Scenario-based Automated Evaluation of Test Traces of Autonomous Systems

Gergő Horányi, Zoltán Micskei, István Majzik

Budapest University of Technology and Economics,
Dept. of Measurement and Information Systems, Budapest, Hungary
[horanyi.gergo,micskeiz,majzik]@inf.mit.bme.hu

Abstract. Testing the robustness and safety of autonomous systems (like domestic or manufacturing robots) is a challenging task since these systems can make decisions on their own depending on their environment. We proposed a model based testing approach to capture the context and basic safety-related behavioural requirements of such systems, and to generate test data representing stressful contexts. During the execution of these tests in a real or simulator based test environment, the captured test traces shall be checked by comparison with the requirements in order to detect the violation of any requirement in each situation. In this paper we analyse this test evaluation problem and propose a method that can be used for efficient comparison.

Keywords. Robustness testing, test evaluation, test comparator, test coverage

1 Introduction

Autonomous systems can make and execute decisions to achieve their goals without direct human control [1]. A significant part of these systems, for example autonomous robots used in the household or manufacturing, operate in real, uncontrolled environment, thus they must properly react to unexpected combinations of environmental objects and events: they shall be *robust* to be capable of correctly handling unforeseen situations and *safe* to avoid harmful effects with respect to humans. This way the evaluation of their robustness (precisely, the degree to which they can function correctly in the presence of invalid inputs or stressful environmental conditions) and functional safety forms an important part of their verification and validation.

To support the testing of the robustness and safety of the context-aware behaviour of autonomous robots, in the R3-COP project [2] we developed a model based testing concept [3]. It is characterized by three main components: (1) context and requirements modelling to represent test requirements, (2) a search based test generation method to derive stressful contexts for robustness testing, and (3) an automated off-line test evaluation approach. We focused on the systematic generation of the stressful contexts that can be derived from the context model and the behavioural requirements, in order to satisfy robustness related test goals and related coverage criteria.

Having these test contexts generated, the tests can be executed in a real or simulator based environment: each generated test context is set as initial context of execution, then the autonomous system is started to perform its mission, and the events, actions and context changes are recorded in a *test trace* until the mission is ended.

According to this testing concept, the evaluation of the test traces has several challenges. Each requirement shall be mapped to a test oracle that is able to compare the sequence of contexts, events and actions recorded in a test trace to the ones allowed by the requirement. Since the test contexts represent complex situations that may match several requirements, each test trace shall be compared to each requirement. Moreover, this comparison shall be started in each step of the test trace (even in an overlapped way) since it may happen that during a test execution an initial context of another requirement evolves. The comparison of context objects and their properties shall take into account the hierarchy of object types and the abstract relations (like "tooClose" and "near" relations mapped to physical distances) that are included in the context model and in the requirements.

To address these challenges, we propose solutions (algorithms and tools) to solve the *test evaluation problem* in an efficient way: for context matching, we use a graph based algorithm that is optimized for matching multiple graphs at the same time, and adapt it to hierarchical context models with multiple valuations. To present our solution, first we provide a brief overview of our testing framework (Section 2), and then introduce the running example used in the paper (Section 3). The problems of the test evaluation are analysed and the basic ideas of our solution are presented in Section 4. The main parts of our test evaluation approach, the context matching and the scenario matching algorithms are described in Section 5. The properties of the algorithms are assessed in Section 6.

2 Overview of the Testing Framework

Our robustness testing framework that supports both test generation and test evaluation was published in [3]. This section presents a short overview in order to put the test evaluation problem tackled by the current paper in context. The testing framework (Fig. 1) consists of manual and tool-supported activities. As the behaviour of autonomous robots is context-aware (i.e., it depends not only on the commands the robot receives, but also on the perceived state of its environment), the framework focuses on generating test data that can be used to evaluate the *control module of the robot* in complex situations. Note that the control module of one robot is considered as system under test (SUT), everything else is treated as its environment.

