Model-based integration testing of ROS packages: a mobile robot case study

Juhan Ernits, Evelin Halling, Gert Kanter and Jüri Vain

***Abstract*—We apply model-based testing – a black box testing technology – to improve the state of the art of integration testing of navigation and localisation software for mobile robots built in ROS. Online model-based testing involves building executable models of the requirements and executing them in parallel with the implementation under test (IUT). In the current paper we present an automated approach to generating a model from the topological map that specifies where the robot can move to. In addition, we show how to specify scenarios of interest and how to add human models to the simulated environment according to a specified scenario. We measure the quality of the tests by code coverage, and empirically show that it is possible to achieve increased test coverage by specifying simple scenarios on the automatically generated model of the topological map. The scenarios augmented by adding humans to specified rooms at specified stages of the scenario simulate the changes in the environment caused by humans. Since we test navigation at coordinate and topological level, we report on finding problems related to the topological map.**

I. Introduction

The software for robots gets increasingly complex as computational resources keep increasing at reduced power budgets. Thus it has become possible to develop software that enables robots to cope in realistic human environments. The current research in, e.g. long term behaviour of mobile robots is concerned with changing environments involving humans and human interaction with robots. The research targets specific scenarios e.g. involving recurring robot behaviour over time in dynamic environments as in [14], [1], and techniques to reason about changing scenes, as in [16], [15]. Such problems stem from the dynamic nature of human environments and the need for robots to cope in them.

While the current robotics research advances the frontiers of what can be achieved by robots, we are aware of relatively moderate amount of work done on how to test robot software to ensure that such solutions are robust and actually work as expected. Evaluation and testing such software is often achieved by running extended tests on real hardware and in simulation. But how much testing is enough? When different development teams develop separate components, how can the influence of a changeset on the overall system behaviour be efficiently evaluated?

Most contemporary software for robots uses some kind of data sharing framework to fascilitate interconnection of sensors, various data processing nodes and actuators. While there exist several such frameworks, we target ROS [18] as a representative and widely used such framework.

Department of Computer Science, Tallinn University of Technology Akadeemia tee 15a, 12618 Tallinn, Estonia firstname.lastname@ttu.ee

978-1-4673-9163-4/15/$31.00 ©

The primary focus of the current paper is how to improve the state of the art of integration testing ROS packages involved in high level robot control, such as e.g. localisation and navigation of mobile robots.

In order not to delve into the very basics of integration testing, it is useful to assume that there is some kind of integration testing system in place, e.g. Jenkins [13]. Jenkins attempts to build all software that gets uploaded to the repository and run the existing tests. When the tests pass, the Jenkins instance can be instructed to upload the binaries to a distribution site. Our goal is to support such integration testing scenario and provide feedback whether some lines of code, e.g. the ones that just got updated, were active in certain test scenarios or not.

We take the approach of Robot Unit Testing [6] and extend it in two ways: first we introduce a white box metric of code coverage, in particular statement and branch coverage, as a quality measure of the tests. Second, we combine a technique called model-based testing [19] into the test setup, that allows us to formalise the requirements of the system into a formal model and check the conformance of the formalised requirements to the *implementation under test* (IUT), in our case the appropriate stack of ROS packages together with either a real or simulated set of sensors and actuators.

The experiments involve modelling and testing the navigation and localisation components of the software stack developed in the STRANDS[1] project. The stack was chosen because it involves multiple layers of functionality on top of the standard ROS `move_base` mobile base package that is responsible for accomplishing navigation, it is open sourced, accessible on GitHub, contains a working simulation environment built using Morse [7], and many existing quality assurance techniques are actively used in the project, including unit tests and a Jenkins based continuous integration system.

A. Test metrics

In order to evaluate and compare different test methods, it is important to quantitatively measure the results. There exist several metrics for software tests, like the number of code errors found, number of test cases per requirement, number of successful test cases etc, that are difficult to apply in our setting where the requirements are not completely clear. Instead we will use the metric of statement and branch coverage of the ROS packages that we are interested in and we prefer tests that excercise a larger percentage of the robot code.

It is important to keep in mind that 100% statement coverage does not guarantee that the code is bug free. On

[1]Spatio-Temporal Representation and Activities for Cognitive Control in Long-Term Scenarios (STRANDS). The project is funded by the EC 7th Frameworks Programme, Cognitive Systems and Robotics, project reference 600623.

the other hand, code coverage is a metric that gives some idea about how much of the code gets touched by the tests.

