Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

A Generic Regression Testing Method for Model-based Development

TDK report

Author:

Gábor Molnár

Advisor:

dr. Zoltán Micskei

2015.

Contents

Kivonat

A regressziós tesztelés célja, hogy meggyőződjünk arról, hogy a korábban működő funkciók az elvégzett módosítások mellett is működőképesek maradnak. Ezt általában a meglévő tesztkészlet ismételt futtatásával érjük el. A rendszer méretétől és a tesztektől függően ez sok időt vehet igénybe, valamint olyan részeit is ellenőrizheti a rendszernek, ami nem is módosult.

Ennek optimalizálására már léteznek módszerek, ilyen például a tesztkészlet minimalizálás, a tesztkiválasztás, illetve a teszt prioririzálás. Továbbá léteznek eszközök is, melyek ezt elvégzik, bár általában ezek egy programnyelvre működnek.

Modellalapú szoftverfejlesztés esetén azonban a rendszerrel kapcsolatos információt modellekben tároljuk, melyekből előállítható forráskód, tesztkód, sőt akár a dokumentáció is. Mivel a modell, mint központi elem jelenik meg a fejlesztés során, így a regressziós tesztelés is elsősorban a modellen végzendő el.

Példaképp tekintsünk egy olyan esetet, amikor modellek alapján generálunk teszteseteket a fejlesztendő rendszerhez. Ilyenkor a rendszerrel kapcsolatos változások a modellekben jelenik meg. Regressziós tesztelés során e változások alapján el kell tudnunk dönteni minden tesztesetről, hogy érdemes-e újrafuttatni. Bár a probléma hasonló a klasszikus, forráskód alapú regressziós teszteléshez, modellalapú rendszerrel kapcsolatban lényegesen kevesebb kutatás található az irodalomban. Ezért a célom egy olyan általános regressziós tesztelő módszer megalkotása, amely többféle bemeneti modellel is működik.

A fenti követelmények és problémák kezelésére:

1. definiáltam egy általános modellt, amely leírja a tesztkiválasztás (RTS) problémáját,

2. bemutattam, hogy a már meglévő optimalizálási algoritmusok módosításával a modellekhez tartozó tesztesetek csoportosíthatóak (újrafuttatandóság szempontjából) egy-egy módosításhoz,

3. kidolgoztam egy módszert, amelyben a rendszer- és a tesztmodellek megfeleltethetőek ennek az általános optimalizációs modellnek.

A módszer kiértékelése érdekében implementáltam azt egy Eclipse alapú prototípusban. Az eszköz képes a változtatások alapján kategorizálni a teszteseteket, illetve kiválasztani közülük néhányat, melyekkel elérhető a maximális tesztfedettség.

Továbbá alkalmaztam a módszert egy esettanulmányban, amelyben a bemeneti modellt a mobil robotok fejlesztése során használt kontextus és konfigurációs modellek jelentik.

A modellbemenetek mellett, a módszer általánosságát hangsúlyozandó, elkészítettem egy kódfedési jelentés (code coverage report) és az optimalizációs modell közötti leképezést is.

Abstract

Regression testing is the activity performed to make sure that previously working features will remain functional after the changes are made. Usually such testing includes executing all the tests available in the test suite. Depending on the size of the software and the testing performed, this can take significant amount of time, and it may also test parts of the system which haven't been changed.

There already are some techniques to optimize regression test runs such as test suite minimization, regression test selection or test prioritization. There also are a few tools which are able to perform this optimization, though their targets are usually a specific programming language and source code.

Model-driven software development is a methodology where all the information of the system are represented in models. These models are then used to generate the source code or other artifacts, such as tests or even documentation. In such development methodology the main artifact for regression testing is the model, not the source code.

Consider a system that uses models to generate test cases. Suppose the system and the models change. Based on the changes and the models, we need to be able to tell whether a test is needed to be rerun or not. So far most research effort has been conducted on optimizing source code based regression testing, therefore my aim is to design and implement a general method that is able to optimize regression test runs for different kinds of input domains.

To address the above requirements and challenges I have:

1. defined a generic model that is able to represent the regression test selection problem,
2. showed how existing optimization algorithms can be adapted to categorize and select tests for a given change,
3. developed a method in which the system and the test models are mapped to this general test optimization model.

To evaluate the method I have implemented an Eclipse-based prototype tool. The tool is capable of identifying the tests which may be used for regression testing and is able to select some of them to provide the maximal test coverage.

I have also conducted a case study in the context of the development of mobile robots.

To show the method is indeed generic and could work on a variety of inputs, I have designed a mapping between test models and a code coverage report.

Introduction

Nowadays the quality of software products are getting more and more important. We are getting used to products or services being always available when needed, they even become part of our everyday life (let's just imagine a day of work without email or Internet access). According to [10], software quality is the "capability of a software product to satisfy stated and implied needs when used under specified conditions". So a software of higher quality satisfies our needs better than another with less quality.

A way of developing quality software comes through verification and validation. One of the most widely applied verification and validation technique is testing. Software testing is defined in [10] as "the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior". So to perform testing one would need to execute a program with given inputs (from a given state), and then needs to be to evaluate whether the behaviour is matching the expected, or not.

Testing is widely applied through the software development lifecycle, for example developers may perform unit testing of their work, or testers can create acceptance tests to verify behavior of the complete software. On the other hand, most software needs to have some changes after the first release, that is when regression testing comes in place.

Regression Testing

In [10], regression testing is defined as "selective retesting 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". In other words regression testing makes sure that no previously working features are broken as a side-effect of the changes. The usual way of performing regression testing is to *re-run all tests* available in the test suite. However, this approach may yield unacceptable execution time (and cost) when a larger software is tested; it may cause hours of test runs for even the smallest change.

The need for optimizing regression test runs appeared quite early in the history of software development, for example [7] was published in 1989. Usually there are two goals for the optimization:

- The total execution time is not significant, though if there is an error in the system, it should be highlighted as soon as possible. This is what *test prioritization* deals with.
- Compared to the size of the change, re-testing the whole system takes significant time. In this case one would want to reduce the number of executed tests; this could be achieved with *test suite minimization* or *regression test selection.*

Categorization of Tests

To define the problems listed above further classification of the tests are needed, as defined by Leung and White [7]:

re-usable: This category has all the test cases which exercises unmodified part of the system. It is important to identify them, because their execution may be omitted as they would produce the same result as before.

re-testable: This category contains tests which are either changed or they verify a changed part of the system. These tests are first class candidates for re-run, though a reduction may be conducted if there is an overlap in coverage.

obsolete: These tests can no longer be re-used due to a change in specification or program structure, so that these tests no longer verify what they were created for. In some cases they may no longer have the correct input-output relation.

new structure: These tests contribute to the overall structural coverage of the system in the current revision. It can be useful for the test engineers to point out if there are uncovered elements in the system, so that these tests can be created.

new specification: This category corresponds to new test cases which were created to verify the new elements of the specification. In this thesis this category will have less emphasis, as the solution is not based on the system specification but the code and the models of the system.

With the above definitions, the usual regression testing problems can be defined in the following way:

Test Suite Minimization aims for identifying obsolete test cases and deleting them from the test suite.

Regression Test Selection aims to identify a minimal covering set of re-testable and new tests while maintaining the maximal coverage.

Test Prioritization disregards the categorization, as it executes all tests available. The usual approach is to execute the test next that has the maximal additional coverage, or fault revealing capability (based on previous runs).

Test Suite Minimization and Regression Test Selection have quite similar goals; they both aim to reduce the execution time by not running tests that does not provide valuable coverage. The main difference is that Test Suite Minimization permanently removes all the unnecessary tests (for every possible change), whereas Regression Test Selection uses the change as an input and identifies tests that are related to it.

In this report I mostly work on RTS, as:

1. Usually we do not want to remove tests we already created; they may prove useful for a certain change in the future, and storing them usually doesn't have much associated cost.

2. While Test Prioritization helps increasing confidence of a build after the first few tests, it does not reduce the total execution time if all the tests are passing. RTS would only run the necessary tests, so compared to Test Prioritization execution time can be saved.

