# Model-based Test Generation for Robotic Software: Automata versus Belief-Desire-Intention Agents

**Article** · September 2016

CITATIONS
2

READS
96

**3 authors**, including:

Dejanira Araiza-Illan
Agency for Science, Technology and Research (A*STAR)
**28** PUBLICATIONS **116** CITATIONS
SEE PROFILE

Kerstin Eder
University of Bristol
**106** PUBLICATIONS **758** CITATIONS
SEE PROFILE

# Model-based Test Generation for Robotic Software: Automata versus Belief-Desire-Intention Agents

Dejanira Araiza-Illan*, Anthony G. Pipe† and Kerstin Eder‡

**Abstract**

Robotic code needs to be verified to ensure its safety and functional correctness, especially when the robot is interacting with people. Testing the real code in simulation is a viable option. It reduces the costs of experiments and provides detail that is lost when using formal methods. However, generating tests that cover interesting scenarios, while executing most of the code, is a challenge amplified by the complexity of the interactions between the environment and the software. Model-based test generation methods can automate otherwise manual processes and facilitate reaching rare scenarios during testing. In this paper, we compare the use of Belief-Desire-Intention (BDI) agents as models for test generation, with more conventional, model-based test generation, that exploits automata and model checking techniques, and random test generation methods, in terms of practicality, performance, scalability, and exploration ('coverage'). Simulators and automated testbenches were implemented in Robot Operating System (ROS) and Gazebo, for testing the code of two robots, BERT2 in a cooperative manufacture (table assembly) task, and Tiago as a home care assistant. The results highlight the clear advantages of using BDI agents for test generation, compared to random and conventional automata-based approaches. BDI agents naturally emulate the agency present in Human-Robot Interaction (HRI). They are thus more expressive and scale well in HRI applications.

## 1 INTRODUCTION

As robot software designers, we must demonstrate the safety and functional soundness of robots that interact closely with people, if these technologies are to become viable commercial products [8]. Robot code needs to be verified to eliminate runtime bugs, such as floating point and memory allocation issues. Additionally, a robot's code must be verified and validated when it controls hardware components, interacts with the environment, including with people, and other software. The interaction of all these elements introduces complexity and concurrency, and thus the possibility of unexpected and undesirable behaviours [16, 3].

Available verification techniques include formal methods such as model checking and theorem proving, testing in simulation, and real-life experiments. These techniques differ in the exhaustiveness of the state space exploration, and the level of realism and detail. A proof of failure or compliance with requirements can be provided by formal methods, based on exhaustive exploration of an abstract model (e.g. in [25]). Testing in

*Department of Computer Science and Bristol Robotics Laboratory, University of Bristol, UK. E-mail: dejanira.araizaillan@bristol.ac.uk

†Faculty of Engineering Technology and the Bristol Robotics Laboratory, University of the West of England, Bristol, UK. E-mail: tony.pipe@brl.ac.uk

‡Department of Computer Science and Bristol Robotics Laboratory, University of Bristol, UK. E-mail: kerstin.eder@bristol.ac.uk

simulation explores a large degree of detail that is lost in the abstract models used to manage the state space in formal verification, but cannot be performed exhaustively. The robotic code can be tested in simulation, in a combination of the real robot (hardware-in-the-loop) and a simulated environment, or in a combination of a real environment (human-in-the-loop) and a simulated robot [21, 17, 22].

In our previous work, we tested real robot code in simulation, as a viable and cost-effective option for verification [3, 4, 2]. Automation of the testing process and a systematic exploration of the code under test within Human-Robot Interaction (HRI) scenarios was achieved through coverage-driven verification (CDV), a method that guides the generation of effective tests, according to feedback from coverage metrics [23]. In [3, 4], we illustrated how a CDV testbench, comprising a test generator, driver, self-checker and coverage collector, can be integrated into a simulator using the Robot Operating System (ROS)[1] and Gazebo[2].

In test generation, random (pseudorandom in practice to ensure repeatability) sampling methods allow a wide exploration of the state space [13]. Constraints can be used to focus the test generation to create valid, meaningful and interesting tests [17]; consuming significant engineering effort in practice. Model-based test generation methods can automate the otherwise manual test and constraint writing processes, by exploring a model that reflects the designer's intent and the interactions between the robot, human and environment [3, 4, 2]. These models are analysed in a fully automated manner to construct tests, e.g. through model checking. Controlling the exploration of the model facilitates reaching scenarios that are rarely exercised when using random sampling methods.

Many languages and formalisms have been proposed for model-based test generation in general, e.g., UML and process algebras for concurrency [15], or Lustre and MATLAB/Simulink for data flow [24]). Their suitability for testing code within the HRI domain is yet to be determined. In [3, 4], we constructed probabilistic timed automata (PTA) models for the model checker UPPAAL[3]. Model exploration was achieved by formulating reachability temporal logic properties (e.g., 'eventually the code reaches the terminating states'). In [2], we proposed Belief-Desire-Intention (BDI) agents to model the code and the interactions. BDI agents naturally reflect the agency present in the environment (e.g. people) and the robot's autonomous capabilities.

This paper focuses on comparing BDI agents against other types of formal models for test generation, in terms of practicality, performance, and scalability, within the context of software for robots in the HRI domain:

1. How does the use of BDI agents compare with formal models such as PTA, for model-based test generation in the HRI domain?

