigate and recover in a highly dangerous situation), space missions (reconnaissance on the surface of Mars for example).

The framework can also be extended to support autonomous vehicle scenarios. The context model can be extended to a new model capable of the description of road networks.

The framework could be extended to test agents of more of an abstract kind. As the monitor not closely attached to the robotic system (uses a network protocol to access information), the simulation could be easily changed.

It could be also demonstrated, how this framework helps to design new robotic systems. This would mean that this monitoring framework could be part of a larger software system

of robot or mechatronic design. This way the framework could support the design of robotic construction or control and navigation algorithms.

The evaluation component can be further extended. The logged data and abstract test case could be used as training data for machine learning methods. Currently available machine learning methods would allow using graph-based or pattern-based approaches. The trained machine learning components could be used to generate new abstract test data or to predict the possible outcomes of test execution based on historical data and test case properties.

The framework could be also further developed to be ready for ROS 2, also with some new features of Gazebo 8. Gazebo 8 supports the feedback of contacts which may be used for more precise collision detection.

9. Appendix

9.1. Geometric format

9.1.1. Pose

The pose is generally composed of position and orientation. The definition of position is straightforward (in 3D, it is defined with three components x, y, z). The

9.1.2. Twist

The twist is generally composed of linear and angular velocity components.

9.1.3. Geometric state

The state provided by simulators (and therefore by Gazebo) is composed of the pose and twist.

9.2. Implementation details

9.2.1. VIATRA-CEP as standalone application

Problem arises that VIATRA-CEP is easily configurable for Eclipse OSGi applications, but standalone applications are not currently in extensive use. From an implementation viewpoint, the standalone configuration is not well documented. This framework resolves the problem by registering the required plugins and Eclipse providers at startup. The code below presents the code fragment required at wrapper class instantiation to use the VIATRA-CEP engine throughout the program.

```
EStructuralFeature.Internal.SettingDelegate.Factory.Registry.INSTANCE.put(
                                "org.eclipse.viatra.query.querybasedfeature",
                                new QueryBasedFeatureSettingDelegateFactory()
                        );
WellbehavingDerivedFeatureRegistry.registerWellbehavingDerivedPackage(
    AutomatonPackage.eINSTANCE);
SingletonQueryGroupProvider groupProvider;
groupProvider = new SingletonQueryGroupProvider(DerivedFeatures.instance());
QueryGroupProviderSourceConnector sourceConnector =
    new QueryGroupProviderSourceConnector("org.eclipse.viatra.cep.standalone.connector",
        groupProvider, true);
QuerySpecificationRegistry.getInstance().addSource(sourceConnector);
```

There is no intention to make the monitor component into an OSGi (more specifically, Equinox) application. The explanation for this is to allow the use of monitor in a more limited software environment.

9.2.2. VIATRA-CEP temporal patterns

In 5.1.6.2, the geometric states was defined for the robot, while 5.1.6.6 presented the idea of describing temporal patterns. 6.1.3.3 described the idea, how to implement each pattern. In the following section, the geometric state detection temporal patterns are listed and briefly described.

9.2.2.1. Preliminaries

Before each temporal pattern could be implemented, the building query must be defined as query events. These are using the queries previously defined to match geometric events. Also, it must be also defined when a match is not satisfied.

The query event definition of far from object state.

```
queryEvent atomicFarFromObject() as findFarObject found
queryEvent notAtomicFarFromObject() as findFarObject lost
```

The query event definition of close to object state.

```
queryEvent atomicCloseToObject() as findCloseToObject found
queryEvent notAtomicCloseToObject() as findCloseToObject lost
```

The query event definition of collision with object.

```
queryEvent atomicCollisionToObject() as findCollisionToObject found
queryEvent notAtomicCollisionToObject() as findCollisionToObject lost
```

9.2.2.2. Implementation of FAR->CLOSE pattern

This event is raised just as the robot was previously in a far state and closes to another object.

```
complexEvent transitionFarToClose(){
    as (notAtomicFarFromObject -> atomicCloseToObject){+}[1000]
    context immediate
}
```

