R5-COP

Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems

Incremental testing of behaviour

BME

Project	R5-COP	**Grant agreement no.**	621447
Deliverable	D34.20	**Date**	31/07/2016
Contact Person	Zoltan Micskei	**Organisation**	BME
E-Mail	micskeiz@mit.bme.hu	**Diss. Level**	PU

Document History			
Ver.	Date	Changes	Author
0.1	21/04/2016	Initial structure of the content	D. Honfi (BME)
0.2	2/5/2016	Added Overview section	Z. Micskei (BME)
0.3	17/5/2016	Updated approach	D. Honfi (BME)
0.4	27/6/2016	Description about implementation	G. Molnar (BME)
0.5	13/7/2016	Finished evaluation section	D. Honfi (BME)
0.6	18/7/2016	Finalize document for internal review	Z. Micskei (BME)
0.7	22/7/2016	Internal review	A. Dabrowski (PIAP)
1.0	28/7/2016	Updated document based on review	Z. Micskei (BME)

Note: Filename should be

"R5-COP_D##_#.doc", e.g. „R5-COP_D91.1_v0.1_TUBS.doc"

Fields are defined as follow

1. Deliverable number ***.***

2. Revision number:

draft version **v**

approved **a**

version sequence (two digits) ***.***

3. Company identification (Partner acronym) *****

Content

Content 3
List of images 5
List of tables 6
List of Acronyms 7
1 Introduction 8
1.1 Summary (abstract) 8
1.2 Purpose of Document 8
1.3 Partners Involved 8
2 Overview 9
2.1 Background 9
2.2 Incremental Testing of Autonomous Robots 10
2.2.1 Basis of the Approach 10
2.2.2 Reconfiguration 11
2.2.3 Inputs and Outputs 13
2.3 Required Descriptions and Languages 13
3 Approach for Incremental Testing 14
3.1 The Envisaged Approach 14
3.2 Description for Incremental Testing 15
3.2.1 Generic Model for Incremental Testing 15
3.2.2 General Mapping of Input Models 16
3.2.3 Using Context Models as an Input for Incremental Testing 16
3.2.4 Using Configuration Models as an Input for Incremental Testing 16
3.2.5 Description of Change and Reconfiguration 17
4 Implementation of the Approach 18
4.1 Overview of the Architecture 18
4.2 Possible Use Cases 18
4.3 Details of the Implementation 19
4.3.1 General Structure 19
4.3.2 Layout of the Projects 19
4.3.3 Model Adapters 20
4.3.4 RTS Engine 21
4.3.5 Implementation of the UI 21
4.3.6 Summary of Details 22
5 Detailed Examples of the Approach 23
5.1 Preliminaries 23

5.2 Example with a NIST test room 26
5.3 Example with two NIST test layouts....27
5.4 Example with a demonstrator context....29
6 Evaluation of Scalability 32
6.1 Setup....33
6.2 Results 33
7 Conclusions 37
8 References 38

List of images

Figure 1. Creating test data (at the bottom) from context models (at the top) 11
Figure 2. Change in tests due to reconfiguration in the context or environment 12
Figure 3. Effects of change in the robot configuration 12
Figure 4. Overview of the incremental testing methods 14
Figure 5. Metamodel for describing incremental testing 15
Figure 6. Architecture of the prototype tool 18
Figure 7. Component model of the core plugins in the implemented tool 20
Figure 8. Greedy approximation algorithm for the Set Cover Problem 21
Figure 9. The robot metamodel used during the examples 23
Figure 10. The example robot instance on a simplified representation 24
Figure 11. The NIST context metamodel 24
Figure 12. Mapping metamodel for different context elements and the capabilities of the robot 25
Figure 13. Schematic layout of Test Room 1 26
Figure 14. Part of the generated test selection model 27
Figure 15. Example NIST test room layout 1. 28
Figure 16. Example NIST test room layout 2. 28
Figure 17. Part of the generated test selection model for the two layouts 29
Figure 18. The output of the tool for the room with two layouts 29
Figure 19. Meta-model of the demonstrator context 30
Figure 20. Example instance of the demonstrator context 30
Figure 21. The generic test selection model for the example with the demonstrator context 31
Figure 22. The output for the example with the demonstrator context 31
Figure 23. The example room instances 32
Figure 24. Execution time of change detection with various model sizes 34
Figure 25. Execution time of change detection with various number of changes 35
Figure 26. Execution time of RTS with various number of changes 36

List of tables

Table 1: Change detection time with different model sizes 34
Table 2: Change detection time with different sizes of changes.....35
Table 3: RTS execution time with different sizes of changes 36

List of Acronyms

DSL	Domain-specific Language
EMF	Eclipse Modeling Framework
MDD	Model-Driven Development
NIST	National Institute of Standards and Technology
RTS	Regression Test Selection
UML	Unified Modeling Language

1 Introduction

1.1 Summary (abstract)

WP34 of R5-COP aims at supporting the off-line and on-line verification of the behaviour of R5-COP systems by elaborating methods and tools for incremental testing and runtime monitoring. In a design or maintenance phase reconfiguration not all behaviour is affected by a change, therefore not all functionality needs to be re-tested. By excluding tests checking unmodified parts significant effort can be spared. In WP34 *methods and tools* are developed *for selecting, adapting and extending test cases from existing test suites* in an incremental way, in order to check the changed components or properties. The gaps in the coverage of the existing test suites are identified, which drives the adaptation of existing test cases and the generation of new test cases to cover the changes.

Deliverable D34.10[1] described the languages that are proposed for the description of the relation of test cases, system components and properties. A generic model was described that can represent requirements (e.g. in scenarios), the context of the system (e.g. in context ontologies and metamodels), and the internal components (e.g. in architecture and capability models).

The current deliverable D32.20 summarizes the results of Task 34.2: finalization of the models and approach, development of a prototype tool implementing the incremental testing, and preliminary evaluation of the tool.

1.2 Purpose of Document

The document describes the generic model designed for representing testing related artefacts (e.g. context, configurations, requirements and existing tests). It presents the approach to categorize existing tests after a reconfiguration. Next, the deliverable details a tool implementing the incremental testing method. Finally, the applicability and scalability of the tool are evaluated using examples connected to the demonstrators.

The incremental testing tool will be used in demonstrators (motivating examples are also described in this deliverable). The application of incremental testing will be evaluated in *Task 34.5: Integration and assessment*, and described in deliverable *D34.50 Assessment of the on-line verification and incremental testing*.

1.3 Partners Involved

Partners and Contribution	
Short Name	**Contribution**
BME	Definition, implementation and evaluation of the approach
PIAP	Review of the document

[1] Note: to be self-contained this deliverable will contain the relevant parts of D34.10.

2 Overview

This section describes the general concept of incremental testing in order to put into context the selection and definition of the description languages presented in the subsequent sections of the document.

2.1 Background

Quality has always been 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.

The kind of incremental testing approach planned in the project is usually referred as regression testing in the literature. 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” [1]. 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 regression testing techniques were proposed over the years, particularly to select only a subset of the test suite, what is relevant for the 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 document we use the categorization of tests introduced by Leung and White [22]:

- *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 [23]* comes in sight, when the total execution time of tests is not relevant, however discoverable errors shall be highlighted as soon as possible. By using *Test Suite Minimization (TSM)[18][20]* or *Regression Test Selection (RTS)* [28][29] the goal is to reduce the number of executed tests, especially when re-testing 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 minimal associated execution cost. This document 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 [2] , dynamic slicing [3], graph-based representation [19]. Several tools exist implementing RTS for source code. For example, SoDA [32] is a tool for C/C++ repositories, while ChOPSJ [31] is available for code written in Java.

In the past decade, the increasing adoption of models as development artefacts 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” [8]. These models are commonly composed using *domain-specific languages* (DSL). DSLs are special

languages for a particular problem domain. The model artefacts describe the system itself and could also serve as inputs for the testing process. Development of a system can be started by creating these models, however, for an existing robotic system theses model artefacts can be also produced afterwards. (The list of required model types will be detailed in Section 2.2.3.)

As MDD is conducted in an incremental manner, model artefacts – similarly to the source code – tend to change in time. The changes in the model artefacts 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 artefacts and system parts, regression test selection can be applied on model level rather than on the generated code.

Numerous approaches have been presented tackling this problem, still most of them is tied to specific models (e.g., UML [34][9] or finite state machines [20]) that requires reimplementation of the RTS algorithms for different domains.

2.2 Incremental Testing of Autonomous Robots[2]

To create reconfigurable robotic systems, not only the development but also the verification and testing activities have to take into account reconfiguration and changes. Time and resources required for testing can be reduced if testing is performed incrementally.

In classical software engineering terminology, the re-use of previous tests and test results is denoted as “regression testing”. Here “incremental testing” is used to address the stepwise extension/change of the functional scope of the subject under test throughout successive testing phases. To increase efficiency in testing, the previous tests and test results are reused and testing is focused on the changed part of the reconfigured system.

2.2.1 Basis of the Approach

In the preceding R3-COP project, BME developed a model-based system level testing method for testing the context-aware behaviour of an autonomous robot [24]. The test goal is to check the safe execution of a robot mission (e.g., transportation of goods without collision) in various contexts (e.g., in the presence of obstacles, humans, other robots and various environment objects). Accordingly, test contexts (arrangements of objects, obstacles etc.) shall be constructed systematically. To do this, we model the scenarios (describing the requirements against the robot) and the potential contexts of the robot (environment object types with their relations and constraints). On the basis of these models, our tool generates systematically the models of test contexts in which the mission of the robot can be checked. These generated test context models can be mapped to the configuration of a real test environment, a simulated environment (like in ROS+Gazebo), or internal representation of perceived context of the robot in ROS (depending on the implementation). Various test generation strategies can be supported, like generating extreme contexts for robustness testing.

Figure 1 presents an example from an autonomous forklift. The left hand side depicts the context model representing that a forklift can move on segments and can interact with people, pallets and other forklifts. On the bottom left a scenario states that if a person moves close to the robot (it is in the so called “warning” range), then the robot has to react with an alarm sound. Another requirement is that if the person is too close (in the “danger” range), then the robot has to stop. From these models the test generator tool creates models describing test contexts (one of them is depicted on the bottom of Figure 1). Test context models place different objects and persons around the robot to verify that it can handle multiple,

[2] Text from D34.10

possibly conflicting requirements (e.g., when one person is in warning range, and another is in danger range).