Model-driven Development

According to [2], "Model-Driven Development (MDD) is a development paradigm that uses models as the primary artifact of the development process." These models then can be validated for certain properties or requirements, and even source or test code can be generated from them. This development methodology is widely used for safety critical systems, as it can drive formal verification and the generated code is supposed to contain fewer bugs than the manually written (provided the code generator is correct). Apart from the safety critical domains it is also gaining popularity, mostly because of the higher level of abstraction and automation it provides (e.g. repetitive code can be generated).

Although existing RTS techniques could be applied on the generated code, their feedback may be hard to apply on the models. For example if a test revealed an error in the generated code the developers needed to find which elements of the model are producing that code, and then they need to change the model accordingly. After correcting the error, they need to re-generate the source codes, and they also need to verify that the error has indeed been corrected.

This method may be hard to apply for non-trivial errors and long-running tests, thus the models themselves seem to be the primary artifact for regression testing.

Available Tools

Although regression testing has quite some history as a field of research, there are only a few tools published, even fewer which are open-source.

To enlist some tools mentioned in [4, 12]:

ATAC is a tool for C programs initially developed between 1992 and 1994.

TestTube is a research tool for programs in C, developed in 1994. It is said to be inefficient in [12].

DejaVu is a research tool from Rothermel and Harrold [9]. It is not publicly accessible.

SoDA is a tool capable of performing regression test selection of huge C/C++ repositories [11]. Apparently this is the only maintained open-source tool of its kind.

It is apparent that, most research and development effort has been conducted for code-based techniques and tools [4, 12], especially for C/C++ applications, there only are a few papers with respect to the model-based application [13, 5].

Goals

In this report my primary goal is to design a method of regression testing applicable in a model-driven development methodology. As we have seen, this method should primarily rely on the models themselves.

Since there can be several models created even in a single project, therefore this method should also be easy to adapt for various input models.

Results

To achieve these goals, I have designed a model-based method for the RTS problem. This way the optimization becomes independent of the input model, and various input models can be adapted for the same optimization algorithm also.

To verify that the theory works in practice, I have also implemented an Eclipse-based prototype tool using on the model designed. The tool is able to select a reduced set of test cases based on the model representation. It can also create models from OpenCover code coverage reports.

Chapter 1 introduces the typical use-case of regression testing in a model-based environment. Then Chapter 2 describes the data structure created to separate the optimization algorithm from the input. Chapter 3 provides insights on the architecture, design and some implementation details of the prototype tool. Chapter 4 evaluates the method in two different domains:

1. a model-based representation of a test specification for autonomous robots,
2. and optimizing regression testing of source code through an XML code coverage report.

Finally, Chapter 5 concludes the report, and provides opportunities for future work.

Chapter 1

Motivation

As a motivating example for the report, this chapter introduces a scenario in which test cases are generated from various models. The problem domain comes from the R3-COP[1] and R5-COP[2] EU research projects, in which the Fault Tolerant Systems Research Group (FTSRG) conducts research about the verification of autonomous systems.

An autonomous system (AS) is a system that is able to make and execute decisions to achieve its goal without full, direct human control or intervention [3]. The usual example of an autonomous system is an autonomous robot, or an autonomous vehicle, though automatically scaling cloud applications requires some autonomous feature too. Autonomous robots operate in various environments, usually even in presence of human beings. Therefore their safety and robustness properties are an important aspect of development.

Testing autonomous systems, including the robots, are not easy, due to the autonomous decision-making capability the results are not necessarily predictable or deterministic. For example a robot may take different paths to accomplish its mission. Though there may be events which are either required or forbidden to happen, e.g. a moving robot must notify the human approaching, or the robot must not hit it if the human is in the way.

In [8] a model-based system-level black-box testing approach is presented (shown in Fig. 1.1). This approach aims at verifying safety and robustness properties. Test execution can also happen in different environments; to verify safety and robustness of the software by itself a simulation environment can be used. As the simulation may not be a perfect representation of the real behavior, the tests can also be conducted in a prearranged test room physically.

To drive the testing approach, firstly the requirements should be formalized.

1.1 Modelling the Requirements in the R3-COP Project

We have already seen that autonomous systems behave somewhat differently than the average, deterministic software. Therefore designing tests for them is different than simply providing the initial state, input and the expected output.

The initial state needs to be part of the environment the system operates in, the input is defined as a mission to complete. In this case the expectation is to maintain safety

[1]Resilient Reasoning Robotic Co-operating Systems – `http://www.r3-cop.eu/`

[2]Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems – `http://www.r5-cop.eu/en/`

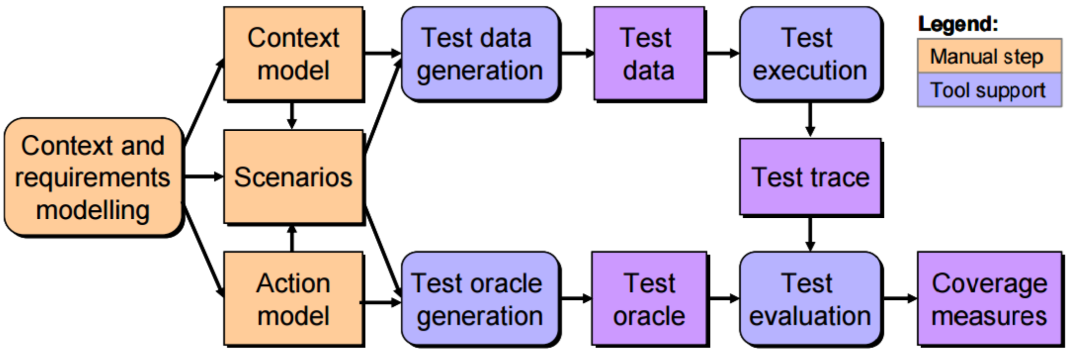


Figure 1.1. Overview of the Testing Approach (from [8])

and robustness constraints even in stressful conditions. This manifests itself as activities the robot is expected to perform, and other events that are forbidden to happen (without violating the requirements). Also due to the non-deterministic nature, the result of these tests need to be analysed statistically too.

To describe these kind of behavioural requirements a domain model had been designed [8]. This model is based on the UML 2 Sequence Diagrams, with certain extensions made.

Fig. 1.2 shows an example of this scenario model.

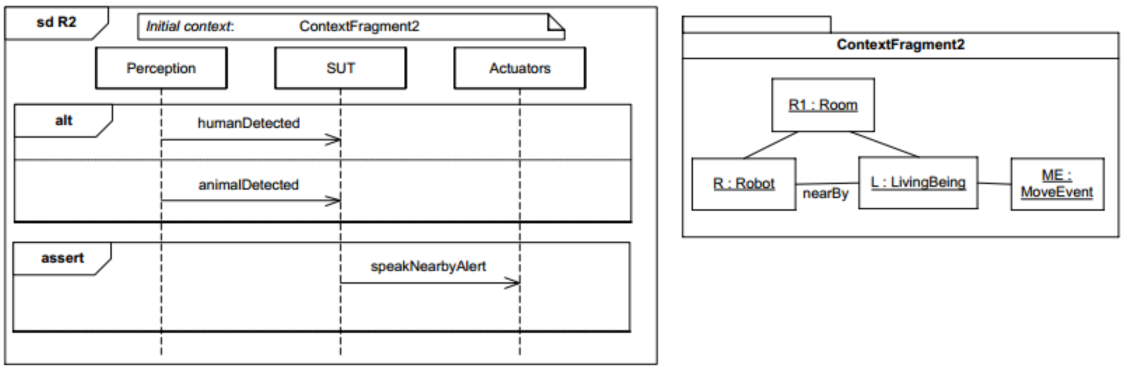


Figure 1.2. A Sample Scenario Model (from [8])

A significant extension is the precondition and the assertion part of the model, ie. given the pattern described in the precondition part, the pattern in the assert part must or must not appear.

