ROS-Industrial quality-assured robot software components - ROSIN

Testing-Based Validation Infrastructure for ROS

D3.3

Grant Agreement No.	732287
Start date of Project	1 January 2017
Duration of the Project	48 months
Deliverable Number	D3.3
Deliverable Leader	ABB (task lead), ITU (WP lead)
Dissemination Level	PU
Status	1.0
Submission Date	31/12/2018
Author	Jon Tjerngren, ABB (jon.tjerngren@se.abb.com) Zhoulai Fu, ITU (zhfu@itu.dk) Andrzej Wąsowski, ITU (wasowski@itu.dk)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732287.

PU=Public, CO=Confidential, only for members of the consortium (including the Commission Services), CI=Classified, as referred to in Commission Decision 2001/844/EC.

Modification Control

Version #	Date	Author	Organisation
1.0	18-12-19	J. Tjerngren	ABB

Release Approval

Name	Role	Date
Carlos Hernandez Corbato	WP2 Leader/Coordinator	18-12-18
Andrzej Wąsowski	WP3 Leader	18-12-19
Alexander Ferrein	WP4 Leader	18-12-21
Thilo Zimmermann	WP5 Leader	19-01-07
Martijn Wisse	Coordinator	19-01-08

History of Changes

Section, page number	Change made
1.0	Version submitted to European Commission

Table of Contents

1. Introduction 4
1.1 Purpose 4
1.2 Executive Summary 4
2. Model-in-the-Loop Testing 5
2.1 ABB Robot Setups 6
2.1.1 RobotWare 6
2.1.2 RobotStudio 7
2.2 Model-in-the-Loop Testing Setup 8
2.2.1 Windows Environment 9
2.2.2 Ubuntu Environment 13
2.3 Model-in-the-Loop Testing Example 15
2.4 Conclusions and Further Work 17
3. Shape Testing 18
3.1 Introduction 18
3.2 Example 19
3.3 Algorithm, Implementation Plan and Limitations 22
3.3.1 Algorithm 22
3.3.2 Implementation Plan 23
3.3.3 Limitations 23
3.4 Preliminary Results 24
3.5 Related Work 25
3.6 Conclusion 27
4. Outlook 28
5. Appendices 29
5.1 Appendix A 29
5.2 Appendix B 31

1. Introduction

1.1 Purpose

The deliverable *D3.3 Testing-Based Validation Infrastructure for ROS* is the first version of the ROSIN report on advances on tasks *T3.5 Model-in-the-loop Testing and T3.4 Automated Unit Test Generation*. Two demonstrator videos are attached to this document, corresponding to the two main parts of the document. The first shows the Model-in-the-loop Testing implementation, the second is a capture of a public demonstration of the shape testing strategy at the ROS-Industrial conference in Stuttgart.

D3.3 is the first presentation of this work. It will be revised by M42 (*D3.4 Testing-based validation infrastructure for ROS, version 2).*

1.2 Executive Summary

In this deliverable we address two key levels at which robotics software needs to be tested:

- The functional level, which includes interaction with the physical hardware and environment. This level is addressed in the section devoted to Model-in-the-loop testing
- The platform level, which includes ensuring stability of the basic ROS platform, standard nodes, etc. This level is addressed in the section devoted to the shape testing.

We summarize the content of the two sections below:

Model-in-the-loop Testing. Contains a brief introduction to the Model-in-the-loop testing concept, which consists of both an automation aspect and a runtime aspect. A prototype package is described, which has been implemented to manage simulations during automated scenarios. And for the runtime aspect, three ease-of-use packages (not developed in the ROSIN project) for ABB robots has gone through ABB's internal open-source process (done in the ROSIN project) and been made openly available for free. Namely the RobotWare StateMachine Add-In[7], as well as the two C++ communication libraries abb_librws[12] and abb_libegm[13]. Finally, the current implementation of the Model-in-the-loop testing is described and exemplified by executing a robot trajectory. This is also shown in the Model-in-the-loop testing demonstrator video.

Shape Testing. In the past stage of the project, ITU has explored testing and code scanning methods for finding bugs in ROS code. As the result of the experience gathered, we proposed shape testing – a new strategy of testing ROS nodes. The strategy relies on using the so- called sanitizers to detect bugs in ROS code. Sanitizers are pre-existing code instrumentation methods provided by modern compilers. We generalize the main program of a ROS node (the launched code) to allow executing it with various parameter values. We use an off-the-shelf tool, a *fuzzer*, to generate the random executions. The executions with random parameters will crash (a bug report) when a sanitizer detects a violation. We report preliminary results, showing that this strategy can be effective in finding ROS bugs. The report includes links to some of the bugs that we have found and reported during the investigation phase.

2. Model-in-the-Loop Testing

Simulations are commonly used to verify that systems are behaving as expected, as well as decreasing the development time of applications. However, these simulations are not always set up to be automatically executed and as such they might only be run on an ad-hoc basis. Additionally, the simulation environments are sometimes hard-coded, which can lead to a decrease in flexibility when changing the setup of the systems being tested.

Model-in-the-loop (MIL) testing is an approach based on using system models to achieve more variable simulation setups, as well as also allowing for the testing to become automated. This leads to a more structured way of performing the simulations, which is intended to increase the quality of the testing. It is also important that the system models are accurate and that they have been validated against real systems.

Simulations are an important tool when developing industrial robotic applications, and MIL testing allows for both automation and virtualization of the testing in a continuous integration manner. Thus, it is an essential technology for reaching high quality applications.

The focus of task T3.5 has been to create a functional MIL testing setup targeting ABB robots.

RobotStudio[1] is ABB's simulation and offline programming software for ABB robots. It allows for simulations to be as close as possible to real ABB robot setups, and many of the different types of ABB robots are available for simulation.

RobotStudio has been chosen to be used in task T3.5 to show that it is possible to integrate such simulation software in a MIL testing setup, as well as benefiting from the perks of RobotStudio when working with ABB robots. However, there are several challenges to address before this is possible. For example, how to automate RobotStudio simulations as well as how to ease the runtime control of the robot during such simulations.

For ROS-Industrial, which is aiming to increase the use of ROS in industrial applications, it is a relevant and important proof of concept that demonstrates how proprietary information can be included in open-source environments. In general, this would be a good development for ROS that would allow for important industrial robot hardware, and software, to be more easily included into ROS and following standard processes.

[1] https://new.abb.com/products/robotics/robotstudio

2.1 ABB Robot Setups

A basic understanding of ABB robot setups is required to be able to implement a MIL testing setup for ABB robots, especially when including RobotStudio. This is because real ABB robot setups consists of several components and many of them also have a representation in virtual setups, simulated in RobotStudio. A real ABB robot setup basically consists of a robot controller, which is installed with a system that runs on a specific version of RobotWare[2] (see below). Additionally, the setup also contains one or more mechanical units.

2.1.1 RobotWare

RobotWare is the operating system for ABB robot controllers and it provides all the required features for fundamental robot operations and programming. From a user's perspective they are used when editing and executing robot programs, and for specifying system configurations. Robot programs are written in RAPID[3], which is ABB's native robot programming language, and they are stored as modules in the systems. There are many interfaces implemented in RobotWare and some of them can also be simulated, in virtual controllers, in RobotStudio. Two of them have been selected for enabling the runtime aspect of the MIL testing, and they are Robot Web Services[4] and Externally Guided Motion [689-1] (see [2]). The former is a free feature in RobotWare, while the latter requires the purchase of a license when using it with real robot controllers. Figure 1 shows a sketch of the different RobotWare components that has been utilized for the MIL testing.

RobotWare Add-Ins

A RobotWare Add-In[5] is an independent package that extends the functionality of a robot controller's system. It can for example consist of a collection of system configurations and RAPID modules that are loaded during system installation. Add-Ins provides a way to setup systems with a baseline of configurations and robot programs, and they can be used for both real and virtual systems. They can also greatly reduce the effort required when getting started with ABB robots.

Robot Web Services (RWS)

RWS (see [4]) provides access to many services and resources in ABB robot controllers, and it is based on HTTP and WebSocket communication. Controller systems runs a RWS server in the background, which an external source can access. The services and resources can be used for general interaction with controllers during runtime. For example, to load, start, stop and reset robot programs, check controller statues and reading and writing of IO-signals.

Externally Guided Motion [689-1] (EGM)

EGM (see [2]) can be used during runtime for responsive motion control. It works by streaming motion references, from an external source, to an ABB robot controller in a closed UDP communication loop. Virtual controllers include the same supervision of the EGM references as real controllers, which is useful for MIL testing. EGM is optional, and the applicability of EGM depends a lot on the use case in question, as well as that it is currently only available for six axes robots.

[2] Application Manual – Controller Software IRC5 (3HAC050798-001)
[3] Technical Reference Manual – RAPID Overview (3HAC050947-001)
[4] http://developercenter.robotstudio.com/webservice/api_reference
[5] Application Manual – RobotWare Add-Ins (3HAC051193-001)

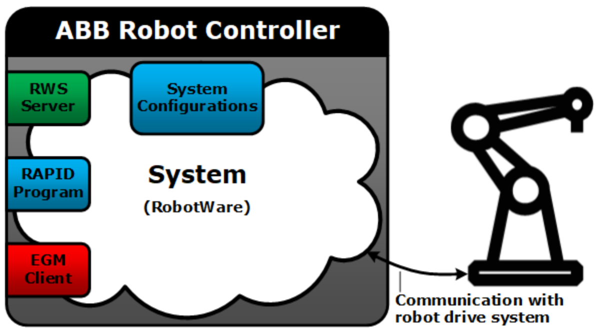


Figure 1: Sketch of RAPID program, system configurations and the RWS and EGM interfaces.

2.1.2 RobotStudio

RobotStudio is, as mentioned previously, ABB's simulation and offline programming software for ABB robots. Simulations in RobotStudio are essentially managed via a representation of the world called a station. A station consists of the environment that, for example, is made up of mechanical units (e.g. robots) and both static (e.g. tables) and dynamic (e.g. workpieces) objects. Stations also contains of one or more virtual robot controllers, which represents real robot controllers. Each virtual controller has a virtual system that simulates parts of the RobotWare operating system as well as the mechanical units. Mechanical units are associated with one of the virtual controllers. The interaction with virtual system are identical to the interaction with real systems in real robot controllers. This increases the likelihood of seamless transfers from successful simulations to real applications. This is the major reason for utilizing RobotStudio in the MIL testing for ABB robots.