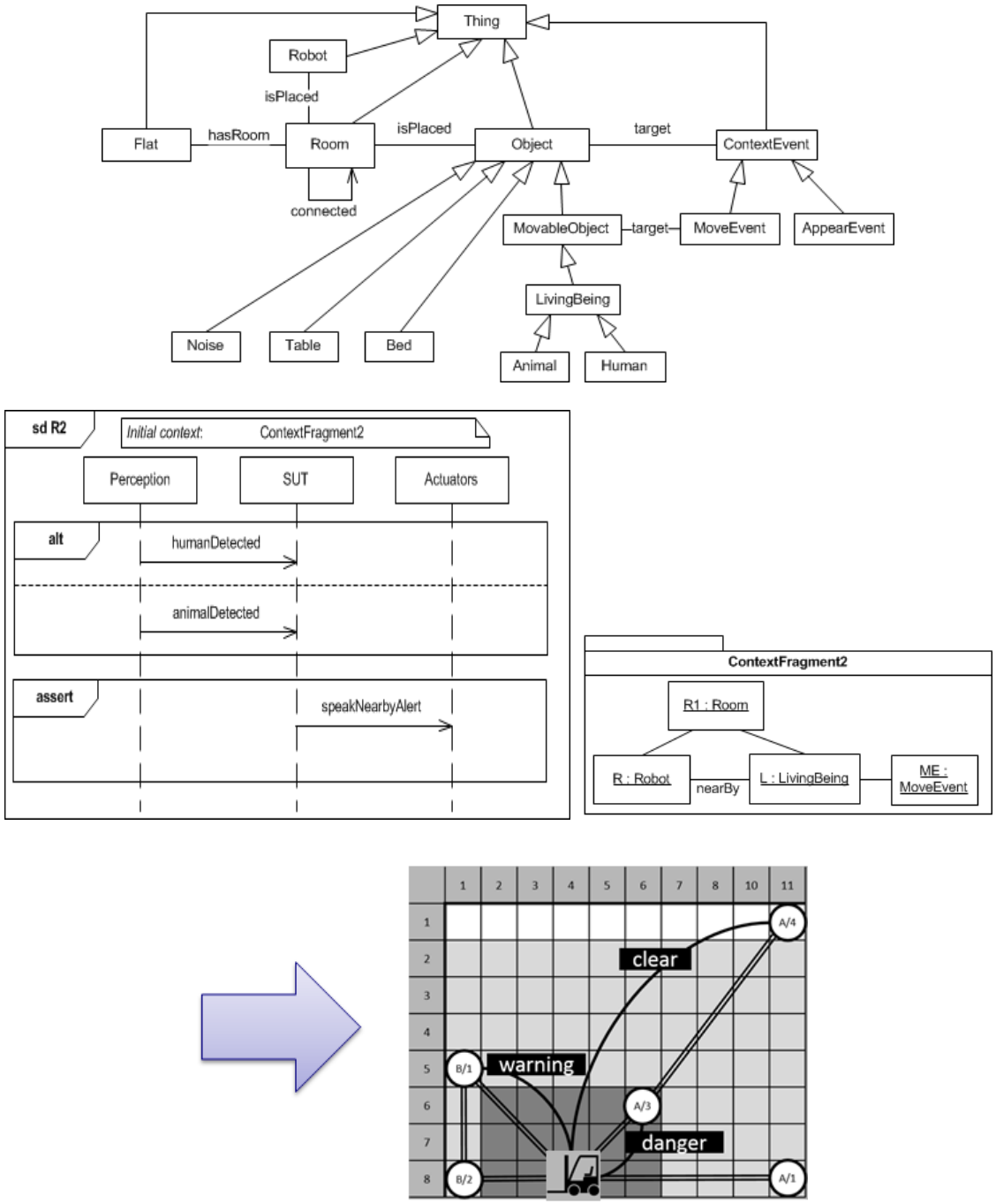

Figure 1. Creating test data (at the bottom) from context models (at the top)

2.2.2 Reconfiguration

In a reconfiguration, the context or the configuration of the robot can change. For example, a new type of object can appear in the environment, a requirement is modified, or a new type of sensor is added to the robot. In these cases the most basic strategy is to run all previous tests (called retest-all). However, this is not an optimal solution, as some of the previous

tests are re-usable, re-testable or obsolete. Moreover, new tests may be needed, i.e., it has to be identified which new tests are required after a reconfiguration.

The following possible reconfiguration scenarios are investigated:

- The context or the requirements of the robot changed: In this case the related models are changed and it is identified (1) which previously generated test data are invalid now, and (2) which part of the new context model is not covered by the existing tests. Figure 2 presents an example: part of the context model is removed (e.g., the robot will be used in a different context), thus one of the previous tests is obsolete.

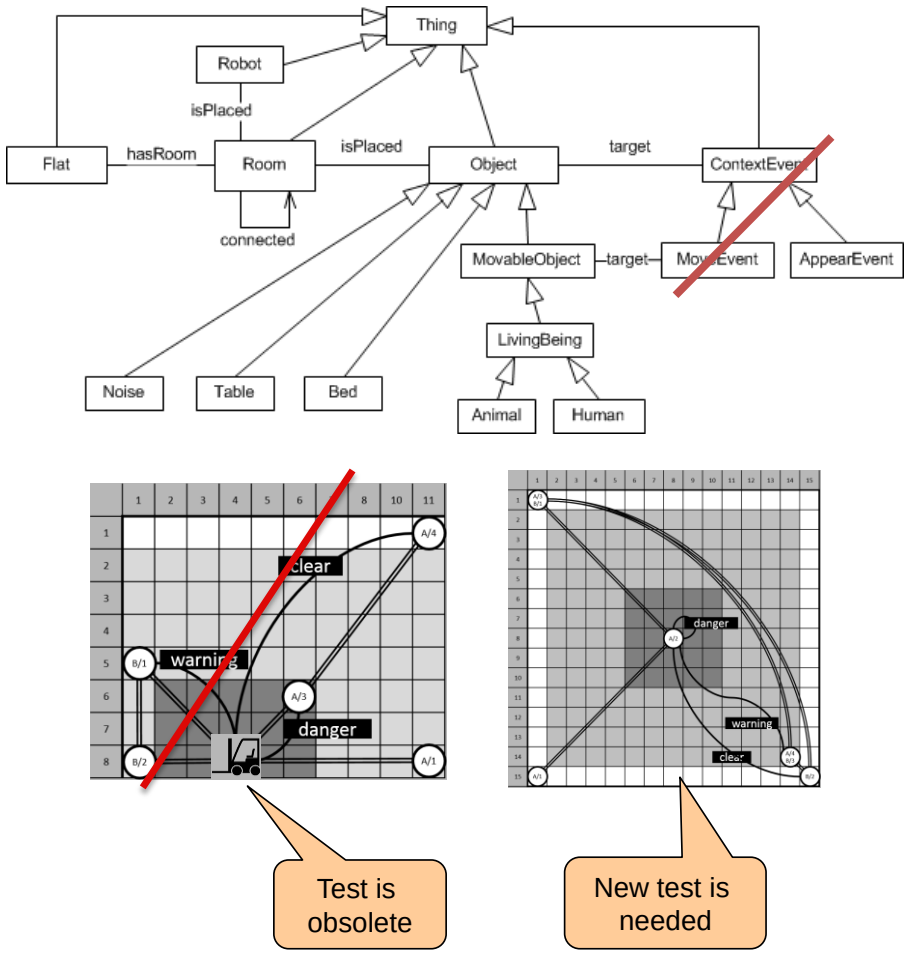

Figure 2. Change in tests due to reconfiguration in the context or environment

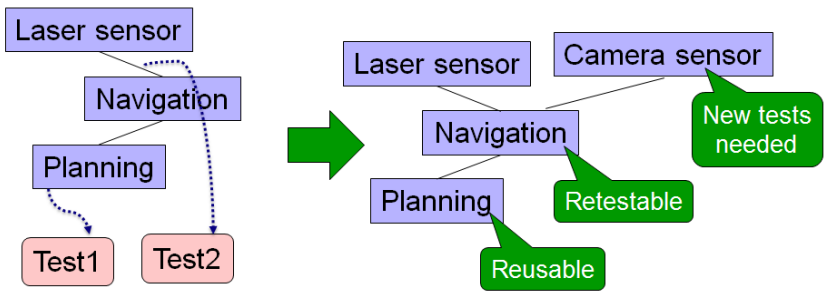

Figure 3. Effects of change in the robot configuration

- The configuration of the robot changes: For example, a new type of sensor or navigation method is added to the robot. In this case, if there is a mapping between the tests and the components/skills exercised by these tests (e.g., when during a given test the

robot uses a laser sensor then a mapping between the test and the laser sensor is recorded), then based on the description of the configuration the tests can also be classified. (Note that the description of the configuration could be obtained from the skill composer tool of WP 3.5.) Figure 3 presents a simplified example.

2.2.3 Inputs and Outputs

In order to perform this kind of analysis in a demonstrator, the following inputs are needed:

- Description of the demonstrator's components: list of the major components and the dependencies between them (e.g., details of the architecture or the skill models developed in the project).
- Description of tests: mapping of existing tests to components (e.g., an integration test suite checking the communication between the sensors and the navigation module).
- Description of context: description of the environment of the robot used in testing.

The outcome is a method for

- the identification of the tests that need to be executed after a reconfiguration,
- the identification of those parts of the system which are not covered by the existing tests.

2.3 Required Descriptions and Languages

Several types of artefacts shall be captured to form the inputs for test analysis for incremental testing. These (types of) artefacts can be grouped into two categories as follows:

1. The so-called *common artefacts* that have to be captured for test analysis:
    - *Context elements*: As the verification of context-aware autonomous behaviour is addressed, the context elements include environment objects, their properties, the potential relations among them (e.g., abstract relations as "close to", "lying on"), and the constraints among them (including physical constraints as well as domain-specific logic constraints).
    - *Configurations*: As re-configuration is a key concept both for incremental testing and on-line verification, the (current) configuration of the system and its changes shall be captured. Configuration can be considered as a hierarchical structure of skills (from which an application is built), software components (that realize one or more skill), and hardware devices (that are used by the software components using specific interfaces).
2. The specific artefacts that are relevant for incremental testing:
    - *Tests*: Tests are captured as basic entities.
    - *Testables*: The term testable is a common artefact that includes everything that can be addressed (covered) by a test: Context fragments (relevant subset of context elements), requirements, configuration fragments (relevant subset of configuration elements), source code snippets, etc. are represented under this common term.
    - *Mapping*: The mapping is a relation "tests" among tests and testables.

3 Approach for Incremental Testing[3]

This section introduces the elaborated approach of incremental testing for autonomous robots by presenting the vision and the detailed description.

3.1 The Envisaged Approach

The existing tools and approaches usually concentrated on one programming or modelling language as the input source for incremental testing. However, as we have seen in the previous sections, in R5-COP there could be multiple levels and types of reconfiguration. Instead of performing incremental testing separately for each of the change types, we could apply a unified approach, as basically they all belong to the same problem.

