Model-based Regression Testing of Autonomous Robots

Dávid Honfi, Gábor Molnár, Zoltán Micskei and István Majzik

Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
{honfi,micskei,majzik}@mit.bme.hu

Abstract. Testing is a common technique to assess quality of systems. Regression testing comes into view, when changes are introduced to the system under test and re-running all tests is not practical. Numerous techniques have been introduced to select tests only relevant to a given set of changes. These are typically based on source code, however, model-based development projects use models as primary artifacts described in various domain-specific languages. Thus, regression test selection should be performed directly on these models. We present a method and a case study on how model-based regression testing can be achieved in the context of autonomous robots. The method uses information from several domain-specific languages for modeling the robot's context and configuration. Our approach is implemented in a prototype tool, and its scalability is evaluated on models from the case study.

1 Introduction

Nowadays quality is a crucial aspect of software systems development. The employment of different verification and validation techniques is a possible way of achieving higher quality. One of the most commonly used techniques is testing, which intends to evaluate whether the behavior of the system under test meets its requirements. As the system develops, changes are introduced, which may require re-testing functions of the system. In these cases regression testing could be used as a solution.

Regression testing is the "selective re-testing of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements" [22]. Regression testing can be performed on any testing level (i.e., module, integration, etc.), and it can cover both functional and non-functional requirements. Re-running every test after each modification is resource and time-consuming. Thus a trade-off must be made between the confidence gained from regression testing and resources used. For this reason, several techniques were proposed over the years, particularly to select only a subset of the test suite, what is relevant for the

The final publication is available at link.springer.com in SDL 2017, LNCS 10567.

current change, or to identify those new parts of the system, which are not covered by existing tests. To discuss test selection and identification, in this paper we use the categorization of tests introduced by Leung and White [26]:

- *Re-usable* tests that exercise unmodified parts of the system.
- *Re-testable* tests that are changed or are able to cover changed parts in the system.
- *Obsolete* tests that cannot be used anymore due to changed specification or system structure.
- *New structure* tests that contribute to the overall coverage of the current, new system structure.
- *New specification* tests that verify new elements in the current specification.

Three common approaches exist for regression testing. *Test Prioritization* [27,37] is usually applied, when the total execution time of tests is not relevant, however discoverable errors shall be highlighted as soon as possible. When using *Test Suite Minimization (TSM)* [20,24] or *Regression Test Selection (RTS)* [21,35] the goal is to reduce the number of executed tests, especially when retesting the whole system requires significant amount of time. Moreover, RTS uses optimization for selecting the minimal subset of these tests that have maximal test coverage with a minimal associated execution cost. Our paper focuses on RTS, which uses the actual changes as an input to identify *re-testable* tests.

One testing criteria of RTS is reaching the maximal coverage possible. In the domain of RTS for source code, numerous approaches have been presented that define various coverage metrics: code executed by tests [1], dynamic slicing [2], graph-based representation [21]. Several tools exist implementing RTS for source code. For example, SoDA [40] is a tool for C/C++ repositories, while ChOPSJ [38] is available for code written in Java.

In the past decade, the increasing adoption of models as development artifacts led to the birth of a new approach called *Model-Driven Development* (MDD). MDD is "a development paradigm that uses models as the primary artifact of the development process" [7]. These models are commonly composed using *domain-specific languages* (DSL). DSLs are special languages for a particular problem domain. The model artifacts describe the system itself and could also serve as inputs for the testing process. As MDD is conducted in an incremental manner, model artifacts – similarly to the source code – tend to change in time. The changes in the model artifacts influence the system functions and properties (as models drive the synthesis of software, hardware, configuration, parameterization etc. of the system), this way these changes can be used to trigger re-testing the influenced parts of the system. In an MDD setting, having the relation between (changed) model artifacts and system parts, regression test selection can be applied on model level rather than on the generated code.

We encountered this situation in the context of the *Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems* (R5-COP) project[1]. The project worked with several industrial demonstrators: autonomous robots that

[1] `http://www.r5-cop.eu`

need to be re-tested after reconfigurations due to changes in their functionality or their components. We developed a model-based approach that uses several domain-specific languages to model the capabilities of the robots and their tests contexts, and created an RTS model to represent the artifacts of the regression testing domain, among others the tests, testables, and coverage relations. The specific input models and artifacts as well test elements (e.g., test cases, test setups) can be mapped to this representation, and the test classification and regression test selection algorithms can be implemented uniformly on the basis of this model. The approach was implemented in a prototype tool using the Eclipse framework and its various modeling components.

The rest of the paper is structured as follows. Section 2 details the autonomous robot case study. Section 3 presents the approach that was developed to support regression test selection. Section 4 presents the implementation of the approach in a prototype tool. Section 5 evaluates the scalability of the approach and the implemented tool.

2 Presentation of the Case Study

An autonomous system can be defined as one that makes and executes decisions to achieve a goal without full, direct human control [12]. Notable characteristics shared by the different kinds of autonomous systems include reasoning, learning, adaptation and context-awareness. A typical example of an autonomous system is an autonomous robot, which is working in a real, uncontrolled environment, possibly in the presence of humans.