2. Are BDI models scalable to generate effective tests for different types of HRI applications?

In this paper we use two case studies to evaluate the practicality, performance and scalability of the proposed BDI-based test generation approach, a human-robot cooperative manufacture (table assembly) task, and a home care assistant scenario. We developed corresponding PTA and BDI models for the code under test, sensors, and interacting humans, based on the work presented in [3, 4, 2], measuring the time overhead of model development, and model exploration setup. Subsequently, we generated tests from the models, to be run in simulation to gather statistics on coverage of the code, the safety and functional requirements (through assertion triggering), and combinations of human-robot actions from all the possibilities in the tasks (cross-product

[1] http://www.ros.org/
[2] http://gazebosim.org/
[3] http://www.uppaal.org/

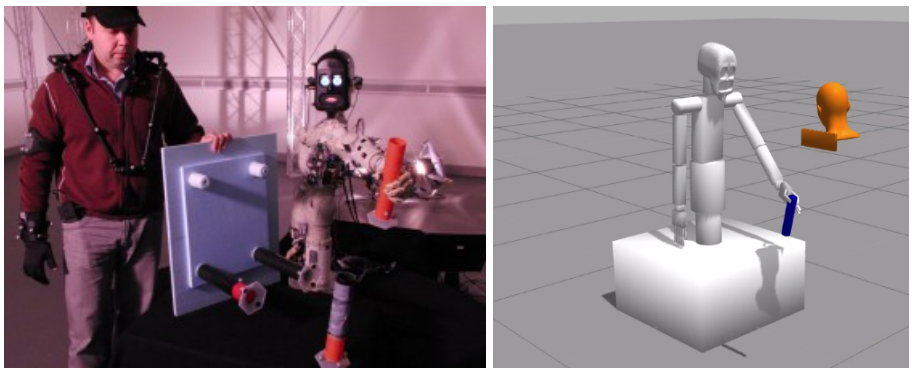

Figure 1: BERT2 robot, table to assemble, and their simulation versions

or situational coverage [1, 4]). Our results demonstrate that BDI agents are effective and scalable models for test generation in the HRI domain. Compared to traditional automata-based test-generation approaches, BDI agents achieve better code coverage, and similar requirement and cross-product coverage. Also, BDI agents are easier to implement and explore than automata models.

# 2 CASE STUDIES

In this section we present the two case studies we will use: a cooperative manufacture scenario and a simple home care assistant scenario.

## 2.1 Cooperative Manufacture Assistant: Table assembly

A table consists of 5 components, 4 legs and the table top, as shown in Fig. 1. The robot, BERT2 [14] (also in Fig. 1), should hand over legs to the person, one at a time, when the person asks the robot. The robot lets go only if it decides the human is ready. It keeps count of the number of legs the person has taken, reporting that a table has been completed if four legs have been handed over within a time threshold.

The handover of the legs is the critical part in this scenario. In a successful assembly, a handover starts with a voice command from the person to the robot, requesting a table leg. The robot then picks up a leg, holds it out to the human, and signals for the human to take it. The human issues another voice command, indicating readiness to take the leg. Then, the robot makes a decision to release the leg or not, within a time threshold, based on a combination of three sensors: "pressure," indicating whether the human is holding the leg; "location," visually tracking whether the person's hand is close to the leg; and "gaze," visually tracking whether the person's head is directed towards the leg.

The sensing combination is the Cartesian product of "gaze", "pressure" and "location" readings, $(g, p, l) \in G \times P \times L$. Each sensor reading is classified into $G, P, L = \{\bar{1}, 1\}$, with 1 indicating the human is ready according to that sensor, and $\bar{1}$ for any other value. If the human is deemed ready, $GPL = (1, 1, 1)$, the robot should decide to release the leg. Otherwise, the robot should not release the leg and discard it. The robot will time out while waiting for either a voice command from the human, or the sensor readings, within specified time thresholds.

A ROS 'node' in Python, with 264 statements, implements the high-level robot control of the assembly task. The code was structured as a finite-state machine (FSM) using the SMACH modules [6]. This allows extracting abstract models of the code, and an efficient implementation of control flow.

### 2.1.1 Requirements List

We considered the following selected set of safety and functional requirements from [2] and the standards ISO 13482 (personal care robots), ISO 15066 (collaborative robots) and ISO 10218 (industrial robots):

1. The robot shall always discard or release a leg within a time threshold, whenever it reaches the sensing stage and determines the human is ready or not (functional).
2. If the gaze, pressure and location sense the human is ready, then a leg shall be released (functional).
3. If the gaze, pressure or location sense the human is not ready, then a leg shall not be released (functional).
4. The robot shall always discard or release a leg, when activated (functional).
5. The robot shall not close its hand when the human is too close (safety).
6. The robot shall start and work in restricted joint speed of less than 0.25 rad/s (safety).

### 2.1.2 ROS-Gazebo Simulator

The ROS-Gazebo simulator, available online[4], includes: *(a)* the robot code under test corresponding to the high-level control that enacts the table assembly protocol; *(b)* control of the human actions (as a Python node) to stimulate the robot for testing; *(c)* sensor models (implemented as Python nodes) for pressure, gaze, location and voice commands; *(d)* physical models of the robot, human and objects in Gazebo; *(e)* other code such as a kinematic trajectory planner.

## 2.2 Home Care Assistant

A Tiago robot, from PAL Robotics[5], operates in the lower floor of a house. This robot has a PMB2 differential drive-base with a laser scanner, a body with a lifting torso, an arm with 7 degrees of freedom and a gripper as the end effector, a head with a camera, a microphone, and a speaker.

In the home care scenario, the robot is in charge of taking care of a person with limited mobility, by bringing food to the table ('feed'), clearing the table ('clean'), checking the fridge door ('fridge'), and checking the sink taps ('sink'). The robot's code is programmed to execute corresponding motion sequences obeying the corresponding commands, formed from assembling other primitives such as 'go to fridge', 'go to table', or 'open the gripper'. Whenever the person asks the robot to execute a command that is not in the list of known ones, the robot will not move. The robot moves to default location, denominated 'recharge', after completing a command, and should remain there until the person asks it to do something else. We assume the person will not ask the robot to perform more than three feasible tasks within a 10 minutes interval.