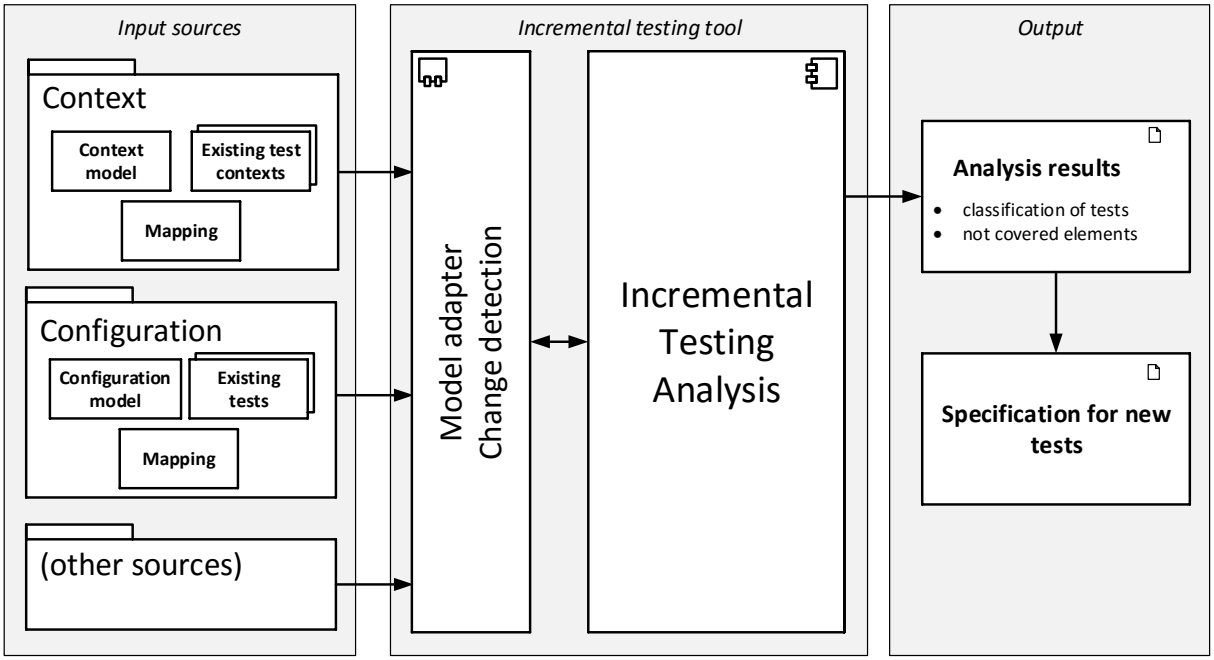

Figure 4. Overview of the incremental testing methods

We recommend to develop a *common, general incremental testing approach*, and connect the specific test types (test contexts from context models, module/integration tests for components, etc.) using special adapters to this core. Figure 4 depicts the approach in detail.

- The incremental testing analysis component is the central element of the approach. It defines a very general model for representing the tests and tested elements. The regression testing algorithms (test selection or coverage identification described in the previous section) work on this general model.
- A model adapter is responsible for connecting the different sources, like context or configuration models and tests to the general analysis component. This adapter should be developed for each source type and is responsible for converting the models and tests to the internal representation of the analysis component. This component is also responsible for detecting changes in the sources.
- The outcome of the analysis is a classification of tests as described in Section 2 and the coverage information of the source elements (e.g. there is a class in the context model

[3] Text mainly from D34.10, but the generic model is updated with new elements.

that is not present in any of the existing test contexts). This information can later be used to create new tests either manually or automatically.

The next section will detail the required description formats for these elements.

3.2 Description for Incremental Testing

In this section, we provide a detailed description of the generic model for our incremental testing approach. The section also presents the main concepts and the required inputs for genericity and incrementality.

3.2.1 Generic Model for Incremental Testing

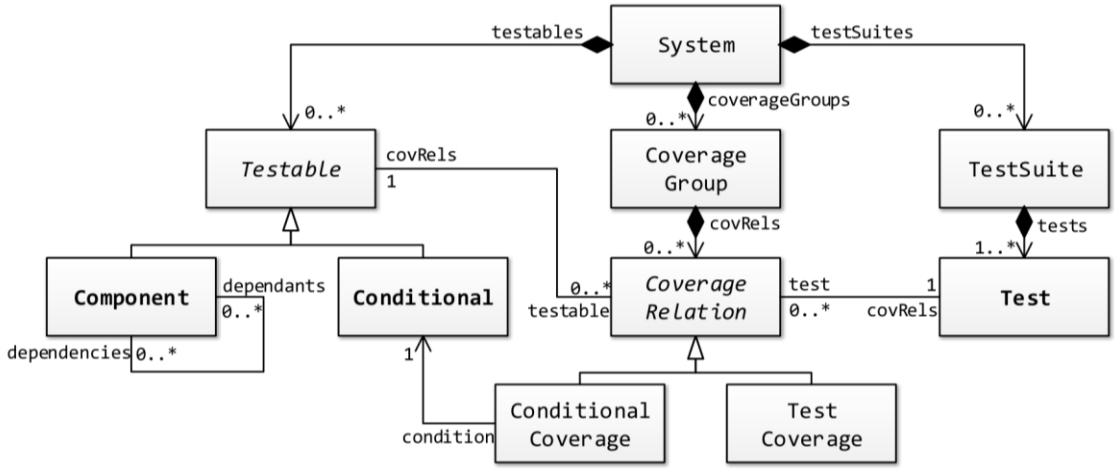

Figure 5. Metamodel for describing incremental testing

As described in the previous sections, three concepts are relevant in case of incremental testing: 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 generic RTS model contains four main concepts that is eligible to describe the underlying artefacts of arbitrary systems 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, which requires individual handling during the RTS process.
- *Test:* represents an executable test case in the system.

Other crucial concepts are also used in the whole generic RTS model, which is shown in Figure 5. Metamodel for describing incremental testing. The main component of the generic model is the *system*. A system consists of *testables*, *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 tests connected to testables through coverage relations of coverage groups. A coverage relation connects a testable and a test (denoted with association). An instance of 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 a conditional element additionally (marked with association), that requires individual handling of condition values during regression test selection. A coverage group hold 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 generic RTS model using a special attribute.

3.2.2 General Mapping of Input Models

In order to produce this generic model a mapping is needed where the inputs are the system and test models, and the result is the generic model itself. The transformation should use unique identifiers to trace back elements to the original models. This transformation is partially specific to the domain actually used in order to have a domain-independent RTS model.

Notice that changes in the original models shall be represented in the generic RTS model. To tackle this question, our approach employs *checkpointing of models*, which is a common model versioning technique [5]. 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 without intervention of the user.

By using the mapping, the selection becomes independent from the input models. The implementation of the RTS is bound to the generic RTS model this way it is not necessary to (re-)implement it on the basis of the specific model artefacts and coverage models.

3.2.3 Using Context Models as an Input for Incremental Testing

When the reconfiguration is performed in the environment or application domain of the robot, it can be reflected with changes in its context model. In this case, the incremental testing analysis should find those test contexts, which

- contain instances of modified context model elements,
- are invalid, because they contain instances, whose type has been deleted.

Moreover, the approach should identify not covered context model elements.

Context models are basically class models, which describe the environment of the robot under tests. The requirements for using contexts models as input sources for incremental testing are the followings.

- Context models should be specified as UML class diagrams or Eclipse Ecore models.
- Test contexts should be specified as instances of the context model.
- The mapping of tests and testable is not required to be given separately, as the instanceOf relation between an instance objects and its meta-element can be used for this purpose.

Thus, in case of context model, it is relatively straightforward to use them as inputs for the envisaged incremental testing approach.

3.2.4 Using Configuration Models as an Input for Incremental Testing

When the reconfiguration is in the capabilities or components of the robot, then it can be captured with changes in the configuration or skill model. As the skill model of R5-COP is still in development, we could not yet use directly it, but the following general requirements can be formulated.

- The configuration model (components, skills, etc.) should be given as a graph-based model, preferably a UML or Ecore model. It should describe the hierarchy and dependency relations between the configuration elements.
- The list of test projects, test suites or individual test cases should be specified.

- The mapping of tests and configuration elements needs to be specified. A relation should exist between a test and a configuration element, when
    - the test checks directly the element (e.g. a module test is written for a given component),
    - the test needs the given element for its execution (e.g. an integration test requires also the service provided by the component to start).

If the configuration model is given as a UML element, then the list of tests and the mapping can also be incorporated in the model. Otherwise, the mapping can be specified in a textual format, e.g. an XML file.

3.2.5 Description of Change and Reconfiguration

So far the models describe only one given context or configuration. However, a crucial part is to include the changes induced by a reconfiguration of the systems. Thus, the concept of "change" should somehow appear in the descriptions used in incremental testing. There are three fundamental ways to achieve this.

1. *Annotate the model*: annotate the source models with tags or stereotypes describing new, changed or deleted elements.
2. *Trigger-based support*: if the modelling environment supports hooks and triggers to notify about model manipulation, then the changes can be detected in this way.
3. *Calculate diff between models:* if an old and new version of the model is given, then the difference can be calculated

The first solution could be quite cumbersome. For a simple model, annotating it by hand could be done once, but maintaining the annotations through several changes in a large model is not preferable.

The second solution could only be used, if the input sources (context or configuration models) are created in a modelling tool, and the tooling supports change detection. Such functionality exists for instance in the Eclipse-based modelling tooling.

The third option can be used without any special modelling environment support and it does not require extra effort from the user either. However, calculating differences in large graph models in not trivial.

4 Implementation of the Approach

4.1 Overview of the Architecture

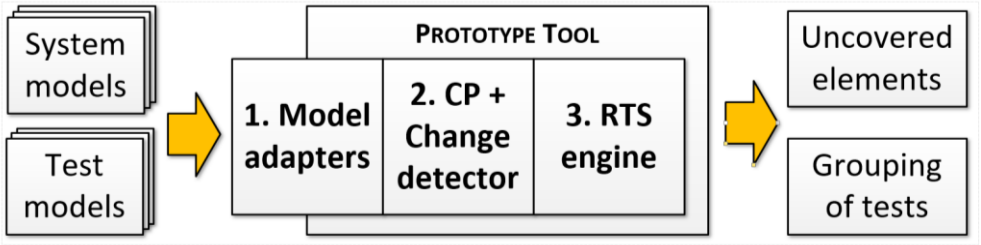

Figure 6. Architecture of the prototype tool

The approach presented in the previous sections is implemented in a tool using the Eclipse Modeling Framework. To ensure independence between the input models and the RTS, the tool was given a layered architecture as shown in Figure 6.

As the input models can be arbitrary, adapters are required for defining the mapping to the generic RTS model. A *Model Adapter* consists of transformations that map the domain models to the RTS model. The tool 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 [7].

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. 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 the well-known version control systems like Git and SVN.

The third layer of the tool is the *RTS engine*. This layer performs the generic RTS 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.

4.2 Possible Use Cases

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 analysis of impact to identify the effects of particular changes,
3) re-testing the system after changes have been applied.

Re-testing time should be reduced as low as possible along with maintaining the same fault-detection capability of the test suites.

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 frequent 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 domain-specific languages shall define the mappings and transformations to the RTS optimization model, that can be used later by the test engineers.

4.3 Details of the Implementation

This section gives insights into the prototype implementation of the generic incremental testing approach.

4.3.1 General Structure

The Eclipse Modeling Framework (EMF) is commonly used as one of the "de facto standard" toolkits for model-driven development projects. Hence, for wide-range applicability, the prototype tool also uses EMF for handling domain and RTS optimization models.

The prototype tool is built using the popular Eclipse framework due to the fact that EMF is also an extension of Eclipse. Eclipse allows implementation and addition of plug-ins as bundles that can be loaded into the framework runtime. All layers of the implemented tool are bundles and have their own level of isolation inside the framework. Bundles are able to provide services and extension points, thus the prototype tool is easily extensible.