In the first steps of our testing process, the context (the environment of the SUT), the events (the inputs the SUT can receive from its perception components), and the actions (the operations the SUT can execute using its actuators) are modelled. Then on the basis of these models the safety related requirements of the SUT are captured in graphical scenarios with the help of domain experts.

The context of the robot is represented by a *context metamodel* that includes the types of environment objects (including dynamic objects that may appear, disappear

or move), their properties, relations and constraints. The test requirements are captured using a language that combines a *context view* (on the basis of the context metamodel) with a *scenario based behaviour specification* (using a limited subset of UML2 Sequence Diagram elements to refer to the sequence of events and actions of the SUT). According to our goals, each test requirement formalizes a basic rule with respect to the expected safe behaviour of the SUT: it fixes an *initial context fragment* (that captures a condition regarding the initial context of the robot using a relevant combination of objects, properties and relations from the context metamodel), a sequence of initial events, actions and interim contexts that form the *trigger (condition) part* of the behaviour, and finally an *assert part* that specifies the events, actions and final context that shall (or shall not) happen after the trigger part occurred. For example, a test requirement may specify that in case of an approaching human in a room (specified in the initial context), when she/he is detected in a dangerous area (trigger part), the robot shall issue a sound alarm (assert part).

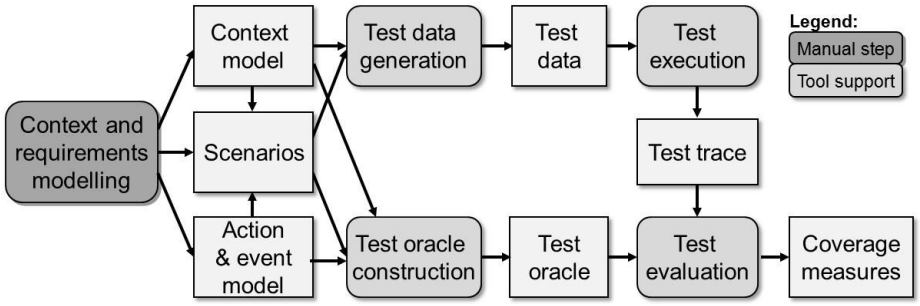

Fig. 1. Overview of the robustness testing framework

The test data generation is a systematic search (using metaheuristics [4]) for test contexts that satisfy our testing goals: (1) extension of the initial context fragments with extra environment objects to check the behaviour of the robot in case of unexpected objects, (2) systematic combination of initial context fragments from the requirements to check the behaviour in complex situations, and (3) generation of combinations of objects that violate semantic constraints (with respect to property values, multiplicity of objects etc.) to check the behaviour in these stressful situations.

Next, the SUT is observed in each of these environments (in a simulated or a real setup), and execution traces of the SUT are collected and evaluated against the requirements using the method presented in this paper.

3 Running example

A simplified household vacuum cleaner robot will be used as a running example throughout the paper. This section gives an overview of the context and requirement models created for the example in order to help understand the test evaluation problem and our proposed solutions.

Fig. 2 illustrates part of a context metamodel that can be created for our example robot. The metamodel contains concepts representing the *static objects* in the robot's environment (e.g., Furniture or Human) and *dynamic context events*, namely AppearEvent, DisappearEvent, and MoveEvent that can be associated with the objects that are concerned (e.g. specifying that an object will appear during the execution). Instances of the context metamodel form the test data generated by the testing framework.