Another relevant metric in the setting of ROS is performance, i.e. how much CPU and memory resources get used by certain nodes under certain scenarios. We will only use the code coverage metric in the current paper as we have no reference for evaluating the performance results in the context of the chosen case study and monitoring performance has been addressed in e.g. [17] previously.

For computing code coverage of Python modules we use the tool `coverage.py` [4]. The approach is programming language agnostic, for example, `llvm-cov` or `gcov` can be used for measuring C++ code coverage in ROS with little additional effort.

B. Model-based testing

Model-based testing is testing on a model that describes how the system is required to behave. The model, built in a suitable machine interpretable formalism, can be used to automatically generate the test cases, either offline or online, and can also be used as the oracle that checks if the IUT passes the tests. Offline test generation means that tests are generated before test execution and executed when needed. In the case of online test generation the model is executed in lock step with the IUT. The communication between the model and the IUT involves controllable inputs of the IUT and observable outputs of the IUT. For example, we can tell the robot to go to a node called Station, and we can observe if and when the robot achieves the goal.

There are multiple different formalisms used for building models of the requirements. Our choice is Uppaal timed automata (TA) [5] because the formalism naturally supports state transitions and time and there exists the Uppaal Tron [11] tool that supports online model-based testing.

II. Related work

A. Testing in ROS

Testing is required in the ROS quality assurance process[2], meaning that a package needs to have tests in order to comply with the ROS package quality requirements. However, the requirement is compulsory only for centrally maintained ROS packages and it is up to the particular maintainers to choose their quality assurance process.

The ROS infrastructure supports different levels of testing. The basic testing methodology used in ROS is unit testing. The most used testing tools in ROS unit testing are *gtest* (Google C++ testing framework), *unittest* (Python unit testing framework) and *nosetest* (a more user-friendly Python unit testing framework, which extends the unittest framework). Using the aforementioned tools is not a strict requirement as the tools are agnostic to which testing framework is used. The only requirement is that the used testing framework outputs the test results in a suitable XML format (Ant JUnit XML report format).

ROS has support for higher level integration tests as well. Integration testing can be done using the *rostest* package, which is a wrapper for the `roslaunch` tool. Rostest allows specifying complex configurations of tests, which enables integration testing of different packages.

The ROS build tool (`catkin_make`) has built-in support for testing and it is fairly simple to include tests for the package. The main concern, however, is with creating the tests as it is up to the developer to write the tests for their packages. This can be difficult, because many robotics packages deal with dynamic data (e.g. object detection from image stream) and testing with dynamic data is more challenging than unit testing simple functions. For this reason, many developers neglect creating unit tests.

To our knowledge there is relatively little research published on testing robot software in ROS, but we think it deserves further attention, since it is not trivial to apply the testing techniques known in the field of software engineering to robot software. Bihlmaier and Wörn [6] introduced RUT (Robot Unit Testing) methodology to bring modern testing techniques to robotics. They outline the process of testing robots utilizing a simulation environment (e.g. Gazebo or MORSE) and control software to test robot performance and correctness of the control algorithm without actually running the tests on real hardware. Our approach follows theirs, but we have firstly introduced quantified measurement of Python code in the context of ROS and secondly embedded online model-based testing into the ROS framework, that enables not only to drive the system through scenarios deemed interesting by the developers, but also checks if the behaviour conforms to the models, i.e. formalised requirements.

Robotic environments entail uncertainty, and testing in the presence of uncertainty is a hard problem, especially automatically deciding whether a test succeeded or failed. There is an attempt to address the issue in [8] where some ideas in the future handling of uncertainty in testing are outlined. The main emphasis is on using probabilistic models to specify input distributions and to accommodate environment uncertainty in the models. We take a different approach to accommodating uncertainty by abstracting behaviour and measuring whether goals are reached within reasonable time limits.

B. Robot monitoring and fault detection

Several fault detection and monitoring approaches in conjunction with robotic frameworks have been proposed, e.g. [10], [12], [17], that enable to detect various faults in robot software. These complement our approach, as we introduce monitoring conformance to certain aspects of specifications that we have encoded into our model, e.g. that the robot makes reasonable progress from topological location to another connected topological location. Our approach differs from the above in the sense that in addition to monitoring, we also provide control inputs to the system. In fact, we get the continuous patrolling feature for free, as we generate the model from the topological map.