The tool is implemented using eclipse Plugins. Since Eclipse version 3, the platform is based on an OSGi framework implementation, Equinox. This has recently become the reference implementation of OSGi. The previously mentioned Eclipse plug-ins are OSGi bundles.

An OSGi framework extends the Java Virtual Machine with a dynamic component model. An OSGi bundle has its own classloader and memory space, so bundles provide a level of isolation inside the framework. Bundles can also provide services, export packages, and express dependencies to import the packages that another bundle exported.

Eclipse plug-ins provide further extension points that dependant plug-ins can register to. These extension points are used e.g. to derive menu contributions in the user interface. Each plug-in implemented has a containing project, this way the source code is easily reusable, and the scope of the dependencies can be limited to the code that uses it.

4.3.2 Layout of the Projects

Complying with Eclipse conventions, all plugin are named using a fully qualified name, with a prefix indicating the authors and the tool.

Plugins ending with suffix “.model” contain the EMF models created. EMF provides support for automated generation of graphical editors, they are contained in edit and editor projects.

Separate projects have been created for IncQuery patterns (“.queries” suffix). The aim of this separation is that these projects need IncQuery dependency, not all the projects invoking it.

The core logic of the tool is implemented in plugins with suffix ".tool", the ".tool.changes" plugin contains integration code for the IncQuery patterns, similarly "tool.ui.button" contains the UI controls contributed to the IDE.

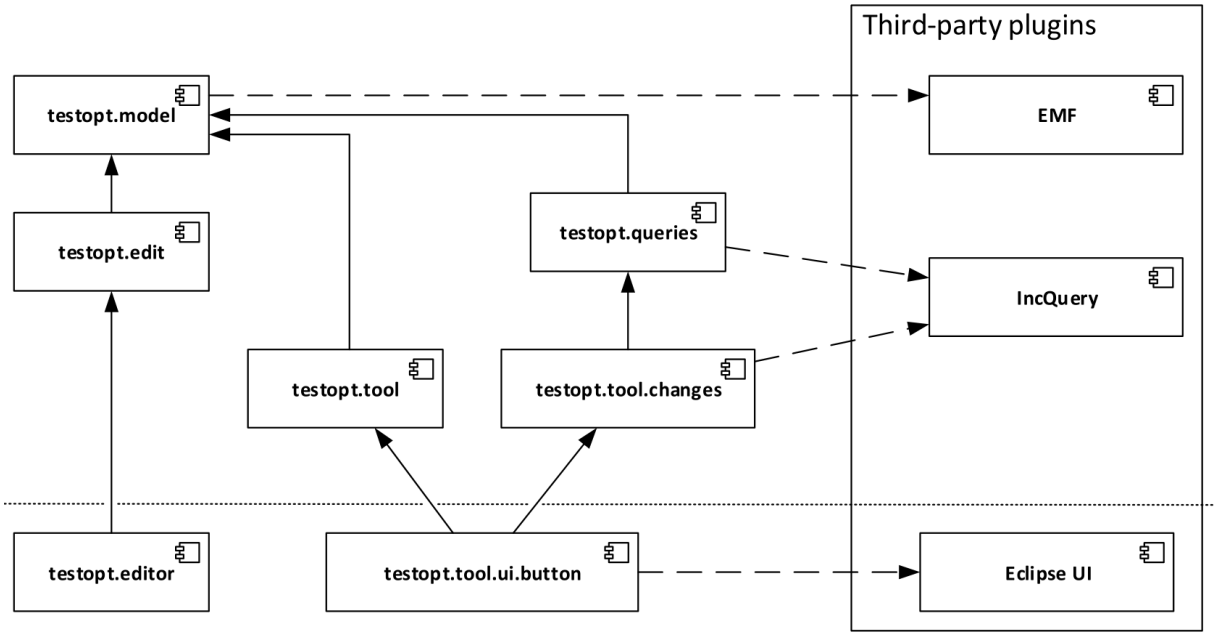

Figure 7. Component model of the core plugins in the implemented tool

4.3.3 Model Adapters

Our generic approach requires model adapters in order to work independently from the domain models. These adapters describe model-to-model mappings and can be implemented using the interfaces provided by the prototype tool. The adapters are registered into the tool via metadata files indicating the type of model and adapter to use. The adapters use VIATRA, an EMF-based model transformation framework in order to conduct the conversion from the domain models to the generic RTS optimization model. We decided to choose VIATRA as it is considered one of the state-of-the-art incremental model transformation frameworks. As the tool uses graphs in the background algorithms, arbitrary domain models and languages can be handled. Using VIATRA requires the definition of patterns that can be matched to different domain model elements. A simple example is shown below, where the first pattern searches for modified elements in the mode, and next, the second patterns finds the tests that that are re-testable.

```
pattern hasChanged(obj : Modifiable) {
    Modifiable.modification(obj, ::Change);
}

pattern testToBeReRun(test : Test) {
    // test has changed
    find hasChanged(test);
} or {
    // tests may be re-run for changed components
   find tests(testable, test);
   find componentToBeReTested(testable);
}
```

Then, a transformation with VIATRA can be defined for each match of the patterns, hence making able to map input model elements to new elements of the RTS optimization model.

The checkpointing layer is also an extensible part of the system. The prototype implementation in the tool currently uses the file system with time stamps for model versioning. However, this layer can be developed further to collaborate with the well-known version control systems like Git and SVN.

The detection of changes between checkpoints is conducted by comparison. The prototype tool compares all of the input models in the last checkpoint to the current versions to indicate the changes in the already generated RTS optimization model. The comparison process uses EMF-Compare, which is a model comparator API for EMF provided by the modelling framework natively. One may notice that only the different versions of the optimization models could be enough to compare instead of all the inputs, however the RTS optimization models do not provide information about changed. For this reason, the tool implements a generic algorithm to compare all the input model elements between checkpoints and to indicate changes in the optimization model using the unique identifiers. The result of the comparison may yield four different outcomes: none, addition, deletion or change. The changes are indicated in the optimization model using a simplified model manipulator API provided by VIATRA

4.3.4 RTS Engine

The RTS engine layer is responsible for conducting the RTS. The prototype tool uses a simple set cover approximation algorithm to solve this problem shown on Figure 8). As the Set Cover Problem is known to be NP-hard, this algorithm has a greedy nature: it would always select the next minimal covering subset, thus can turn out to be suboptimal. The approximation factor of the algorithm is ln(n+1), where n denotes the size of the optimization model. This value can be acceptable even for many changes in large models. The layer is also flexible, hence new algorithms can easily be implemented to solve the problem of RTS.

Input: base set U of size n to cover
Input: $\mathcal{S} = \{S_1, S_2, \ldots, S_k\} \mid \forall i \colon S_i \subseteq U$; i.e. the set of subsets to use
Input: cost function $c \colon \mathcal{S} \mapsto \mathbb{R}^+$ that returns a positive number for each element of $\mathcal{S}$
Output: A covering set with cost at most $(\ln n + 1)$ times worse than the optimal
Variable : $C \leftarrow \{$the set of already covered elements of $U\}$
Definition: Let the `value` of a subset S_i be the cost it would have, if it was chosen next; i.e. the average cost of the newly covered elements: $c(S_i)/\,|S_i \setminus C|$.

```
1 Selected ← ∅;
2 while U \ C ≠ ∅ do                          // there are uncovered elements
3 |  Find S_i ∈ (S \ Selected) that has a minimal value(S_i);
4 |  C ← C ∪ S_i;
5 |  Selected ← Selected ∪ {S_i};
6 end
7 return Selected;
```

Figure 8. Greedy approximation algorithm for the Set Cover Problem

4.3.5 Implementation of the UI

In the current phase of the implementation, both the IncQuery engine and the test selection are started from the GUI by the user. To provide controls for the user, We have implemented a separate project, with suffix “ui.button”. To register the buttons, we have used the Command API of Eclipse. The key idea of the API is to separate the configuration into reusable parts:

- Command: a declarative description of the component
- Handlers: responsible for the actual behaviour (i.e., it is the Java code being called)

- Menu Contributions: declarative definition of menu entries, be it the main menu or a context menu entry. Even buttons on the toolbar are defined this way. The placement and the nature of the menu contribution is defined via a special URL.

With these parts, for a command a single handler could be called from multiple menu contributions, e.g., from the toolbar and also from the context menu.

4.3.6 Summary of Details

This section described how the tool is implemented in the Eclipse framework. Figure 7 shows the plugins responsible for the core functionality and their relationships with each other. So far the prototype of the tool with all of its plugins, including the model builders and transformers, consists of 23 plugins.

5 Detailed Examples of the Approach

This section intends to present the elaborated approach through detailed examples. The examples use a predefined metamodel set, which we specially defined for demonstrational purposes.

5.1 Preliminaries

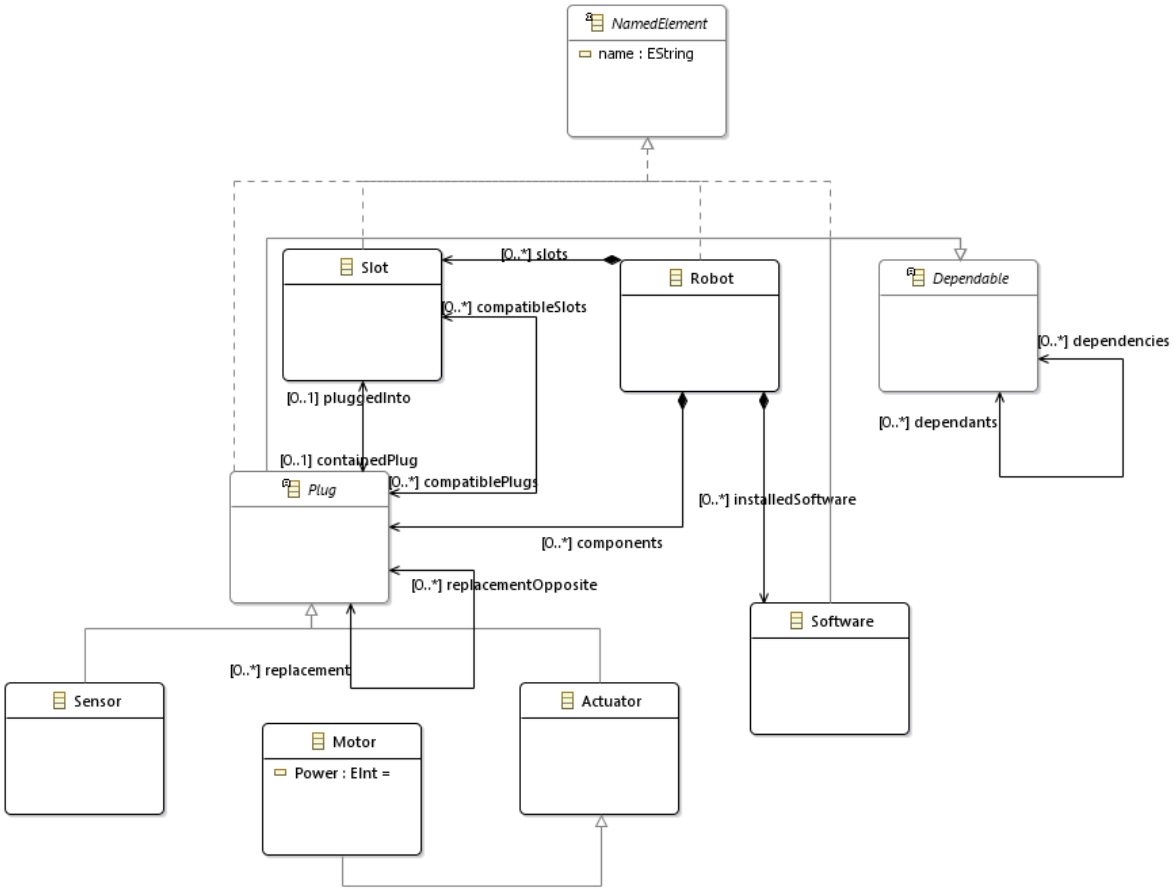

Figure 9. The robot metamodel used during the examples

We designed several metamodels and instance models in order to present the application of the approach. First, we elaborated a robot metamodel, which could be general enough for demonstrational purposes. The metamodel on the robot is found on Figure 9. The model includes 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. Figure 10 shows the simplified capability model of the designed sample robot, which is used during the following examples.