Another extension is the concept of *initial context* which describes the environment the robot operates in. The model of this environment is called the *context model*. In terms of testing the initial context is the starting position of the test, which the testers create and then start the robot in.

A sample context model can be seen in Fig. 1.3. The metamodel describes the environment the robot operates in, and the objects and event it needs to recognise. In this example the context model was created for a robot vacuum cleaner and represents rooms in a flat.

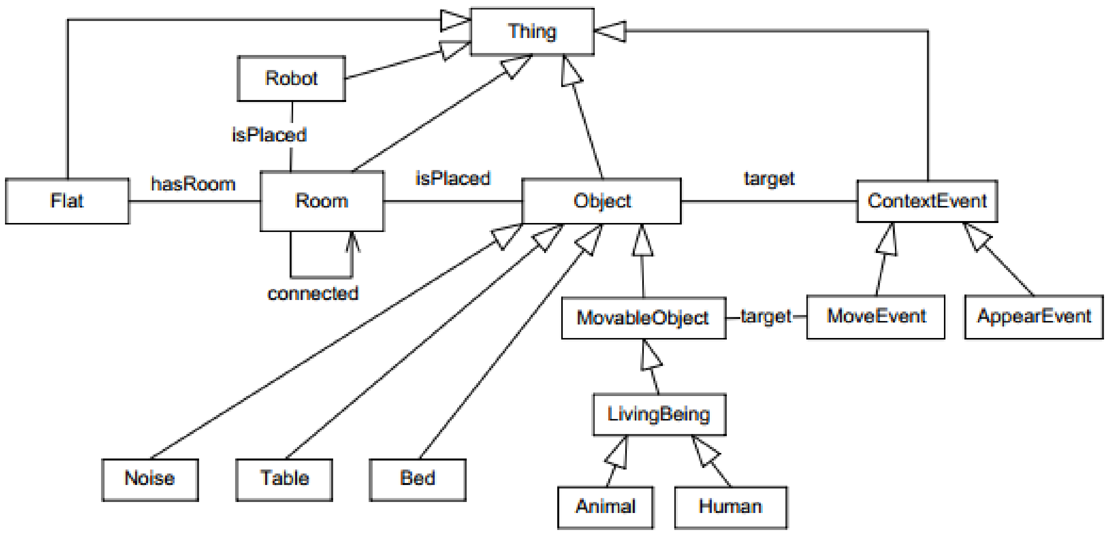
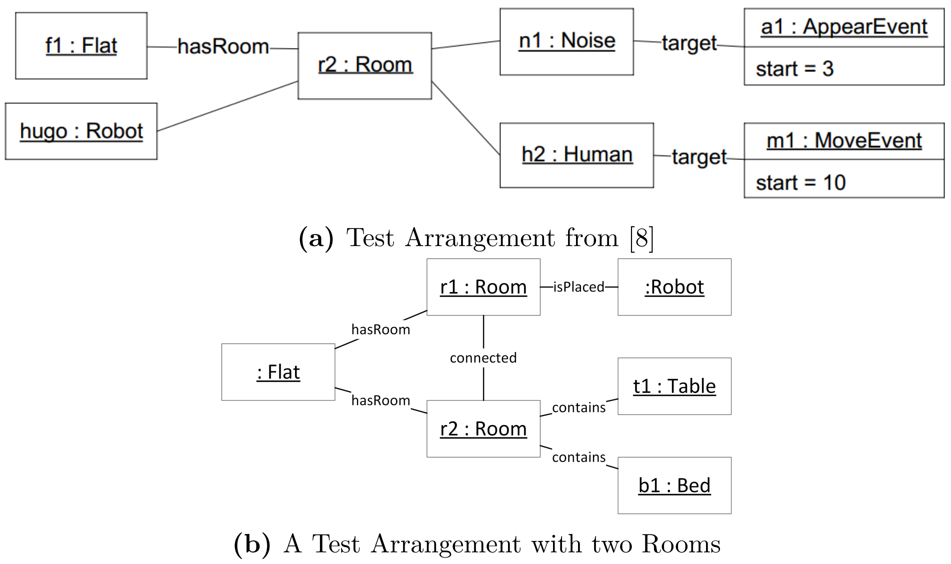


Figure 1.3. A Context Metamodel Describing a Room (from [8])

1.1.1 Test Generation from Context Models

The key idea of context modelling is to be able to define a test context in an abstract way, e.g. there is furniture in the room, or other kind of obstacles with abstract distance (ie. near or far). From this abstract model then concrete and different test arrangements can be generated for given input criteria, such as the one being described in [1].



(a) Test Arrangement from [8]

(b) A Test Arrangement with two Rooms

Figure 1.4. Sample Test Arrangements for the Robot Vacuum Cleaner

1.2 Regression Testing using Generated Test Cases

Let's consider that these context models need to have a change (e.g. the robots now need to recognise a sofa). In that case some of the existing test cases may become obsolete, and some new test cases (containing a sofa) may need to be generated.

Also re-running all the tests may be unnecessary, as the ones which do not test the components related to the change would have the same outcome.

In this model-based testing domain, we could apply the same regression testing techniques that are already developed for source code and other kind of software, the difference is just instead of source code, now we have model elements.

1.2.1 Significant Model elements for Regression Testing

To drive regression testing, one must identify which elements of the models are items to cover, and which elements act as the test to cover them. In this case it is obvious; the generated test arrangements would be used as tests, and all other elements, including objects of the context model, scenarios and constraints from sequence diagrams are items to be tested.

Though this mapping is not yet complete; some test arrangements may contain the same objects, though they may be significantly different. For example having or not having an edge (the *isNear* relation in this case) in the model may test two different edge-cases of the behavior. The number of instances a class has may also be significant, e.g. the behavior is different if the robot notices a few or a lot of people nearby, or entirely different if there are no one nearby.

The latter case can be modelled by assigning a special constraint to the test layout, ie. the layout has an implicit constraint of having a certain amount of objects, for example the three test cases would have their implicit constraint as having two, five, or zero people accordingly.

1.3 Summary

In this chapter we have seen a model-based approach for testing autonomous systems. There are various modelling activities involved in order to represent the requirements formally:

- the behavioral requirements are captured in a scenario model, which is based on the UML 2 Sequence Diagrams,
- the environment of these requirements are captured in context models.

The method to be presented in Chapter 2 is not specific to these models, though this use-case can provide a good motivational example.

Chapter 2

Modelling Regression Testing

As we have already seen reducing re-run time of regression test runs have two main ways, Regression Test Selection and Test Suite Minimization. We have also seen that they are quite analogous to each other: RTS on the scope of the whole system can solve the Minimization problem as well.

Firstly, I am going to present the existing techniques for regression testing source code. As we will see there are various ways of creating a representation of the source code and its test, so I have created an adaptable data model that separates the nature of the input and the optimization algorithm. Finally, I am going to present how this data model can be used for model-based inputs.

2.1 Existing Algorithms for Source Code

RTS needs to maintain the same fault detection effectiveness that the whole test suite had. This can be achieved by checking which tests verify the same code element, and eliminating this redundancy. This is also called code coverage of the test. So in other words RTS aims for maintaining the maximal coverage with the least associated cost of execution.

This coverage can be built and represented in several different ways [12]. The simplest representation is a binary matrix whose rows correspond to the program elements and columns for tests, and have 1 in a cell (i, j), if the ith program element is covered by the jth test.

Another way of building coverage is from control or data-flow graphs, which tend to yield better results according to [4].

All of these approaches operate on the relation between the test and the program element it verifies (let's call this coverage from now on). Over this generalized coverage the Set Cover Algorithm needs to be applied to select the reduced set of tests.

In the following sections I am going to present how the data structure and the algorithm can be separated to make the optimization problem independent of the approach the coverage was built from.

2.2 Requirements for an Adaptable Data Structure

Since the focus of this work is on application of regression testing on model-based inputs, the internal representation for RTS should be easily convertible from input models. There also are various model transformation tools which are able to convert between different meta-models, so it seem to be the easiest to create a model to represent the RTS problem domain.