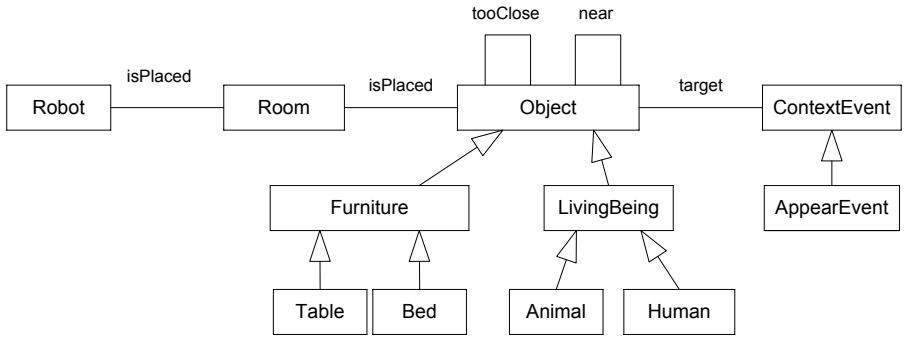

Fig. 2. Part of a context metamodel

The metamodel may include *abstract relations* that represent partitions of concrete property values relevant for the requirements (e.g., the *near* association between two Objects in the example represents a partition of the concrete distance). Once test data are selected, a post-processing creates concrete test data by selecting values (for the undefined properties in the model) that satisfy the abstract relations.

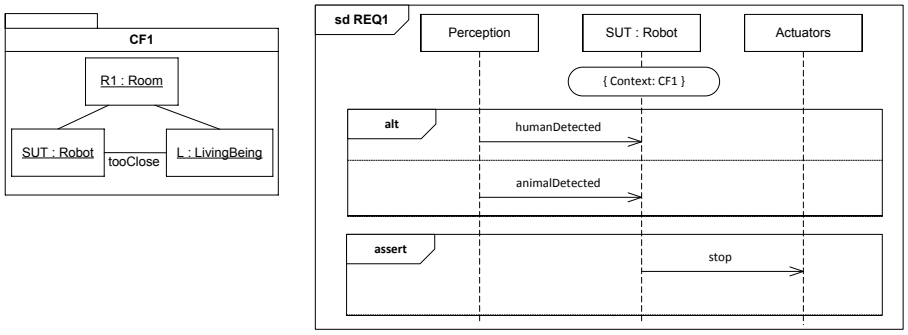

Fig. 3. Example scenario with a context fragment

Fig. 3 and Fig. 4 depict example requirements. Requirements capture what events the control component of the robot can receive from its sensors, and what actions should it send in response to the actuators. In the first requirement, the context fragment on the left hand side is referenced in the beginning of the event view's sequence

diagram, making it its initial context fragment. The trigger part of the scenario consists of an *alternate* construct with the *humanDetected* or *animalDetected* events, and the assert part (the mandatory behaviour that shall happen once the trigger part occurred) is sending the *stop* action.

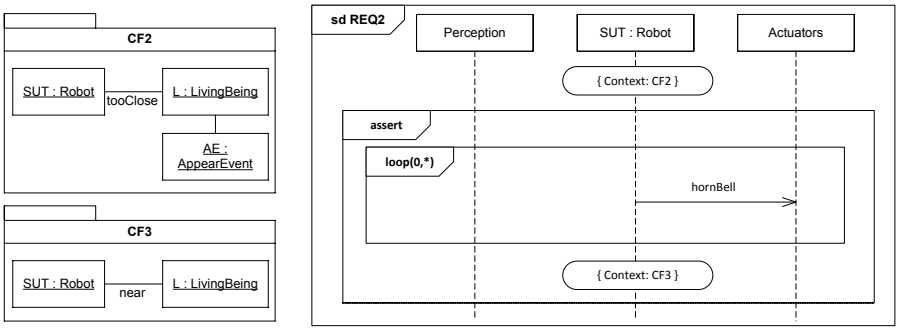

Fig. 4. Example scenario with a final context

To make the evaluation of traces with respect to the scenarios possible, an operational semantics of the language was defined [5]. It allows the construction of an *observer automaton* for the whole scenario, where the SUT receives events and sends actions, see Fig. 5.