The autonomous robots case study was performed in the R5-COP project. The project focused on reconfigurable robots coping with quickly changing environments and conditions. The verification of autonomous robot systems is an essential part of their development process due to their safety-critical nature. Thus testing and regression testing are crucial tasks during their development.

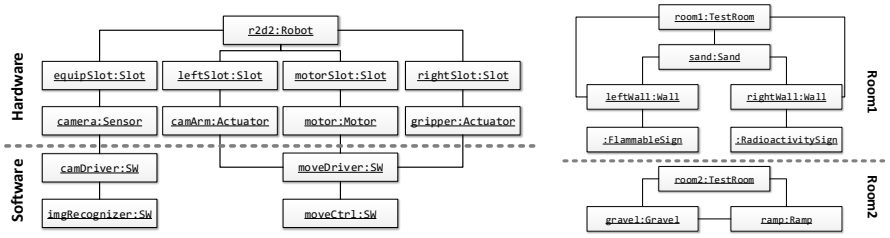
Testing autonomous systems is particularly challenging due to the facts that their behavior is highly context-aware and their context contains a large number of possible situations [39]. Full behavior specification can be impractical due to the complexity of the behavior and the diversity of the system environments. Therefore a typical solution is to specify high-level properties and scenarios and evaluate these to detect violations of (safety) requirements [18]. Robots are placed in different situations (either in physical test environment or simulator), and properties are checked at runtime using monitors or off-line via trace analysis.

One of the industrial demonstrators of the project was an emergency response robot, a special type of mobile robots that is capable of performing certain activities in an environment that may possess the risk of human injury (e.g., critical tasks in handling explosives). The verification process of the completely built robots is usually conducted in special test rooms. These rooms are able to pose challenges for different capabilities of the robot through different terrain and obstacle types. The rooms use standardized elements (e.g., alleys, ramps) [5,23,29] that can be assembled in different configurations, and several tasks can

be performed on each element (just crossing it, crossing it by following a line, reading a sign, etc.).

The changes in the requirements of the robots may trigger modifications in the configuration (replacing a component) or the test rooms (using a new element for testing a new functionality). This is very similar to the maintenance of the test suites of software, hence regression testing could be applied also in this domain: the robot can be thought as the system under test, while a layout of test room or a particular element of a test room is a test case for the robot.

Our testing approach [28] used a model-based, system-level black-box testing method. We modeled both the capabilities of the robots and the test rooms. Based on the NIST guidelines [29], we defined the following main types of model elements for test rooms: 1) mobility terrain, 2) obstacle, 3) visual target. The capabilities of the robot are also captured in a model that describes both hardware and software elements and the dependencies amongst them. According to the model a robot has slots where hardware elements (e.g., sensor, actuator, motor) can be mounted. Robots also have several different software elements installed that control hardware elements. Due to space constraints the full meta-models are not included, but they can be found in the project's deliverables [33,34].



(a) The robot instance (b) The test room instances

Fig. 1. Example instance models for robot configuration and test context

Figure 1a shows the simplified capability model of a sample robot, while Figure 1b presents two sample test room instances. The robot model used in this example contains both hardware and software elements. The robot itself has four slots (left, right, motor, equipment). The motor slot is connected to the motor, which enables the robot to move. The right slot is connected to an arm that has a gripper to grasp objects. The left slot has an arm connected, which holds a camera. The camera is plugged into the equipment slot. Both actuators (camera arm, gripper) and the motor are controlled by a movement controller through a movement driver software. The camera has an image recognition software that communicates with the sensor using a special driver software. The terrain in the first room ($room1$) is sand, which is located between two walls (left and right). The left wall has a flammable warning sign, while the right one has a radioactivity sign on it. The second room ($room2$) has a gravel terrain and contains a ramp.

Let us consider the situation, when the specification of the robot is modified: a new `camera` is designed for the robot. Without any regression test selection, this change would trigger re-execution of all tests in both rooms. In a real scenario this may take high amount of time as the same room is used often with different layouts, thus would require multiple rearrangements. However, if only the camera is changed, it may be enough to the test the robot in $room1$.

To perform regression test selection it is crucial to have a mapping of coverage, which connects the test rooms with the capabilities of the robot. For example, the image recognizer component can be tested by the signs on the walls and the motor can be tested by the different terrain types. An RTS algorithm would be able to identify the minimal number of tests that are required to re-run to cover the modified parts of the system.

Selecting the right level of abstraction for the models and the goal of the testing was a non-trivial design decision. We performed multiple iterations with the industrial partners and designed several versions of the system and test models. Some models captured multiple possible configurations of the test rooms with different elements and selected tests based on which test room or which test room element is relevant for a given robot skill or component. Other models worked with a small, fixed number of test setups that were actually assembled at the partner's location, and varied what combinations of exercises should the robot perform in each test room. Therefore we needed an approach that can work with different input modeling languages and can be quickly adapted to new ones, without having to re-implement the whole regression testing algorithm.

The next section presents the approach we developed for the case study. Regression test selection was performed on similar domain-specific models by 1) defining an RTS model and 2) mapping the elements of the domain-specific inputs models to the elements of this RTS model. This approach was able to support regression testing in the presented setting.