The requirements for such model are as follows:

- The model should separate the RTS algorithm independent of the actual form of coverage (e.g. a matrix or a graph). It should achieve this through the abstract notion of coverage.
- The model should easily be extensible for future use-cases for regression testing or applications which require special terminology or elements. For example multiple kinds of coverage (e.g. statement or conditional) may need to be represented in a single model.
- While the model is general for RTS and easily extensible, it must not require an extension to operate for use-cases anticipated, such as the one in Chapter 1.

2.2.1 Support for Different Coverage

The extensibility of coverage is important, as it may need to represent multiple concepts. Consider a program (code), its tests and the code coverage measured. In this case one can not entirely rely on either statement or branch coverage, each has its own meaning: a test suite with full statement coverage may not highlight issues with conditions, while 100% branch coverage as a metric may ignore statements that were not executed because of an exception thrown (despite the branches are covered for both cases).

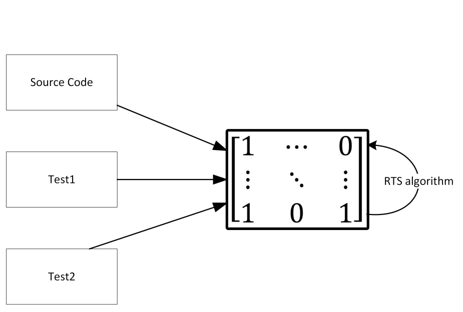
Thus raises the need to have different kind of coverage in the model. Other future use may also need further extensions in terms of coverage, e.g. in this example, one may want to introduce more advanced coverage, such as MC/DC.

2.2.2 Support for Boundary Value Analysis

Boundary Value Analysis (BVA) is a method of creating test cases. It requires partitions (inputs that produce the same behavior) in the input domain, and it verifies behaviour on the edge-cases of those partitions. For this BVA takes three values: each one from the first partition, at the limit and the other partition.

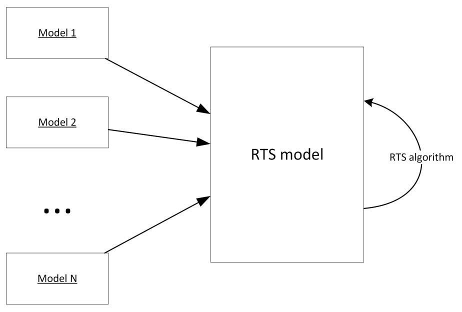
Let's consider the *sign function* as an example, it returns -1 for all negative numbers, 0 for 0, and 1 for positive numbers. Therefore it's partitions are $(-\infty, 0)$, 0, $(0, \infty)$. BVA then may produce the following test cases:

- An element of the first partition, such as -0.01, and the expected output is -1.
- The function for 0 returns 0 indeed.
- Analogous to the first test, for input 0.01 (and every other member of the third partition) the return value is 1.



Source Code
Test1
Test2
$\begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 1 & 0 & 1 \end{bmatrix}$
RTS algorithm

(a) An Algorithm Using Coverage Matrix



(b) A Model-based Input and the Adapted Data Model

Figure 2.1. Visualizing RTS Algorithms

If one would only check coverage for these, the partitions may overlap. Though BVA highlights separate kind of errors, for example a mistaken comparison of `input <= 0` instead of the correct `input < 0` would be highlighted, whereas branch coverage could still be 100%.

The concept of BVA is similar for model inputs, an element or an edge may or may not be present, or it may have difference whether it appears once or multiple times. Unless the test model have some construct for this behavior, test cases for the edge cases may be omitted from the re-run, potentially ignoring a bug which the original test suit would have highlighted.

2.3 A Domain Model for the RTS problem

As we have seen in 2.1, there are various algorithms and data structures for the RTS problem. The schematics of such algorithm can be seen on Fig. 2.1a. In this case the program has some test cases, the coverage algorithm uses a matrix to represent the coverage relationship.

Using an abstract data model for the RTS problem enables us to separate the optimization (basically the Set Cover Algorithm) and the representation of the input (be it a coverage matrix or a data-flow graph for example). Fig. 2.1b shows the schema for the adapted data structure for model inputs. The abstract data model also naturally provides a representation if the inputs are model elements, ie. only a transformation is needed between the domain models.

Fig. 2.2 shows the model I have designed to address the requirements shown in 2.2. The model contains the following main concepts:

Test: is the class to represent an executable test case. It stores the duration of the test run (from which execution cost is derived), and the last known outcome as well.

Testable: is an object that is being verified by tests. To support further extensions this is modelled as an interface.

Component: is a class that implements *Testable*. It also supports dependencies: if a component is changed, all dependent components must be tested again.

Condition: is a *Testable* that represents a branch in the execution path. It is associated with *Conditional Coverage* that covers a single execution path (e.g. `true` outcome of an *if* statement, or the case when a certain model element is present).

2.4 Evaluation of Regression Test Selection over the Model

Let us revisit the motivational example to show how this proposed model can represent the regression testing problem. In 1.2.1 we have already seen which elements are significant for regression testing. Namely test arrangements would become *tests*, while elements of the context model and the scenarios would become *testables*.

As an example the test arrangements in Fig. 1.4 would produce the test model seen in Fig. 2.3. *testData1* and *testData2* (the bigger boxes) represents the test arrangements, while the others are created either for the elements of the context model, or for requirements of the scenarios to satisfy.

As there are some common elements in both arrangements (e.g. robot, flat or room) they are connected to both test data nodes. Similarly the distinct elements (bed, table, human or the events) are connected to only one of them.

Now let's consider a change in the concept of a room (e.g. rooms must have a roof from now on). Since both tests cover the `Room` *Testable* it is enough to run one of them to verify the changes, suppose `testData1`. `testData2` can be omitted, because the change does not affect functionality related e.g. `Bed`.

If a new context element would be added (e.g. `Sofa`), then it would appear as an isolated *Testable*, meaning further tests would need to be created to verify the functionality.