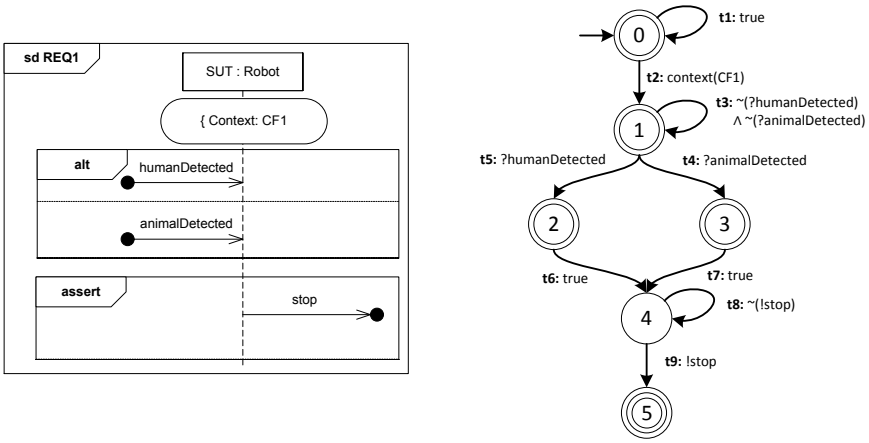

Fig. 5. Observer automaton generated from the scenario

The transitions of the observer automaton are labelled with guards that refer to context fragments, events and actions that are allowed by the requirement scenario. *Trivial accepting states* (denoted with double circles) mark that the processing is still inside the trigger part of the scenario, thus if a trace stops here then the requirement is neither satisfied nor violated (i.e., it is inconclusive). *Reject states* (denoted with sin-

gle circles) are inside the assert part; stopping here means that some mandatory behaviour is still missing, thus the requirement is violated. Finally, *stringent accepting states* (denoted with triple circles) represent that the trace successfully reached the end of the assert part, thus the requirement was triggered and satisfied.

4 The Test Evaluation Problem

According to our test strategy, each segment of the test trace captured during the test execution shall be evaluated against each of the requirements to identify whether any requirement is violated during the test. The observer automaton belonging to a requirement is used as a *test oracle*. Matching is examined in each step of the test trace, and depending on the state of the observer automaton where the matching stops, inconclusive, violated and satisfied requirements are collected.

The two subtasks of matching are related to the two views included in the requirements: context and event/action sequences. In case of matching the contexts, the changes represented by *dynamic events* need special handling. In case of an AppearEvent or a DisappearEvent, its 'timing' property determines the relative time when the specified change in the context shall occur, and the matching can take this time into account by dynamically (i.e., "on-the-fly" during the matching procedure) updating the context that shall be matched. In case of MoveEvents, we assume that interim context fragments are used to specify when a moving object becomes relevant from the point of view of the requirement (e.g., a moving human reaches the dangerous area). According to these considerations, sequence(s) of static contexts fragments can be derived from each requirement, and these precisely include the occurrences of the dynamic events given in the initial context fragment.

This way we can formulate the context matching problem in a more abstract way as follows. Context fragments in requirements, as well as context configurations recorded in a test trace are instances of the context metamodel. Instances of a metamodel can be commonly represented as labelled graphs: objects are mapped to graph vertices (where vertex labels determine the type of the object), and the relations among them are mapped to graph edges (where edge labels determine the type of the relation). This way the sequences of context fragments from requirements are represented as so-called *requirement graph sequences*, while the configurations in the test trace are represented as *configuration graph sequences*.

In the following, let us analyse the main challenges of the test evaluation, and present the basic ideas of our solutions.

- *Matching all requirements from each step of the test trace*: To check potential violations of any requirement in each segment of the test trace, matching of each requirement shall be examined (by trying to match first its initial context fragment) in each step of the test trace. Moreover, in each step the requirements that were already partially matched in the previous steps, shall be checked for progress (continuation or failure of the matching). Accordingly, a configuration graph (representing a configuration in a given step of the test trace) shall be matched to multiple requirement graphs. To solve this problem, we use a graph matching algorithm