3 Approach

RTS algorithms usually employ the following common concepts: 1) testable, 2) test and 3) coverage to handle the system under test (testables like elements of source code, model, etc.) and the tests that cover elements of the system. However, creating a compound representation is far from trivial and can be accomplished in various ways [43]. The forthcoming part of this section defines a representation that can be used for model-based regression test selection.

Several typical ways exist to define the coverage model. The most simple one is a binary matrix with program elements in its rows and tests in the columns. The matrix has 1 in cell (i, j) if the ith program element is covered by the jth test. However, if our inputs are DSL models and not just program lines or list of methods, a different, model-based representation is more suitable.

The main requirements of the RTS model were the followings. The RTS model shall 1) be easily extensible for different artifacts of various models and DSLs, and 2) separate the RTS algorithm from the core RTS concepts. To fulfill

these requirements we developed an RTS model, which represents the generic concepts of RTS that can be mapped from the concrete artifacts (models and tests) of the input domain. The RTS model represents the data model that is required to conduct test selection for different models as input artifacts.

3.1 RTS Model

An RTS algorithm uses three main concepts: 1) elements in the system, 2) tests that exercise parts of the system and 3) a coverage relation that drives the selection process. Our proposed RTS metamodel contains four main concepts that is eligible to describe the underlying artifacts for the RTS algorithm.

- *Testable:* an abstract element that is verified by tests.
- *Component:* a type of *Testable* that supports dependencies; changing a component triggers all dependents to be re-tested.
- *Conditional:* a special type of *Testable* that represents a conditional element in the system (e.g., a branch or a condition in a decision), which requires individual handling during the RTS process (e.g., each value of the condition must be tested with a specific test case).
- *Test:* represents an executable test case in the system.

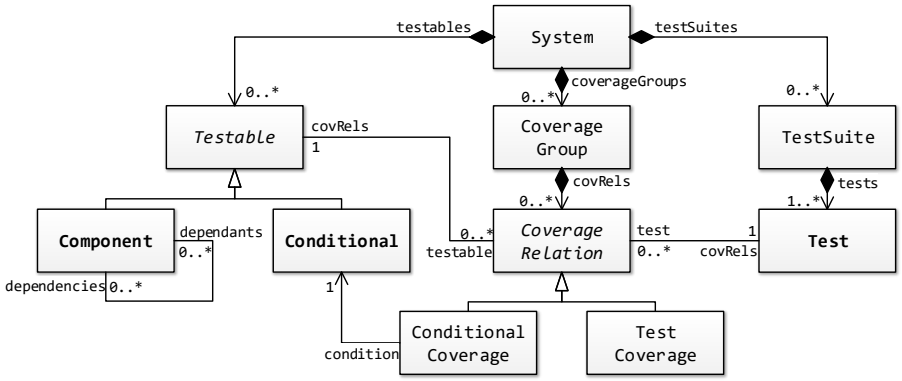


Fig. 2. The structure of the RTS metamodel

The full RTS metamodel is shown in Figure 2. The main component of the model is the *system*. A system consists of *testable*s, test suites and coverage groups. A testable instance could be a component or a conditional element, which were already presented. Components can depend on each other, thus there is a self-association defined. A test suite consists of *test*s connected to testables through coverage relations of coverage groups. A *coverage relation* connects a testable and a test (denoted with association). An instance of the coverage relation could be conditional coverage or simple test coverage. Simple test coverage defines no special conditions on the notion of coverage, thus can be fulfilled by simply covering an element. On the contrary, conditional coverage also covers

elements but uses additionally a conditional element (marked with association), that requires individual handling of condition values during regression test selection (e.g., covering both the inclusion and the absences of an input model element in the tests). A coverage group holds together relations that have similar meaning in the domain being used, which alleviates their handling. Furthermore, testables, test suites, tests and coverage relations are modifiable meaning that they store whether the given element in the system has been changed since the last run or not. This change is represented in the RTS model using a special attribute.

3.2 Mapping of Input Models

In order to produce an instance of this metamodel a mapping is needed where the inputs are the system and test models, and the result is an instance of the RTS model itself. The transformations should use unique identifiers to trace back elements to the original models. These transformations are specific to the domain-specific models used as inputs. By using the mapping, the selection becomes independent from the input models. The implementation of the RTS is bound to the RTS model this way it is not necessary to (re-)implement it on the basis of the specific model artifacts and coverage models.

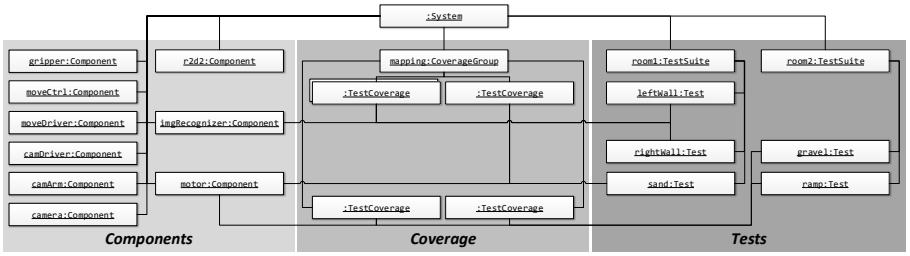


Fig. 3. The transformed sample RTS instance model

Notice that changes in the original models shall be represented in the RTS model. To tackle this question, our approach employs *checkpointing of models*, which is a common model versioning technique [4]. Hence, when a checkpoint during the model development is reached, the automatic mapping to the common RTS model is triggered with calculating the changes between checkpoints. These changes are applied to the RTS model incrementally and indicated on each modifiable element using the according attribute automatically.