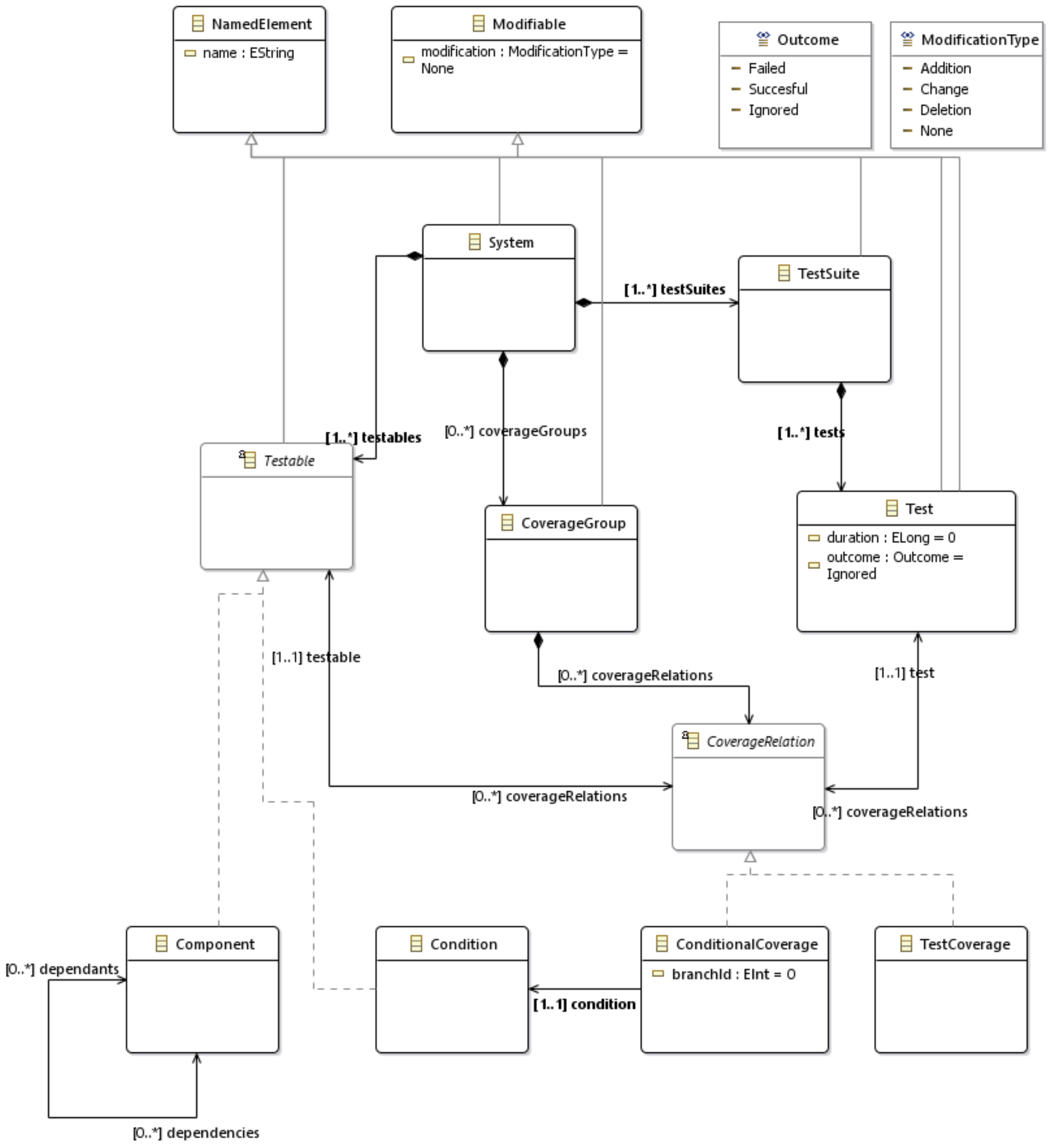


Figure 2.2. The Model Designed for RTS

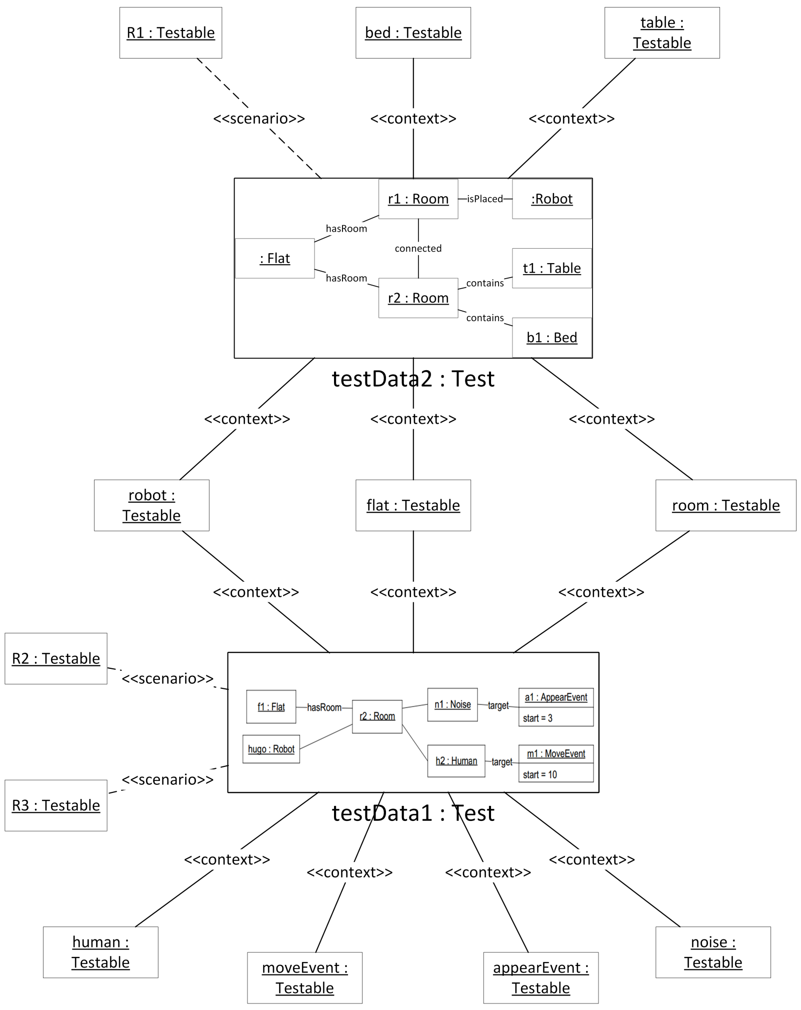


Figure 2.3. Test Model Resulting from the Example Arrangements in Chapter 1

Chapter 3

A Prototype Framework

This chapter depicts the main design principles behind the implementation of the tool. These principles provides maintainability and extensibility.

3.1 Requirements for the Architechture

Most existing tools are designed for a single specific platform or programming language, which limit their applicability. To address this limitation, we need to separate the regression testing algorithm from the handling of the input. As we have seen in Chapter 2 this separation can be a model.

Since I have implemented one of only the simplest algorithm for the Set Cover Problem, the algorithm is designed to be easily replaceable. This enables experiments with different, more complex set cover algorithms.

Analogous to the different input platforms, the output of the test generation needs to be easily extensible and replaceable.

3.2 Architecture

These requirements suggest a modular, plugin-based approach. Fig. 3.1 depicts the architecture:

Model Builder: is a component responsible for converting an input (e.g. an EMF model) into the test optimization model. This also means each different input domain needs to have its own Model Builder.

Optimizer: performs the optimization on the test model. Though Model Builder is unique for each input domain, the Optimizer is general for all input domains and algorithms.

Set Cover Algorithm: is responsible of solving the Set Cover Problem. It is extracted from the Optimizer to make it easily replaceable.

Executor: is an adapter for test execution frameworks (e.g. JUnit). This also needs to be implemented for each framework.

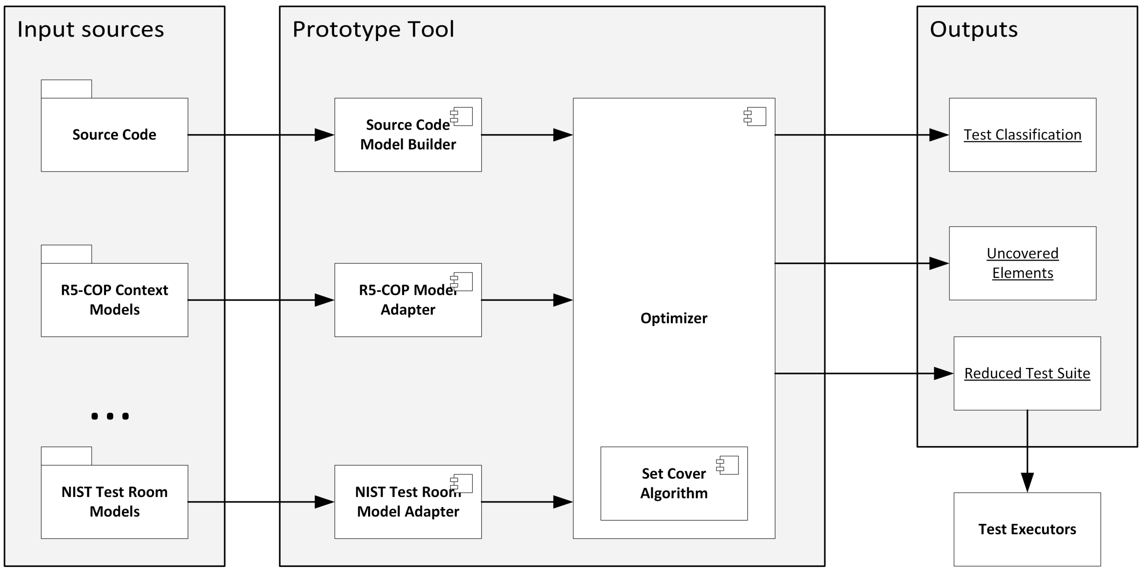


Figure 3.1. The Architecture of the Tool

3.3 Examples of Reusability

Consider our previous example of the R5-COP test model from Chapter 1; to integrate that into the framework one would need to implement a new Model Builder component. Since the input is also a model, one would need to provide mappings between the model elements, ie. what type of elements will be represented as tests or components.