9.2.2.3. Implementation of CLOSE->FAR pattern

This event is raised just as the object previously closed to an object, but then moves far.

```
complexEvent transitionCloseToFar(){
    as (notAtomicCloseToObject -> atomicFarFromObject){+}[1000]
    context immediate
}
```

9.2.2.4. Implementation of CLOSE->COLLISION pattern

This event is raised just as the object previously closed and the collides with an object.

```
complexEvent closeTransitCollision(){
    as(notAtomicCloseToObject -> atomicCollisionToObject){+}[1000]
    context immediate
}
```

9.2.2.5. Implementation of COLLISION->CLOSE pattern

This event is raised just as the object collided but start to move further away.

```
complexEvent collisionTransitClose(){
    as(notAtomicCollisionToObject -> atomicCloseToObject){+}[1000]
    context immediate
}
```

10. References

[1] NIST, "Reference Architecture fo Cyber-Physical Systems," [Online]. Available: https://www.nist.gov/programs-projects/reference-architecture-cyber-physical-systems.

[2] ARTEMIS, "Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems," 2017.

[3] S. Russell and P. Norvig, Artificial Intelligence, A Modern Approach, Prentice Hall, 2010.

[4] ANSI/RIA, *R15.06-2012*, 2012.

[5] EN, *MSZ EN ISO 10218-1:2011*, 2011.

[6] S. Roland and N. Illah R., Autonomous Mobile Robots, The MIT Press, 2004.

[7] ANSI, "ANSI/RIA R15.306:2016," 2016. [Online]. Available: https://webstore.ansi.org/RecordDetail.aspx?sku=RIA+TR+R15.306-2016.

[8] J. Hsu, "When It Comes to Safety, Autonomous Cars Are Still "Teen Drivers"," *Scentific American*, 2017.

[9] M. Zoltán, "Languages and frameworks for specifying test artifacts," 2013.

[10] IEEE, 24765 Systems and software engineering - Vocabulary, 2010.

[11] ISTQB, Glossary.

[12] D. Nelly, "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* , vol. 30, no. 12, p. 859, 2004.

[13] R. Wiki, "http://wiki.ros.org," OSRF. [Online].

[14] Orocos, "The Orocos Project," KU Leuven, [Online]. Available: www.orocos.org.

[15] RT-Middleware, "OpenRTM-aist," AIST, [Online]. Available: http://www.openrtm.org/openrtm/.

[16] OSRF, "ROS 2 design," [Online]. Available: http://design.ros2.org/.

[17] Gazebo, "Gazebo simulation," [Online]. Available: http://gazebosim.org/.

[18] OSRF, "Simulation Description Format," [Online]. Available: http://sdformat.org/.

[19] WAMP, "WAMP protocol," [Online]. Available: http://wamp-proto.org/.

[20] WAMP, "WAMP protocol comparison," [Online]. Available: http://wamp-proto.org/compared/.

[21] F. Huang, Web Technologies for the Internet of Things, University of Aalto, 2013.

[22] E. Foundation, "Eclipse Modeling Framework," [Online]. Available: https://www.eclipse.org/modeling/emf/.

[23] D. István, "VIATRA CEP," [Online]. Available: https://wiki.eclipse.org/VIATRA/CEP.

[24] MATLAB, "ROS support from Robotoics System Toolbox," [Online]. Available: https://www.mathworks.com/hardware-support/robot-operating-system.html.

[25] "Acceleo Eclipse plugin," [Online]. Available: http://www.eclipse.org/acceleo/.

[26] S.-K. László, A. György and C. Ferenc, Háromdimenziós grafika, animáció és játékfejlesztés, Computerbooks, 2003.

[27] NHTSA, "United States Department of Transportation," [Online]. Available: https://www.nhtsa.gov/technology-innovation/vehicle-cybersecurity.