Figure 3 depicts how this mapping was defined for the robot case study example presented on Figure 1. One transformation was defined for the model of robot capabilities, which transformed every element into a *Component* in the optimization model. The approach supports dependencies between components, and these dependencies are used to find affected components transitively during the regression test selection for a given change. Additionally, the test rooms were

transformed into test suites and tests. One may notice that we used *TestRooms* as test suites and elements as *Tests*, though they can be handled differently as the level of abstraction is changed (e.g., using the whole room as a test). Finally, a third, simple model (not shown on Figure 3) was used to describe the mapping between robot capabilities and test room elements. This mapping model is translated into coverage elements in the RTS model.

Note that even the RTS model uses simple and compact concepts, these were enough to represent the regression testing problem in the current case study. For other case studies, the RTS model could be extended with other concepts.

3.3 Usage Scenarios

The approach can be used in two phases of an MDD development. First, the approach is intended to be used by *Test Engineers* during the development and maintenance phase of models as their common tasks are 1) identifying untested elements in the system, 2) performing impact analysis to identify the effects of particular changes, 3) re-testing the system after changes have been applied. Re-testing time should be reduced along with maintaining the same fault-detection capability of the test suite. This is where the presented approach emerges by 1) highlighting untested parts of the system calculated from the coverage relationships 2) detecting changes and impacts through dependencies of components and 3) selecting tests to re-run. Test engineers only employ the approach and do not develop or extend it.

Second, the presented approach shall also be used by developers of domain-specific languages as their tasks include 1) identifying elements of the DSL that correspond to tests and testables, 2) identifying how test coverage could be defined from elements and 3) implementing a transformation to a specific test model. These tasks are supported by providing the definition of the main concepts in the presented approach for generic regression test selection. In an MDD setting, developers of DSLs shall define the mappings and transformations to the RTS model, that can be used later by the test engineers.

4 Implementation

The approach is implemented in RTSMOT (RTS MOdeling Tool), a tool using the Eclipse Modeling Framework. To be able to handle several, different input models, the tool was given a layered architecture as shown in Figure 4.

As the input models can be different domain-specific models, adapters are required for defining the mapping to the RTS model. A *Model Adapter* consists of transformations that map the domain models to the RTS model. RTSMOT provides interfaces for these transformations, hence only the knowledge of domain models is enough to implement them. For transformations, the adapters use VIATRA, a state-of-the-art incremental model transformation framework [6]. Using VIATRA requires the definition of patterns that can be matched to different domain model elements. Then, a transformation with VIATRA can be defined for each

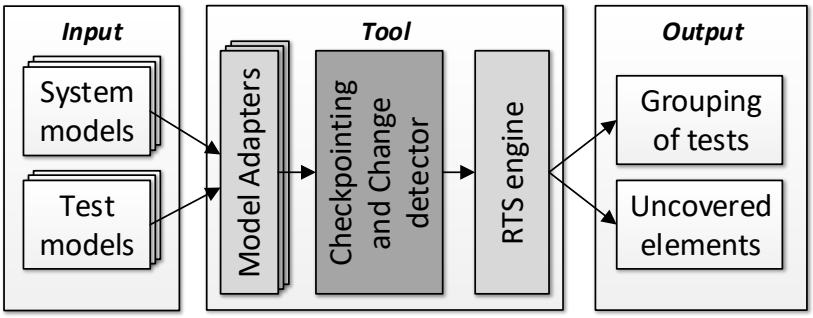


Fig. 4. Architecture and workflow of the prototype tool

match of the patterns, hence making able to map input model elements to new elements of the RTS model.

The model checkpointing technique, which is used in the presented approach demands for another layer in the architecture; the *Checkpointing and Change Detector* component provides the ability to create checkpoints during model development. At each checkpoint, this layer is also responsible for detecting changes in input models and indicating them on elements of the generic RTS model. The change detector marks all changes, i.e. all differences between the two versions of the model in the checkpoints.This process is performed with unique identifiers of elements that allows tracing between the input models and the generic RTS model. The prototype implementation currently uses the file system with time stamps for model versioning. However, this layer can be developed further to collaborate with well-known version control systems like Git and SVN.

The third layer of the RTSMOT tool is the *RTS engine*. This layer performs the actual test selection by using a replaceable algorithm subcomponent making the prototype tool more flexible. The algorithm yields the identification of elements in the RTS model, which are affected by changes in a checkpoint. Then, the algorithm selects test cases that are able to cover changed parts in the system. Also, the layer reports the uncoverable (but changed) and uncovered elements. The tool currently uses a simple greedy approximation algorithm for Minimal Set Cover as the problem of RTS can be reduced to this [19]. Further details of the implementation can be found in the project's deliverable [33].