If we had another input domain, only the Model Builder component would need to be implemented, the Optimizer, the Set Cover Algorithm and the Executor could remain exactly the same.

On the other hand, if we had a more effective algorithm, we would only need to replace that part of the tool.

3.4 Tool Functionality

Currently the prototype version of the tool has a single model builder component that is capable of building a test model from the coverage report of OpenCover (4.2). For model inputs model transformation would be used.

Then based on the test model (and the changes set), the optimizer component can identify which elements in the test model have been affected by the changes, and then it can provide a set of test cases that covers them. In case there were uncoverable elements, the tool can report them as well.

So far evaluation of the functionality was conducted on simple, hand-built models and generated models (of a few hundred elements) as well.

3.5 Implementation Details

The tool has been implemented in Java, as an extension of the Eclipse development environment as a set of plugins. This provides future extensibility between the components, e.g. set cover algorithms can be interchanged. The Eclipse platform also provides the integration with source control systems (ie. the Platform API is being used for querying commits).

On top of Eclipse, Eclipse Modelling Framework (EMF) is the de-facto modelling environment. It also provides various features such as persistence, basic code generation and a visual editor.

Usually in model-based applications appear the need to find model elements of a certain property. For example in this tool, one would need to find `Testables` that are changed, or highlight others dependent on them. There are various tools capable of evaluating model queries, I have used Eclipse IncQuery[1], which is mainly developed in FTSRG as well.

Analogous to model queries, there usually are use-cases that require conversion between different domain models. Model transformation deals with this matter: one can define patterns in the input model that are translated into another in the output model. In case of this tool, model builders that use an EMF model as their inputs can be implemented with a model transformation only. Similarly, I chose VIATRA[2], a transformation framework that is developed in FTSRG and integrates with IncQuery.

So far the prototype of the tool with all of its plugins (including the model builders and transformers) is about 3000–3500 LoC in size.

[1]`https://www.eclipse.org/incquery/`
[2]`https://eclipse.org/viatra/`

Chapter 4

Case Studies

For the purpose of evaluating the tools in various input domains, I have conducted two case studies. 4.1 uses a model-based approach like Chapter 1. The aim of using another model is to demonstrate that only the model builder part of the architecture (3.2) needs to be designed and developed, the optimization part is unchanged.

On the other hand, 4.2 uses a code coverage report as an input to the optimization problem. This use-case is intended to present that the method is applicable even if the input is not model-based.

As this report does not deal with the efficiency of the algorithms, the case studies are only intended to demonstrate the applicability of this model-based regression testing method.

4.1 Optimizing NIST Test Rooms

Final system verification of mobile robots are usually conducted in a special test room that provides certain challenges for the robot. Emergency response robots are a special kind of mobile robots which are capable of performing a certain activity in a hostile environment which may pose harm for humans (e.g. recovering survivors in an earthquake, perform tasks in nuclear plants, identify and disarm explosives).

Amongst certain guidelines for evaluating these robots, National Institute of Standards and Technology (NIST) created a guideline [6] in order to make these rooms comparable. These guidelines are based on ASTM standard objects, such as the terrain or the obstacles in the rooms are predefined.

For example mobility exercises are performed over various terrain, such as continuous ramps[1], crossing ramps[2], stepfields[3], or over sand, gravel and mud.

There are various other tests, such as exercising the manipulator (door opening), obstacles: gaps in the floor[4] or stairs[5]. To test the image recognition there are certain visual targets or signs the robot may need to recognise (e.g. sign for flammable or radioactive objects).

If there is a change in the requirements of the robots, some modification to these rooms may be necessary as well. This is analogous to the need of maintaining the test suite of

[1]ASTM E2826-11 – `http://www.astm.org/Standards/E2826.htm`
[2]ASTM E2827-11 – `http://www.astm.org/Standards/E2827.htm`
[3]ASTM E2828-11 – `http://www.astm.org/Standards/E2828.htm`
[4]ASTM E2801-11 – `http://www.astm.org/Standards/E2801.htm`
[5]ASTM E2804-11 – `http://www.astm.org/Standards/E2804.htm`

a software, though changing the room definitely involves more manpower and time. The aim of this case study is to apply regression test optimization methods for the changes of the test rooms.

Regression testing can also identify which parts of the robot (and its software) is not verified in a certain room. This can provide hints for the developers/testers of the robot for future test rooms. This is similar to the one described in Chapter 1, though in this case we are not interested in which tests can be eliminated while testing the robot thoroughly, the focus is rather on what needs to be changed in the test room, to accommodate the changes in the requirements.

4.1.1 Creating Domain Models for the Case Study

To be able to represent the problem domain formally, I have designed domain-specific models for both the test rooms and the robot capabilities as well.

Domain Model for Test Rooms

This domain model has to describe test rooms built, e.g. what type of terrain and obstacles are used, how they are laid out.

Based on the guideline the following categories present themselves:

- Mobility Terrain: any type of terrain a robot may operate in,
- Mobility Obstacle: gap or wall,
- Visual Target: an object for the image recognition.

Fig. 4.2b shows a sample room with walls, obstacles and a sign.

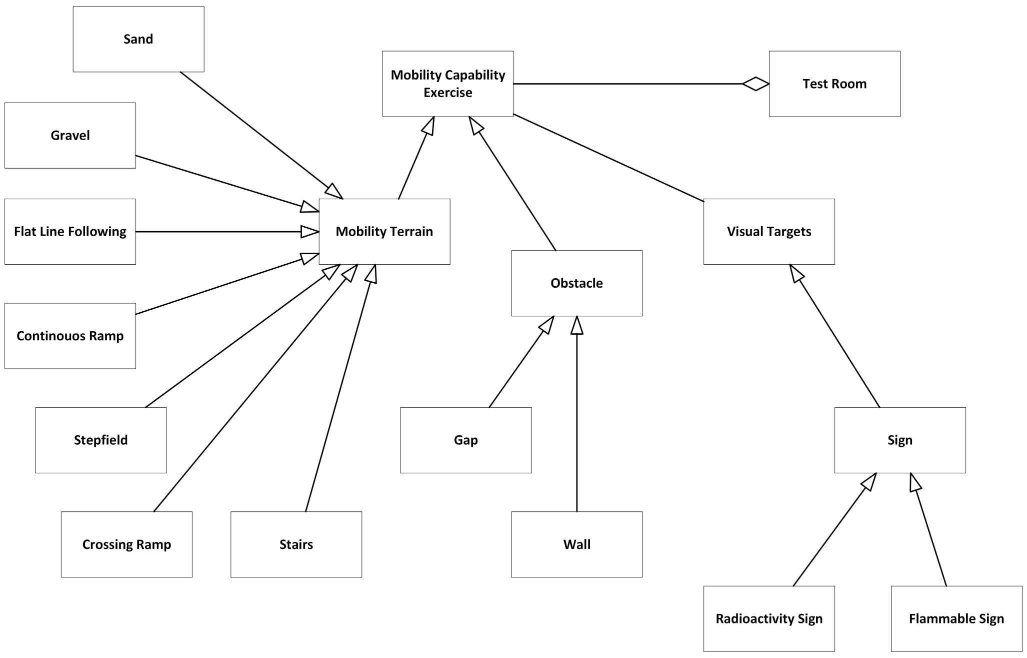
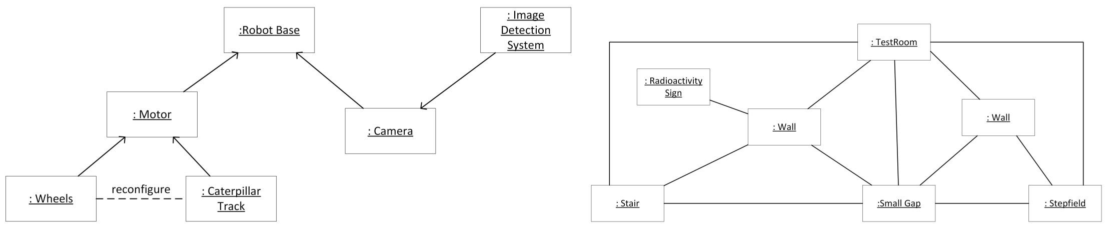


Figure 4.1. Model for Test Rooms

A Model to Represent Robot Configuration

The purpose of this model is to describe what components a robot has (including both hardware and software), what are the dependency between them. These components are exercised by the obstacles in the test room.