III. Modelling robot requirements with timed automata

The overall test setup used in the context of model-based testing with Uppaal Tron as the test engine and dTron as the adapter generation framework is given in Fig. 1. The model

[2] http://wiki.ros.org/QAProcess

contains the formalisation of the requirements of the IUT and the environment. We model the topological map of the environment and encode distances as deadlines. The adapter is responsible for translating messages from the model to postings to appropriate topics in ROS, and vice versa. The dTron layer allows the adapter to be distributed across multiple computers while ensuring that measuring the time stays valid.

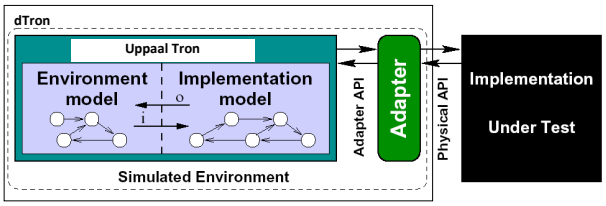

Fig. 1. The test setup involving the Uppaal Tron test engine and the distributed adapter library dTron.

The test configuration used in the current work consists of test execution environment dTron and one or many test adapters that transform abstract input/output symbols of the model to input/output data of the robot. The setup is outlined in Fig. 1. Uppaal Tron is used as a primary test execution engine. Uppaal Tron simulates interactions between the IUT and its environment by having two model components – the *environment* and the *implementation* model. The interactions between these component models are monitored during model execution. When the environment model initiates an input action i Tron triggers input data generation in the adapter and the actual test data is written to the robot interface. In response to that, the robot software produces output data that is transformed back to model output o. Thereafter, the equivalence between the output returned and the output o specified in the model is checked. The run continues if there is no conformance violation, i.e. there exists an enabled transition in the model with parameters equivalent to those passed by the robot. In addition to input/output conformance, the $rtioco_e$ checking supported by Uppaal Tron also checks for timing conformance. We refer the reader to [5] for the details.

A. Uppaal Timed Automata

Uppaal Timed Automata [5] (TA) used for the specification of the requirements are defined as a closed network of extended timed automata that are called *processes*. The processes are combined into a single system by the parallel composition known from the process algebra CCS. An example of a system of two automata comprised of 3 locations and 2 transitions each is given in Fig. 2.

The nodes of the automata are called *locations* and the directed edges *transitions*. The *state* of an automaton consists of its current location and assignments to all variables, including clocks. The initial locations of the automata are graphically denoted by an additional circle inside the location.

Synchronous communication between the processes is by hand-shake synchronisation links that are called *channels*. A channel relates a pair of transitions labelled with symbols for input actions denoted by e.g. `chA?` and `chB?` in Fig. 2, and output actions denoted by `chA!` and `chB!`, where `chA` and `chB` are the names of the channels.

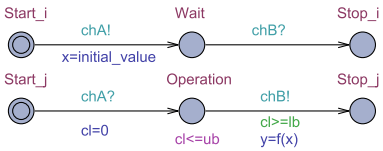

Fig. 2. A sample system with two Uppaal timed automata with synchronisation channels `chA` and `chB`. The automaton at the top denotes Process_i and the one below Process_j. In addition to the automata, the model also includes the declarations of channels `chA` and `chB`, integer constants `lb=1`, `ub=3`, and `initial_value=0`, integer variables `x` and `y`, a clock `cl`, and a function `f(x)` defined in a subset of the C language.

In Fig. 2, there is an example of a model that represents a synchronous remote procedure call. The calling process Process_i and the callee process Process_j both include three locations and two synchronised transitions. Process_i, initially at location Start_i, initiates the call by executing the send action `chA!` that is synchronised with the receive action `chA?` in Process_j, that is initially at location Start_j. The location Operation denotes the situation where Process_j computes the output `y`. Once done, the control is returned to Process_i by the action `chB!`

The duration of the execution of the result is specified by the interval $[lb, ub]$ where the upper bound ub is given by the *invariant* `cl<=ub`, and the lower bound lb by the *guard condition* `cl>=lb` of the transition Operation $\rightarrow$ Stop_j. The *assignment* `cl=0` on the transition Start_j $\rightarrow$ Operation ensures that the clock cl is reset when the control reaches the location Operation. The global variables `x` and `y` model the input and output arguments of the remote procedure call, and the computation itself is modelled by the function `f(x)` defined in the Uppaal model.

Please note that in the general case these inputs and outputs are between the processes of the model. The inputs and outputs of the test system use channels labelled in a special way described later. Asynchronous communication between processes is modelled using global variables accessible to all processes.

