UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakultät für Elektrotechnik, Informatik und Mathematik
Heinz Nixdorf Institut und Institut für Informatik
Fachgebiet Softwaretechnik
Warburger Straße 100
33098 Paderborn

Coverage Criteria for Testing DMM Specifications

Master's Thesis

Submitted to the Software Engineering Research Group
in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by
Svetlana Arifulina
Röderweg 5
34295 Edermünde-Besse

Thesis Supervisor:
Prof. Dr. Gregor Engels

Thesis Reviewer:
Prof. Dr. Wilhelm Schäfer

Co-Supervisor:
Dipl.-Inf. Christian Soltenborn

Paderborn, September 2011

Declaration

(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original Declaration Text in German:

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

City, Date	Signature

Contents

List of Figures

List of Tables

Acronyms

CPA	Critical Pair Analysis
DMM	Dynamic Meta Modeling
GROOVE	GRaphs for Object-Oriented VErification
GTR	Graph Transformation Rule
GTS	Graph Transition System
OCL	Object Constraint Language
RMM	Runtime Meta Model
SUT	System Under Test
TDSS	Test-Driven Semantics Specification
UQS	Universally Quantified Structure

1 Introduction

Development of large modern software systems, involving participation of experts from different problem domains and development on high quality level with least time efforts, cannot be separated from the usage of models. Models provide beneficial properties, like formal representation, abstraction, separation of concerns, reuse, and are widespread in software engineering nowadays. For example, models are successfully used for communication purposes. Visual modeling languages serve as a convenient means to put together diverse experts, so that it is possible for them to understand each other. Domain-specific languages designed exactly for a particular problem domain enable experts to perform modeling on their own and conveniently exchange information with colleagues from other domains.

In this context, one should be able to judge about the quality of developed models, since error-free artifacts contribute to the quality of software as well. In order to design large and complicated models in a given language efficiently and to be able to check their correctness, the language should be analyzable, that implies it should have a formal definition.

Static concepts of a language can be expressed by a meta model. The problem of a formal definition of dynamic concepts can be solved by using existing approaches for a formal description of language's semantics, like *Dynamic Meta Modeling (DMM)* [4, 11]. Applying DMM, it is possible to formally specify the behavioral semantics of a visual modeling language syntactically based on a meta model, and automatically check whether models designed in this language possess stated properties of correctness.

Soltenborn et al. claim in [17] that DMM semantics specifications were used for model analysis with not enough consideration of their own quality. However, the question of the quality is of great importance, since a semantics specification containing erroneously specified behavior may lead to unreliable results. Consequently, an approach to develop high quality semantics specifications called *Test-Driven Semantics Specification (TDSS)* was proposed in [17]. The idea of this approach is to define a set of example models, on which the conformance between the actual behavior computed for each model based on a corresponding DMM semantics specification and the expected behavior for this model specified by the developer is checked. Error detection in a semantics specification during this process yields its higher quality.

The comparison between testing of software systems and semantics specifications is presented in Figure 1.1. In testing of software systems, a test case consists of an input and expected result. A software system is a system under test (SUT) here. It is run on the given input, and a test result is then compared to be equal with the expected one. For TDSS, an example model corresponds to the test input and its expected behavior corresponds to the expected result forming a single test case. A DMM semantics specification serving as SUT in TDSS and an example model are used by a tool to compute an actual behavior for the given test case. The actual behavior should contain the expected one, in order for the test to pass.