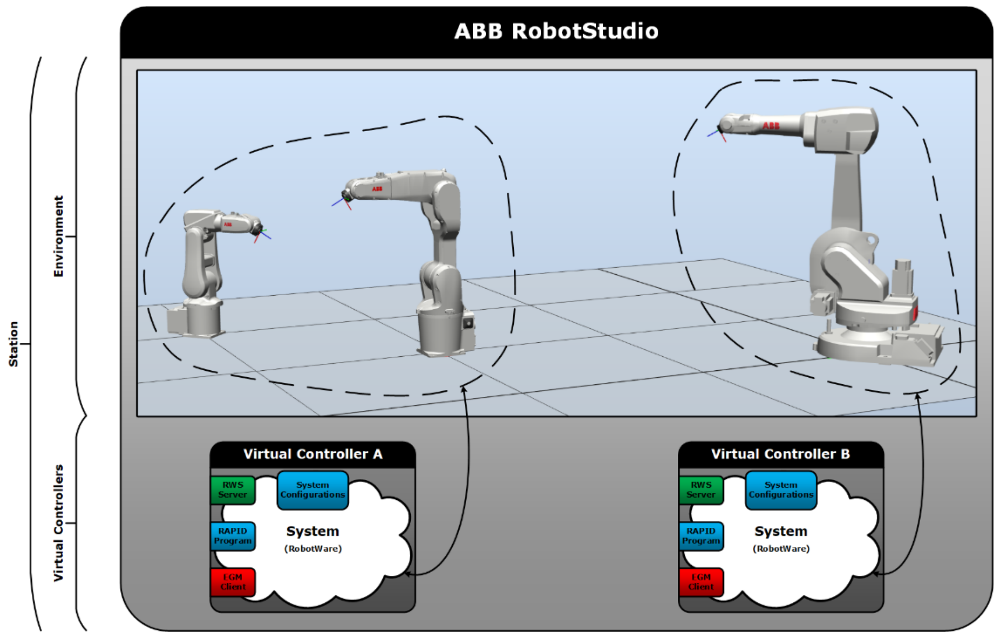


Figure 2: Sketch of a station in RobotStudio, with the environment and two virtual controllers.

RobotStudio Add-Ins

A RobotStudio Add-In is an independent package that extends the functionality of the RobotStudio software. Add-Ins are implemented in C#, and they can be used to realize a wide range of extra features. For example, setting up and managing a background server that provides the means to automatically executing RobotStudio simulations, as well as switching between different virtual controllers and robot types.

2.2 Model-in-the-Loop Testing Setup

The approach that has been chosen to implement the MIL testing involves at least two different environments. This is due to that RobotStudio is only available in Windows, while ROS is primarily used in Ubuntu. Currently the physical setup involves two separate computers, one running RobotStudio in Windows and the other running ROS in Ubuntu. It would also be possible to run ROS in Windows by, for example, using a virtual machine. Figure 3 shows a sketch of how the Windows and Ubuntu environments are connected.

The intention is to use the Windows machine as a server, which provides access to RobotStudio simulations via a RobotStudio Add-In. A prototype of such an Add-In (see the MIL Manager Add-In below) has been implemented and it facilitates the automation aspect of the MIL testing, and it is in the Windows section of the figure. Additionally, a limited number of RobotStudio stations has been created to exemplify how different robot models and virtual controllers can be handled. Also, the systems, in the virtual controllers, are based on a RobotWare Add-In (see the StateMachine Add-In below) that simplifies and enables the usage of the RWS and (optionally) EGM interfaces during ongoing simulations. This facilitates the runtime aspect of the MIL testing, and it is also in the Windows section of the figure.

The intention with the Ubuntu machine, is to include it into a continuous integration setup so that it can receive robot specific simulation jobs, automatically execute them and finally report back the simulation results. Components are then required to enable the interaction with both the automation and the runtime aspects of the RobotStudio simulations. For example, one that can send requests to the RobotStudio Add-In to start and stop simulations, and this would be the manager in the Ubuntu section of the figure. Also, the simulation jobs can consist of one or more components that utilizes the runtime interfaces to control the robot during ongoing simulations. This would be the active job in the Ubuntu section of the figure.

This approach allows access to ABB proprietary robot models, which are used internally inside RobotWare, in the virtual controllers.

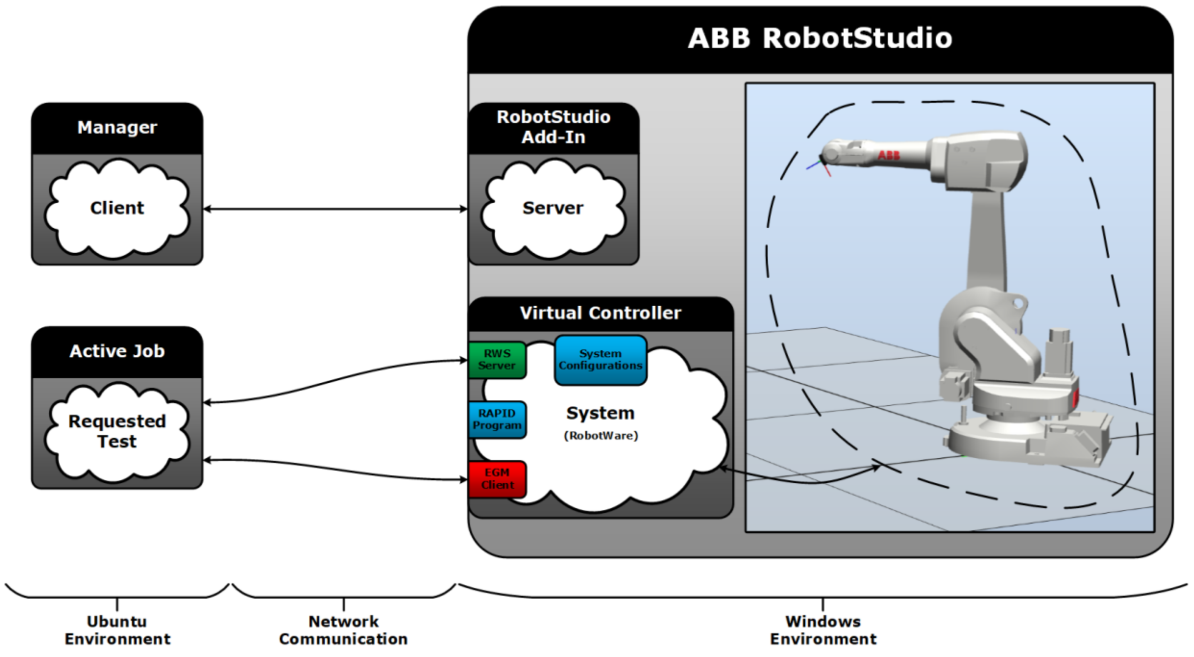


Figure 3: Sketch of a MIL testing setup.

2.2.1 Windows Environment

The Windows environment primarily requires RobotStudio to be installed. Then there is also a need for a RobotStudio Add-In, a RobotWare Add-In, and a couple of RobotStudio stations. Each station requires at least one virtual controller with a virtual system. These things together can enable a MIL testing setup, which in the end can potentially be transferred to a real robot controller and real use cases.

RobotStudio MIL Manager Add-In

A prototype RobotStudio Add-In has been implemented in C# to enable a higher-level control of RobotStudio simulations to enable automation of the MIL testing. The Add-In is currently called MIL Manager Add-In, and it provides several services via a TCP server. A user interface is also included that allows for manually sending requests to the server. The services that are provided can be split into the following services for stations, and services for simulations:

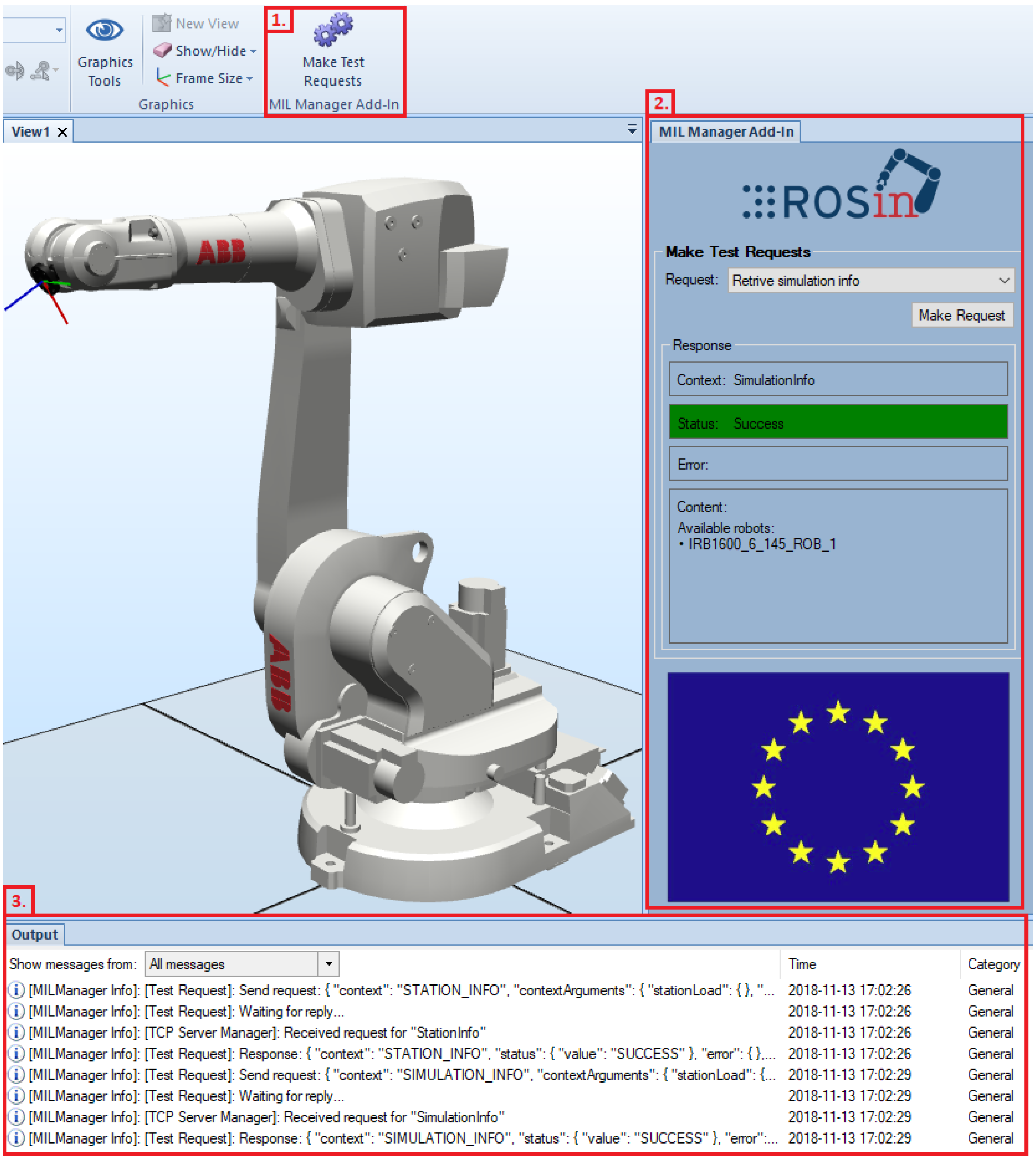
- Stations:
    - Retrieve information about available stations.
    - Load a specific station (e.g. to change robot model).
    - Close the current station.
- Simulations:
    - Retrieve information about available robots.
    - Start a simulation, with option to set robot start state.
    - Stop a simulation.
    - Wait for a simulation to finish.
    - Collect simulation results.

The communication messages are specified with Google's Protocol Buffers[6] format. See Appendix A for the message definitions. An external TCP client can use the messages to access the services provided by the Add-In's TCP server.

The MIL Manager Add-In's TCP server is started in the background of the RobotStudio application as soon as the Add-In is loaded into RobotStudio. Depending on the received request, then the Add-In will either process it in the background or forward it to the main RobotStudio thread. For example, requests that affect the simulation needs to be handled by the main thread, while a request to retrieve information about available stations can be processed in the background.