Formally the Uppaal timed automata are defined as follows. Let Σ denote a finite alphabet of actions $a, b, \ldots$ and C a finite set of real-valued variables p, q, r, denoting clocks. A guard is a conjunctive formula of atomic constraints of the form $p \sim n$ for $c \in C, \sim \in \{\geq, \leq, =, >, <\}$ and $n \in \mathbb{N}^+$. We use $G(C)$ to denote the set of clock guards. A timed automaton A is a tuple $\langle N, l_0, E, I \rangle$ where N is a finite set of locations (graphically denoted by nodes), $l_0 \in N$ is the initial location, $E \in N \times G(C) \times \Sigma \times 2^C \times N$ is the set of edges (an edge is denoted by an arc) and $I : N \rightarrow G(C)$ assigns invariants to locations (here we restrict to constraints in the form: $p \leq n$ or $p < n, n \in \mathbb{N}^+$. Without the loss of generality we assume that guard conditions are in conjunctive form with conjuncts including besides clock constraints also constraints on integer variables. Similarly to clock conditions, the propositions on integer variables k are of the form $k \sim n$ for $n \in \mathbb{N}$, and $\sim \in \{\leq, \geq, =, >, <\}$. For the formal definition of Uppaal TA full semantics we refer the reader to [5].

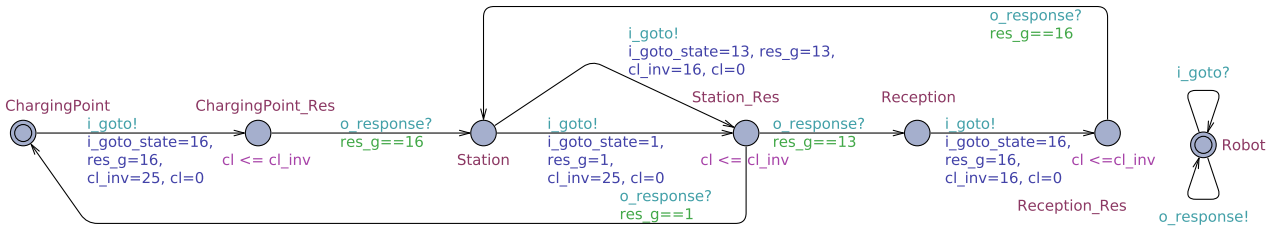

Fig. 3. Automatically generated timed automaton representation of the topological map containing locations “ChargingPoint”, “Station” and “Reception”.

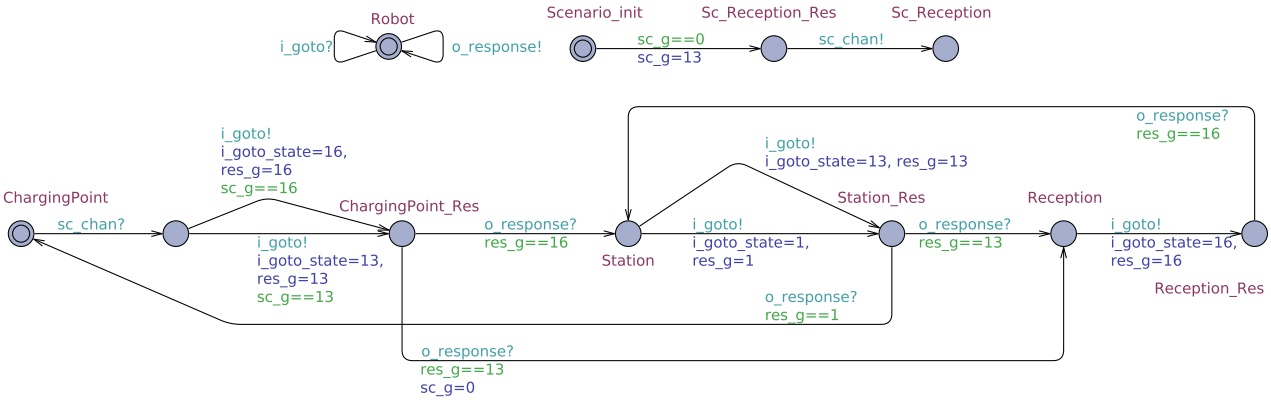

Fig. 4. Automatically generated timed automaton denoting the topological map (below) and the desired scenario (above).