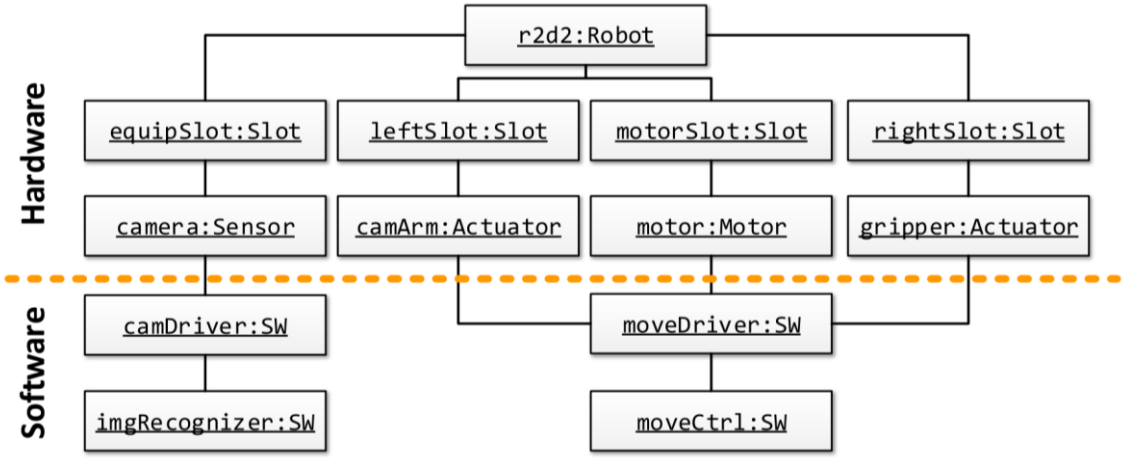

Figure 10. The example robot instance on a simplified representation

The robot model 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.

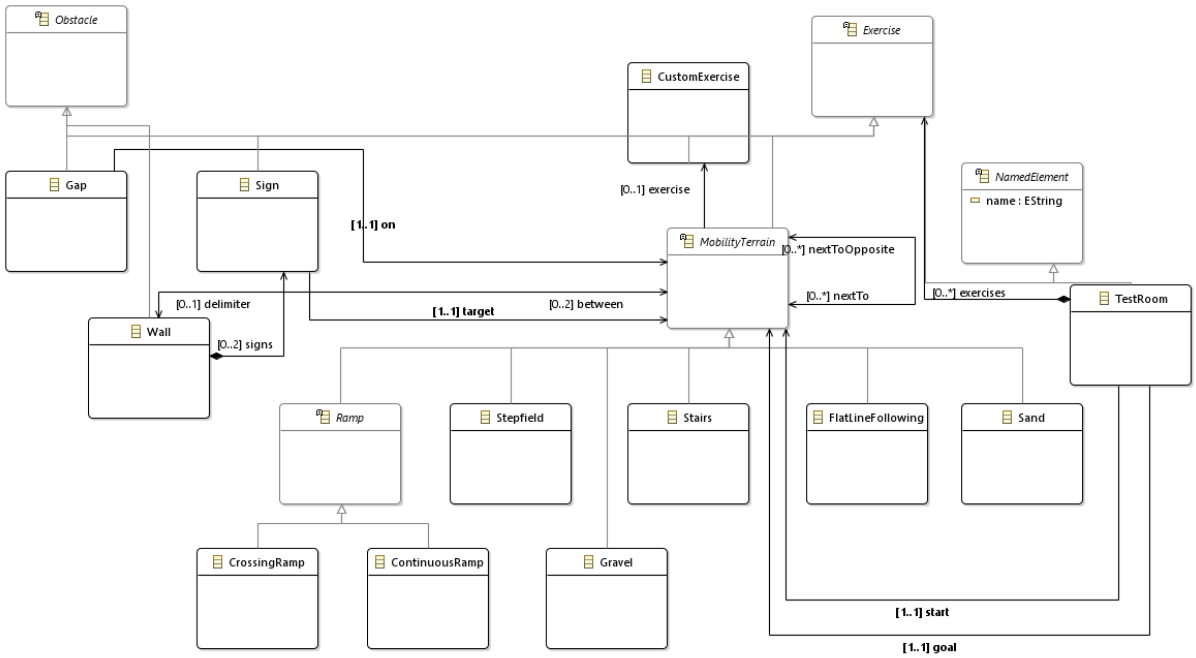

Figure 11. The NIST context metamodel

We defined a context metamodel based on the guidelines for autonomous robot test rooms created by the National Institute of Standards and Technology (NIST) [6][25]. Note this standard is used also elsewhere in the project, e.g. for the validation of HMI interfaces presented in D24.40 “Usage patterns, tests and validation results”.

We extracted the main concept from the guidelines and the attached examples. The elaborated metamodel is found on Figure 11. The main element of a test room metamodel is the exercise, which can be:

- a specific type of terrain tile, on which the robot under test moves,
- a gap between terrain tiles,
- a sign on the wall or on a tile,
- a custom exercise that is defined during the design of a concrete instance.

We defined 7 different types of terrain tiles, which are the following:

- Crossing ramp: a small-heighted ramp passes across the tile covering a cable tunnel.
- Continuous ramp: a ramp, which connects two tiles with different heights.
- Stepfield: a tile filled small steps with the same heights.
- Stairs: a tile with stairs that connects two tiles with different heights.
- FlatLineFollowing: a simple flat, plastic tile with a line that can be followed.
- Sand: a tile filled with sand.

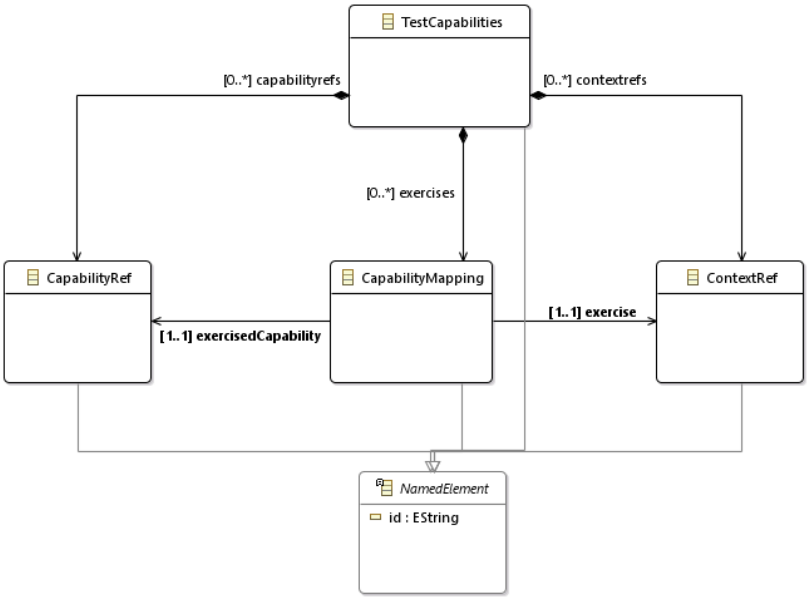

Figure 12. Mapping metamodel for different context elements and the capabilities of the robot

In order to have a coverage definition in our generic incremental test selection model, a mapping metamodel shall be designed first to connect the concepts of the context and the components under test. The elaborated metamodel is found on Figure 12. The main element is a container that stores mappings. A mapping consists of exactly one capability reference and one context reference. These references are pointing to corresponding model elements by their identifiers.

In summary, we defined three metamodels for the examples:

1) a capability model that describes the robots being developed and tested,
2) a context model that represent test rooms for the robots under test, and
3) a mapping metamodel, which is able to connect the elements of the contexts with the capabilities of the robots.

5.2 Example with a NIST test room

In this example, we present a scenario, where the elaborated incremental testing approach for model-based system testing may help reducing execution time and cost. The test room depicted on Figure 13 consists of 8 tiles separated with a wall (indicated with orange line). The first tile has a simple gravel terrain and serves as the starting point for the robots. Tile 2 is a staircase with increasing height in the direction of the arrow. The third and seventh tile is a simple gravel terrain similarly to tile 1. The fourth tile is a simple ramp with increasing height. Tile 5 consists only of a crossing ramp. Tile 6 has a sand terrain and contains a cone as an exercise. Tile 8 has gravel as its terrain and contains a red button as exercise, where the robot finishes the test room. Each tile and exercise on a tile may serve as a test case for the robot being tested. Note that due to using ramps and staircases a real test room based on this scenario should have different vertical levels but the current abstract context meta-model does not contain this information. These details could be later added in a concretization step [24].

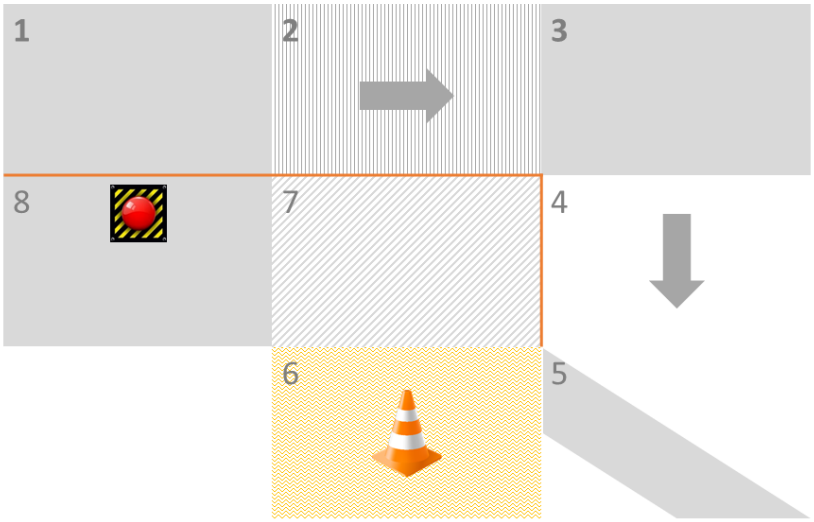

Figure 13. Schematic layout of Test Room 1

In order to complete the overview of the scenario, a mapping shall be made between the elements of the robot and the exercises in the test room. For this example, we define this mapping as follows. The other parts of the context and the capabilities are out of scope for the example.