Figure 4 shows the MIL Manager Add-In's user interface in RobotStudio and it is mainly used for testing the Add-In. In the figure, then the area marked by 1. is a button that opens the testing interface marked by area 2. The area marked by 3. shows log messages indicating what is going on in the Add-In.

[6] https://developers.google.com/protocol-buffers/



Make Test Requests

Request: Retrive simulation info

Make Request

Response

Context: SimulationInfo

Status: Success

Error:

Content:
Available robots:
• IRB1600_6_145_ROB_1

Show messages from: All messages

```
                                                                                                              Time                 Category
[MILManager Info]: [Test Request]: Send request: { "context": "STATION_INFO", "contextArguments": { "stationLoad": { }, "...   2018-11-13 17:02:26  General
[MILManager Info]: [Test Request]: Waiting for reply...                                                                        2018-11-13 17:02:26  General
[MILManager Info]: [TCP Server Manager]: Received request for "StationInfo"                                                    2018-11-13 17:02:26  General
[MILManager Info]: [Test Request]: Response: { "context": "STATION_INFO", "status": { "value": "SUCCESS" }, "error": { },...    2018-11-13 17:02:26  General
[MILManager Info]: [Test Request]: Send request: { "context": "SIMULATION_INFO", "contextArguments": { "stationLoad": {...     2018-11-13 17:02:29  General
[MILManager Info]: [Test Request]: Waiting for reply...                                                                        2018-11-13 17:02:29  General
[MILManager Info]: [TCP Server Manager]: Received request for "SimulationInfo"                                                 2018-11-13 17:02:29  General
[MILManager Info]: [Test Request]: Response: { "context": "SIMULATION_INFO", "status": { "value": "SUCCESS" }, "error":...     2018-11-13 17:02:29  General
```

Figure 4: The MIL Manager Add-In's user interface.

RobotWare StateMachine Add-In

A RobotWare Add-In, the StateMachine Add-In[7], has been developed by ABB, during another European project called SYMBIO-TIC[8]. It has been designed for use with both real robot controllers and virtual controllers. The Add-In's purpose is to ease the use of ABB robots during both initial set up, as well as during runtime. The intention is to use it in combination with external, to the robot controller, systems. For example, a MIL testing setup can be viewed as such an external system.

The Add-In eases the use of ABB robots by auto-loading several RAPID modules and system configurations, which makes up a ready to run RAPID program. If the Add-In is selected during system installation, then the Add-In inspects the system specifications (e.g. single/multiple robots and if the EGM RobotWare option is present or not), and finally it loads corresponding RAPID modules and system configurations. The RAPID program has been designed to be interacted with via the RWS and (optionally) EGM interfaces.

The StateMachine Add-In has been made freely available online (see [7]) during the ROSIN project, and this has included going through ABB's internal process for open-sourcing software. All the included RAPID modules and system configurations are possible to extract from the Add-In.

When the state machine is started, then it waits until a request is received to change state. This is done by triggering any of the IO-signals defined in the system configurations. In Figure 5, the dashed area in the left side is affected during system installation while the right side illustrates the state machine's different states and transitions.

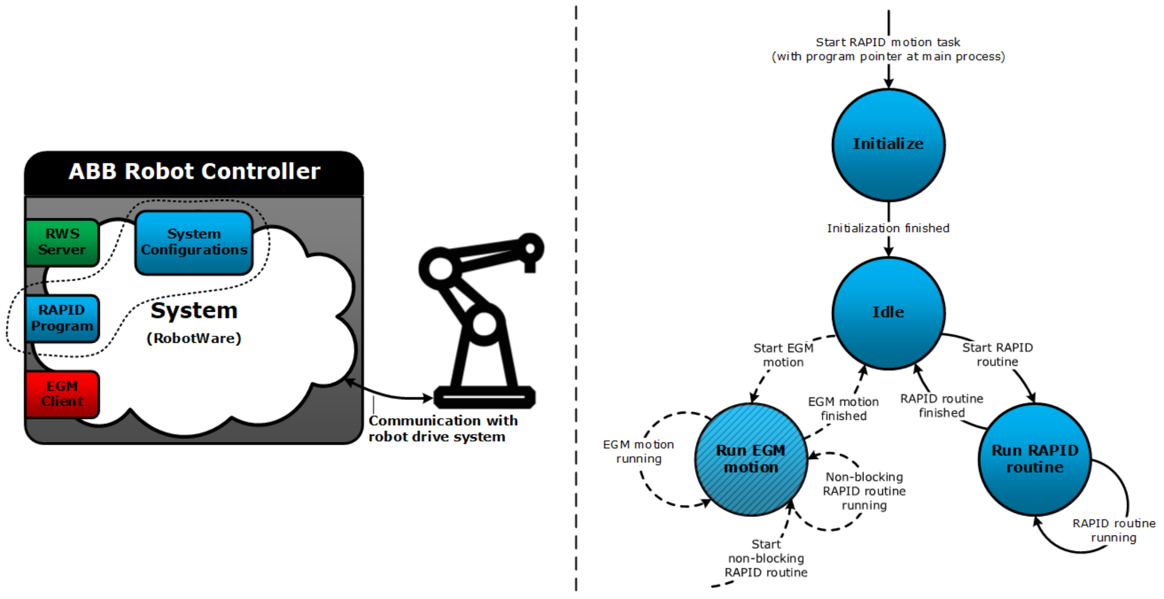


Figure 5: Sketch of the StateMachine Add-In. Note: The EGM section is only included if the EGM option exist in the system.

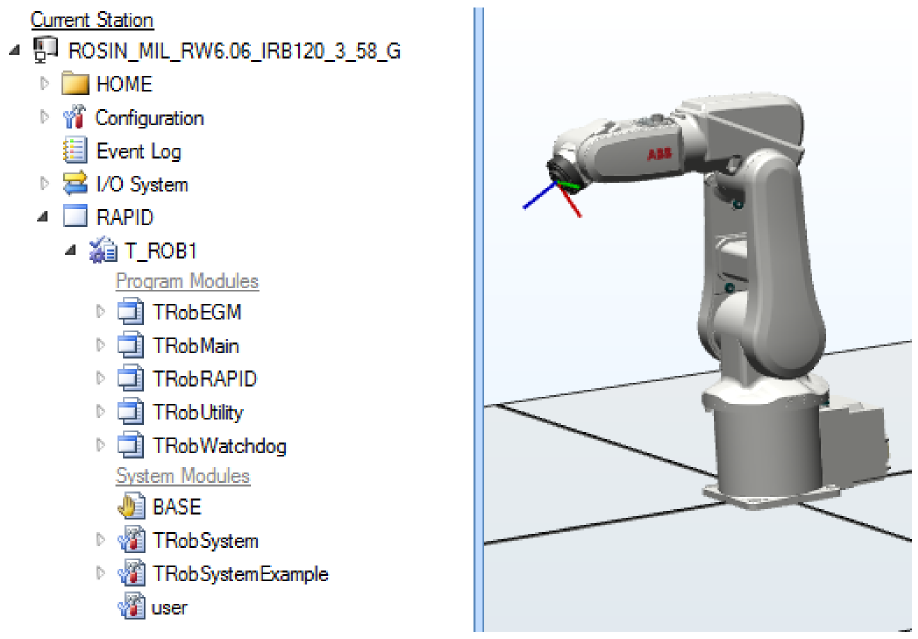
See the online documentation[9] for more details.

All in all, this is in line with the approach chosen for the MIL testing setup and as such it has been used to generalize the RobotStudio stations used for exemplifying the MIL testing.

[7] https://robotapps.robotstudio.com/#/viewApp/7fa7065f-457f-47ce-98d7-c04882e703ee
[8] http://www.symbio-tic.eu/
[9] https://robotapps.blob.core.windows.net/appreferences/docs/2093c0e8-d469-4188-bdd2-ca42e27cba5cUserManual.pdf

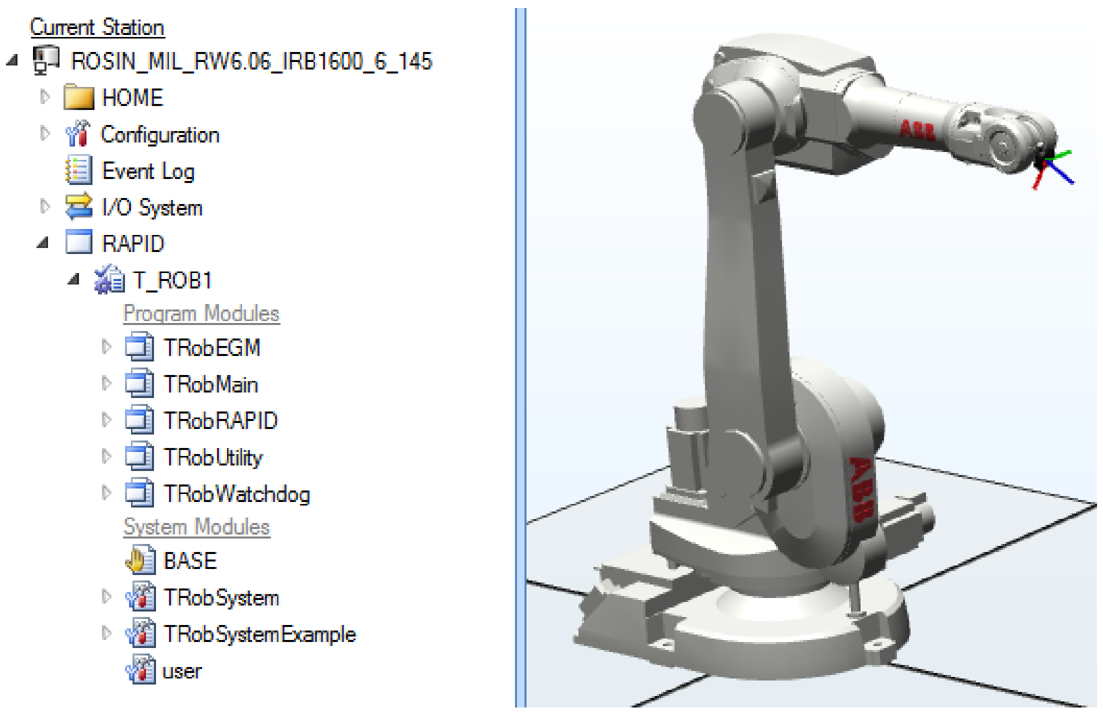
RobotStudio Stations

Two RobotStudio stations has been created for exemplifying the MIL testing simulations. One station with an IRB120[10] robot, and one with an IRB1600[11] robot. Figure 6 and Figure 7 shows the RobotStudio stations, along with the modules loaded by the RobotWare StateMachine Add-In.



Current Station
- ROSIN_MIL_RW6.06_IRB120_3_58_G
  - HOME
  - Configuration
  - Event Log
  - I/O System
  - RAPID
    - T_ROB1
      - Program Modules
        - TRobEGM
        - TRobMain
        - TRobRAPID
        - TRobUtility
        - TRobWatchdog
      - System Modules
        - BASE
        - TRobSystem
        - TRobSystemExample
        - user

Figure 6: The RobotStudio station for an IRB120 robot.