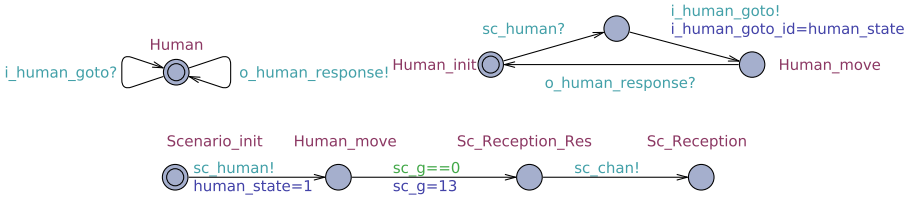

Fig. 5. A scenario involving models of humans

B. Modelling the topological map

One of the general requirements of a mobile robot is that it should be able to move around in its environment. We relate the requirement to the topological map and state that *the robot should be able to move to the nodes of the topological map*. The details of how the topological map gets constructed for the particular case study are given in [9], but for the purpose of the current requirements, we assume that each node of the topological map should be reachable by the robot and that from each node, it should be possible to reach adjacent nodes without having to visit any further nodes.

Since the topological map is an artefact frequently present in mobile robots, we chose to automate the translation of the topological map to the TA representation.

In Fig. 3 there is an example of a TA model of the environment of a robot specifying where the robot should be able to move and in what time the robot should be able to complete the moves. The environment stipulates that when the robot moves from the node called ChargingPoint to Station on the topological map, it synchronises on the communication channel `i_goto` with the robot model. This corresponds to passing the command to the robot to go to the state number 16, which denotes the Station node on the topological map. The destination node number is assigned on the transition to a parameter `i_goto_state` that is passed along with the synchronisation command to the test adapter which in turn will pass the command to move to the Station node on to the robot. There is an additional assignment on the transition, `res_g=16` which denotes that the current goal is node 16, i.e. the Station node. The assignment `cl_inv=25` means that the maximum time allowed for the robot to be on its way from ChargingPoint to Station is 25 time units. The clock `cl` is then reset to 0. The automaton transitions to the intermediate location ChargingPoint_Res where it awaits reaching the Station node. When Station is reached, the robot will indicate it to the test adapter which converts the indication to the response `o_response` that is passed back to the model. The guard `res_g==16` only allows the transition to be taken for the goal node 16. While on the second transition the guard does not influence the behaviour as there is no other value `res_g` can have taken, there is a choice on transitions starting from the Station node. It is possible either to go back to the ChargingPoint node when taking the transition below with the assignment `res_g=1` or go to the Reception node by taking the transition above with the assignment `res_g=13`. Then, the

guards `res_g==1` and `res_g==13` will restrict which will be the valid transition after the robot has reached the goal. In this way the model will be able to distinguish which node it started off to. If the robot for some reason wonders to a wrong node or is kidnapped on the way without covering the sensors, it will be detected as a conformance failure. Also, if the robot takes too much time to reach another node, the model will trigger an error. The time restrictions are enforced by invariants `cl<=cl_inv` at the intermediate states. The robot is modelled as an automaton with a single location and the edges synchronising on the IUT input and output messages – those denoted by the `i_...` and `o_...` channels. The model of the robot is input enabled, i.e. it does not restrict any behaviour, it is up to the implementation – the robot software.

In Fig. 4 there is a model where the behaviour is restricted with a *scenario*. The model contains an automaton corresponding to the topological map, an automaton corresponding to a scenario above it, and an input enabled single location automaton denoting the robot. Now there is one additional intermediate state to facilitate synchronisation with the scenario over the `sc_chan` channel. It is important to note that when synchronisation over the `sc_chan` takes place, an integer variable `sc_g` is assigned the value 13. After such synchronisation, when the map model is at ChargingPoint_Res location, the only option to proceed is to the Reception location. In this way a goal is set that is not an immediate neighbour of the current node on the map.

If multiple such combinations of pairs are enabled, one is chosen either randomly or according to some test coverage criterion involving model elements. We have omitted the clock resets and invariants from Fig. 4 for brevity. The automaton with a single location denotes the model of the robot.

In our approach both models are automatically generated from the topological map file in the yaml format. The time delays allowed to transition from a node to node on the topological map are computed based on the distances along the edges between nodes and are computed with a margin to accommodate time spent on turning. The resulting model is able to detect situations when the robot gets stuck for example when moving too close to a low wall in simulation or when there is a link on the topological map that is not present in the environment. In the case of scenarios we compute the length of the shortest path between each pair of nodes specified in the scenario and add appropriate time restrictions to the invariant of the intermediate location introduced between the nodes on the topological map automaton stipulated by the scenario.