A small dog cohabits the operational space. To avoid dangerous collisions with the dog, the robot checks the readings from the laser scan and stops if any object is too close (within a proximity of 20 cm). Figure 2 shows the simulated environment and the robot.

A ROS 'node' in Python, with 265 statements, implements the robot actions executed as directed by the human commands. The low level motion in Gazebo is executed within

[4]https://github.com/robosafe/mc-vs-bdi/table
[5]http://tiago.pal-robotics.com/

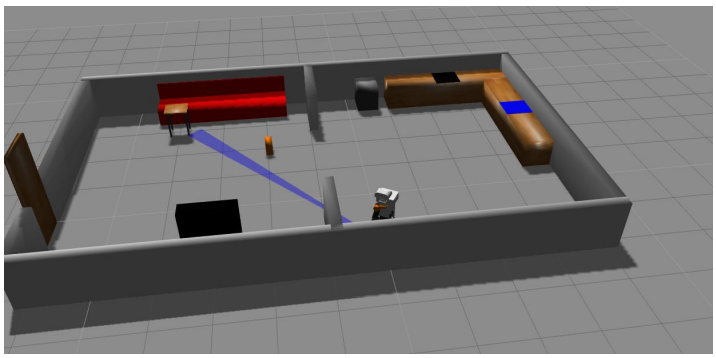

Figure 2: Tiago, and the simulated living space with a dog (orange).

the code, where angle correction, motion towards the goal, and collision avoidance are alternated. The code was also structured as an FSM.

#### 2.2.1 Requirements List

We considered requirements similar to the first scenario, also inspired by the standards ISO 13482, ISO 15066 and ISO 10218:

1. If the robot gets food from the fridge, it shall eventually bring it to the table (functional).
2. If the robot is idle waiting for the next order, it shall go to or remain in the recharge station (functional).
3. The robot shall start and operate in restricted motion speed of less than 0.25 mm/s (safety).
4. The robot shall not stay static in any other location that is not the recharge station for a long period of time (functional).

#### 2.2.2 ROS-Gazebo Simulator

Similarly to the other case study, the ROS-Gazebo simulator, available online[6], includes: *(a)* the robot code under test corresponding to the high-level control that enacts the motion sequences to bring food to the table, clean the table, check the fridge, and check the sink; *(b)* control of the human commands (as a Python node) to stimulate the robot for testing; *(c)* control of the dog's pseudorandom motion (i.e. controlled through a seed); *(d)* a Python node for processing laser sensor readings; *(e)* physical models of the house, robot, and dog, in Gazebo; *(f)* other code downloaded from Tiago's repository[7].

### 2.3 Testbench Components

We implemented a CDV testbench including a test generator, a driver, a checker, and coverage collection, for both case studies, based on our previous work [3, 4, 2].

[6] https://github.com/robosafe/mc-vs-bdi/tiago
[7] http://wiki.ros.org/Robots/TIAGo

| | | |
|---|---|---|
| `tell` | `leg` | Human voice A1 for 5 s |
| `receivesignal` | | Human waits for max. 60 s |
| `tell` | `humanReady` | Human voice A2 for 2 s |
| `set_param` | `location=0` | Move hand to $x + 0.03$, $y - 0.01$ |

Figure 3: Example of an abstract test (LHS) and its concretization (RHS) to indirectly stimulate the robot code under test.

#### 2.3.1 Test Generation

Verifying robot software requires valid, meaningful and realistic orchestration of timed sequences of data inputs that generate interesting scenarios in HRI applications, increasing the complexity of the simpler "valid data problem" [19, 17]. We have proposed a two-tiered test generation process to alleviate the complexity challenge, where abstract tests or timed sequences are generated first, and then valid data is instantiated [3, 4, 2]. Tests stimulate a simulated environment, which indirectly propagates the stimulus to the code under test. In the case of the table assembly task, the tests determine the human actions and also instantiate location, gaze and pressure values. In the case of the home care assistant scenario, the tests choose the commands sent by the human only. An example of a test is shown in Fig. 3, for the table assembly task.

A test can be produced via pseudorandom sampling, constraint solving (concolic), and model search or exploration (model-based), to mention some of the most popular methods. In unconstrained, pseudorandom test generation, all the test inputs for stimulating the code are chosen from sampling their domains, from carefully chosen probability distributions to maximize coverage and fault detection [10]. Constraint solving and search methods have been employed to bias testing towards valid and rare events, and specific kinds of scenarios for the robot systems [13, 17]. Model-based approaches perform biasing in a more systematic and automatic manner: a model is explored to synthesize a test sequence (e.g., one that complies with a property or specification in model checking [9, 20]). For this reason, model-based approaches achieve high coverage (or exploration) of the code and the simulation of an HRI scenario, compared to pseudorandom sampling [3, 4]. Nonetheless, pseudorandom sampling allows finding cases that might not have been found when exploring an abstract model, hence providing a wider variety of tests. We detail our model-based test generation setup with automata and BDI agents in the next section.

#### 2.3.2 Driver

The driver routes each one of the concrete inputs to the respective simulator component.

#### 2.3.3 Checker

We implemented assertion monitors to check each one of the requirements, using the SMACH module in Python [3]. The assertion monitors can refer to abstract events (e.g., the robot moved and the human responded) and quasi-continuous parameters such as the robot's speed, as needed to encode the requirements. If a monitor is triggered, the results of the checks are recorded for coverage collection. The assertions are checked when a change of events occurs, or after every specified $\Delta t$ time interval.