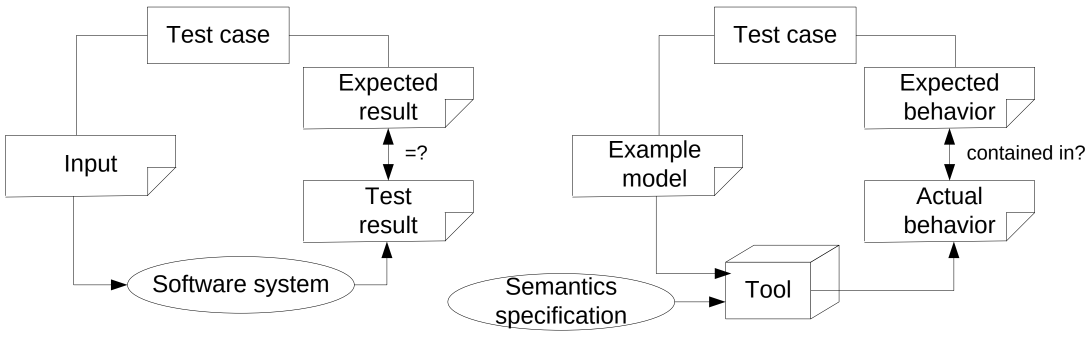


Figure 1.1: Testing of software systems and semantics specifications [17]

Testing in TDSS aims to check the correspondence of a DMM semantics specification with developer's expectations. Having achieved good quality of tests in TDSS, the developer can hope that the quality of the developed DMM semantics specification has also improved, since either no errors were found in the tested parts of the specification and so these parts are assumed to be correct or some errors were revealed and fixed increasing the quality. Thus, one possible solution to tell more about the quality of DMM semantics specifications could be a definition of some criteria that characterize a degree to which these specifications were tested for correctness during their development.

1.1 Problem statement

In order to understand precisely how the quality of tests executed in TDSS can influence the quality of DMM semantics specifications, the structure of these specifications and the process of testing in TDSS are examined closely.

Each DMM semantics specifications is implemented as a set of graph transformation rules (GTRs) called a *ruleset*. GTRs model behavior of language constructs described in a meta model. Each GTR defines how a certain graph, e.g. an example model, should be changed after the application of this rule and consists of a precondition and a postcondition. Precondition is a graph defining a state, when a rule matches, i.e. when this graph is found within the current state graph, the corresponding rule can be applied and the current state is changed in a way defined in the postcondition. A particular matching of the precondition is defined as *matching context*.

As an example consider a DMM rule depicted in Figure 1.2. This rule creates an intern for the given internship. The precondition is a graph representing the internship, for which an intern should be hired, and the postcondition is a graph with the intern belonging to this internship. Black signify elements to preserve in the pre- and postcondition, green refer to new elements in the postcondition.

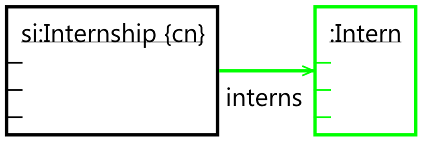


Figure 1.2: Example of a DMM rule

During testing in TDSS, example models exercise a certain part of a DMM semantics specification, which is applicable on them. If this part contains errors, a corresponding test fails, pointing out that this part of the specification has to be reworked by the developer. Depending on the properties of the used example models, a given semantics specification is tested to a certain extent. A larger scope of tested semantics could mean a semantics specification with a better quality, because more potential errors could be detected.

Since an implementation of semantics specifications is accessible and a proportion between the tested and the whole parts of a semantics specification is under consideration, the idea of test coverage in white box testing from software engineering seems to be appropriate for application in this case. Coverage expresses a degree to which a coverage item has been exercised by a test suite. Coverage items represent different structural elements of a semantics specification. A test suite is composed from test cases, see Figure 1.1. Coverage analysis consists in a measurement of coverage achieved during the testing of a semantics specification regarding some criteria determined by the coverage item. If an achieved level of coverage does not satisfy a predefined one, additional test cases are needed.

Two criteria have been proposed in testing of DMM semantics specifications. The coverage criterion that serves as a minimum measure of quality is *rule coverage*. A rule coverage of 100% means that every rule from a given ruleset has been used at least once while testing the semantics on a given set of example models. Rule coverage guarantees that at least one application of each rule is correct and tests only several contexts, which all together use each rule at least once. In contrast, the second criterion, called *all-instances coverage*, ensures that all possible contexts which can occur are tested. This implies that every possible model that can be created in a language has to serve as example model in TDSS.