In Fig. 5 we add the human factor to the environment. As the simulator, Morse, supports models of humans, we can create scenarios where humans are either moved around or "teleported" to desired locations. We leave the actual locations to be specified at the adapter level and specify in the scenario automaton when which configuration of humans should be present in certain rooms. This way we can change the environment in chosen rooms and emulate varying patterns involving humans.

We can think of the scenarios as *high level test cases* in the context of integration testing. In the example above there is a test case for moving along a predefined adjacent set of nodes, one for moving to a remote location on the topological map, and one moving throught rooms with humans present.

In addition to actual testing, such scenarios run in cycles can be used for generating data, e.g. representing long term behaviour in simulation. As we can leave certain parts of the scenario less strictly specified, the model-based testing tool will vary the scenarios randomly.

Once a test failure is encountered, the search for the causes is currently left to the user. If the problem is repeatable after a certain patch set, the problem obviously may be related to the patch set, but it may also be the problem with the simulator or wrong estimates of deadlines in the model. The process of getting the requirements related to the model right requires work, that can be considered the overhead in our approach.

IV. Test execution

In order to connect the model to the robot we need an *adapter* (sometimes called *harness*) that connects the abstract messages from the model to the concrete messages comprehensible by the robot and vice versa. We use the dTron tool to fascilitate connecting our ROS adapter to Uppaal Tron. We refer to the dTron setup as the *tester*.

A. dTron

dTron[3] [3] extends the functionality of Uppaal Tron by enabling distributed and coordinated execution of tests across a computer network. It relies on Network Time Protocol (NTP) based clock corrections to give a global timestamp (t_1) to events arriving at the IUT adapter. These events are then globally serialized and published to other subscribers using the Spread toolkit [2]. Subscribers can be other IUT adapters, as well as dTron instances. Subscribers that have clocks synchronised with NTP also timestamp the event received message (t_2) to compute and if necessary and possible, compensate for the messaging overhead $D = t_2 - t_1$. The parameter D is essential in real-time executions to compensate for messaging delays in test verdict that may otherwise lead to false-negative non-conformance results for the test-runs.

B. ROS test adapter

We have created a test adapter that can be used as a template for test adapters in any similar dTron-based MBT setup. The template adapter is implemented in C++ and the specific case study inherits from the template adapter and implements the one specific to the case study. The template adapter has the implementations for receiving and sending messages between the tester and the adapter – a generic prerequisite for performing model-based testing with dTron.

In essence, the test adapter specifies what is to be done when a synchronisation message is received from the tester. In the mobile robot case, the model specifies the goal waypoint where the robot must travel. Upon receiving the waypoint information via a synchronisation message (`i_...` channel), it is either passed as a goal to the standard ROS `move_base` action server or the action server responsible for topological navigation – `topological_navigation`, depending on which level we choose to run the tests. In the implementation the goal topic is specified in the configuration of the adapter.

[3]http://www.cs.ttu.ee/dtron

After publishing the goal, the adapter waits for the action server to return a result for the action (i.e. whether it was successful or not). In case the action was successful, the adapter sends a synchronisation message (`o_...` channel) back to the tester and waits for a new goal to be passed on to the action server. If the goal was not achieved, the adapter does not send a synchronisation to the tester and the tester will detect a time-out and report the test failure.

V. Experimental results

We specified different scenarios, i.e. high level test cases, and ran the tests in two different simulated environments[4] and repeated same scenarios by sending the goals to the `move_base` and `topological_navigation` action servers. The results are summarised in Table I and the highest coverage results for the particular test case for each package are highlighted in cases they can be distinguished. The "Total" columns represent total number of statements of Python code in the package, "Missed" columns represent the number of statements missed, and the "%" columns represent a combined statement and branch coverage percentage. That is why there are same statement counts but different percentages in the results of e.g. the `localisation` node.

Initially we tested the code coverage by manually specifying a neigbouring node on the topological map to `move_base`. Then we proceeded giving the same topological node to the robot as a goal via the `topological_navigation` action server. These are the rows marked by *manual goal*.

Then we repeated the same neighbouring node goals, but controlled the robot from the model (rows marked by *model*). It is expected that the code coverage is very close in these cases.

Next we specified a scenario with the goal node not being a neighbour. It can be seen from the results that significantly more code was exercised in the topological navigation node in the case of scenarios with goals passed to the `topological_navigation` action server.

Next we tested the scenario when human models are moved to different locations in a room and robot is given tasks to enter and leave the room. We managed to run the tests when passing goals to `move_base`, but the `topological_navigation` case failed because the robot got stuck with "DWA planner failed to produce path" error from `move_base`. We cannot confirm the problem to be a code error, as it can also be related to a local simulator configuration. But we have successfully demonstrated that it is possible to produce code coverage results in cases where the tests succeed and point out scenarios where the goals are not reached.