that is optimized for matching multiple graphs: the requirement graphs that are to be checked for matching a configuration graph in a given step are represented together in a so-called *decomposition structure*. In a decomposition structure the isomorph subgraphs (from multiple graphs) are represented only once, and this way the re-use of partial matching is supported. Re-use is efficient when the requirement graphs contain similar patterns, which is expected when the behaviour of a robot in a given environment (e.g., in a living room, where similar setup of objects appear in case of several requirements) is specified. In Fig. 6 two requirement graphs (CF1 and CF2' on the left) and their decomposition structure (on the right) are illustrated. The dashed rectangles are individual subgraphs stored in the decomposition structure, while the dashed lines represent how a complex subgraph is decomposed into simpler ones. For example, the graph representing CF1 is decomposed into one which contains only a Room vertex, and one with a Robot and LivingBeing vertices. This latter subgraph consisting of the vertices Robot and LivingBeing can be found in both requirements, but it is represented only once, thus its matching detected in the first requirement graph shall not be checked again when the second requirement graph is checked.

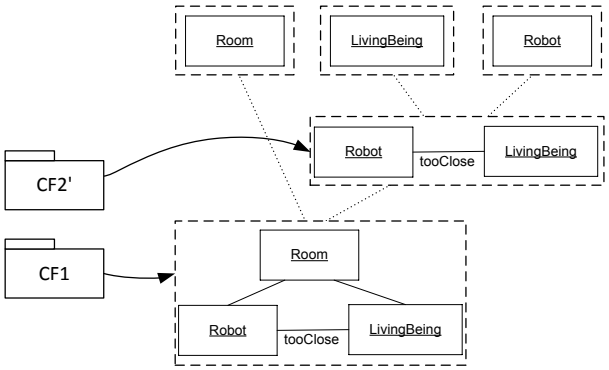

Fig. 6. Requirement graphs from Fig. 3 and Fig. 4 and their decomposition structure

It may happen that the same requirement can be matched from multiple steps of a test trace, even in an overlapped way (e.g., when the robot moves close to several objects). To solve this problem, the test evaluation runs several instances of the observer automata to check the matching. Each observer automaton has a loop transition in its initial state, this way the matching can be started at any step of the test trace, as there will be a run of the automaton that skips any potential prefix. This also solves the problem of matching one requirement overlapping with itself.

- *Handling the potential valuations:* If the observer automaton contains a context related transition with several potential matching to the configuration graph (i.e., with different *valuations* of graph elements), then an automaton instance is created for each possible valuation. To keep track of the potential valuations that may be applied at the same time, these are represented in a separate data structure linked to the decomposition structure. For example in Fig. 7 the LivingBeing element in the requirement can be matched either to a Human or an Animal in the trace.

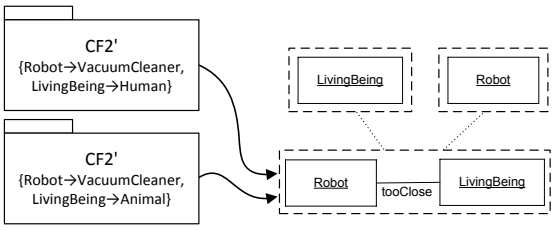

Fig. 7. Handling different valuations of the same graph structure

- *Matching abstract relations:* The mapping between the abstract relations and the concrete values (in the trace) shall be considered. To reconstruct the abstract relations, we perform a pre-processing step on the test trace which derives the valid and relevant abstract relations on the basis of the concrete values.
- *Matching the hierarchy of object types*: The hierarchy of the types of context objects shall also be considered: a subtype instance in the trace shall match its ancestor type in the requirement. To match the labels of vertices and edges (i.e., to provide valuations of graph elements), the so-called *compatibility relation* is introduced (instead of the direct equivalence of labels), that conforms to the type hierarchy defined in the context metamodel.
- *Handling dynamic changes:* As introduced above, there are dynamic objects specified in the initial context fragment that appear/disappear with a given timing. Since the requirements can also contain a sequence of events, actions, and interim contexts that not necessarily include the precise timing of their occurrence (only their ordering), the relation between these and the occurrence of the dynamic changes is not known in advance. Therefore, the matching procedure shall insert these changes into the requirement graph sequence "on-the-fly" when the timing of the test trace (relative to the start of matching) equals the timing property of a dynamic event. Formally, the representation of the necessary interim context is inserted into the observer automaton in the form of a new state with a context switch, and also the subsequent context fragments are updated.
- *Nondeterministic observer automaton*: A requirement may contain alternatives in the behaviour, this way one state in the observer automaton may have more successor states. The evaluation shall consider all possible runs simultaneously.