#### 2.3.4 Coverage Collection

We implemented code (statement) coverage over the Python node under test, instrumented via the 'coverage' Python module[8]. We also implemented assertion coverage,

[8] http://coverage.readthedocs.org/en/coverage-4.1b2/

counting if a monitor was triggered per test run [3]. Ideally, all the assertions must be covered by our test suite.

Finally, we implemented cross-product or Cartesian product coverage over abstract actions of the robot and its environment ($Human \times Robot$), deemed to be fundamental for the scenarios; for example, combinations of requesting 1 to 4 legs, and the robot deciding to release all, some or none in the manufacture task. Some of these combinations should not be feasible, such as the human requesting a leg but not being ready and the robot choosing to release a leg. Encountering infeasible combinations during testing provides evidence of functional failures. Ideally, each feasible $Human \times Robot$ combination from our defined set should be encountered (covered) at least once with our test suite.

# 3 MODEL-BASED TEST GENERATION

In model-based testing, a model of the system under test or the testing goals is derived first, followed by its traversal to generate tests from its paths [24]. In this section, we describe two types of model-based test generation, using BDI agents [2], and PTA model checking [3, 4].

## 3.1 BDI Agent Models and Exploration

BDI is an intelligent or rational agent architecture for multi-agent systems. BDI agents model human practical reasoning, in terms of 'beliefs' (knowledge about the world and the agents), 'desires' (goals to fulfil), and 'intentions' (plans to execute in order to achieve the goals, according to the available knowledge) [5]. Recently, we have shown that BDI agents are well suited to model rational, human-like decision making in HRI, for test generation purposes [2].

We employed the Jason interpreter, where agents are expressed in the AgentSpeak language. An agent is defined by initial beliefs (the initial knowledge) and desires (initial goals), and a library of 'plans' (actions to execute to fulfil the desires, according to the beliefs). A plan has a 'head', formed by an expression about beliefs (a 'context') serving as a guard for plan selection, and a 'body' or set of actions to execute. New beliefs appear during the agents' execution, triggered (sent) by other agents, or by the execution of plans (self-beliefs) [5].

We model an HRI scenario using a set of BDI agents, representing the robot's code, sensors, actuators, and its environment (e.g. human, dog). Then, we add BDI verification agents that control the execution of the HRI agents, by triggering (sending) beliefs to activate plans and create new desires in the other agents. We manually select a set of beliefs for the verification agents to send to other agents, every time we run the multi-agent system. Each system execution with a different set of beliefs will activate a corresponding sequence of plans in the agents, and thus a test comprising a sequence of actions. We extract only the environment components of the resulting sequence, creating an abstract test that activates the environment to indirectly stimulate the robot's code in simulation. This structure with the HRI and verification agents is shown in Fig. 4.

## 3.2 PTA Models and Model Checking

Model checking is the exhaustive exploration of a model to determine if a logical property is satisfied or not. Examples or counterexamples can be provided as evidence of the satisfaction or violation proof. Model checking has been proposed as a model-based method for test generation, using these examples or counterexamples as tests, from a model of code [3, 4] or high-level system functionality [25].

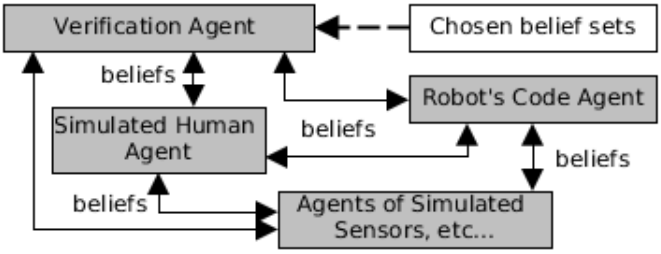


Figure 4: BDI agents (grey) and HRI in the verification setup.

In UPPAAL, we start by formalizing a model of the HRI in terms of PTA. The non-deterministic component allows modelling uncertainty in the human actions, and sensor errors. As robots interacting with humans are expected to fulfil goals in a finite amount of time, the timing counters in the PTA allow emulating these timing thresholds. Consequently, we construct PTA for the robot's code, sensors, actuators, and its environment, as we did with BDI agents. The execution of these automata are synchronized by communication events, and 'guards' or conditions to transition from a state to another, according to system variables and events. Our models are abstractions of the actual HRI to manage the state space.

To derive tests from the PTA, logical properties are formalized using TCTL (Timed Computation Tree Logic), and automatically checked by the UPPAAL model checker. For example, we would specify properties where 'eventually the robot reaches the specified location', or 'eventually the interaction is completed', for our case studies. Formulating the right properties to achieve high model coverage requires a good understanding of formal logics and the PTA model. Once model checking takes place, UPPAAL produces an example if the property is satisfied, comprising sequences of states from all the PTA combined. We exclude the robot code contribution from the sequences to obtain abstract tests that indirectly stimulate the robot's code in simulation.

# 4 EXPERIMENTS AND RESULTS

We tested the robot's code in both scenarios, the table assembly task and the home care assistant, to verify their respective requirements. We generated tests from pseudorandom sampling and the model-based methods described in Section 3. Coverage was collected according to the models in Section 2.3.4.

## 4.1 Experimental setup

The simulator and testbench were implemented in ROS Indigo and Gazebo 2.2.5. The tests were run on a PC with Intel i5-3230M 2.60 GHz CPU, 8 GB of RAM, running Ubuntu 14.04. We used Jason 1.4.2 for the BDI models. We used UPPAAL 4.1.19 for model checking. All the related data (models, tests and results) is openly available online[9].