5 Evaluation

We evaluated the applicability of the approach and the capabilities of RTSMOT to answer the following research question: *Could the prototype of the approach scale up to models found in the case study domain?*

5.1 Study Design

Method In order to measure the scalability, the change detection and test selection capabilities are evaluated. Evaluating the change detection requires the input models to change between two checkpoints. The evaluation of test selection

also uses the RTS model, which can be extended and scaled up in three ways: 1) components, 2) tests and 3) coverage. Moreover, the RTS evaluation demands for creating elements with predefined connections (coverage), thus making it a more complex scenario. We used upscaled model instances of models presented in Section 2.

Setup for Change Detection The change detection can be evaluated from two aspects: 1) size of the input models to compare, 2) size of the change. Six different sizes of input models are defined for the evaluation: 16, 32, 64, 128, 256 and 512. These models were created by adding new component instances to the robot. Note that these sizes are the numbers of newly added components to the original robot instance model seen on Figure 1a. Additionally sizes of the changes are defined in a smaller scale for this experiment: 1, 2, 4, 8, 16 and 32. According to the industrial partners in the R5-COP research project, these model sizes can be relevant in the autonomous robot domain. A significant aspect of the scalability is that how much time it takes to detect changes with different sizes of models and changes. Hence the evaluation addresses the following comparisons: 1) execution time with different sizes of inputs (number of changes here is 1), 2) execution time with different number of changes between checkpoints (size of input models here is set to 512).

Setup for RTS The time that RTS takes during the test selection is a crucial part of the approach as it should not take unfeasible amount of time (e.g. running RTS and the selected tests should not take longer than re-running the whole test suite). Thus, the RTS model with 512 elements is used in this part of the evaluation with various amount of changes ranging from 1 to 512 on a logarithmic scale. Furthermore the number of dependencies to a changed component may affect the time required for running the RTS. This analysis also uses the model with 512 elements with the number of changes tied to one. However, the number of dependants to a single component is modified on a logarithmic scale from 1 to 512.

5.2 Results

The values presented in this section were obtained from executing RTSMOT on a notebook with a 2-core CPU running at 3.0 GHz and 8 GBs of RAM. During the evaluation, every measurement was repeated 30 times and the average values are presented here. Before each measurement a warm-up session was conducted in order to avoid outlier values caused by initialization processes in the Eclipse framework. The data analysis was performed using R [32], while execution times were measured by using stopwatches in code. In order to use statistical measures, the normality of the results for each repetition was checked. All check yielded that the 30-times repeated results follows a normal distribution.

Figure 5 presents the relationship between the number of model elements on a logarithmic scale and the change detection time in milliseconds. The results show that as the size of the model is incremented, the detection time also increases. Table 1 summarizes these values including a confidence interval (CI) on 95% confidence level obtained using the one-sample t-test. The confidence intervals

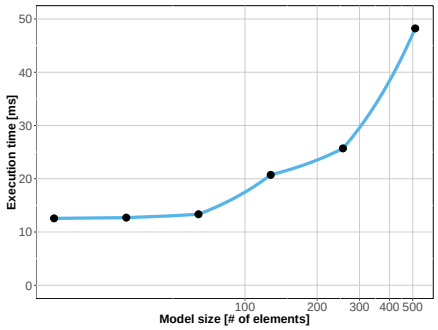


Fig. 5. Execution time of change detection with various model sizes

Size [#]	Avg. time [ms]	CI
16	12.56	[10.3, 14.83]
32	12.7	[10.38, 15.02]
64	13.33	[11.48, 15.18]
128	20.73	[14.93, 26.53]
256	25.7	[22.3, 29.1]
512	48.23	[43.69, 52.78]

Table 1. Change detection times with different model sizes

do not show large deviations, and the border values of the CIs grow with the average times. The presented change detection times may be thought feasible in the domain of the study. We also measured change detection time on larger models in order to determine the effects on practical applicability. We used two models containing 8192 and 16384 elements, from which the results were 5,59 and 22,02 seconds respectively, which are still convenient response times.

In terms of the relationship between the size of changes and the execution time of change detection, the results are promising. Figure 6 shows that there is a clear linear correlation between the number of changed elements and the related execution time. This is due to the linear search algorithm used in the background. Changing this algorithm to a model pattern detection-based technique may improve the performance.

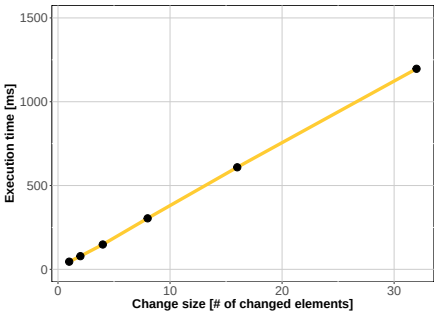


Fig. 6. Execution time of change detection with various number of changes

Size [#]	Avg. time [ms]	CI
1	45.67	[41.70, 49.63]
2	78.53	[75.6, 81.46]
4	148.17	[144.82, 151.51]
8	304.27	[285.24, 323.3]
16	608.83	[584.84, 632.82]
32	1196.53	[1151.51, 1241.56]

Table 2. Change detection times with different sizes of changes

Table 2 presents the results from the analysis of the relationship between the number of changes and the detection time. Note that the values are increasing linearly with the number of changes. Moreover the confidence intervals (CI) also show this relationship. The intervals were obtained again on 95% confidence level

using the one-sample t-test. To sum up, these results show a clearly identifiable linear relationship between the number of changes and the change detection time. The maximum value was slightly more than one second even on the largest models used, thus can be thought as a promising and feasible result.