Current Station
- ROSIN_MIL_RW6.06_IRB1600_6_145
  - HOME
  - Configuration
  - Event Log
  - I/O System
  - RAPID
    - T_ROB1
      - Program Modules
        - TRobEGM
        - TRobMain
        - TRobRAPID
        - TRobUtility
        - TRobWatchdog
      - System Modules
        - BASE
        - TRobSystem
        - TRobSystemExample
        - user

Figure 7: The RobotStudio station for an IRB1600 robot.

[10] https://new.abb.com/products/robotics/industrial-robots/irb-120
[11] https://new.abb.com/products/robotics/industrial-robots/irb-1600

2.2.2 Ubuntu Environment

Applications (e.g. ROS nodes) running in the Ubuntu environment needs to be able to interact with the RobotStudio software running in the Windows environment. This is spit into controlling RobotStudio itself (for automation purposes), as well as interacting with a robot during an ongoing simulation. The first aspect is possible via communication with the RobotStudio MIL Manager Add-In, while the second is enabled via the RWS and (optionally) EGM interfaces.

RWS Communication Library

A C++ communication library called abb_librws[12] has been developed by ABB in the SYMBIO-TIC European project. This library, similarly to the StateMachine Add-In, has been made open-source during the ROSIN project and is now openly available on GitHub for use with ABB robot controllers supporting RWS.

The main intention with this library is to simplify setting up, and managing, RWS communication channels from an external computer to an ABB robot controller. The library contains C++ classes that create clients, which can connect to RWS servers, interpret the RWS message protocol, as well as exposing APIs to a user. The library currently provides methods for:

- Reading/writing of IO-signals.
- Reading/writing of RAPID data.
- Reading of RAPID data properties.
- Starting/stopping/resetting the RAPID program.
- Subscriptions (i.e. receiving notifications when resources are updated).
- Uploading/downloading/removing files.
- Checking controller state (e.g. motors on/off, auto/manual mode and RAPID execution running/stopped).
- Reading the joint/Cartesian values of a mechanical unit.
- Registering as a local/remote user (e.g. for interaction during manual mode).
- Turning the motors on/off.
- Reading of current RobotWare version and available tasks in the robot system.

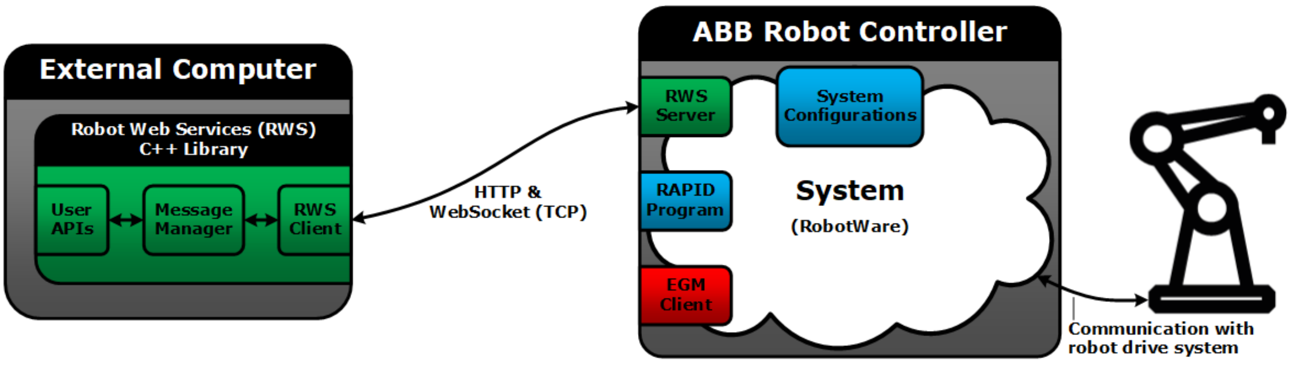


Figure 8: Sketch of the relationship between abb_librws and an ABB robot controller.

[12] https://github.com/ros-industrial/abb_librws

EGM Communication Library

Another C++ communication library called abb_libegm[13] has also been developed by ABB in the SYMBIO-TIC European project. This library has also been made open-source during the ROSIN project and is now openly available on GitHub for use with ABB robot controllers supporting EGM.

The main intention with this library is to simplify setting up, and managing, EGM communication channels from an external computer to an ABB robot controller. The library contains C++ classes that create servers, which can be connected to by EGM clients (in the robot controller), interpret the EGM message protocol, as well as exposing APIs to a user.

If a RAPID program starts an EGM communication session, then the robot controller's EGM client sends out a request for new motion references. When this request is received by the EGM server, then it will reply with new motion references telling the robot where to move to. The actual references depend on which C++ class is used. For example, there is one class for processing a queue of trajectories, and then the references are calculated by interpolation between the points in each trajectory.

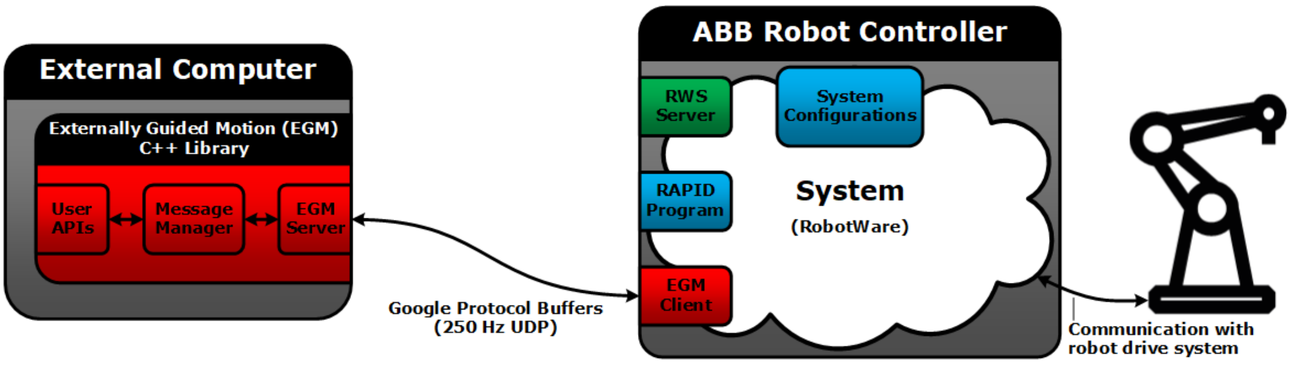


Figure 9: Sketch of the relationship between abb_libegm and an ABB robot controller.

ROS Setup

Currently, the Ubuntu environment consists of two mock-up ROS nodes for testing the MIL testing setup. Both nodes are related to the runtime aspect of the MIL testing. One of the nodes utilizes the abb_librws library for all the general interaction with the simulation, like setting IO-signals (via RWS) to command a running RAPID program to start and stop EGM motions. The other node utilizes the abb_libegm library for controlling the robot during simulation, when an EGM communication session has started.

These nodes are only used for trying out the MIL testing setup. See Appendix B for the complete source code of the nodes.

So far, the primary focus has the creation of the communication infrastructure that facilitates the MIL testing setup. However, one piece is still missing, which is a component that can automatically control the RobotStudio simulations, via the previously mentioned RobotStudio MIL Manager Add-In. The intention is to continue working with that component in the next phase of the ROSIN project. Currently, only the Add-In's user interface has been used to manually load RobotStudio stations, starting and stopping the actual simulations, and collecting simulation results.

[13] https://github.com/ros-industrial/abb_libegm

2.3 Model-in-the-Loop Testing Example

The following figures will illustrate how a manually triggered MIL testing scenario can look like. Figure 10 shows the initial state of the IRB1600 RobotStudio station, after it has been loaded via the RobotStudio MIL Manager Add-In. The robot will be moved by an online generated motion trajectory, from the EGM mock-up ROS node.

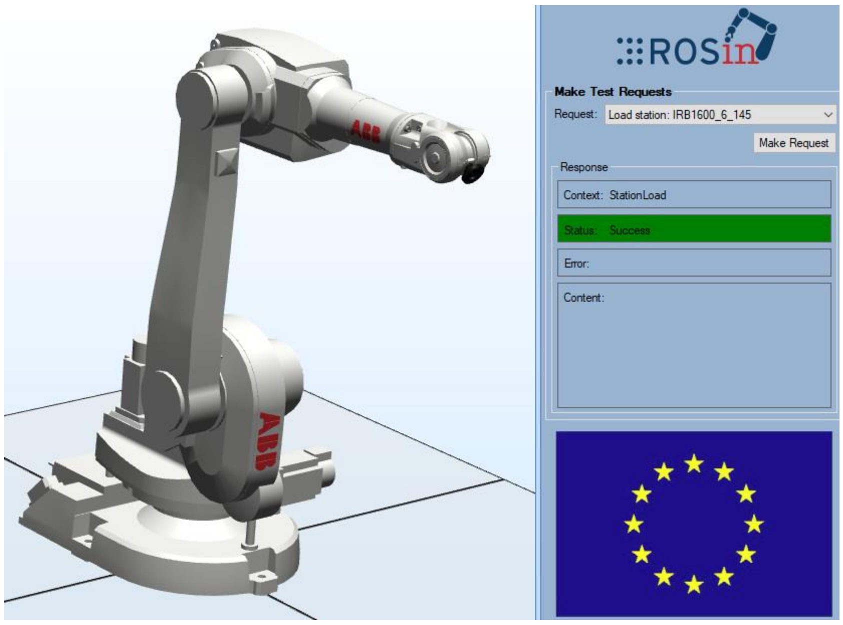
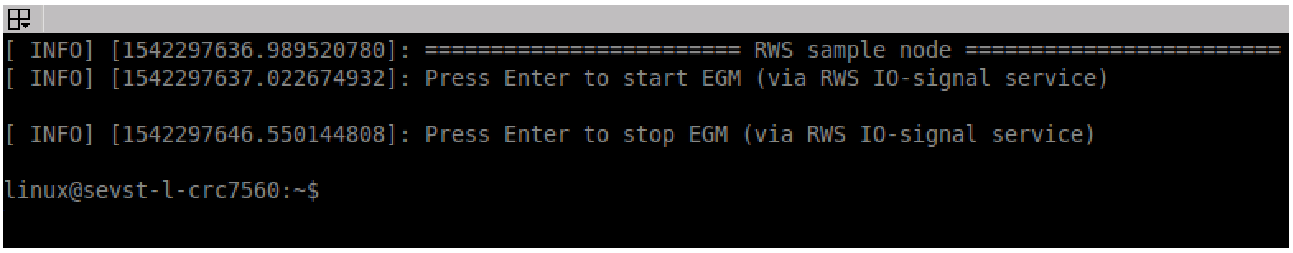


Figure 10: Initial state of the IRB1600 RobotStudio station.

After the station has been loaded, then a request is sent to start a simulation. The two ROS mock-up nodes are started in the Ubuntu environment, and Figure 11 and Figure 12 shows the execution progress (run in parallel). The EGM sample node automatically waits for an EGM communication session starts, and then it will send motion references until the EGM session is stopped. The RWS sample node waits for user input before sending an EGM start request via RWS to the robot controller, and then it waits again for user input to send an EGM stop request.