### 4.1.1 Cooperative Manufacture Assistant

Firstly, we generated 160 tests from pseudorandom sampling the set of all possible human actions a number of times chosen by a seed.

[9]https://github.com/robosafe/mc-vs-bdi/results

With the model checking method in Section 3.2, we generated 91 valid (satisfied) TCTL properties, for which example traces were produced automatically, and abstract test were extracted. These properties covered combined robot and human actions to set up the gaze, pressure and location sensor readings, such that $GPL = (1, 1, 1)$ for all the leg requests, $GPL \neq (1, 1, 1)$ in at least one of the leg requests, requesting 1 to 4 or no legs, the human being bored, and the robot timing out while sensing or while waiting for a signal. We cloned 69 of these for a total of 160 abstract tests.

With the BDI method, we generated 131 abstract tests by manually selecting belief sets from a possible total number of $2^{38}$, following the procedure explained in Section 3.1. We covered the human actions to set up the gaze, pressure and location sensor readings, such that $GPL = (1, 1, 1)$ for all the leg requests, $GPL \neq (1, 1, 1)$ in at least one of the leg requests, requesting 1 to 4 or no legs, and getting bored. We cloned 29 of the abstract tests, selected at random, for a total of 160 tests.

Each test ran for a maximum of 300 seconds.

#### 4.1.2 Home Care Assistant

We generated 50 tests from pseudorandom sampling the set of all possible human actions a number of times chosen by a seed.

With the model checking method, we generated 23 TCTL properties and the consequent abstract tests. We covered combinations of the human requesting feeding, cleaning, checking the fridge, checking the sink, and any other invalid order, for 1 to 3 times, with the corresponding robot actions, and also the robot being idle. We cloned these 23 tests at least once for a total of 50 abstract tests.

With the BDI agents, we generated 62 abstract tests by sampling belief sets from a possible total number of $2^5$. We covered combinations of the human requesting feeding, cleaning, checking the fridge, checking the sink, and any other invalid order, for 1 to 3 times. We discarded 12 repeated tests to get a total of 50.

Each test ran for a maximum of 700 seconds.

### 4.2 Code Coverage Results

Figures 5 and 6 show the code coverage percentage reached by each produced test, and the accumulated coverage, for both scenarios. It is clear that model-based methods produce tests that reach high levels of code coverage faster, compared to pseudorandom test generation. Additionally, when comparing both model-based approaches, we observed that the BDI-based approach reached the highest percentages of coverage (more than once out of 160 tests for the table assembly scenario), despite us diligently specifying TCTL properties that we thought would achieve similar results.

### 4.3 Assertion Coverage Results

The assertion coverage results are shown in Table 1 for both scenarios. We recorded the number of tests where the requirement was satisfied (P), not satisfied (F) or not checked (NC) for Reqs. 1 to 3 and 5 in the table assembly, and 1 to 4 in the home care scenario. Req. 4 in the table assembly is only triggered if the robot is activated, thus the NC column result indicates tests where this did not happen. Req. 6 is triggered only if a violation has happened, hence the NC column result indicates that a violation did not occur.

The results for the cooperative assembly scenario indicate that Reqs. 1 to 3 are violated by the system, as the robot occasionally fails to make a decision to release or not release a leg within the specified time threshold. These failures came to light mostly from the model-based tests' results, again showing the advantages of model-based test

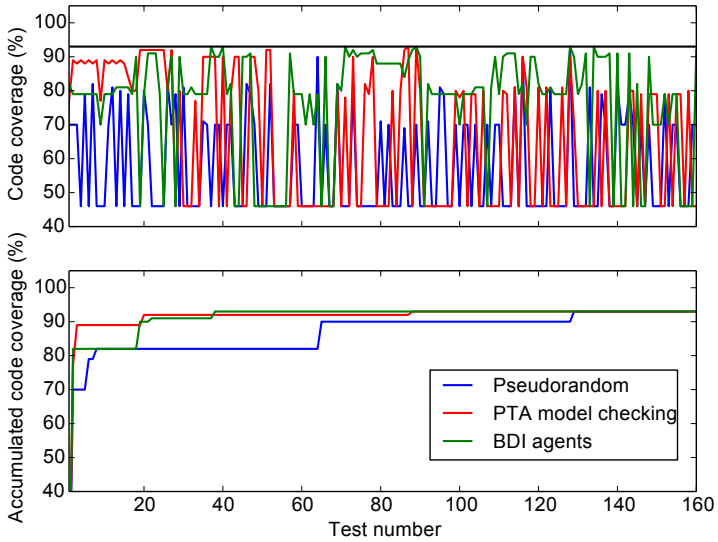


Figure 5: Code coverage results for the cooperative assembly assistant.

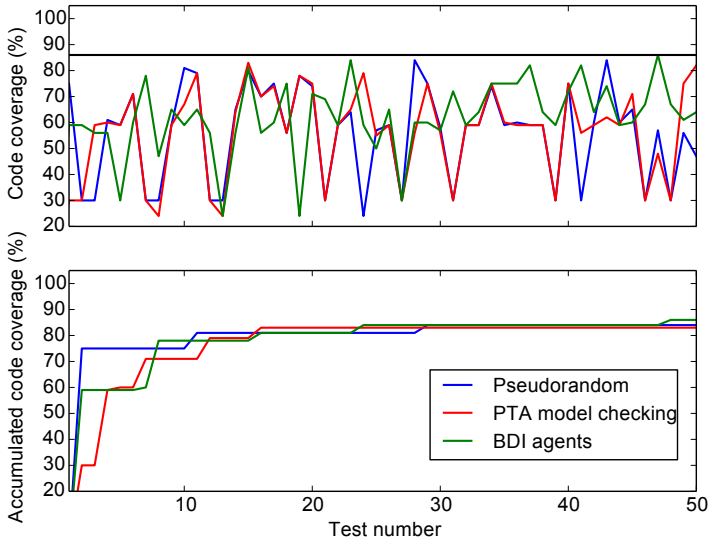


Figure 6: Code coverage results for the home care assistant.