As mentioned earlier, the RTS execution time is also a crucial part of the process. To evaluate its performance the execution time was measured with different number of changes on a previously used model in the case study (containing 512 elements). Figure 7 depicts the results from this evaluation with the sizes of changes on a logarithmic scale. It can be seen that no dependency exists between the number of changes and the RTS execution time because even when all the model elements were changed the time remained almost the same.

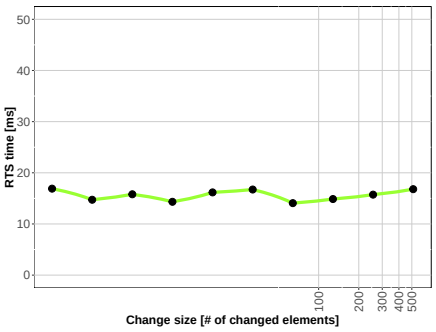


Fig. 7. Execution time of RTS with various number of changes

Size [#]	Avg. time [ms]	CI
1	16.9	[10.63, 23.17]
2	14.73	[10.78, 18.68]
8	14.33	[10.12, 18.55]
32	16.73	[12.1, 21.38]
128	14.87	[10.2, 19.53]
512	16.8	[9.12, 24.48]

Table 3. RTS execution times with different sizes of changes

Table 3 reveals the details of this evaluation containing the average times and their confidence intervals (CI) with the previously used one-sample t-test on 95% level of confidence. The values are almost equal in all cases and do not show large deviations. However larger CIs exists, which is due to the first and second measurements that had longer execution times as the modeling framework did not cache the required model elements until the third run (though a warm-up run was conducted to avoid this effect). In brief, these execution times are acceptable for the domain of autonomous robots even on relatively large models.

As described in Section 5.1 we also analyzed how the RTS execution time is affected by the number of dependencies belonging to a changed component. Based on the results, the pattern-based dependency analysis that is implemented in RTSMOT turned out to be effective: the execution times were roughly the same that were presented in Table 3. Thus the execution time of RTS can be thought as independent from the number of dependencies to a component.

The evaluation of these complex cases was performed to answer the *RQ*. The results produced by RTSMOT that implements the generic RTS approach are promising and scale up without significant increase of execution time even for these larger model sizes. Hence the presented approach and the prototype tool can scale to real models used in the autonomous robot domain.

6 Related Work

Regression test selection has a very broad area of research as it can be executed in numerous ways [36,17]. Engström et al. conducted a survey [13], where the regression test selection techniques for source code are gathered and assessed based on their evaluations. They had two conclusions to emphasize: 1) empirical evidence is not very strong on evaluating RTS techniques, 2) RTS techniques have to be tailored to the given context as no generic technique can be found.

Yoo et al. [43] also conducted a survey on regression test selection techniques, and identified new trends in the research of this area. According to them, model-based RTS techniques emerge for two reasons: 1) the higher level of regression testing and 2) the easier scalability. Some of these techniques use EFSM or UML, however some use other approaches like graphs or a specific internal model.

Methods Using EFSM EFSMs add variables and conditional execution to the basic FSM semantics. This enables them to model software behavior better. Chen et al. [11] provide a way to use regression testing on EFSM models. The changes (elementary modifications) they cover are defined on a transition of the state machine; either addition, deletion or a change. Korel et al. also use the EFSM semantics in their work [24]. They also use the notion of elementary modification to describe changes on the input models. Vaysburg et al. presented a technique [41] that uses dependency analysis on EFSM system models. Their approach is able to capture various kind of interaction between elements, which is used as an input for the regression test selection process. Almasri et al. employed EFSMs to conduct impact analysis in model-based systems [3] in order to reduce maintenance costs and to identify critical parts of the system. They also defined model and data density metrics, which are found to be major influencing factors to the number of components involved in a change.

Methods using UML Wu and Offut [42] provide an approach for regression testing component-based software based on their UML diagrams. The diagrams applied are class diagrams, collaboration diagrams and statecharts. Somewhat similarly Briand et al. [9,8] provide another approach that classifies test cases into the usual categorization; obsolete, re-testable or re-usable. They have also implemented a tool (RTSTool) to evaluate it. Farooq et al. propose an approach for UML state machines [15] and an Eclipse-based tool as well [14]. From these papers we have seen that UML diagrams are already used for regression testing purposes, there are even some tools implemented. Pilskalns et al. propose [31] an incremental test generation method for UML diagrams that transforms the input to a graph, on which then a test selection algorithm is run to identify re-testable test cases. Traon et al. [25] also use an internal model (test dependency graph) to represent the input towards the test selection algorithm. They are also mapping UML class diagrams to this graph. It is not clearly expressed, whether these techniques can be used for another model inputs (apart from UML) as well. Chen et al. use [10] UML activity diagrams to identify test cases that are affected by the modifications in release of a software. They employ activity

diagrams as the specification and only separate two different types of regression tests (targeted and safety) unlike other, more generic approaches.