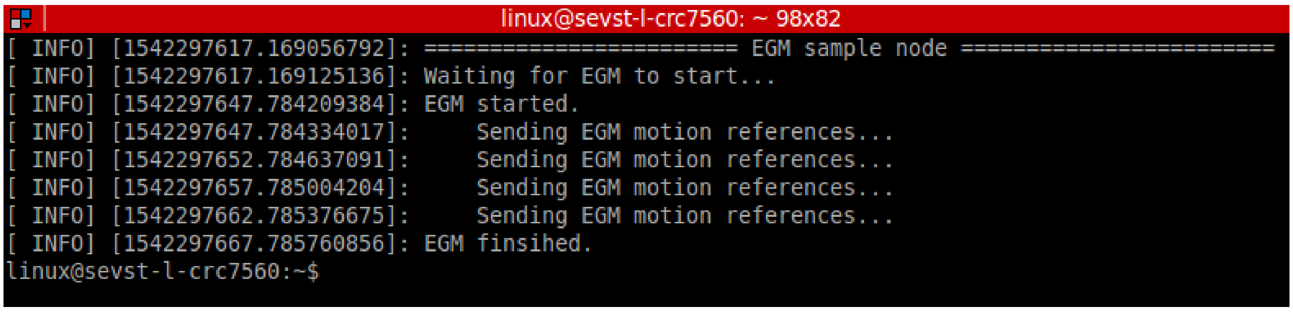


```
[ INFO] [1542297636.989520780]: ======================== RWS sample node ========================
[ INFO] [1542297637.022674932]: Press Enter to start EGM (via RWS IO-signal service)

[ INFO] [1542297646.550144808]: Press Enter to stop EGM (via RWS IO-signal service)

linux@sevst-l-crc7560:~$
```

Figure 11: Execution flow of a RWS sample ROS node.



```
linux@sevst-l-crc7560: ~ 98x82
[ INFO] [1542297617.169056792]: ======================== EGM sample node ========================
[ INFO] [1542297617.169125136]: Waiting for EGM to start...
[ INFO] [1542297647.784209384]: EGM started.
[ INFO] [1542297647.784334017]:     Sending EGM motion references...
[ INFO] [1542297652.784637091]:     Sending EGM motion references...
[ INFO] [1542297657.785004204]:     Sending EGM motion references...
[ INFO] [1542297662.785376675]:     Sending EGM motion references...
[ INFO] [1542297667.785760856]: EGM finsihed.
linux@sevst-l-crc7560:~$
```

Figure 12: Execution flow of an EGM sample ROS node.

Figure 13 shows the robot's state after the simulation has finished. The blue line is a trace of the robot's tool centre point during the execution of the trajectory.

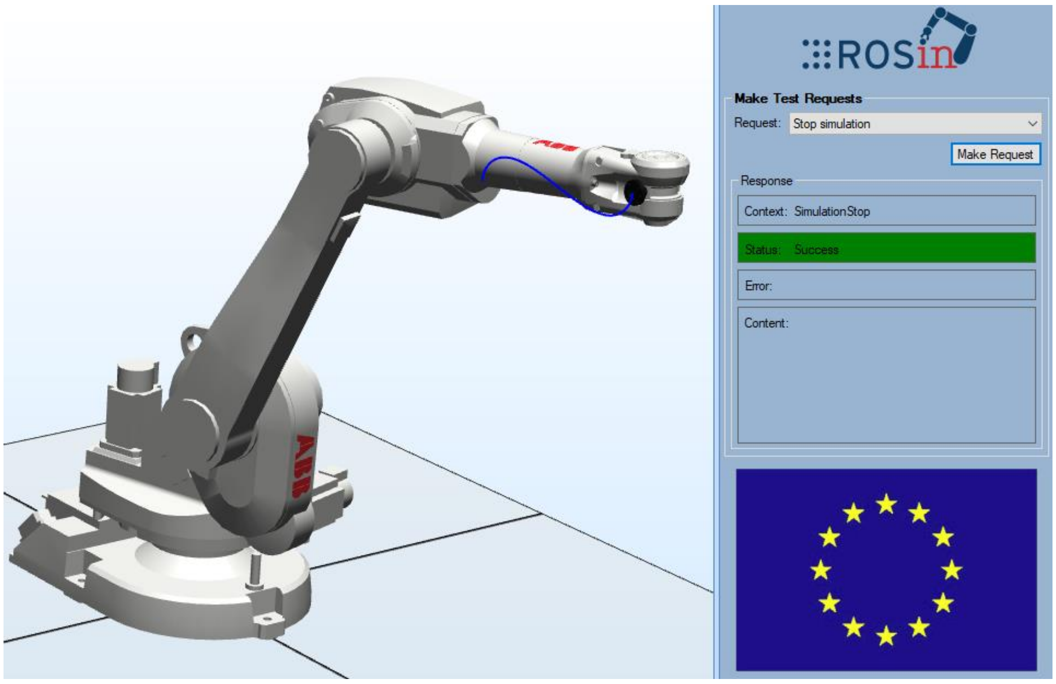
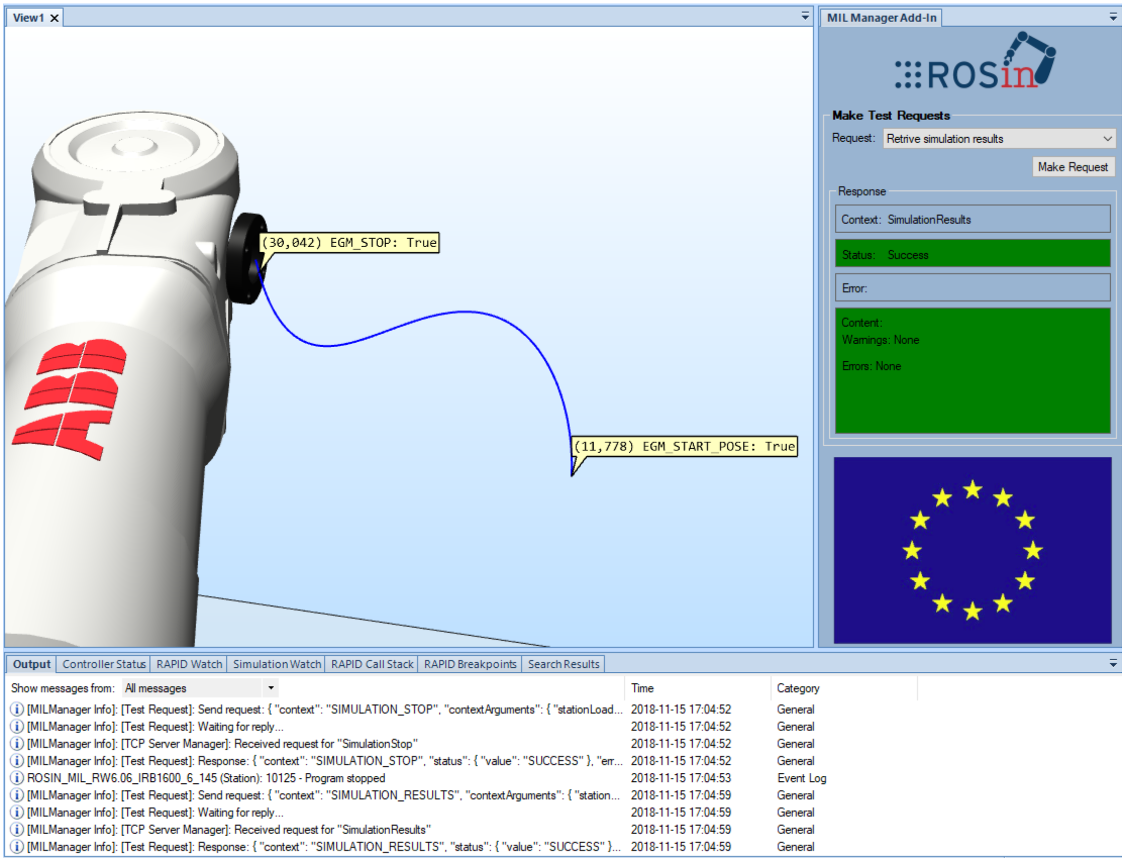


Figure 13: Robot state after the simulation has finished.

Figure 14 shows two event tags where, during the simulation, the RWS sample node's requests were received. The figure also shows the simulation results, retrieved via the MIL Manager Add-In.



```
                                                                                              Time                 Category
[MILManager Info]: [Test Request]: Send request: { "context": "SIMULATION_STOP", "contextArguments": { "stationLoad...   2018-11-15 17:04:52  General
[MILManager Info]: [Test Request]: Waiting for reply...                                                                2018-11-15 17:04:52  General
[MILManager Info]: [TCP Server Manager]: Received request for "SimulationStop"                                         2018-11-15 17:04:52  General
[MILManager Info]: [Test Request]: Response: { "context": "SIMULATION_STOP", "status": { "value": "SUCCESS" }, "err...  2018-11-15 17:04:52  General
ROSIN_MIL_RW6.06_IRB1600_6_145 (Station): 10125 - Program stopped                                                      2018-11-15 17:04:53  Event Log
[MILManager Info]: [Test Request]: Send request: { "context": "SIMULATION_RESULTS", "contextArguments": { "station...   2018-11-15 17:04:59  General
[MILManager Info]: [Test Request]: Waiting for reply...                                                                2018-11-15 17:04:59  General
[MILManager Info]: [TCP Server Manager]: Received request for "SimulationResults"                                      2018-11-15 17:04:59  General
[MILManager Info]: [Test Request]: Response: { "context": "SIMULATION_RESULTS", "status": { "value": "SUCCESS" }...     2018-11-15 17:04:59  General
```

Figure 14: Simulation events and simulation results.

2.4 Conclusions and Further Work

The main conclusion in this task is that it is possible to run a functional MIL testing setup in the approach chosen here. However, most of the work has so far been related to the communication infrastructure and the Windows environment, and therefore the Ubuntu environment requires additional effort. Especially for making the MIL testing process automated and connecting it to the continuous integration services from Task 3.2. All the tests have currently only been performed in an isolated setup with two computers, and the intention is to continue with the implementation in the next phase of the ROSIN project.

The inclusion of the RobotStudio software, and the usage of interfaces existing in real robot controllers means that, in the end, it is straightforward to move from successful simulations to a real physical setup. This is an important aspect, since this can allow for a large reduction in the time required to program a real physical robot station. Additionally, three ease-of-use packages has been made openly available for free. Namely the RobotWare StateMachine Add-In, which has been released in ABB’s distribution service for such Add-Ins, as well as the two C++ communication libraries abb_librws and abb_libegm that has been released on ROS-Industrial GitHub repositories. This makes it possible for anyone to implement similar setups. The RobotWare Add-In and the C++ libraries needs to be continuously updated, both to improve them as well as making sure they continue to work with new RobotWare versions.

Another conclusion is that this approach can be a feasible solution for accessing proprietary robot models, although it is up to a company to provide this kind of setup. This would, most likely, require further investigation and evaluation from a business perspective, which is not included in the scope of this task.

3. Shape Testing

3.1 Introduction

We now turn our attention from testing high-level robot requirements with models and simulators, to broadly understood quality of the ROS platform and the ROS ecosystem. We want to help the ROS community to detect and control errors and bugs in the platform code much more efficiently than it is done today. This includes hunting and eliminating low level errors, like exceptions being thrown, null pointers being dereferenced, or memory buffers being overrun. Unfortunately, the term “Quality Assurance” (QA) tends to be synonymous with “modern software bureaucracy” in some ROS developers’ minds. They view QA as a tedious procedure for improving code quality toward an abstract goal of “software reliability”. A recent study[14] shows that the ROS community is reluctant to implement QA procedures unless they are automated, e.g., through testing platforms such as GoogleTest, Unittest, Travis CI, or Jenkins.