generation over pseudorandomly generated tests. Req. 5 is also violated occasionally, as the person's hand is allowed to get close to the robot gripper when it closes. To improve this issue, the person's hand would need to be tracked, and the robot gripper stopped accordingly. Req. 4 is observed in nearly half of the model-based tests. Finally, Req. 6, inspired by the standard ISO 10218-1 for industrial robots, is satisfied in all tests. For this scenario, more faults in our assumptions about the system's functionality and requirement violation were found through model-based test generation methods than with pseudorandom test generation.

For the home care scenario, requirement violations have been observed with all three test generation methods. If the robot collides with the dog, the collision causes the

Table 1: Requirement (assertion) coverage

| Req. | Pseudorandom | | | PTA model checking | | | BDI agents | | |
|---|---|---|---|---|---|---|---|---|---|
| | P | F | NC | P | F | NC | P | F | NC |
| Cooperative Manufacture Assistant (160 tests per method) | | | | | | | | | |
| 1 | 19 | 4 | 137 | 37 | 50 | 73 | 45 | 16 | 99 |
| 2 | 3 | 0 | 157 | 35 | 10 | 115 | 31 | 5 | 124 |
| 3 | 24 | 0 | 136 | 97 | 0 | 63 | 63 | 2 | 95 |
| 4 | 57 | – | (103) | 99 | – | (61) | 72 | – | (88) |
| 5 | 33 | 31 | 96 | 128 | 2 | 30 | 79 | 4 | 77 |
| 6 | – | 0 | (160) | – | 0 | (160) | – | 0 | (160) |
| Home Care Assistant (50 tests per method) | | | | | | | | | |
| 1 | 16 | 7 | 27 | 16 | 7 | 27 | 17 | 6 | 27 |
| 2 | 19 | 7 | 24 | 19 | 7 | 24 | 21 | 7 | 22 |
| 3 | 38 | 11 | 1 | 38 | 11 | 1 | 40 | 8 | 2 |
| 4 | 13 | 36 | 1 | 13 | 36 | 1 | 10 | 38 | 2 |

robot to fall over without recovery, which will prevent the robot from succeeding. This is reflected by the failure of Reqs. 1, 2 and 4. The scenario could be improved by providing self-localization and mapping (SLAM) capabilities, along with a predictive collision avoidance mechanism against dynamic entities such as the dog. Req. 3 is not satisfied as a velocity limit is not enforced in the motion of the base. In this scenario, the requirement coverage performance was quite similar for all the three methods.

## 4.4 Cross-Product Functional Coverage Results

Table 2 shows the coverage results for reachable combinations $Human \times Robot$ of interest. The considered environment (human) actions for the table assembly task were: requesting 1 to 4 legs or none, and being bored before sending the second signal. Correspondingly, the robot's actions for the table assembly comprised: releasing legs if the sensors read $GPL = (1, 1, 1)$, discarding legs if the sensors read $GPL \neq (1, 1, 1)$, and timing out. The considered human actions for the table assembly task comprised: requesting food on the table, cleaning the table, checking the fridge and checking the sink, at least once; requesting food on the table or cleaning the table at least twice; requesting some other invalid action and no valid actions; and requesting invalid actions at least once in combination with valid actions. The home care robot's actions comprised the activation of the corresponding action sequences.

The results for the cooperative assembly scenario using pseudorandom test generation show the difficulty of reaching some of the coverage points, due to the complexity of the interaction protocol to activate the robot. In this scenario, model checking slightly outperforms BDI agents, as TCTL properties were derived explicitly to cover each one of the rows in the table at least once. In the home care scenario, the coverage performance was similar for all the three methods, as the human action space is small and this increases the likelihood of generating a diversity of effective tests.

Table 2: Reachable cross-product coverage

| | *Human* | *Robot* | Pseudorandom | PTA model checking | BDI agents |
|---|---|---|---|---|---|
| Cooperative Manufacture Assistant (160 tests per method) | | | | | |
| 1 | 4 legs | $GPL = (1, 1, 1)$ for at least 1 leg | 0 | 24 | 24 |
| 2 | 4 legs | $GPL \neq (1, 1, 1)$ for at least 1 leg | 0 | 30 | 31 |
| 3 | 4 legs and/or bored | Timed out | 2 | 43 | 32 |
| 4 | 3 legs | $GPL = (1, 1, 1)$ for at least 1 leg | 0 | 20 | 12 |
| 5 | 3 legs | $GPL \neq (1, 1, 1)$ for at least 1 leg | 7 | 38 | 22 |
| 6 | 3 legs and/or bored | Timed out | 10 | 38 | 27 |
| 7 | 2 legs | $GPL = (1, 1, 1)$ for at least 1 leg | 2 | 13 | 5 |
| 8 | 2 legs | $GPL \neq (1, 1, 1)$ for at least 1 leg | 6 | 28 | 10 |
| 9 | 2 legs and/or bored | Timed out | 9 | 11 | 7 |
| 10 | 1 leg | $GPL = (1, 1, 1)$ | 14 | 11 | 9 |
| 11 | 1 leg | $GPL \neq (1, 1, 1)$ | 10 | 14 | 14 |
| 12 | 1 leg | Timed out | 10 | 2 | 1 |
| 13 | 1 to 4 legs | Timed out | 72 | 38 | 75 |
| 14 | No leg | Timed out | 62 | 2 | 3 |
| Home Care Assistant (50 tests per method) | | | | | |
| 1 | At least 1 *feed* | At least 1 *feed* | 5 | 9 | 6 |
| 2 | At least 1 *clean* | At least 1 *clean* | 14 | 3 | 14 |
| 3 | At least 1 *fridge* | At least 1 *fridge* | 4 | 12 | 4 |
| 4 | At least 1 *sink* | At least 1 *sink* | 8 | 2 | 9 |
| 5 | At least 2 *feed* or *clean* | At least 2 *feed* or *clean* | 5 | 22 | 5 |
| 6 | Other commands | Idle | 13 | 2 | 12 |