Generic approaches A closely related approach for model-based RTS is presented by Zech et al. in [44] and [45]. Their approach uses the generic MoVe model versioning platform and calculates deltas from changes between model versions. The difference between the approaches is that theirs employs a domain-specific model obtained from the expanded delta, while our approach uses a generic RTS model. Fourneret et al. presented a generalized model-based regression testing technique in [16]. They extract behavior from the input models to supply impact analysis during the RTS process. These behaviors are extracted from guards or actions when using state charts, and from Object Constraint Language constraints in case of class diagrams. This process is clearly similar to the approach presented by Zech, although behavioral extraction is made additionally. Orso et al. provides an approach [30] on how to use metadata from external components to supply regression test selection process both on code and model.

7 Conclusions

This paper presented a model-based regression test selection (RTS) approach that was developed for the system-level testing of reconfigurable, autonomous robots. This technique uses an RTS model to enable the handling of multiple input models specified in different domain-specific languages. In order to use the approach on different input domains, simple transformations are needed, which can be defined by the potential users of the approach. This includes test engineers and domain-specific language developers.

The paper also introduced the architecture of a prototype tool called RTS-MoT that implements the approach using the Eclipse framework and its modeling platform EMF. The scalability of the approach was evaluated on models from the case study. The results showed that the tool can scale to larger models and even after several changes the test selection is performed quickly.

The developed approach was able to capture the regression testing problem of the case study. However, an important lesson was that it required numerous iterations with the industrial partners to find the right level of abstraction of the models representing the capabilities, context and test setups of the robots. Several versions of the input model languages were developed targeting different testing goals (e.g. testing using rooms with different configurations, testing using different exercises in a fixed test room). In these iterations the layered architecture of the tool and the usage of small model adapters that can be quickly developed proved to be a really useful design decision.

Future work includes several directions. For example, the approach is able identify elements in the models for which no test exists, but offers no solution for the user. We are working on to automatically generate test setups including the missing elements using search-based techniques.

Acknowledgment

This work was partially supported by the ARTEMIS JU and the Hungarian National Research, Development and Innovation Fund in the frame of the R5-COP project.

References

1. Aggrawal, K., Singh, Y., Kaur, A.: Code coverage based technique for prioritizing test cases for regression testing. ACM Software Engineering Notes 29(5), 1–4 (2004)
2. Agrawal, H., Horgan, J.R., Krauser, E.W., London, S.: Incremental regression testing. In: Int. Conf. on Software Maintenance. vol. 93, pp. 348–357 (1993)
3. Almasri, N., Tahat, L., Korel, B.: Toward automatically quantifying the impact of a change in systems. Software Qual J pp. 1–40 (2016)
4. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. Int. J of Web Information Systems 5(3), 271–304 (2009)
5. ASTM International: Standard Terminology for Evaluating Response Robot Capabilities (2016), E2521-16
6. Bergmann, G., Dávid, I., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z., Varró, D.: VIATRA3: a reactive model transformation platform. In: Theory and Practice of Model Transformations, pp. 101–110. Springer (2015)
7. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers, 1st edn. (2012)
8. Briand, L., Labiche, Y., He, S.: Automating regression test selection based on UML designs. Information and Software Technology 51(1), 16 – 30 (2009)
9. Briand, L., Labiche, Y., Soccar, G.: Automating impact analysis and regression test selection based on UML designs. In: Int. Conf. on Software Maintenance. pp. 252–261 (2002)
10. Chen, Y., Probert, R.L., Sims, D.P.: Specification-based regression test selection with risk analysis. In: Conf. of the Centre for Advanced Studies on Collaborative Research. pp. 1–14 (2002)
11. Chen, Y., Probert, R.L., Ural, H.: Regression test suite reduction using extended dependence analysis. In: 4th Int. Workshop on Software Quality Assurance. pp. 62–69. SOQUA '07, ACM (2007)
12. Connelly, J., Hong, W., Mahoney, R., Sparrow, D.: Challenges in autonomous system development. In: Proc. of Performance Metrics for Intelligent Systems Workshop (PerMIS'06) (2006)
13. Engstrm, E., Runeson, P., Skoglund, M.: A systematic review on regression test selection techniques. Information and Software Technology 52(1), 14 – 30 (2010)
14. Farooq, Q., Iqbal, M., Malik, Z., Riebisch, M.: A model-based regression testing approach for evolving software systems with flexible tool support. In: IEEE Int. Conf. on Engineering of Computer Based Systems. pp. 41–49 (2010)
15. Farooq, Q.u.a., Iqbal, M.Z.Z., Malik, Z.I., Nadeem, A.: An approach for selective state machine based regression testing. In: Proc. of the 3rd Int. Workshop on Advances in Model-based Testing. pp. 44–52. A-MOST, ACM (2007)
16. Fourneret, E., Cantenot, J., Bouquet, F., Legeard, B., Botella, J.: SeTGaM: Generalized technique for regression testing based on UML/OCL models. In: Int. Conf. on Software Security and Reliability. pp. 147–156. IEEE, United States (2014)