These automated platforms rely on existence of a test harness (harness for short). Figure 15 shows an example harness for the geometry2 package, which implements the basic vector-based view of a 3-dimensional universe in ROS. The coordinate stored in the variable `v_simple` following a spatial transformation (Lines 2–3) is expected to get close to the location (−9, 18, 27) within a given tolerance EPS (Lines 4–6). The harness is executed, usually more than once, during software evolution and regression tests.

```
TEST(TfBullet, Transform) {

  tf2::Stamped<btTransform> v1(btTransform(btQuaternion(1,0,0,0), btVector3(1,2,3)),
  ros::Time(2.0), "A");
  btTransform v_simple = tf_buffer->transform(v1, "B", ros::Duration(2.0));
  EXPECT_NEAR(v_simple.getOrigin().getX(),  -9,  EPS);
  EXPECT_NEAR(v_simple.getOrigin().getY(),  18, EPS);
  EXPECT_NEAR(v_simple.getOrigin().getZ(),  27, EPS);
  ...
}
```

Figure 15: A Test harness in the geometry2 package, taken from test_tf2_bullet.cpp.

Unfortunately, the process of writing the test harness is very expensive. The entire geometry2 package includes only 20 test harnesses in total (in the melodic-devel branch). Since a single harness covers at most a small portion of run-time scenario, the testing platforms provide only very limited coverage: no matter how frequently they execute the test harness, it always exercises the same executions. Therefore, these testing platforms are ineffective in terms of reliability: A ROS package is reliable if most, if not all, run-time scenarios have been tested.

Recently, we have been studying a new methodology for testing ROS packages. During the research, we have found several bugs in ROS packages, including one in roscpp_core (from the

[14] Adam Alami, Yvonne Dittrich, and Andrzej Wasowski. 2018. Influencers of quality assurance in an open source community. In *Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*.
61–68. https://doi.org/10.1145/3195836.3195853

Kinetic distribution). Our approach does not need a test harness. Instead, it leverages any piece of code that interacts with the ROS package, which we call a Main program. Acquiring a Main is a more natural process than writing harness — all packages users can contribute to a Main, but few people who are both acquainted with the code API and expected calculation results, can write the test harness. Our approach automatically mutates the Main program, via the fuzzing techniques, to cover a substantial portion of run-time scenarios. We call this approach *shape testing*.

The vision of this project is two-fold. First, our scientific objective is to advance the state-of-the-art QA procedures for ROS by developing shape testing as an efficient, fully automated testing platform that complements existing ones. Second, our ultimate goal to integrate shape testing into the existing ROS ecosystem. This document presents the approach, reports our progress and preliminary results.

3.2 Example

Suppose that we want to test the ROS "helloworld" package turtlesim. One may well start by looking for a test harness, but it does not exist in the repository (as of today, Dec. 14, 2018). Instead, the package comes with three tutorial programs that interact with the package. For example, the program draw_square.cpp in the package illustrates how one can move the turtle drawing a square. We call the program Main. Unlike a test harness, a Main program does not have assertions. So, if we run it for testing, bugs can go undetected for there is it no crash. For example, as explained in our pull request https://github.com/ros/ros_tutorials/pull/46, the simulated turtle draws 3/4 of a circle and then spins forever, which is unexpected. The turtle should draw a full square, even though this is not explicitly specified. The specification is hidden, or at least not reflected in the code. This problem—to determine whether a test has failed or passed—is known as an *oracle problem*[15] in software engineering research.

Oracle Problem.

Oracle is a mechanism to detect whether a program or a test executes as expected with respect to some property. If a correctness specification exists, it can be used as an oracle (for example a post-condition property). However, formal specifications rarely exist for real software. When no specifications are given, program crashes have frequently been used as an oracle for finding bugs. A ROS component, on the other hand, can silently produce wrong results without crashing. Thus, when testing ROS, program crashes cannot be used as a simple, readily available oracle.

Sanitizers can be used to mitigate the oracle problem. A sanitizer is a compiler feature that injects run-time checks, or assertions, to capture certain known buggy conditions. For example, the *Undefined Behavior Sanitizer* (UBsan) injects assert (z != 0) before a = y / z; and the *Address Sanitizer* (ASan) checks almost every memory access with extra code added. While sanitizers do not provide functional specification as harness does, they guard the code against local, sometimes security-related conditions, such as use-after-free, array-index-out-of-bound, or race conditions. Sanitizers are free, mature tools allowing us to immediately enhance code quality with negligible efforts, just by adding a compiler option we get a simple and effective oracle for free.

[15] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2015), 507–525.

Now, assume that we have applied a sanitizer on all programs in the turtlesim package. If we attempt to run draw_square.cpp to trigger crashes, we still face the problem that running the Main covers only small portion of run-time scenarios (the same problem the test harness faces). The challenge, then, is to leverage Main in some way to generate more coverage.

Coverage Problem.

Effective testing usually requires high coverage of run-time scenarios. Testing with few Main or harness covers a small portion of potentials run-time scenarios, thereby missing bugs that larger-scale testing would detect.

One way to improve coverage is to generate a large amount of *different* executions from draw_square.cpp. Consider the forward function in draw_square.cpp, listed in Figure 16 (a). Before reaching a user-defined goal, the simulated turtle should move with a linear speed 1.0 and angular speed 0.0 (Line 3). If we replace the two constants 1.0 and 0.0 by another pair of randomly chosen floating-point numbers (x, y), we obtain a different version of draw_square.cpp. In this way, we can have as many versions as we need, say, draw_square_1.cpp, ... , and draw_square_N.cpp, corresponding to randomly chosen (x_i , y_i) ($1 \leq i \leq N$).

```
void forward(ros::Publisher twist_pub) {
if (hasReachedGoal()) { ... }
else commandTurtle(twist_pub, 1.0, 0.0);

}
...
int main(int argc, char** argv){
...
}
```

(a)

```
double __g1, __g2;
void forward(ros::Publisher twist_pub) {
  if (hasReachedGoal()) { ... }
  else  commandTurtle(twist_pub, __g1, __g2);
}
...
int original_main(int argc, char** argv){
...
}

int main(int argc, char** argv) {
  if (scanf("%lf,%lf", &__g1,&__g2) != 2) return -
1;
  return original_main(argc, argv);
}
```

(b)

Figure 16: (a) Code segments from draw_square.cpp. (b) A Shape program.

Unfortunately, testing draw_square_i incurs a significant complexity overhead. Compiling 100 000 versions with 0.1 second for each would take 10000 seconds, or approximately three hours. And all this while different draw_square_i can still trigger the same program paths, reducing the benefits of having many versions of the Main program. For example, if we use (1.1, 0) and (1.2, 0) to define draw_square_1 and draw_square_2, the simulated turtles move in the same trajectory, producing the same program path. Select test data to avoid such situations is known as test *data generation problem*[16].

Problem of Test Data Generation.

Each input data (x_i, y_i) corresponds to a Main program draw_square_i.cpp. Some of them the programs can have overlap in terms of coverage. We need to generate (x_i, y_i) "smartly", so to trigger as many run-time scenarios as possible.

[16] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on software engineering* 16, 8 (1990), 870–879.

To address this problem, we introduce two global variables __g1 and __g2 to parameterize the linear and angular speeds. Then, we construct a new main procedure that takes from stdin the values __g1 and __g2 and invokes the original main. We call the derived program a *Shape* (Figure 16 (b)). Generally speaking, shape testing can parameterize any types of data, *e.g.*, strings, function or variable identities. At this stage, we focus on floating-point numbers. The process of shaping not only provides an advantage in saving compilation time, but it also unleashes the power of *fuzzing*, techniques that have found many bugs in open-source and commercial projects, *e.g.*, the Heartbleed vulnerability. Figure 17 illustrates the general structure of fuzzing. Fuzzing techniques need a *fuzz target*, a piece of code that that passes random inputs to the program under test. For example, below is the interface of the fuzz target of LLVM's LibFuzzer.

```
int FuzzTarget(const uint8_t *Data, size_t Size)
{ Do_Something_With_the_program_under_test(Data, Size); return 0;  }
```

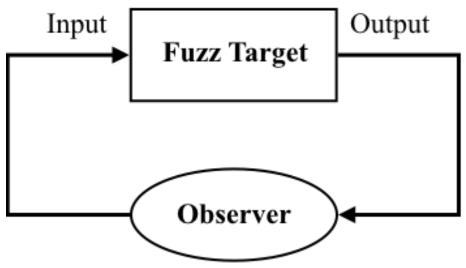


Figure 17: Illustration of fuzzing techniques.

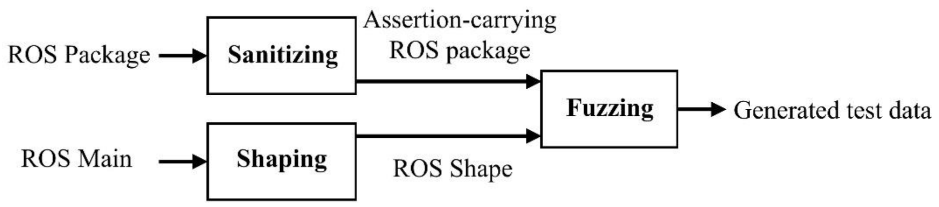


Figure 18: Illustration of Shape Testing.

The Shape program listed in Figure 16 (b) is, by design, a fuzz target that most off-the-shelf fuzzers can use. In short, fuzzing improves the randomly generated inputs via observing the output. Such output can be returned values, status, or more commonly, code coverage. For example, if modifying the input data in a certain way (*e.g.*, by flipping its last bit) triggers additional code coverage, that change can be applied in later iterations.

3.3 Algorithm, Implementation Plan and Limitations

Having conducted several preliminary experiments with a simple prototype, we now have a relatively clear idea about how we should implement the Shape Testing paradigm.

3.3.1 Algorithm

Following our example with turtlesim, we summarize the algorithm with the three steps below. The inputs of Shape Testing are a ROS package R and a Main program M that provides an entry-point for interacting with R.

(1) (Sanitizing.) Inject assertions in R using existing sanitizers. Since sanitizers are compiler options, the output of sanitizing, $R_{asserts}$, is usually in the form of a binary.

$$R_{asserts} \leftarrow \text{sanitize}(R)$$

Sanitizers that can be applied in this step include *Address Sanitizers* (ASan), *Undefined Behavior Sanitizers* (UBSan), *Thread Sanitizers* (TSan), or *Memory Sanitizers* (MSan). A complete list of sanitizers can be found at https://clang.llvm.org/docs/UsersManual.html#controlling-code-generation

(2) (Shaping.) Derive a Shape from M with lightweight program transformation. The shaping process can be done either at the intermediate-representation level (with Clang for example) or source level (with TXL[17]). In the sample case where shaping only concerns numeric constants, then global variables of the associated types will be introduced to parameterize the constants.

$$M_{shape} \leftarrow \text{shape}(M)$$

(3) (Fuzzing.) Fuzz M_{shape} using an existing fuzzer. Linking R_{assert} with M_{shape} gives us a fuzz target. The output of fuzzing is a sequence of generated test data. The simplest fuzzer is pure random data generation. More sophisticated fuzzers are commonly based on program coverage, *e.g.*, AFL and Libfuzzer.

$$x_1, \ldots, x_N \leftarrow \text{fuzz}(M_{shape}, R_{asserts})$$

We illustrate the algorithm in Figure 18. The statement below provides a theoretical basis for arguing the soundness of the algorithm above.