In this capability model, a robot may have *slots*, which represent an option to have a *plug* installed. Plugs on the other hand are any object that one can mount on the robot, such as sensors, actuators or motors. Robots also have software installed. All plugs and software can have dependencies, for example a motor may require power supply, or a sofware library can have dependency in another one. Fig. 4.2a shows a model of a simple robot.



(a) Capabilities of a Sample Robot **(b)** Model of a Sample Test Room

Figure 4.2. Example Models of a Robot and a Test Room

4.1.2 Mapping between Domain and Testing Elements

Although the input models are similar to the ones in Chapter 1, the mapping is somewhat different. In this case elements of the context model are considered tests, and elements in the capability model(s) are considered testables. Coverage is derived based on which item in the room exercises which capabilities (e.g. the terrain verifies parts associated with the mobility). This knowledge can either be implemented in the model builder (provided there are fixed amount of context elements in the guideline), or as a more elegant solution, a separate mapping model could be created.

Another difference is that a test room can exercise multiple kinds of robots, so the testables should be the union of elements in all capability models.

With this mapping the problem of identifying which parts of which robots are not tested are equivalent of finding uncoverable testables.

4.1.3 Evaluating Optimization

Consider the robot in Fig. 4.2a. It can operate either with wheels or caterpillar tracks. Currently the test room contains stairs, though (suppose) there is no element that requires using the caterpillar tracks (assuming stairs verify wheels, but not caterpillar tracks, and rough terrain verifies caterpillar tracks, but not wheels).

In this case the tool could highlight that caterpillar tracks are uncovered, and analysing this fact may provide a hint that some rough terrain (e.g. sand or gravel) needs to be added to the room.

4.2 Using a Code Coverage Report as Input

In this section I am going to demonstrate how regression testing can be performed based on a properly detailed code coverage report of the source code.

Code coverage is the ratio of the amount of source code executed by the test suite and the total amount of source code. It can be represented on various levels, such as method, class, line or even instruction, ie. it provides the percentage of invoked methods, referenced classes, executed source lines or instructions.

It should be noted though this use-case is only a proof of the applicability; source code-based tools provide a more effective approach, as the overhead of using the model is fairly significant for huge codebases (i.e. several MLoC).

4.2.1 Requirements for the Code Coverage Report

The method has certain dependencies for the underlying code coverage tool and the report it produces. One should be able to:

1. identify classes, methods and optionally source code instructions,
2. distinguish between source and test code,
3. derive the coverage relation between source code and tests.

Item 1 defines the necessity to have elements the testing model, Item 2 distinguishes *testables* and *tests*.

Item 3 is almost the same as code coverage, though some tools only maintain the metric, and do not have information about coverage distribution (i.e. which tests cover a single instruction).

4.2.2 Tools Evaluated

For the purpose of this case study firstly I have experimented with open-source Java code coverage tools.

Cobertura[6] is a popular code coverage tool that integrates well with the Maven build environment. EclEmma[7] on the other hand integrates with the Eclipse IDE, though it's harder to invoke during builds.

Unfortunately both tools produced a report that was missing the mapping for contributed coverage, therefore they proved to be unusable for regression testing.

OpenCover

OpenCover is an open source code coverage tool for the .NET platform, unlike the Java tools tested, OpenCover has a contributed coverage feature. Though it is not native to the platform of the optimizer tool, the XML report it produces can be parsed in Java

[6]`http://cobertura.sourceforge.net/`
[7]`http://eclemma.org/`

as well. OpenCover operates on the IL[8] instructions, which is the processor-independent code compiled from .NET languages, such as C#.

OpenCover uses the following points for collecting coverage:

Sequence Point: an instrumented IL instruction. It is the smallest possible element of measuring code coverage.

Branch Point: an instrumented conditional instruction (e.g. `ifs` or `switch-case` instructions), it can have several outcomes.

Method Point: indicates whether a method had been called or not.

Consider the method in List 4.1 and a test suite with inputs 3 and 1. Running this test suite produces full statement and partial branch coverage; the implicit `else` of the `else if` statement is covered (case when `mod == 2`).

Listing 4.1. A Method with Three Branches

```
 1  public static void LogIfDividable(int i)
 2  {
 3      var mod = i % 3;
 4      if (mod == 0)
 5      {
 6          Console.WriteLine("Number is dividable by 3");
 7      }
 8      else if (mod == 1)
 9      {
10          Console.WriteLine("1 is the remainder");
11      }
12  }
```

4.2.3 Mapping to Model Elements

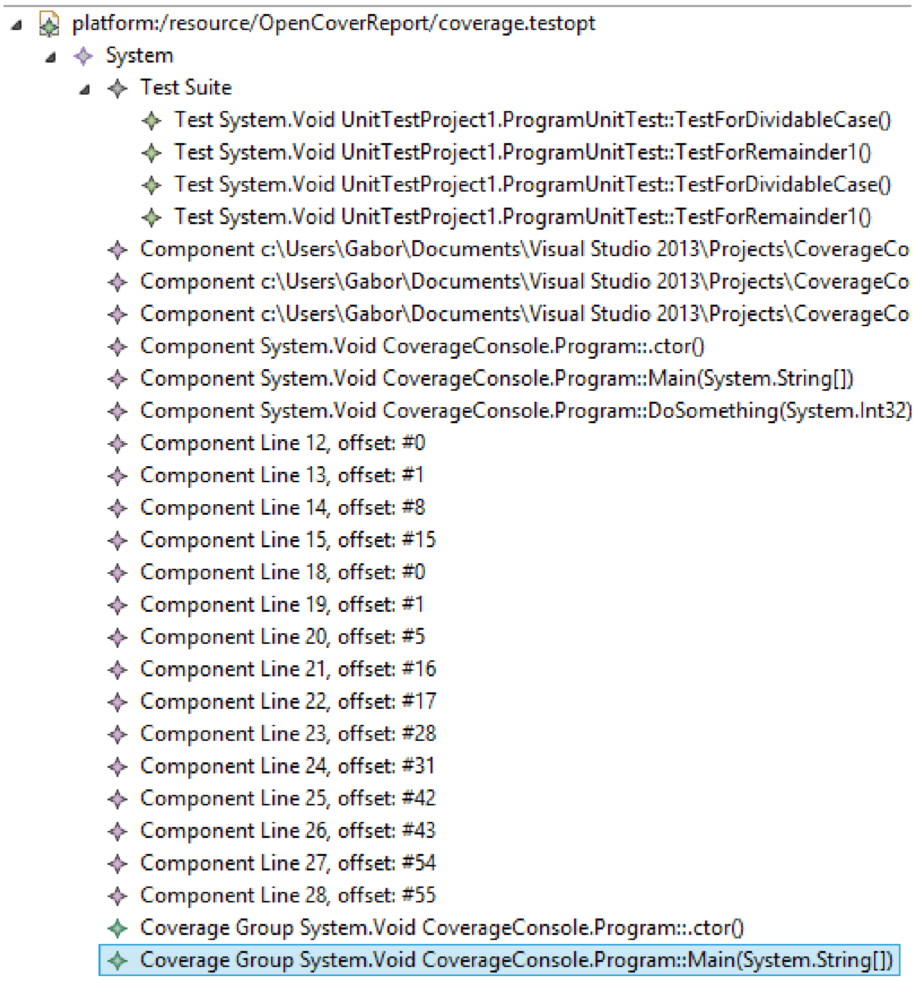
Similarly to 4.1.2, one would need to provide a mapping of the input and their corresponding element in the optimization model. In case of code coverage, the coverage and the tests are mapped obviously (to `Coverage` and `Test`). All the program elements (classes, methods, lines, sequence and branch points) are `Testables`. Fig. 4.3 shows a screenshot from the tool for the code in List 4.1.