## 4.5 Discussion

### 4.5.1 Exploration

Answering our first research question, the coverage results in this section demonstrate that BDI agents perform as well as traditional model checking in model-based test generation. Furthermore, BDI agents outperformed model checking in terms of code coverage. A clear advantage of model checking over BDI agents is the possibility of specifying cross-product coverage points as TCTL properties. This generates specialized "directed" tests to cover a specific coverage target at the expense of the manual effort that needs to be invested into formalizing the properties (further addressed in Sections 4.5.2 and 4.5.3). The advantage of using BDI agents, however, is a higher degree of automation and thus far less manual effort to achieve coverage-directed test generation [23]. Beliefs in the agents can be triggered to target coverage goals, automated further by adding learning-based feedback loops, as we demonstrated in [2].

### 4.5.2 Performance

A comparison of the time to craft the different models, and the time it took to run them to produce the tests, is shown in Table 3. The automata construction in UPPAAL required a longer effort than constructing the BDI agents in Jason, as variables and related guards need to be constructed carefully to reflect the constraints in the actions of an HRI protocol. On the other hand, the syntax of BDI agents offers a more rational structure, where an HRI protocol is easier to construct and thus takes less time. Also, constructing the properties in TCTL for model checking increased the modelling time effort, whereas specifying BDI agent belief sets was a much easier task. Finally, we limited the running time of the BDI model manually, but it could have been further reduced based on the coverage results achieved. However, the model checking time was not under our control, in some cases being quite high compared to exploring the BDI agents. It also varied significantly depending on the properties.

Table 3: Performance of the model-based test generation methods

| | PTA model checking | BDI agents |
|---|---|---|
| Cooperative Manufacture Assistant | | |
| Model's lines of code | 725 | 348 |
| Modelling time | $\approx$ 10.5 hrs | $\approx$ 6 hrs |
| Model explor. time (min/test) | 0.001 s | 5 s |
| Model explor. time (max/test) | 33.36 s | 5 s |
| Home Care Assistant | | |
| Model's lines of code | 722 | 131 |
| Modelling time | $\approx$ 5.5 hrs | $\approx$ 3 hrs |
| Model explor. time (min/test) | 0.001 s | 1 s |
| Model explor. time (max/test) | 2.775 s | 1 s |

#### 4.5.3 Practicality and Scalability

Our results demonstrate that BDI agents are applicable to different HRI scenarios, as exemplified by our two case studies. Hence, we positively answere the second research question. BDI agents model an HRI task with human-like actions and rational reasoning. They are natural to program, by specifying plans of actions. Compared to model checking, we do not need to formulate temporal logic reachability properties, which requires a good understanding of formal logics, and a greater amount of manual effort. In addition, constructing automata, such as PTAs, for larger case studies requires several cycles of abstraction to manage the state-space explosion problem [25].

#### 4.5.4 Limitations

In this paper, the two case studies serve to illustrate our comparison of using BDI agents, instead of model-checking PTAs, for model-based test generation. Both could be extended to obtain comprehensive, richer scenarios in the future. The coverage results in the simple home care scenario are similar with the three test generation methods. This indicates that if the HRI scenarios are not complex enough, then any test generation method is likely to reach acceptable coverage results. Additionally, all the approaches presented in this paper deal with offline test generation, i.e. the tests are computed before the simulation is running, assuming that the models of the system and the environment do not change. Hence, our next step is to extend both our model-based test generation approaches to more complex systems with agency and change.

## 5 RELATED WORK

Test generation methods in robotics have been applied to relatively small sets of data types, e.g. a timing sequence for controllers [17], or a set of inputs for a controller [13]. For these applications, model-based test generation approaches cannot offer clear advantages. When testing a full robot system, however, the orchestration of different timed sequences is more complex and requires model-based approaches such as those presented in this paper.

Within model-based testing, several challenges remain for the HRI domain. In some approaches, testing is performed over a highly abstracted model of the robot code, used

for test generation [15], which lacks the detail of testing the real code in simulation [3, 4, 20]. Other approaches for test generation require the construction of formal models such as hybrid [12, 7] or timed [9, 20] automata, which requires considerable manual effort if automated approaches to extract models are not available. After modelling, abstraction is needed to reduce the model size to a manageable one for model checking [20].

Evidently, many approaches do not consider testing the software's functionality with respect to its environment. HRI and challenging environments in robotics can be better represented by other types of models for test generation, as noted by [18], proposing Markov chains and other probabilistic models, and [2], proposing BDI agents.

A multi-agent framework has been proposed in [11] for model-based testing code. Agent programs explore a UML model to generate all the scenarios of the if-then-else conditions and branches. In this paper, we followed the principles from [2] to use BDI agents in the test environment, rather than to support the robot's decision making, for intuitive and effective test generation.