Lemma 3.3.1. *If the sanitizing procedure does not introduce wrong assertions, each crash in* $R_{asserts}$ *corresponds to a bug in the ROS package* R.

[17] James R Cordy. 2006. The TXL source transformation language. *Science of Computer Programming* 61, 3 (2006), 190–210.

3.3.2 Implementation Plan

As a proof-of-concept, we will first implement the algorithm above into a research prototype. By reusing existing sanitizers and fuzzing techniques, we only need to realize the shaping part, for which we will use Clang's LibTooling (https://clang.llvm.org/docs/LibTooling.html) or write an LLVM IR pass (https://llvm.org/docs/LangRef.html). As the next step, we will parameterize all numerical constants. Then we will write a script to chain sanitizing, shaping, and fuzzing together. As a baseline comparison, we will implement TopicFuzz, a new fuzzer for feeding random inputs to ROS subscriber nodes at run-time. TopicFuzz should be relatively easy to implement, as data passed to the nodes can generally reduce to numerical types.

We will evaluate shape testing incrementally. Below, we denote an instance of Shape Testing system by a pair (*s*, *f*) where *s* is a sanitizer, and *f* is a fuzzer. As an example, we can start with (none, random). Then we move on to (UBSan/ASan, Random), and (UBSan/ASan, AFL). For each step, we will evaluate the system on a small number of packages and also compare the results with our baseline algorithm TopicFuzz (see above). After analyzing the results, we will batch process with additional packages. We will consider integration once evaluation results become convincing. We will write a catkin_make wrapper (for ROS1) for setting the compiler options for sanitizers and fuzzers. Having such a wrapper is essential because we strive to minimize the efforts required by the ROS users.

3.3.3 Limitations

1. At the core of Shape Testing is the premise that a Main program (that runs the tested ROS package) is available. If such a program is of poor quality in the sense that mutating it does not allow for reasonable program coverage, then shaping testing will be ineffective.
2. Shape testing checks only assertions injected from the sanitizers. Such assertions are usually low-level and do not cover functional properties that are typically specified in the test harness. Thus, if a robotic arm moves from point A to C instead of the expected A to B, shape testing is helpless if such wrong move does not trigger low-level faults captured by sanitizers. In this sense, shape testing is a complementary QA measure to MIL testing (See Section 2) which focuses on high-level functional problems. The advantage of shape testing is that it is entirely automatic and comes almost at no cost for the developer. The advantage of MIL testing is that the tester has fine grained control over exercising behavior and over specifying safe behavior. This control however comes at quite some work cost. Shape testing and MIL-testing are likely to be effective in detecting failures caused by different faults. Shape testing focuses on bugs (programmer omissions). MIL testing focuses on design problem (misunderstanding of the problem or the requirements).
3. The process of shaping can have many variations. The problem becomes tricky if we would like to fuzz not only constants but also function or variable identities. (4) We have implicitly assumed that the ROS package is in C/C++ family languages. Sanitizers are available in these languages but how to apply sanitizers to other languages such as python remains unclear to us. E.g., the use-after-free error that ASan detects does not exist in python programs.

3.4 Preliminary Results

In our preliminary experiment, we have prototyped Shape Testing in different settings. We have found bugs in three packages, turtlesim, roscpp_core, and ros_comm. As usual, we reported the bugs as issues or pull requests at GitHub. Table 1 lists a selection of the bugs and the corresponding settings of shape testing. All experiments are performed on a laptop with a 2.6 GHz Intel Core i7 running an Ubuntu 16.04 virtual machine with 4GB RAM.

(Col. 3) We observed the first two bugs as unexpected trajectories. In the first one, the turtle drew 3/4 of a square instead of a full square; in the second one, the turtle should not move toward the -pi/4 direction when its angular speed is 0. The rest of the bugs manifested themselves through the "assertion" that sanitizers injected.

(Col. 4–6) We denote each instance of Shape Testing by a tuple (*s*, *sh*, *f*), where *s* is the sanitizer, *sh* is the shaping process, and *f* is the fuzzer. We already detected the first bug by merely running draw_square.cpp for several seconds.

Table 1: A selection of bugs found during our preliminary study, up to 14 Dec, 2018.

Package	Bug report	Manifestation	Shape Testing Components		
	Bug		Sanitizing	Shaping	Fuzzing
turtlesim	https://github.com/ros/ros_tutorials/pull/46	3/4 square	-	-	-
turtlesim	https://github.com/ros/ros_tutorials/issues/47	wrong direction	-	shape1	random
turtlesim	https://github.com/ros/ros_tutorials/issues/48	assertion	TSan	-	-
roscpp_core	https://github.com/ros/roscpp_core/issues/96	assertion	Asan	-	-

We discovered the second bug with the setting (-, shape1, random). Here, shape1 refers to the shaping process that parameterizes only __g1, namely the linear speed, and "random" refers to pure random data generation. We found that when __g1 is very large, the turtle moved to a wrong direction (as explained above). The other two bugs are detected by using only the sanitizers.

It is worth noting that the way we detected the first two bugs cannot be automated. Thus, we believe that sanitizers will play an essential role in further development of shape testing. As a sub-project, we have collaborated with an expert in sanitizers to test the capability of using sanitizers as the only tool for detecting bugs in ROS packages. We have found many bugs in this way. Below are three bugs in the ros_comm package.

- https://github.com/ros/ros_comm/issues/1530. Detected by ASan
- https://github.com/ros/ros_comm/pull/1546. Detected by UBSan
- https://github.com/ros/ros_comm/pull/1547. Detected by UbSan

As a larger experiment, we have evaluated (-, shape2, AFL) for stress-testing the turtlesim package, using draw_square.cpp as the Main program. The screen-shot in Figure 19 summarizes our results after stress testing turtlesim for more than ten days. The fuzzing process finds 171 paths, including a hang (namely, the program runs out the predefined time-bound). The number of paths corresponds roughly to the number of distinct run-time scenarios triggered. Figure 20 shows various statistics of our finding, which we obtain from the afl-plot tool (http://manpages.ubuntu.com/manpages/xenial/man1/afl-plot.1.html).

3.5 Related Work

Testing is about producing failures[18]. In 1989, Miller *et al.* found that pure random testing was able to 1/4 of their tested Unix Applications [19]. Fuzzing is improved random testing, which incorporate the feedback for generating relevant inputs[20].

Two Java testing platforms are related to Shape Testing. (1) Parameterized test [21] (PT) extends unit testing by introducing parameters. Given a Java method, PT expects users to manually write its unit tests as usual, in addition, parameterize them with a set of data. In this way, a parameterized unit program combined with N data inputs becomes N unit test programs. (2) Randoop[22] is a kind of feedback-guided random testing for Java and .Net. It randomly selects Java methods or constructors. For the input parameters of the chosen method/constructor, Randoop randomly generates its inputs, which also takes into account of result from running previously created data.

Recently, a ROS package hypothesis-ros has been proposed. Its high-level concepts are similar to QuickCheck[23]: Users need to write program properties that are then checked by various strategies of random testing. The package can be useful once the developers have written down the properties. However, the reliance on such properties has the similar drawback as the need of writing harness as they both do not scale up and therefore providing small run-time coverage. As ROS developers are generally reluctant to apply quality assurance procedures unless they are automated[14], the property-based testing probably will encounter difficulty when it comes to tool integration.? Compared with Shape testing, hypothesis-ros does not leverage the powers of sanitizers or fuzzing. Its use of fuzzing remains at the random testing level. As a side note, the original developer of hypothesis-ros has dropped the project; we do not know about the current status of the project.

[18] Bertrand Meyer. 2008. Seven principles of software testing. *Computer* 41, 8 (2008), 99–101.

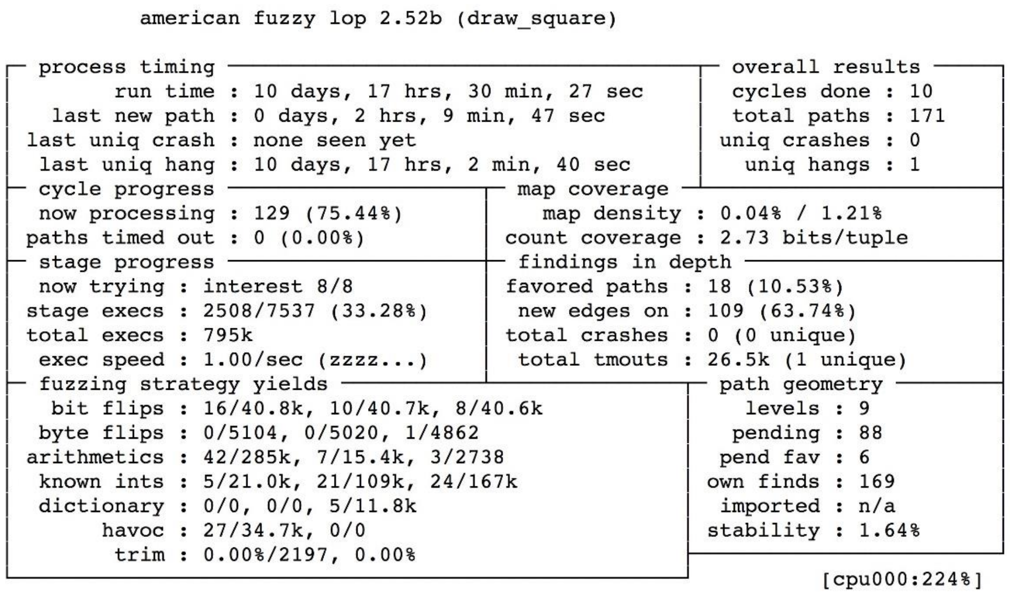
[19] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. Commun. ACM 33, 12 (1990), 32–44.

[20] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. 2018. *Fuzzing for software security testing and quality assurance.* Artech House.

[21] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 253–262.

[22] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion.* ACM, 815–816.

[23] Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices* 46, 4 (2011), 53–64.



```
           american fuzzy lop 2.52b (draw_square)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 10 days, 17 hrs, 30 min, 27 sec    │  cycles done : 10     │
│   last new path : 0 days, 2 hrs, 9 min, 47 sec       │  total paths : 171    │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : 10 days, 17 hrs, 2 min, 40 sec     │   uniq hangs : 1      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 129 (75.44%)      │    map density : 0.04% / 1.21%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.73 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : interest 8/8          │ favored paths : 18 (10.53%)           │
│ stage execs : 2508/7537 (33.28%)    │  new edges on : 109 (63.74%)          │
│ total execs : 795k                  │ total crashes : 0 (0 unique)          │
│  exec speed : 1.00/sec (zzzz...)    │  total tmouts : 26.5k (1 unique)      │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 16/40.8k, 10/40.7k, 8/40.6k           │    levels : 9          │
│  byte flips : 0/5104, 0/5020, 1/4862                │   pending : 88         │
│ arithmetics : 42/285k, 7/15.4k, 3/2738              │  pend fav : 6          │
│  known ints : 5/21.0k, 21/109k, 24/167k             │ own finds : 169        │
│  dictionary : 0/0, 0/0, 5/11.8k                     │  imported : n/a        │
│       havoc : 27/34.7k, 0/0                         │ stability : 1.64%      │
│        trim : 0.00%/2197, 0.00%                     ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000:224%]
```

Figure 19: Result summary after Shape Testing turtlesim for more than ten days. Provided by AFL.