4.2.4 Examples

Suppose our tests verify the standard output of this method. If we changed the string literal for Line 6, only the test associated with that line would need to be executed (test with input 3). Test input 1 could be entirely omitted as its associated program elements are unchanged.

This raises a problem though: tests are associated with the IL instructions, but change is tracked with the lines. Fortunately `Component` and its association, *dependency* may help us; if the components representing the IL instructions are a dependency of the component of the source line, then the change of the source line would automatically imply the change of the IL components, which would select their corresponding tests.

[8](Common) Intermediate Language – `http://www.ecma-international.org/publications/standards/Ecma-335.htm`



```
platform:/resource/OpenCoverReport/coverage.testopt
  System
    Test Suite
      Test System.Void UnitTestProject1.ProgramUnitTest::TestForDividableCase()
      Test System.Void UnitTestProject1.ProgramUnitTest::TestForRemainder1()
      Test System.Void UnitTestProject1.ProgramUnitTest::TestForDividableCase()
      Test System.Void UnitTestProject1.ProgramUnitTest::TestForRemainder1()
    Component c:\Users\Gabor\Documents\Visual Studio 2013\Projects\CoverageCo
    Component c:\Users\Gabor\Documents\Visual Studio 2013\Projects\CoverageCo
    Component c:\Users\Gabor\Documents\Visual Studio 2013\Projects\CoverageCo
    Component System.Void CoverageConsole.Program::.ctor()
    Component System.Void CoverageConsole.Program::Main(System.String[])
    Component System.Void CoverageConsole.Program::DoSomething(System.Int32)
    Component Line 12, offset: #0
    Component Line 13, offset: #1
    Component Line 14, offset: #8
    Component Line 15, offset: #15
    Component Line 18, offset: #0
    Component Line 19, offset: #1
    Component Line 20, offset: #5
    Component Line 21, offset: #16
    Component Line 22, offset: #17
    Component Line 23, offset: #28
    Component Line 24, offset: #31
    Component Line 25, offset: #42
    Component Line 26, offset: #43
    Component Line 27, offset: #54
    Component Line 28, offset: #55
    Coverage Group System.Void CoverageConsole.Program::.ctor()
    Coverage Group System.Void CoverageConsole.Program::Main(System.String[])
```

Figure 4.3. The Optimization Model Representing List 4.1

Chapter 5

Summary

In this report I have presented a method how regression testing techniques can be applied in model-based development. The essence of this method is an abstract model that represents the Regression Test Selection problem. This model also separates the optimization from the nature of the input, thus provides easier integration for future input domains, as the optimization algorithm can be re-used.

I have implemented this method in a prototype tool that can select a set of tests from the test suite that verifies the changed part of the system with a reduced cost, compared to the cost of executing all the tests ('re-run all' approach).

I have also applied this model for different use cases, such as optimizing re-runs for generated test cases in a model-driven development environment. The material for this use-case came from the R3-COP and the R5-COP EU research project. In this domain my method can provide the necessary amount of tests based on how the input models changed.

Another use-case of the method was re-building NIST test rooms for emergency responder robots. In this use-case, I have designed the models to represent the test rooms and the capability of the robots. The method is then used to highlight which parts of the robot are not verified by the test room.

To show that the method and the optimization model is indeed adaptable, I have successfully applied it on performing Regression Test Selection of a C# codebase, based on a code coverage report.

The key point in the applicability of the test model, is to define mappings between the input domain and their part in the testing activity, ie. which model elements will act as tests and which elements are the verified components.

5.1 Future Work

Future work may be conducted in area of NIST test rooms. For now this task is mostly designed as an experiment. Feedback from industrial partners would be needed in order to make this use-case applied in the industry.

Another straightforward extension of this work is to apply the method for more domains, such as component models of distributed applications. It would be interesting to apply regression testing method in conjunction with a highly scalable cloud application with continuous deployment involved.

There are also plans on extending the tool to create a model builder component based on Eclipse JDT, the Java development module of the Eclipse IDE. That way optimizing regression testing of Java projects could be integrated in the IDE.

Once more uses are available for the tool, measurements would need to be conducted over various projects and their changesets to measure scalability and efficiency. There should also be measurements to verify the fault revealing effectiveness of the selected test cases are matching the one in the original test suite. This could be achieved through fault injection, and measuring how many of the injected faults are revealed by the tests.

Obviously more development and coding work needs to be done for the tool itself, which is currently only in a prototype phase.

Bibliography

[1] Ágnes Barta. *Absztrakt tesztadat generálás autonóm és elosztott rendszerekhez.* Tech. rep. Department of Measurement, Information Systems, Budapest University of Technology, and Economics, 2015.

[2] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice.* 1st. Morgan & Claypool Publishers, 2012. ISBN: 1608458822, 9781608458820.

[3] J. Connelly et al. "Challenges in Autonomous System Development". In: *Proc. of Performance Metrics for Intelligent Systems (PerMIS'06) Workshop.* 2006.

[4] Emelie Engström, Per Runeson, and Mats Skoglund. "A systematic review on regression test selection techniques". In: *Information and Software Technology* 52.1 (2010), pp. 14 –30. ISSN: 0950-5849. DOI: `10.1016/j.infsof.2009.07.001`.

[5] Elizabeta Fourneret et al. "SeTGaM: Generalized Technique for Regression Testing Based on UML/OCL Models". In: *SERE 2014, 8th Int. Conf. on Software Security and Reliability.* Best paper award. Paris, Dallas, United States: IEEE, June 2014, pp. 147–156. DOI: `10.1109/SERE.2014.28`.

[6] *Guide for Evaluating, Purchasing, and Training with Response Robots using DHS-NIST-ASTM International Standard Test Methods.* National Institute of Standards and Technology. URL: `http://www.nist.gov/el/isd/ms/upload/DHS_NIST_ASTM_Robot_Test_Methods-2.pdf` (visited on 07/30/2015).

[7] H.K.N. Leung and L. White. "Insights into regression testing". In: *Software Maintenance, 1989., Proceedings., Conference on.* 1989, pp. 60–69. DOI: `10.1109/ICSM.1989.65194`.

[8] Zoltán Micskei et al. "A Concept for Testing Robustness and Safety of the Context-Aware Behaviour of Autonomous Systems". English. In: *Agent and Multi-Agent Systems. Technologies and Applications.* Ed. by Gordan Jezic et al. Vol. 7327. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 504–513. ISBN: 978-3-642-30946-5. DOI: `10.1007/978-3-642-30947-2_55`.

[9] G. Rothermel and M.J. Harrold. "Empirical studies of a safe regression test selection technique". In: *Software Engineering, IEEE Transactions on* 24.6 (1998), pp. 401–419. ISSN: 0098-5589. DOI: `10.1109/32.689399`.

[10] "Systems and software engineering – Vocabulary". In: *ISO/IEC/IEEE 24765:2010(E)* (2010). DOI: `10.1109/IEEESTD.2010.5733835`.

[11] D. Tengeri et al. "Toolset and Program Repository for Code Coverage-Based Test Suite Analysis and Manipulation". In: *14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014).* 2014, pp. 47–52. URL: `http://www.inf.u-szeged.hu/~beszedes/research/6148a047.pdf` (visited on 02/27/2015).

[12] S. Yoo and M. Harman. “Regression testing minimization, selection and prioritization: a survey”. In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120. DOI: 10.1002/stvr.430.

[13] Philipp Zech et al. “A Generic Platform for Model-Based Regression Testing”. English. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 7609. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 112–126. ISBN: 978-3-642-34025-3. DOI: 10.1007/978-3-642-34026-0_9.