- motor ←→ tile_2
- motor ←→ tile_4
- motor ←→ tile_5
- gripper ←→ red_button
- gripper ←→ cone

Using the implemented incremental testing tool, we defined this mapping as an instance model using references of context elements and robot capabilities. This results in a coverage relation that is used during the incremental test selection procedure.

This initial state of the 3 instance models (robot, context, mapping) forms the very first checkpoint of the system development. We used the tool to mark this state as initial. For the purpose of this example, we considered that the specification of some parts of the robot must be changed at some point during the development. More specifically, the motor and the gripper is replaced to a newer version, thus re-testing gains priority to ensure conforming behav-

iour. Without any test selection, the changed parts would trigger re-running all considered test cases (tile_2, tile_4, tile_5, red_button, cone).

Thus, to demonstrate the potential of incremental testing, we introduced the two modifications to the robot instance model and executed a development checkpoint using the implemented tool. The creation of the checkpoint triggers the instantiation of a new generic RTS model indicating the changes carried out in the robot model instance. This model is depicted on Figure 14.

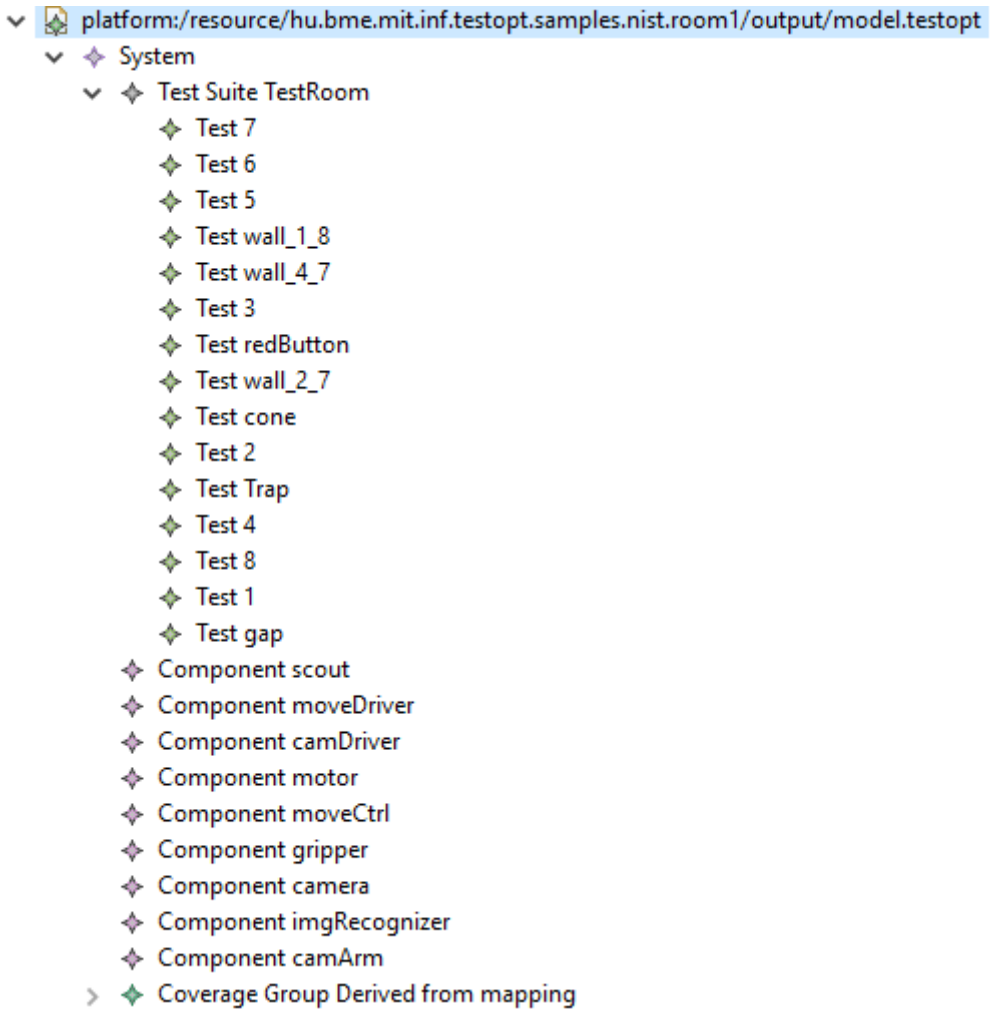

Figure 14. Part of the generated test selection model

The final step of the example is to calculate the tests to re-run for this specific modification. We executed this function of the incremental testing tool, and obtained the following results: only tests tile_4 (marked with Test 4 on figure) and cone (Test cone) must be re-run to re-test both components. Therefore, instead of manoeuvring through the whole room, the robot can be placed on tile 4 and just directed to grab the cone, this shorter path would exercise the changed functionality.

Note the example was simplified for demonstrational purposes, but with adding further constraints to the model more realistic setups can be analysed, i.e. tile_2 and tile_2 testing different capabilities of the motor.

5.3 Example with two NIST test layouts

Consider the following situation. A test facility has only one test room, which always must be organized into different layouts for different test situations. In these cases, frequent changes

of the robot under test demands for repeated reorganizations that consume large amount of time and effort. The current example illustrates this problematic situation. On Figure 15, the first test room layout is depicted. The first and second tile is a simple terrain with gravel and sand, respectively. The third tile is also a gravel, however it contains a cone, which must be placed out of the way by the robot.

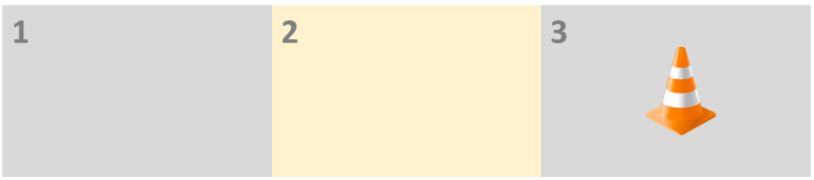

Figure 15. Example NIST test room layout 1.

The second layout of the test room is found on Figure 16. The first two tiles in the room have gravel terrain, while the last one has sand. Furthermore, the second tile contains a radioactivity sign that must be recognized by the robot in order to pass the test suite.

Figure 16. Example NIST test room layout 2.

Similarly to the first example (Section 5.2), a mapping between the context and robot elements must be given. This results in the following coverage relations.

- motor ←→ testRoom1.tile_2
- motor ←→ testRoom2.tile_3
- gripper ←→ testRoom1.cone
- imgRecognizer ←→ testRoom.radioActivitySign

For the purpose of this example, we introduced the same modifications as in the previous example, namely the gripper and the motor is upgraded.

Without any incremental testing, the re-testing all approach must be applied, which requires reorganizations of the one and only test room.

In order to alleviate this situation, we employed the incremental testing tool by defining the two test room instance models and their mapping with the capabilities of the example robot. The crucial part of the yielded generic RTS instance model – generated by thee tool – can be found on Figure 17.

In order to obtain the results from the tool, we executed the regression test calculation. The results are shown on Figure 18. Note that only the two test cases must be re-run to cover the modifications. Both tests are found in layout 2, thus only the second layout must be organized in the test room.

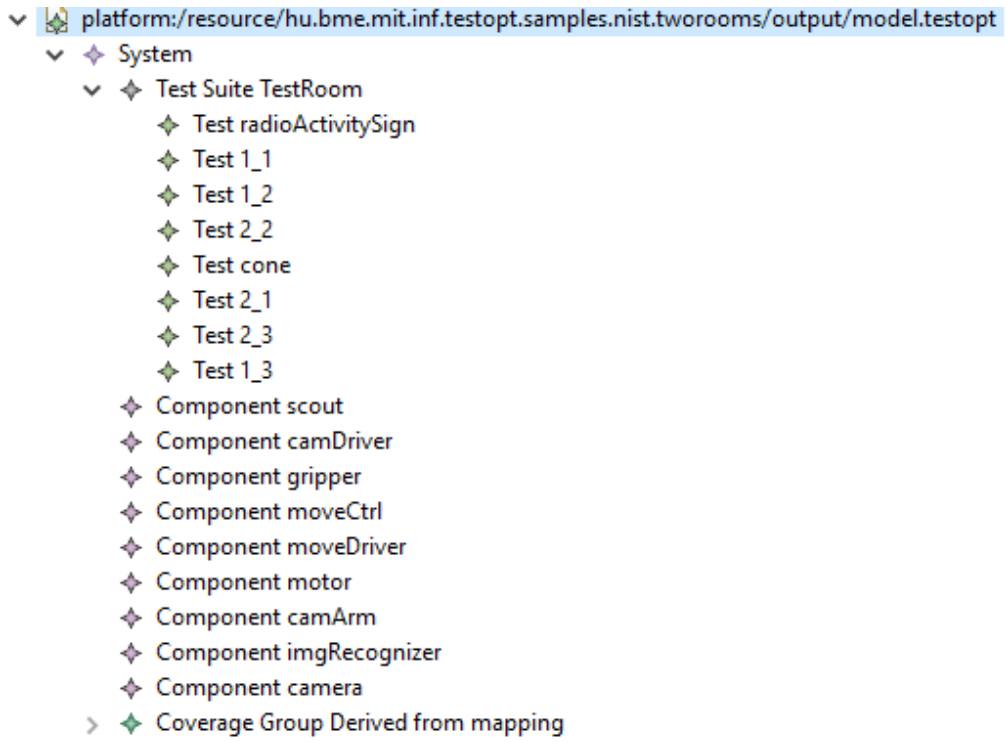

Figure 17. Part of the generated test selection model for the two layouts

```
Test run cost: 1.5
Re-run all cost for changed: 3.0
Tests to re-run:
        radioActivitySign
        2_3
Uncoverable, yet changed elements:
```

Figure 18. The output of the tool for the room with two layouts

5.4 Example with a demonstrator context

To demonstrate the generic applicability of the tool, we created another context meta-model, in which mobile industrial robots (like the robots of the project partner MIR) are involved.

The meta-model (Figure 19) considers the main concept of the context: walls, different types of floors, and objects like boxes or humans. The main element of the model is the TestRoom. A room consists of building blocks (walls and floors) and floor groups. A test room always has a starting and ending floor tile. The floor can contain different test objects. These can be the following: Box, Human, Robot. Note that this metamodel is extensible and is only created for demonstrational purposes.

The next step of the new context model integration was to create an example model. We modelled the following simple test room for this purpose. The test room has three tiles (as shown on Figure 20), tile 1 has another robot placed onto, tile 2 is empty, while tile 3 has a box on it. The starting tile is tile1, while the end is tile 3. The test room has the robot and the box as its exercises. Thus, we created the mapping between the robot capabilities and these exercises in the following way.