# 6 CONCLUSIONS

In this paper, we presented a comparison of two kinds of model-based test generation approaches in the context of HRI scenarios: model checking (for PTA models) and BDI agents, in terms of exploration (code, assertion, and functional coverage), performance, practicality and scalability. We also compared both methods to pseudorandom test generation. The three test generation methods were applied to two case studies, a cooperative table assembly task, and a home care scenario, for which high-level robot control code was tested in respective ROS-Gazebo simulators coupled with coverage-driven automated testbench components [3, 4].

BDI agents allow more realistic, human-like stimulus, whilst simultaneously facilitating the generation of interesting events, to gain good coverage levels, both of the code under test, as well as assertion and functional coverage models. BDI agents perform better (in terms of code coverage) and similarly as the compared model checking method (in terms of requirement and cross-product coverage), and better than pseudorandom test generation (in all types of coverage). Also, BDI agents are easier to specify, and computationally cheaper to execute than model checking. Hence, our results clearly highlight the potential of using BDI agents for test generation, stimulating the code according to realistic scenarios, from complex and detailed models of the environment and the robot.

## 6.1 Future Work

We plan to investigate more efficient exploration methods for the BDI agent model to obtain tests for high and varied coverage. Also, we will apply BDI-based test generation to complex systems with agency and change.

## Acknowledgement

This work was supported by the EPSRC grants EP/K006320/1 and EP/K006223/1, part of the project "Trustworthy Robotic Assistants".

# References


[1] Rob Alexander, Heather Hawkins, and Drew Rae. Situation Coverage – A Coverage Criterion for Testing Autonomous Robots. Technical report, Department of Computer Science, University of York, 2015.

[2] D. Araiza-Illan, A.G. Pipe, and K. Eder. Intelligent agent-based stimulation for testing robotic software in human-robot interactions. In *Proceedings of the 3rd Workshop on Model-Driven Robot Software Engineering (MORSE)*, 2016.

[3] D. Araiza-Illan, D. Western, K. Eder, and A. Pipe. Coverage-driven verification — an approach to verify code for robots that directly interact with humans. In *Proc. HVC*, pages 1–16, 2015.

[4] D. Araiza-Illan, D. Western, K. Eder, and A. Pipe. Systematic and realistic testing in simulation of control code for robots in collaborative human-robot interactions. In *Proc. TAROS*, 2016.

[5] R.H. Bordini, J.F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley, 2007.

[6] Jonathan Boren and Steve Cousins. The SMACH High-Level Executive. *IEEE Robotics & Automation Magazine*, 17(4):18–20, 2010.

[7] T. Dang and T. Nahhal. Coverage-guided test generation for continuous and hybrid systems. *Form Methods Syst Des*, 34:183–213, 2009.

[8] K.I. Eder, C. Harper, and U.B. Leonards. Towards the safety of human-in-the-loop robotics: Challenges and opportunities for safety assurance of robotic co-workers. In *Proc. IEEE ROMAN*, pages 660–665, 2014.

[9] Juhan Ernits, Evelin Halling, Gert Kanter, and Jüri Vain. Model-based integration testing of ROS packages: a mobile robot case study. In *Proc. ECMR*, pages 1–7, 2015.

[10] M. Gaudel. Counting for random testing. In *Proc. ICTSS*, pages 1–8, 2011.

[11] M.S. Geetha Devasena and M.L. Valarmathi. Multi agent based framework for structural and model based test case generation. *Procedia Engineering*, 38:3840–3845, 2012.

[12] A. A. Julius, G. E. Fainekos, M. Anand, and G. J. Pappas I. Lee. Robust test generation and coverage for hybrid systems. In *Proc. HSCC*, pages 329–342, 2007.

[13] J. Kim, J. M. Esposito, and R.V. Kumar. Sampling-based algorithm for testing and validating robot controllers. *International Journal of Robotics Research*, 25(12):1257–1272, 2006.

[14] A. Lenz, S. Skachek, K. Hamann, J. Steinwender, A.G. Pipe, and C. Melhuish. The BERT2 infrastructure: An integrated system for the study of human-robot interaction. In *Proc. IEEE-RAS Humanoids*, pages 346–351, 2010.

[15] R. Lill and F. Saglietti. Model-based testing of autonomous systems based on Coloured Petri Nets. In *Proc. ARCS*, 2012.

[16] Z. Micskei, Z. Szatmári, J. Oláh, and I. Majzik. A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In *Proc. KES-AMSTA*, pages 504–513, 2012.

[17] M. Mossige, A. Gotlieb, and H. Meling. Testing robot controllers using constraint programming and continuous integration. *Information and Software Technology*, 57:169–185, 2014.

[18] Akbar Siami Namin, Barbara Millet, and Mohan Sridharan. Fast abstract: Stochastic model- based testing for human-robot interaction. In *Proc. ISSRE*, 2010.

[19] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):1–29, 2011.

[20] Andrea Patelli and Luca Mottola. Model-based real-time testing of drone autopilots. In *Proc. DroNet*, 2016.

[21] S. Petters, D. Thomas, M. Friedmann, and O. von Stryk. Multilevel testing of control software for teams of autonomous mobile robots. In *Proc. SIMPAR*, 2008.

[22] T. Pinho, A. P. Moreira, and J. Boaventura-Cunha. Framework using ROS and SimTwo simulator for realistic test of mobile robot controllers. In *Proc. CONTROLO*, pages 751–759, 2014.

[23] Andrew Pizialli. *Functional Verification Coverage Measurement and Analysis*. Springer, 2008.

[24] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22:297–312, 2012.

[25] Matt Webster, Clare Dixon, Michael Fisher, Maha Salem, Joe Saunders, Kheng Lee Koay, and Kerstin Dautenhahn. Formal verification of an autonomous personal robotic assistant. In *Proc. AAAI FVHMS 2014*, pages 74–79, 2014.