We also augmented the topological map with a transition through a wall. The test system yielded a test failure based on exceeding the deadline for reaching the node.

We used different versions of code in C1 and C2, that is why there is are slightly different statement counts in similar components. C1 experiments were done on STRANDS packages taken from the GIT repository while C2 experiments were done using current versions of packages available from the Ubuntu Strands deb package repository. In the case of the `actionlib` package, it appears that more code is utilized in the case of topological navigation. The `strands_navigation` package contains only messages in Python, thus there are very little differences in coverage. In the `topological_navigation` module utilisation there is clear correlation between the use of topological goals and code coverage. The `localisation` node uses practically the same amount of code regardless of the scenario. In the case of the `navigation` node the difference is the largest and there is clear correlation between larger code coverage and harder navigation tasks[5].

When interpreting the results it is important to keep in mind that the coverage numbers are for *single high level test cases*. When analysing e.g. the `navigation` package coverage, then the code covered in the `move_base` case is a subset of the coverage in the `topological_navigation` case. Developing a test suite with a higher total code coverage is an iterative process of running the tests and looking at what code has still been missed. In the current case study, the test suite needs to be extended with a scenario with a goal that is the same as the current node, there need to be scenarios triggering the preemting of goals, and triggering several different exceptions. Such behaviour requires extending the model, and perhaps the adapter. While the coverage numbers of the reported scenarios are below 100%, the added value provided by the current approach lies in clear feedback, either in the form of test failure, or in the case of success, what code was used in the particular set of scenarios and what was missed.

VI. Conclusion

We presented a case study of applying model-based testing in ROS and evaluating the results in terms of code coverage of code related to topological navigation of mobile robots. Relying on the empirical evidence, we conclude that the proposed automatic generation of models from topological maps and defining scenarios as sequences of states provides a valuable tool of exercising the system with the purpose to achieve high code coverage. By performing the tests on the `move_base` coordinate level and topological navigation level, we showed that our approach can also be used to validate and discover problems in configurations, such as topological maps. We also showed how to build models of the environment involving human models in simulation. Similar scenarios can be carried out also in real life, but then the test adapter needs to be changed to give humans instructions when and where to move to, and when humans are in place, the adapter can return to giving the robot next goals.

The future work on the model and adapter side involves extending the dynamic reconfiguration of the environment, e.g. connecting collision detection probes in Morse with the test adapter and introducing natural human movement. Improving the code coverage requires insight into the packages and manual extension of model and the adapter to support triggering various exceptions and other specific actions.

[4]C1 corresponds to AAF simulation environment and C2 to Bham SoCS building ground floor simulation environment.

[5]The code and detailed coverage statistics is available at http://cs.ttu.ee/staff/juhan/mobile_robot_mbt/.

TABLE I. THE EXPERIMENTAL RESULTS OF MODEL-BASED TESTING A MOBILE ROBOT IN SIMULATION IN TWO DIFFERENT VIRTUAL ENVIRONMENTS, C1 AND C2.

	actionlib			strands_navigation			topological_navigation			localisation node			navigation node		
	Total	Missed	%	Total	Missed	%	Total	Missed	%	Total	Missed	%	Total	Missed	%
C1 manual goal move_base	1347	733	46	13024	10969	16	1954	1502	23	154	37	72	349	251	24
C2 manual goal move_base	1347	872	35	13024	10902	16	1789	1479	17	154	37	73	344	247	24
C1 model goal move_base	1347	733	46	13024	10969	16	1954	1502	23	154	37	73	349	251	24
C2 model goal move_base	1347	872	35	13024	10902	16	1789	1479	17	154	37	73	344	247	24
C1 manual goal topo-nav	1347	565	58	13024	10832	17	1954	1335	32	154	37	72	349	99	64
C2 manual goal topo-nav	1347	678	50	13024	10864	17	1789	1346	25	154	37	73	344	159	48
C1 model goal topo-nav	1347	598	56	13024	10832	17	1954	1335	32	154	37	73	349	97	65
C2 model goal topo-nav	1347	615	54	13024	10749	17	1789	1311	27	154	37	73	344	98	64
C1 scenario topo-nav	1347	598	56	13024	10832	17	1954	1315	33	154	37	73	349	77	72
C2 scenario move_base	1347	872	35	13024	10902	16	1789	1475	18	154	37	73	344	247	24
C2 scenario topo-nav	1347	613	54	13024	10749	17	1789	1286	28	154	37	73	344	73	73
C2 scenario with humans move_base	1347	871	35	13024	10902	16	1789	1479	17	154	37	73	344	247	24

Acknowledgements

We would like to thank Nick Hawes, Marc Hanheide and Jaime Pulido Fentanes for useful discussions and support in setting up a local copy of the STRANDS simulation environment. We also thank the anonymous referees for their constructive comments. This work was partially supported by the European Union through the European Regional Development Fund via the competence centre ELIKO.