5 The Solution to the Test Evaluation Problem

To apply these solutions, we developed algorithms for (i) matching context graphs using the decomposition approach, and (ii) matching requirement scenarios by using an observer automata execution context that is responsible for starting and evaluating the runs of the observer automata, and storing the results. Due to space constraints here only an overview is presented, the detailed algorithms can be found in [6].

5.1 Matching Context Graphs Using a Decomposition Approach

Our algorithm for searching valuations that result in subgraph isomorphism between a configuration graph and multiple requirement graphs is based on the work of Messmer et al. [7], where the idea of *decomposition structures* was introduced. Here we utilize this idea to represent the requirement graphs in a compacted form: the decomposition structure stores each isomorph subgraph only once, therefore during the search for valuations the isomorph subgraphs have to be processed only once. The resulting valuations are bijective functions between the configuration graph and the requirement graphs (as illustrated in Fig. 8).

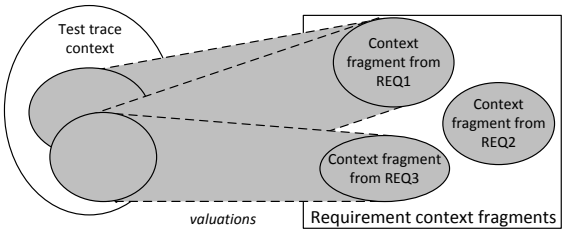

Fig. 8. Finding valuations to match requirements graphs with a configuration graph

The decomposition. The decomposition structure is built by adding the requirement graphs one by one. The algorithm has three inputs. The first input is the current decomposition structure (empty at the beginning), while the second input is the next requirement graph to be added to this structure. The third input is a set of valuations (empty at the beginning), which were already fixed during the previous steps of matching and thus considered as restrictions on the graph. Here a graph may have several valuations or no valuation at all. The decomposition algorithm handles the valuations separately from the structure of the graphs, therefore multiple valuations can be processed together. An example of the decomposition structure is presented on Fig. 6, where *CF1* and *CF2'* context fragments (from Fig. 3 and Fig. 4) are decomposed, potential valuations are illustrated in Fig. 7.

The search algorithm. The search algorithm is used to search for all possible valuations between an input configuration graph and the graphs in a decomposition structure. In comparison with the original algorithm [7], the decomposition structure has been modified: our decomposition structure contains valuations separately (rather than substituting the valuations and storing concrete vertices), which makes the structure more compact, as the similarities of the subgraphs are searched in the structure of the graph and not in the concrete vertices.

5.2 Matching Scenarios using Observer Automata

During test evaluation, the observer automata (generated from the requirement scenarios) are executed as test oracles that compare the sequence of contexts, events and

actions recorded in a test trace to the ones allowed by the requirement. Several instances of these automata are executed simultaneously, and each possible run is considered. For this purpose, we defined an execution context (EC), which is responsible for creation and execution of the observer automata.

An observer automaton has two kinds of transitions: transitions guarded with context information and transitions without context. A transition with a context can fire only if its context is matched to the actual configuration of the test trace. The EC uses the decomposition-based search approach to find a matching.