17. Graves, T.L., Harrold, M.J., Kim, J.M., Porter, A., Rothermel, G.: An empirical study of regression test selection techniques. ACM TOSEM 10(2), 184–208 (2001)
18. Guiochet, J., Machin, M., Waeselynck, H.: Safety-critical advanced robots: A survey. Robotics and Autonomous Systems 94, 43 – 52 (2017)
19. Harman, M.: Making the case for MORTO: Multi objective regression test optimization. In: ICST Workshops. pp. 111–114 (2011)
20. Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM TOSEM 2(3), 270–285 (1993)
21. Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., Gujarathi, A.: Regression test selection for Java software. ACM SIGPLAN Notices 36(11), 312–326 (2001)
22. IEEE: Systems and software engineering – Vocabulary (2010), standard 24765:2010
23. Jacoff, A., Huang, H.M., Messina, E., Virts, A., Downs, A.: Comprehensive standard test suites for the performance evaluation of mobile robots. In: Proc. of the 10th Performance Metrics for Intelligent Systems Workshop. pp. 161–168. PerMIS '10, ACM (2010)
24. Korel, B., Tahat, L., Vaysburg, B.: Model based regression test reduction using dependence analysis. In: Int. Conf. on Software Maintenance. pp. 214–223 (2002)
25. Le Traon, Y., Jeron, T., Jezequel, J., Morel, P.: Efficient object-oriented integration and regression testing. IEEE Tran. on Reliability 49(1), 12–25 (2000)
26. Leung, H., White, L.: Insights into regression testing. In: Int. Conf. on Software Maintenance. pp. 60–69 (Oct 1989)
27. Malishevsky, A.G., Ruthruff, J.R., Rothermel, G., Elbaum, S.: Cost-cognizant test case prioritization. Tech. rep., Department of Computer Science and Engineering, University of Nebraska-Lincoln (2006)
28. Micskei, Z., Szatmári, Z., Oláh, J., Majzik, I.: A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In: Agent and Multi-Agent Systems. Technologies and Applications, pp. 504–513. Springer (2012)
29. NIST: Guide for Evaluating, Purchasing, and Training with Response Robots using DHS-NIST-ASTM International Standard Test Methods (2014), `https://www.nist.gov/el/intelligent-systems-division-73500/response-robots`
30. Orso, A., Do, H., Rothermel, G., Harrold, M.J., Rosenblum, D.S.: Using component metadata to regression test component-based software. Software Testing, Verification and Reliability 17(2), 61–94 (2007)
31. Pilskalns, O., Uyan, G., Andrews, A.: Regression testing UML designs. In: Int. Conf. on Software Maintenance. pp. 254–264 (2006)
32. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing (2013), `http://www.R-project.org/`
33. R5-COP: Incremental testing of behaviour (2016), `http://www.r5-cop.eu/media/cms_page_media/35/R5-COP_D34.20_v1.0_BME.pdf`, d34.20 deliverable
34. R5-COP: Assessment of the On-line Verification and Incremental Testing (2017), `http://www.r5-cop.eu/media/cms_page_media/35/R5-COP_D34.50_v1.1_BME.pdf`, d34.50 deliverable
35. Rothermel, G., Harrold, M.J.: Selecting regression tests for object-oriented software. In: Int. Conf. on Software Maintenance. pp. 14–25. IEEE (1994)
36. Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. IEEE Tran. on Software Engineering 22(8), 529–551 (1996)
37. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. IEEE Tran. on Software Engineering 27(10), 929–948 (2001)
38. Soetens, Q.D., Demeyer, S.: Cheopsj: Change-based test optimization. In: European Conference on Software Maintenance and Reengineering. pp. 535–538 (2012)

39. de Sousa Santos, I., de Castro Andrade, R.M., Rocha, L.S., Matalonga, S., de Oliveira, K.M., Travassos, G.H.: Test case design for context-aware applications: Are we there yet? Information and Software Technology 88, 1 – 16 (2017)
40. Tengeri, D., Beszedes, A., Havas, D., Gyimothy, T.: Toolset and program repository for code coverage-based test suite analysis and manipulation. In: 14th IEEE Int. Working Conf. on Source Code Analysis and Manipulation. pp. 47–52 (2014)
41. Vaysburg, B., Tahat, L.H., Korel, B.: Dependence analysis in reduction of requirement based test suites. In: Proc. of the Int. Symposium on Software Testing and Analysis. pp. 107–111 (2002)
42. Wu, Y., Offutt, J.: Maintaining evolving component-based software with UML. In: European Conf. on Software Maintenance and Reengineering. pp. 133–142 (2003)
43. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22(2), 67–120 (2012)
44. Zech, P., Felderer, M., Kalb, P., Breu, R.: A generic platform for model-based regression testing. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, pp. 112–126. Springer (2012)
45. Zech, P., Kalb, P., Felderer, M., Atkinson, C., Breu, R.: Model-based regression testing by OCL. Int. J on STTT 19, 115–131 (2015)