- gripper ←→ box1
- imgRecognizer ←→ robot1

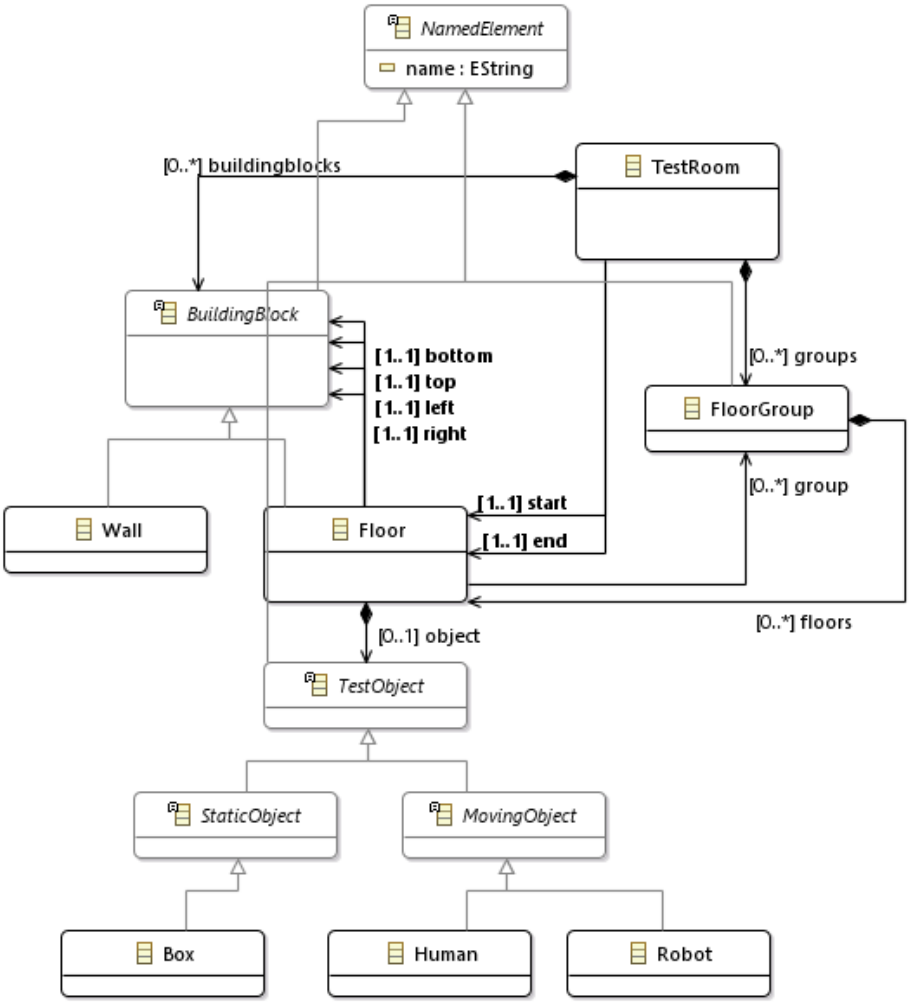

Figure 19. Meta-model of the demonstrator context

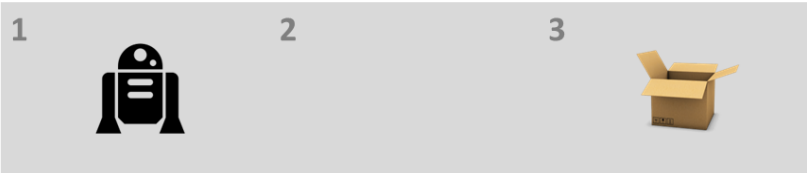

Figure 20. Example instance of the demonstrator context

Consider that a change is introduced to the gripper of the robot under test. Without test selection, both the robot1 and box1 shall be re-tested (complying with the re-test all approach). To avoid this, we applied our prototype generic incremental testing tool. We created the model instances for the new context and for the mapping between the robot capabilities and the elements of the context accordingly to the previously described mapping. In the first model development checkpoint (without any modification on the gripper), the generic regression test selection instance model is generated by the tool. This model instance can be found on Figure 21.

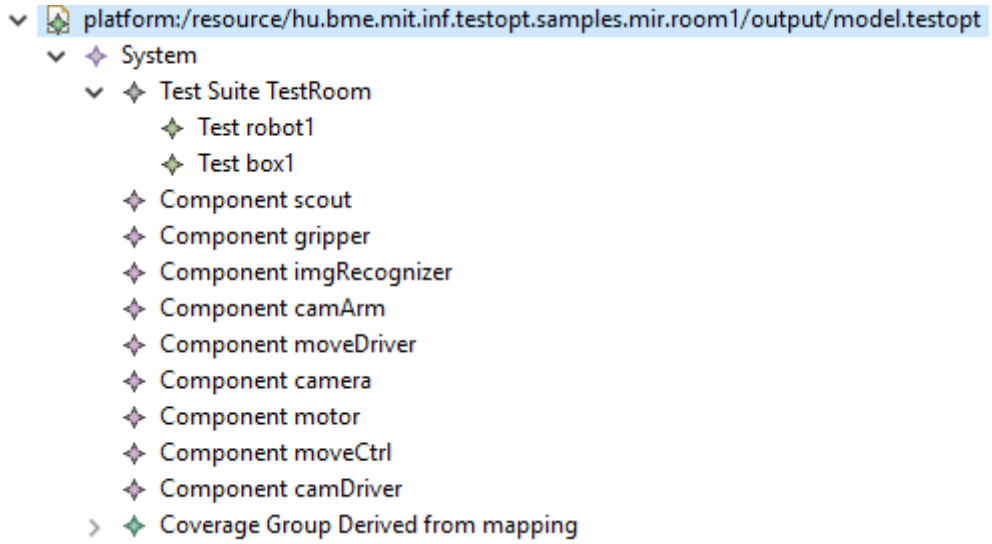

Figure 21. The generic test selection model for the example with the demonstrator context

Note that the only difference between this and the previous test selection models is that the tests are changed. This greatly emphasizes the genericity of the approach as it hides the specific attributes of the context and just copes with concepts that regression test selection algorithms use.

As we modified the gripper in the instance model of the robot under test and executed a checkpoint, the change was indicated in the generic test selection model. Then, we executed the test selection algorithm, which yielded the result found on Figure 22. Thus, the technique spared time and effort by avoiding the run of the robot1 test.

```
Test run cost: 1.0
Re-run all cost for changed: 1.0
Tests to re-run:
        box1
```

Figure 22. The output for the example with the demonstrator context

6 Evaluation of Scalability

As no repositories are maintained, where models and tests generated from test models are collected systematically, we decided to employ our own model for motivational purposes.

The verification of autonomous robot systems is an essential part of their development process due to their safety-critical nature. For example, an emergency response robot is 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 a special test room. These rooms are able to pose challenges for different capabilities of the robot [6][25].

The occurrence of changes in the requirements of the robots may trigger the modifications of the test rooms. 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 particular element of a test room is a test case for the robot. This testing approach [24] is an example for model-based system-level black-box testing.

As the goal of this section is to demonstrate the importance of regression test selection and its applicability, a representation is needed first that describes the problem domain. This representation shall be able to describe both the capabilities of the robots and the test rooms. Based on the guidelines of NIST [25], we defined the following main types of model elements for test rooms. 1) mobility terrain, 2) mobility obstacle, 3) visual target.

Note that we used the metamodels that were presented in Section 5.1. The evaluation uses the robot instance model also introduced in Section 5.1 (Figure 10).

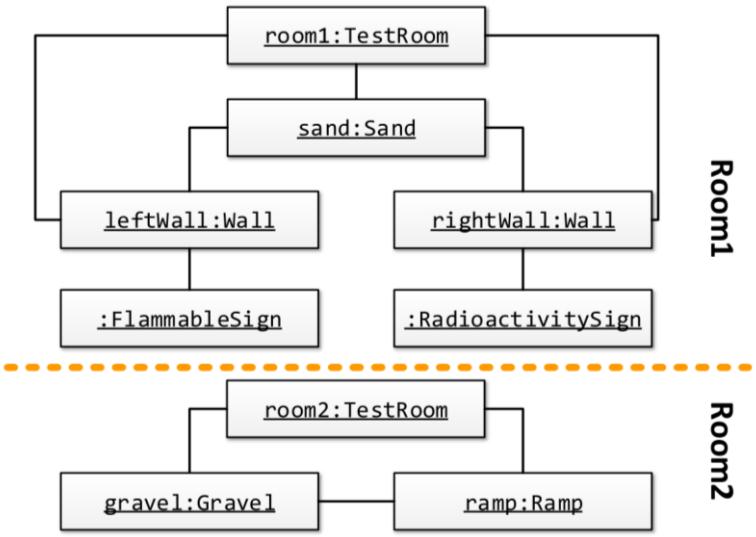

Figure 23. The example room instances

The test rooms are described using a simplified model (Figure 23). 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.

Furthermore, the definition of test cases requires a coverage mapping, which connects the objects in test rooms with the capabilities of the robot. This mapping is defined for the current example as follows: 1) *motor* → *sand*, 2) *motor* → *ramp*, 3) *imgRecognizer* → *leftWall*, 4) *imgRecognizer* → *rightWall*. Thus, the four test cases that exercise different capabilities of the robot are the following: *sand, ramp, leftWall, rightWall*.

In order to evaluate the scalability of the approach, we employed the instance models described in the beginning of this section. In this evaluation our intention is to answer the following question.

Could the prototype of the approach scale up to models found in the autonomous robots domain?

Answering the question demands for the mapping of domain models to the generic RTS model. For the purpose of this study, one transformation was defined for the model of robot capabilities, which transformed every element into a `Component` in the generic RTS model. Additionally, the test rooms were transformed into test suites and tests: 1) `TestRoom` → `TestSuite`, 2) `{Wall,Sign,Gravel,Ramp}` → `Test`. One may notice that we used TestRooms as test suites, though they can be handled differently as the level of abstraction is changed (e.g., using only a layout of few elements as a test suite instead of the whole room). Using this mapping, larger instance models of robots and test rooms are employed to conduct various measurements of scalability. Note that we used upscaled model instances of the example without the second room (room2).

6.1 Setup

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 generic 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.

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 10. 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 does it take 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).

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 generic 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.

6.2 Results

The values presented in this section were obtained from executing the prototype tool 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 [27], while execution times were measured by using stopwatches in code.