So, the proposed coverage criteria are two extremes. On the one hand, it is unlikely practical to achieve all-instance coverage, because the number of models, that can be created based on a given meta model, i.e. all possible combinations of language elements defined by this meta model, is infinite. For example, if a variable is defined in a range of real numbers from 0 to 1, it is impossible to test all-instances, i.e. all real numbers in this range, as their amount is infinite.

On the other hand, rule coverage does not test all different matching contexts in which a rule can apply, as for each GTR only one matching is examined, in spite of the fact that a rule can appear and apply in several ones. For example, the rule in Figure 1.2 can apply multiple times during the test execution after different rules, which produce different matching contexts.

In order to overcome the gap between the mentioned extreme techniques in software testing, intermediate methods, e.g. testing only boundary values of a domain or determining test values randomly with a certain distribution, were invented. They deliver more information, than the simplest testing techniques, and produce a feasible number of tests. So, the problem of this thesis is to investigate some efficient and feasible coverage criteria for testing DMM semantics specifications, which are more powerful than rule coverage and generate a reasonable number of tests comparing to all-instance coverage.

1.2 Goals of the Thesis

Based on the information provided above, major goals of this thesis are:

- Design and definition of coverage criteria for testing DMM semantics specifications situated between two extremes of rule and all-instances coverage;
- Formalization of the developed coverage criteria, extracting their commonalities and underlining their differences;
- Tool support for the proposed solution and its integration into the implementation of the DMM workbench;
- Systematization of the developed coverage criteria regarding their expressiveness and computational effort;
- Explanation and documentation of the results.

1.3 Overview of the Thesis

A brief introduction to the topic, problem statement and pursued goals are illustrated in Chapter 1. Theoretical basis needed for understanding of the topic and content of the thesis is explained in Chapter 2. Inspired from the concepts described in Chapter 2, motivation and requirements for the thesis are presented in Chapter 3. New coverage criteria developed with respect to requirements stated in Chapter 3, their formalization and classification are provided in Section 4. Details of the implementation of the concepts illustrated in Section 4 and its evaluation are discussed in Section 5. An application of the critical pair analysis for quality analysis of DMM specifications having no or few invocations is elaborated in Chapter 6. The main achievements and further perspectives for this thesis are shown in final Chapter 7.

2 Foundations

This chapter provides necessary foundations for the comprehension of this thesis, motivation for its goals and inspiration for its ideas. It begins with an introduction of a running example that will be used throughout the whole thesis, in order to understandably illustrate the concepts and ideas brought up in it, see Section 2.1. An insight into the notion of software quality assurance, software testing as a method of quality control, and test coverage as an estimation of the quality of testing in software engineering follows in Section 2.2. Then a description of the DMM approach and the process of specifying semantics formally with the help of this technique are provided in Section 2.3. For the development of high quality semantics specifications, TDSS approach and its relation to the quality of testing semantics specifications are addressed in Section 2.4.

2.1 Running Example

In this section, a running example used for the comprehensible illustration of discussed approaches and their application is presented. At the beginning, an overall idea and design of the example are described in Section 2.1.1. Then, syntax of a language used in the example is given in the form of a meta model in Section 2.1.2 and its dynamic semantics is specified informally in Section 2.1.3.

2.1.1 Idea & Design

The goal of the running example is to model a particular problem domain from the real world. For description of the problem domain, a visual modeling language will be designed, whose syntax defining language elements and relations among them will be represented by a meta model. In addition, the modeled elements will have their own behavior according to which they change in time with respect to each other. The meaning of these elements and dynamical changes that they undergo during time, i.e. dynamic semantics, will be first specified informally using a natural language.

The content of the running example is described below. It simulates a structure of an organization, a human resources administration in a firm, controlling several elements, i.e. departments and internships with employees and interns. The whole system has a certain dynamics, e.g. departments and internships can be dismissed or created, employees and interns can be hired or transferred. The chosen modeling language is rather simple and aims to illustrate common constructs in modeling the behavioral semantics.