The core algorithm of the EC (that executes an observer automaton) can be considered as a cycle with two separate phases:

- *Silent phase*: In this part the algorithm computes all independent paths of the observer automaton, which consist of subsequent transitions that do not contain any event or action to match (i.e., only context related or *true* guards are found on the transitions). These paths are evaluated to check whether they can be traversed: a path can be traversed if all contexts on the path are compatible with the current configuration in the test trace. The EC creates observer automaton instances for the different paths that can be traversed, and these paths are followed.
- *Event/action matching phase:* After the silent phase, the EC processes the next event/action or context change from the test trace and checks whether it can be matched with the next enabled transition of the observer automaton.

The EC uses the decomposition approach when a context matching is performed. The number of checking a context from the test trace is limited by the maximal length of the paths found in the silent phase, plus one for the event/action matching phase. The EC collects all requirement graphs into one decomposition structure and uses the search algorithm presented in Section 5.1.

As we presented in Section 4, the dynamic events are handled *on-the-fly*: the contexts in the observer automata shall be updated dynamically according to their timing property. For this reason, the EC registers the trace time, when a new observer automaton instance is started. The dynamic events are handled differently in the steps of the silent phase and in the event/action matching phase. As the events and actions of the test trace have exact timing, the related dynamic events can be simply collected and inserted. However, in case of silent steps the EC only knows that silent steps are made before the next event/action step. Therefore the EC shall examine all possible insertions of dynamic events (including also the extreme cases when no dynamic event is inserted and when all events are inserted). This may increase the number of the simultaneous observer automaton instances.

6 Related Work and Assessment

As stated in a recent paper [8], testing autonomous systems is still an unsolved key area. Related research focused first of all on high fidelity simulators [9], excessive field testing [10], and testing the physical aspects [11]. There are only a few frameworks that offer automated test generation. An approach similar to ours is presented

in [12]: it uses metaheuristic search techniques for on-line test generation, calculating the fitness of potential test data on the basis of evaluating the execution of the SUT. In our case, we perform off-line test generation and evaluation to satisfy robustness related testing goals on the basis of a flexible context metamodel and graphical scenarios. This way our approach does not depend on the domain of the SUT.

We mapped the problem of matching contexts to the problem of matching graph sequences, which is a relatively rarely addressed problem [13]. Our test evaluation technique is similar to the related solution that was proposed for processing mobility traces using the GraphSeq tool [14]. However, as we focused on robustness testing in complex situations, we addressed the specific challenge of comparing configurations recorded in a test trace to multiple requirements that contain similar context fragments. Accordingly, we adapted a decomposition technique for searching graph isomorphism in multiple graphs at the same time, and extended it to handle multiple valuations (separated from the decomposition structure), type hierarchy, dynamic events, and abstract relations.

The decomposition based approach offers a significant increase in efficiency [7]: the search is faster than the classical Ullman's algorithm; in best case its expense is $O(IM)$ while in worst case it is $O(NI^M M^2)$, where N is the number of graphs, I is the number of vertices in the configuration graph, and M is the average size of the requirement graphs. In the best case the N graphs are the same, while in the worst case N completely different complete graphs are decomposed. Of course, the decomposition structure has to be constructed that is characterized in worst case by $(N^2 M^{M+3})$. Considering behavioural requirements of a robot operating in a given environment, the common parts in the requirement graphs are relatively frequent.

Another important characteristic of the performance of the test evaluation is the number of observer automata that are executed simultaneously. In our test setup the requirements (and thus the observer automata) are relatively small (i.e., they consist of a small number of states and transitions), but the test traces are typically long. According to our experiments, the number of simultaneous observers does not depend on the length of the test trace, but depends on the structure of the observer automata (mainly the alternative behaviours represented in the scenario), and the number of dynamic events (that may interleave with the recorded events and actions).

7 Conclusions

This paper presented the challenges and proposed solutions for test evaluation of autonomous systems. The need of comparing complex configurations recorded during test execution to multiple requirements (that typically include common context fragments) lead to the use of specific algorithms for matching multiple graphs. As behavioural requirements frequently utilize abstraction (type hierarchy and abstract relations) in the context models, we extended the graph matching with efficient handling of valuations and compatibility relations. The test evaluation tool was implemented and successfully applied in case of various test suites constructed in our framework. Currently we are working on a more extensive validation of our testing framework by

generating, executing and then evaluating tests for the control modules of ROS (Robot Operating System) [15] robot implementations in a Gazebo based simulation environment [16].