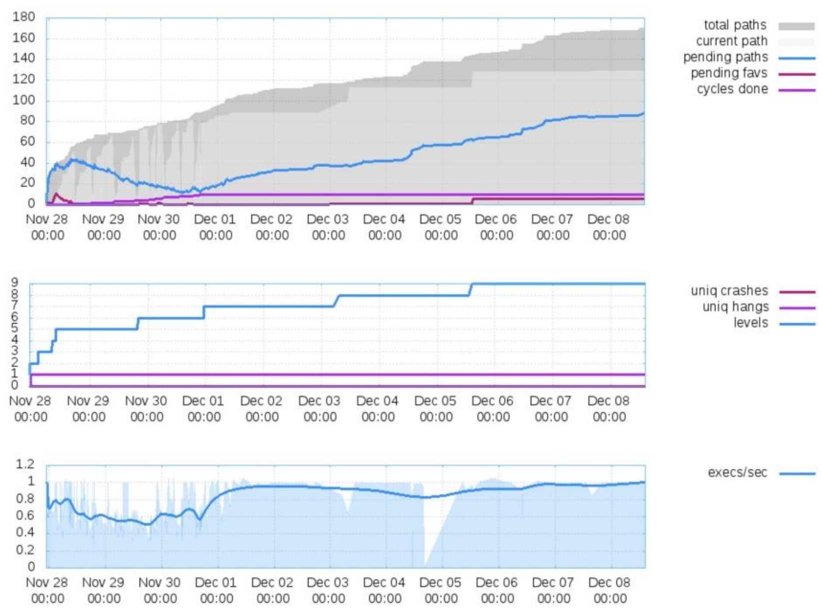


Figure 20: Statistics obtained after Shape Testing turtlesim for more than ten days. Generated by afl-plot.

3.6 Conclusion

Under the current testing infrastructures, ROS developers need to write test harness that executes during the whole software evolution. However, writing a test harness requires expertise and time. For example, the geometry2 package has only 20 test harnesses. To some extent, the open-source ROS project becomes closed-source when it comes to writing tests. The limited number of test harness provide little coverage of run-time scenarios, which is ineffective for enhancing code reliability. In this project, we propose to decouple the harness from testing. We introduce Shape Testing — a new testing methodology that leverages a Main program for testing ROS packages. Unlike the test harness, all users can contribute to the Main program. We write the ultimate goal of shape testing as the following slogan.

If you can run it, we can test it.

To achieve this goal, Shape Testing takes a ROS package and a Main program as inputs. It fulfills the following steps automatically. The ROS package is sanitized at compile time, which injects assertions into the program. The Main program is abstracted into a shape so that concretizing the shape can generate an arbitrary number of new Main programs. The sanitization step is essential as it provides a sort of "crash-able" conditions. The shaping step also plays an indispensable role as it unleashes the power of the fuzzing techniques. It is worth noting that fuzzing would not apply to ROS directly (the the hypothesis-ros package, for example, requires developers write properties before applying random testing).

We have prototyped shape testing with AFL, random testing (also known as the dumb fuzzing), Address Sanitizer, Thread Sanitizer, and Undefined Behavior Sanitizers. In our preliminary bugs, we have found several bugs including one from the roscpp_cpp package. Our most massive experiment has been run for ten days for stress testing the turtlesim package.

This work has opened up a promising avenue of research and represents only a minute proportion of what remains to be done. For the near future, we plan to implement, evaluate, and integrate Shape Testing into the ROS ecosystem.

4. Outlook

We have presented the first steps of ROSIN's testing-based validation strategy for robotic systems and for platform level problems. In the upcoming parts of the project we intend to work with the proposed infrastructure in the following directions:

- For the Model-in-the-loop testing we will continue to work with the automation aspect of the testing, and especially by connect it to the continues integration services related to Task 3.2. The intention is to include the whole chain from receiving a simulation job from the continuous integration service, run the simulation, collect the results and finally report back to the server.
- We will develop the shape testing infrastructure to the point when we can deploy it automatically on many standard ROS nodes in the distribution, so that bugs can be reported to maintainers for many packages. Furthermore, we intend to interface the infrastructure to ROS 2, which is quickly becoming a promising platform for industrial robotics.

5. Appendices

5.1 Appendix A

The RobotStudio MIL Manager Add-In, described in section 2.2.1, uses the following Google Protocol Buffers messages for communication with the Add-In's TCP server.

The messages specify the available requests and response messages that can be used during a communication session.

```
syntax = "proto3";
package proto.mil;

//-----------------------------------------------------------
// Primary
//-----------------------------------------------------------
// Context for requests and responses.
enum Context
{
    UNKNOWN                    = 0;
    STATION_INFO               = 1;
    STATION_LOAD               = 2;
    STATION_CLOSE              = 3;
    SIMULATION_INFO            = 4;
    SIMULATION_START           = 5;
    SIMULATION_STOP            = 6;
    SIMULATION_WAIT_FOR_FINISH = 7;
    SIMULATION_RESULTS         = 8;
}

//-----------------------------------------------------------
// Request Related
//-----------------------------------------------------------
// Required argument for "STATION_LOAD" context.
message StationLoadArguments
{
    string station_name = 1;
}

// Optional argument for "SIMULATION_START" context. If not specified,
// then the current robot state(s) are used for the simulation.
message SimulationStartArguments
{
    message RobotStartState
    {
        string robot_name             = 1;
        repeated double joints_values = 2; // [Unit: Degrees]
    }

    repeated RobotStartState robot_start_states = 1;
}

message ContextArguments
{
    StationLoadArguments     station_load     = 1;
    SimulationStartArguments simulation_start = 2;
}

message MILRequest
{
    Context          context           = 1;
    ContextArguments context_arguments = 2;
}
```

```
//-------------------------------------------------------------
// Response Related
//-------------------------------------------------------------
message Status
{
    enum Status
    {
        UNKNOWN = 0;
        SUCCESS = 1;
        ERROR   = 2;
    }

    Status value = 1;
}

message Error
{
    enum Error
    {
        UNKNOWN                        = 0;
        BAD_REQUEST                    = 1;
        SERVER_BUSY                    = 2;
        STATION_NOT_FOUND              = 3;
        ROBOT_NOT_FOUND                = 4;
        NO_ACTIVE_STATION              = 5;
        INVALID_ARGUMENTS              = 6;
        MISSING_ARGUMENTS              = 7;
        SIMULATION_IS_RUNNING          = 8;
        NO_SIMULATION_HAS_BEEN_STARTED = 9;
        SIMULATOR_NOT_READY            = 10;
    }

    Error value = 1;
}

message StationInfoContent
{
    repeated string available_stations = 1;
}

message SimulationInfoContent
{
    repeated string available_robots = 1;
}

message SimulationResultsContent
{
    repeated string warnings = 1;
    repeated string errors   = 2;
}

message ContextContent
{
    StationInfoContent       station_info       = 1;
    SimulationInfoContent    simulation_info    = 2;
    SimulationResultsContent simulation_results = 3;
}

message MILResponse
{
    Context        context         = 1;
    Status         status          = 2;
    Error          error           = 3;
    ContextContent context_content = 4;
}
```

5.2 Appendix B

Two mock-up ROS nodes has been created for trying out the MIL testing setup. The following source code illustrates the amount of code required to interact with ABB robot controllers, when the aforementioned ease-of-use packages has been utilized.

That is the abb_librws and the abb_libegm C++ libraries. Additionally, these nodes assume that the RobotWare StateMachine Add-In was used for setting up the ABB robot controllers with a ready to run RAPID program.

The RWS mock-up node's source code are as follows:

```
#include <ros/ros.h>
#include <abb_librws/rws_state_machine_interface.h>

int main(int argc, char** argv)
{
  ROS_INFO("======================= RWS sample node =======================");
  ros::init(argc, argv, "rws_node");
  ros::NodeHandle node_handle;

  // Create a RWS interface:
  // * Sets up a RWS client (that can connect to the robot controller's RWS server).
  // * Provides APIs to the user (for accessing the RWS server's services).
  //
  // Note: It is important to set the correct IP-address here, to the RWS server.
  abb::rws::RWSStateMachineInterface rws_interface("192.168.0.197");

  //----------------------------------------------------------
  // Loop for starting and stopping EGM motions
  //----------------------------------------------------------
  while(ros::ok())
  {
    ROS_INFO_STREAM("Press Enter to start EGM Pose mode...");
    std::cin.get();
    rws_interface.services().egm().signalEGMStartPose();

    ROS_INFO_STREAM("Press Enter to stop EGM...");
    std::cin.get();
    rws_interface.services().egm().signalEGMStop();
  }

  return 0;
}
```

The EGM mock-up node's source code are as follows:

```
#include <ros/ros.h>
#include <abb_libegm/egm_trajectory_interface.h>

int main(int argc, char** argv)
{
  ROS_INFO("======================= EGM sample node =======================");
  ros::init(argc, argv, "egm_node");
  ros::NodeHandle node_handle;

  // Boost components for managing asynchronous UDP socket(s).
  boost::asio::io_service io_service;
  boost::thread_group thread_group;

  // Configurations for the EGM interface.
  abb::egm::BaseConfiguration configuration;
  configuration.use_demo_outputs = true;
  // Create an EGM interface:
  // * Sets up an EGM server (that the robot controller's EGM client can connect to).
  // * Provides APIs to the user (for setting the EGM motion reference).
  //
  // Note: It is important to set the correct port number here,
  //       as well as configuring the settings for the EGM client in the robot controller.
  abb::egm::EGMBaseInterface egm_interface(io_service, 6511, configuration);

  if(!egm_interface.isInitialized())
  {
    ROS_ERROR("EGM interface failed to initialize (e.g. due to port already bound)");
    return 0;
  }

  // Spin up a thread to run the io_service.
  thread_group.create_thread(boost::bind(&boost::asio::io_service::run, &io_service));

  //----------------------------------------------------------
  // Loop for executing EGM demo references.
  //----------------------------------------------------------
  while(ros::ok())
  {
    bool wait = true;
    ROS_INFO("Waiting for EGM to start...");
    while(ros::ok() && wait)
    {
      if(egm_interface.isConnected())
      {
        wait = egm_interface.getStatus().egm_state()!=abb::egm::wrapper::Status_EGMState_EGM_RUNNING;
      }

      ros::Duration(0.5).sleep();
    }

    ROS_INFO("EGM started.");

    wait = true;
    while(ros::ok() && wait)
    {
      ROS_INFO("    EGM running...");
      ros::Duration(5.0).sleep();
      wait = egm_interface.getStatus().egm_state() == abb::egm::wrapper::Status_EGMState_EGM_RUNNING;
    }

    ROS_INFO("EGM finsihed.\n");
  }

  // Perform a clean shutdown.
  io_service.stop();
  thread_group.join_all();

  return 0;
}
```