REFERENCES

[1] Rares Ambrus, Nils Bore, John Folkesson, and Patric Jensfelt. Meta-rooms: Building and maintaining long term spatial models in a dynamic world. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*, pages 1854–1861, 2014.

[2] Yair Amir, Michal Miskin-Amir, Jonathan Stanton, and John Schultz et al. The Spread toolkit, 2015. http://spread.org/, accessed in May 2015.

[3] Aivo Anier and Jüri Vain. Model based continual planning and control for assistive robots. In Emmanuel Conchon, Carlos Manuel B. A. Correia, Ana L. N. Fred, and Hugo Gamboa, editors, *HEALTHINF 2012 - Proceedings of the International Conference on Health Informatics, Vilamoura, Algarve, Portugal, 1 - 4 February, 2012.*, pages 382–385. SciTePress, 2012.

[4] Ned Batchelder and Gareth Rees. Coverage.py – code coverage testing for Python, 2015. http://nedbatchelder.com/code/coverage/, accessed in April 2015.

[5] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[6] Andreas Bihlmaier and Heinz Wörn. Robot unit testing. In Davide Brugali, Jan F. Broenink, Torsten Kroeger, and Bruce A. MacDonald, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 8810 of *Lecture Notes in Computer Science*, pages 255–266. Springer International Publishing, 2014.

[7] Gilberto Echeverria, Séverin Lemaignan, Arnaud Degroote, Simon Lacroix, Michael Karg, Pierrick Koch, Charles Lesire, and Serge Stinckwich. Simulating complex robotic scenarios with MORSE. In *SIMPAR*, pages 197–208, 2012.

[8] Sebastian G. Elbaum and David S. Rosenblum. Known unknowns: testing in the presence of uncertainty. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 833–836. ACM, 2014.

[9] Jaime Pulido Fentanes, Bruno Lacerda, Tomás Krajník, Nick Hawes, and Marc Hanheide. Now or later? Predicting and maximising success of navigation actions from long-term experience. In *IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015*, pages 1112–1117, 2015.

[10] Raphael Golombek, Sebastian Wrede, Marc Hanheide, and Martin Heckmann. Online data-driven fault detection for robotic systems. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011, San Francisco, CA, USA, September 25-30, 2011*, pages 3011–3016. IEEE, 2011.

[11] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer, 2008.

[12] Hengle Jiang, Sebastian G. Elbaum, and Carrick Detweiler. Reducing failure rates of robotic systems though inferred invariants monitoring. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 3-7, 2013*, pages 1899–1906. IEEE, 2013.

[13] Kohsuke Kawaguchi, Andrew Bayer, and R.Tyler Croy. Jenkins – an extensible open source continuous integration server, 2015. http://jenkins-ci.org, accessed in April 2015.

[14] Tomás Krajník, Jaime Pulido Fentanes, Óscar Martínez Mozos, Tom Duckett, Johan Ekekrantz, and Marc Hanheide. Long-term topological localisation for service robots in dynamic environments using spectral maps. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*, pages 4537–4542, 2014.

[15] Lars Kunze, Chris Burbridge, Marina Alberti, Akshaya Thippur, John Folkesson, Patric Jensfelt, and Nick Hawes. Combining top-down spatial reasoning and bottom-up object class recognition for scene understanding. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*, pages 2910–2915, 2014.

[16] Lars Kunze, Keerthi Kumar Doreswamy, and Nick Hawes. Using qualitative spatial relations for indirect object search. In *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*, pages 163–168. IEEE, 2014.

[17] Valiallah Monajjemi, Jens Wawerla, and Richard T. Vaughan. Drums: A middleware-aware distributed robot monitoring system. In *Canadian Conference on Computer and Robot Vision, CRV 2014, Montreal, QC, Canada, May 6-9, 2014*, pages 211–218. IEEE Computer Society, 2014.

[18] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[19] Mark Utting and Bruno Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007.