Acknowledgements. This work was supported by the ARTEMIS JU and the Hungarian National Development Agency (NFÜ) in frame of the R3-COP project. The first author was also supported by the Magyar Fejlesztési Bank through the Habilitas programme.

References

1. Connelly, J., Hong, W., Mahoney, R., Sparrow, D.: Challenges in Autonomous System Development. In: Proc. 4th IEEE Int. Workshop on Safety, Security and Rescue Robotics, NIST, Gaithersburg (2006)
2. R3-COP: Resilient Reasoning Robotic Cooperating Systems. ARTEMIS-2009-1 Project No. 100233, http://www.r3-cop.eu/
3. Micskei, Z., Szatmári, Z., Oláh, J., Majzik, I.: A Concept for Testing Robustness and Safety of the Context-Aware Behaviour of Autonomous Systems. In: Proc. 1st Int. Workshop on Trustworthy Multi-Agent Systems, Springer LNCS 7327, pp. 504–513, (2012)
4. Szatmári, Z., Oláh, J., Majzik, I.: Ontology-Based Test Data Generation Using Metaheuristics. In: Proc. 8th International Conference on Informatics in Control, Automation, and Robotics (ICINCO 2011), pp. 217–222, SciTePress (2011)
5. Majzik, I., Micskei, Z., Oláh, J., Szatmári, Z.: Models, Languages and Coverage Criteria for Behaviour Testing of Individual Autonomous Systems. R3-COP deliverable D4.2.1 Part I, http://www.r3-cop.eu/.
6. Horányi, G., Majzik I.: Automated Evaluation of the Test Traces of Autonomous Systems. Technical report, BME (2013) https://www.inf.mit.bme.hu/en/research/publications/TR-2013-Trace-Evaluation
7. Messmer, B. T., Bunke, H.: Efficient Subgraph Isomorphism Detection: A Decomposition Approach. IEEE Trans. on Knowledge and Data Engineering, 12:2, pp. 307–323 (2000)
8. Weiss, L.G.: Autonomous Robots in the Fog of War. IEEE Spectrum, 48.8, pp. 30–57 (2011)
9. Scrapper, C., Balakirsky, S., Messina, E.: MOAST and USAR-Sim: A Combined Framework for the Development and Testing of Autonomous Systems. Proc. SPIE 6230, (2006)
10. Kelly, A., et al.: Toward Reliable Off Road Autonomous Vehicles Operating in Challenging Environments. Journal of Robotics Research, 25:5-6, pp. 449–483 (2006)
11. Michelson, R. C.: Test and Evaluation for Fully Autonomous Micro Air Vehicles. In: ITEA Journal, 29.4, pp. 367–374 (2008)
12. Nguyen, C.D., Perini, A., Tonella, P., Miles, S., Harman, M., Luck, M.: Evolutionary Testing of Autonomous Software Agents. In: Proc. AAMAS (1) pp. 521–528, (2009)
13. Bunke, H. et al.: Matching Sequences of Graphs. In: A Graph-Theoretic Approach to Enterprise Network Dynamics. Progress in Computer Science and Applied Logic, vol. 24, pp. 131–143, Birkhäuser (2007)
14. Nguyen, M. D., Waeselynck, H., Riviere, N: GraphSeq: A Graph Matching Tool for the Extraction of Mobility Patterns. In 3rd Int. Conf. on Software Testing, Verification and Validation, ICST 2010, pp. 195–204, IEEE Computer Society, (2010)
15. Robot Operating Systems (ROS) Wiki. http://www.ros.org/wiki/
16. Gazebo – Multi-Robot Simulator. http://gazebosim.org/