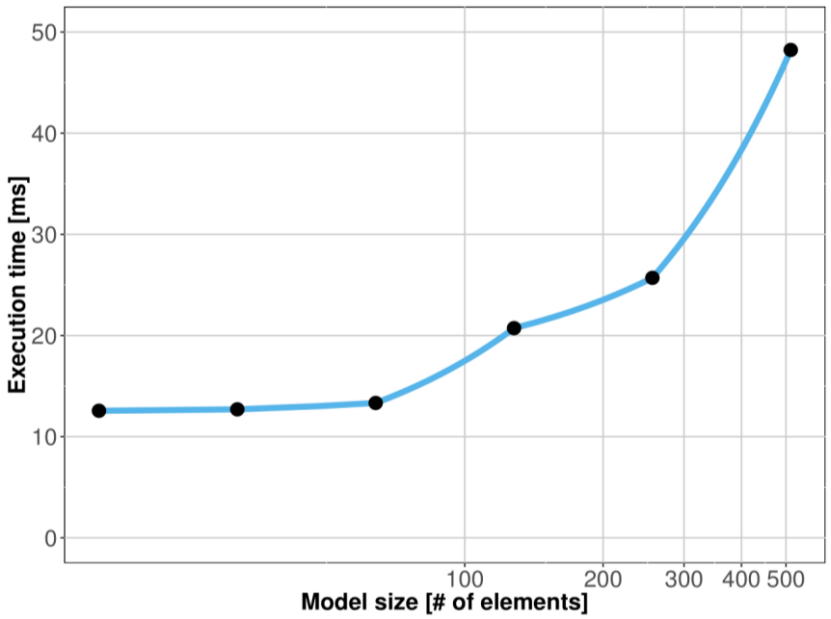

Figure 24. Execution time of change detection with various model sizes

Figure 24. Execution time of change detection with various model sizes 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. The growth seems to be exponential, however as the x-axis is logarithmic while the y-axis is linear, this actually denotes a simple linear correlation between the two variables.

Table 1 summarizes these values including a confidence interval (CI) on 95% confidence level obtained using the one-sample t-test. The confidence intervals 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.

Size [# of elements]	Average 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 time with different model sizes

In terms of the relationship between the size of changes and the execution time of change detection, the results are promising. Figure 25 also uses a logarithmic scale and shows that there is a clear linear correlation between the number of changed elements and the related execution time. This is due to the algorithm used in the background, which is a linear search. Changing this algorithm to a model pattern detection-based technique may improve the performance.

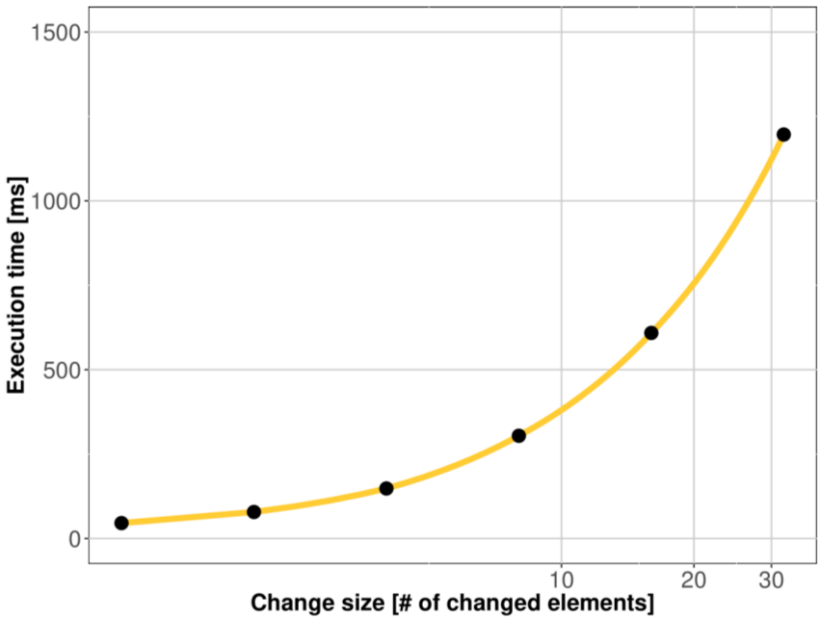

Figure 25. Execution time of change detection with various number of changes

Table 2: Change detection time with different sizes of changes 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.

Size [# of elements]	Average 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 time with different sizes of changes

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 26. Execution time of RTS with various number of changes 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.

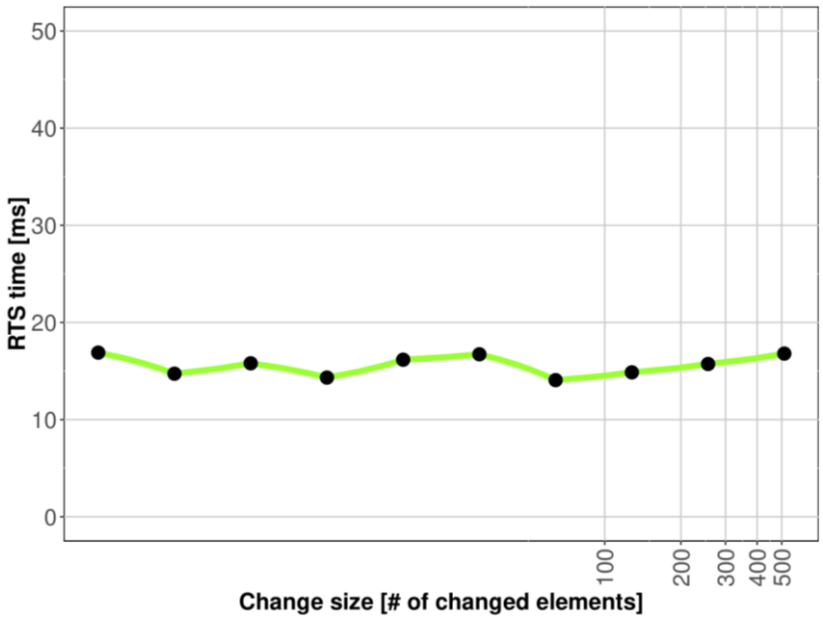

Figure 26. Execution time of RTS with various number of changes

Table 3: RTS execution time with different sizes of changes 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 modelling 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.

Size [# of elements]	Average 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 time with different sizes of changes

As described previously, we also analysed 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 the prototype tool, 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 the prototype tool 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.

7 Conclusions

The deliverable elaborated a method that can be used for the selection, adaptation and extension of test cases. The operation of this method is driven by the analysis of the new requirements (formalized in scenarios), the changes in the context of the system (formalized in context ontologies and metamodels), and the changes in the internal components (formalized in architecture and capability models). The gaps in the coverage of the existing test suites are identified, which drives the adaptation of existing test cases and the generation of new test cases to cover the changes.

The main contributions of the work were the following:

- A general concept of test analysis was introduced and the corresponding languages to capture tests (test cases) and testables (context and configuration elements) and their mapping were defined.
- A tool was designed that can perform the incremental testing analysis. Using model adapters the core incremental analysis component is independent from the actual domain, and only these light-weight adapters had to be created when testing a new system. A prototype implementation was also detailed that is based on the Eclipse framework, the de facto modelling environment used in industry.
- Preliminary evaluation of the applicability and scalability of the method and the tool was presented. The evaluation used context and capability model inspired by the NIST autonomous robot test room standard.

The use of approach will be further evaluated in D34.50 “Assessment of the on-line verification and incremental testing” at the end of the project.

8 References

[1] Systems and software engineering – vocabulary. ISO/IEC/IEEE 24765:2010(E)

[2] 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)

[3] Agrawal, H., Horgan, J.R., Krauser, E.W., London, S.: Incremental regression testing. In: ICSM. vol. 93, pp. 348–357. Citeseer (1993)

[4] Almasri, N., Tahat, L., Korel, B.: Toward automatically quantifying the impact of a change in systems. Software Qual J pp. 1–40 (2016)

[5] Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. Int. J of Web Information Systems 5(3), 271–304 (2009)

[6] ASTM International: Standard test method for evaluating emergency response robot capabilities: Mobility: Confined area terrains: Continuous pitch/roll ramps

[7] Bergmann, G., Dávid, I., Hegedüs, A., Horváth, A., Ráth, I., Ujhelyi, Z., Varró, D.: VIATRA3: a reactive model transformation platform. In: Theory and Practice of Model Transformations, pp. 101–110. Springer (2015)

[8] Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers, 1st edn. (2012)

[9] Briand, L., Labiche, Y., He, S.: Automating regression test selection based on UML designs. Information and Software Technology 51(1), 16 – 30 (2009)

[10] Briand, L., Labiche, Y., Soccar, G.: Automating impact analysis and regression test selection based on uml designs. In: Software Maintenance, 2002. Proceedings. Int. Conf. on. pp. 252–261 (2002)

[11] Chen, Y., Probert, R.L., Sims, D.P.: Specification-based regression test selection with risk analysis. In: Proc. of the 2002 Conf. of the Centre for Advanced Studies on Collaborative Research. pp. 1–. IBM Press

[12] 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)

[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: Engineering of Computer Based Systems, IEEE Int. Conf. and Workshops on. 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 '07, ACM (2007)

[16] 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)

[17] Harman, M.: Making the case for MORTO: Multi objective regression test optimization. In: ICST Workshops. pp. 111–114 (2011)

[18] 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)

[19] 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. In: ACM SIGPLAN Notices. vol. 36, pp. 312–326. ACM (2001)

[20] Korel, B., Tahat, L., Vaysburg, B.: Model based regression test reduction using dependence analysis. In: Software Maintenance, 2002. Proceedings. Int. Conf. on. pp. 214–223 (2002)

[21] Le Traon, Y., Jeron, T., Jezequel, J., Morel, P.: Efficient object-oriented integration and regression testing. Reliability, IEEE Tran. on 49(1), 12–25 (Mar 2000)

[22] Leung, H., White, L.: Insights into regression testing. In: Software Maintenance, 1989., Proceedings., Conf. on. pp. 60–69 (Oct 1989)

[23] Malishevsky, A.G., Ruthruff, J.R., Rothermel, G., Elbaum, S.: Cost-cognizant test case prioritization. Department of Computer Science and Engineering, University of Nebraska-Lincoln, Techical Report (2006)

[24] 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)

[25] NIST: Guide for evaluating, purchasing, and training with response robots using dhs-nist-astm international standard test methods

[26] Pilskalns, O., Uyan, G., Andrews, A.: Regression testing UML designs. In: Software Maintenance, 2006. ICSM '06. 22nd IEEE Int. Conf. on. pp. 254–264 (Sept 2006)

[27] R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing (2013), http://www.R-project.org/, ISBN 3-900051-07-0

[28] Rothermel, G., Harrold, M.J.: Selecting regression tests for object-oriented software. In: Software Maintenance, 1994. Proceedings., Int. Conf. on. pp. 14–25. IEEE

[29] Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. Software Engineering, IEEE Tran. on 22(8), 529–551 (1996)

[30] Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. Software Engineering, IEEE Tran. on 27(10), 929–948 (2001)

[31] Soetens, Q.D., Demeyer, S.: Cheopsj: Change-based test optimization. 15th European Conf. on Software Maintenance and Reengineering 0, 535–538 (2012)

[32] 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)

[33] 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)

[34] Wu, Y., Offutt, J.: Maintaining evolving component-based software with UML. In: Software Maintenance and Reengineering, 2003. Proceedings. 7th European Conf. on. pp. 133–142 (2003)

[35] Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22(2), 67–120 (2012)

[36] Zech, P., Felderer, M., Kalb, P., Breu, R.: A generic platform for model-based regression testing. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, LNCS, vol. 7609, pp. 112–126. Springer Berlin Heidelberg (2012)

[37] Zech, P., Kalb, P., Felderer, M., Atkinson, C., Breu, R.: Model-based regression testing by OCL. Int. J on STTT pp. 1–17 (2015)