Behavioral constructs that the running example is intended to illustrate are:

- Sequence of actions following one another, where an action is something what someone can do or something what can happen in the real world;
- Recursive execution of a single action or an action sequence;
- Choice among several actions at a certain execution point;
- Usage of some actions in different application contexts;
- Application of some actions on different objects of the same type;
- Conditions for an execution of a certain action.

The constructs listed above are key elements for describing the dynamic semantics of a visual modeling language. For example, considering modeling of business processes, a business process consisting of several steps requires a sequence of actions. A repetition of a single step or sequence of steps is needed, when depending on some condition or task, a certain activity is done several times or repeated until the overall activity is finished. A choice among several possibilities is essential, especially in a case, when several mutual exclusive functionalities are available. Some functionality can be used at different places alternating with other actions during one activity. An example of the same action that can be applied to several objects of the same type could be the same action applied to different items of a certain kind. So, sequencing, conditioning and repetition, which are also main structures used in programming languages, should be embraced by the running example.

2.1.2 Syntax

A meta model describing syntax of a modeling language for the running example is illustrated in Figure 2.1.

The meta model consists of the main class *HR* depicting a human resources administration (HR-administration), which manages departments and internships in a firm represented by classes *Department* and *Internship.* These classes inherit from an abstract class *NamedElement*, highlighted with orange color and characterized by a string attribute *Name.* According to the meta model, classes *Department* and *Internship* are containments for classes *Employee* and *Intern* correspondingly which denote to the firm's staff. Employees and interns are identified by IDs and so inherit from the abstract class *IDElement* highlighted with white color. Regarding the cardinalities, each HR-administration may have none or several departments and internships, each department has at least one employee working there, an internship can exists even without any interns or several interns may correspond to one internship.

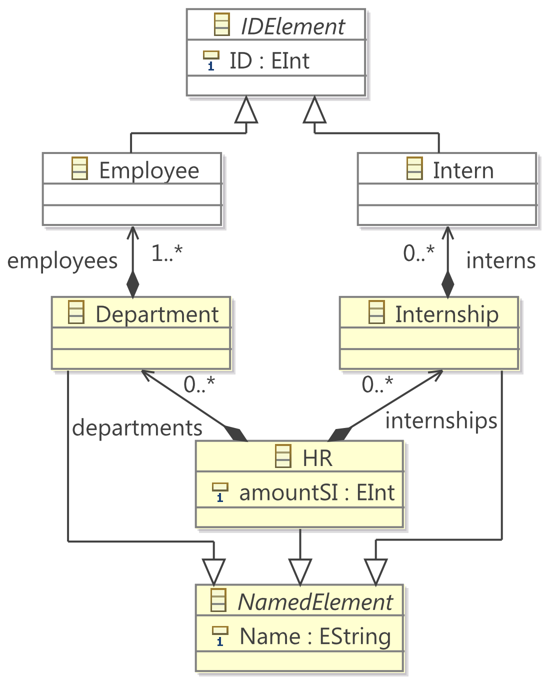


Figure 2.1: Meta model for the running example

So, the syntax of the modeling language used in the example has been depicted and explained above. The dynamic relations among the modeling elements will be presented in the next section.

2.1.3 Semantics

Dynamic semantics represents changes which modeled elements undergo in time. The dynamic semantics for the running example contains the following main activities: an existing internship should be dismissed, unnecessary departments and internships should be dismissed, a special internship should be initiated, and the firm should be reorganized. Detail descriptions of activities can be found below.

2.1.3.1 Dismiss Internship

If an existing internship with at least one intern should be dismissed, then all interns from this internship should be transferred to a suitable existing department. After the transfer, the internship is not needed anymore. In this activity, such constructs for behavior modeling could be involved: a recursive execution of a single action (transfer is repeated until no more interns are left), an action applied several times on different object of the same type (transfer is perform for each intern separately), or a choice among several actions at a certain execution point (still transfer or already delete).

2.1.3.2 Clear Firm

The next activity refers to a discharge of some empty structures in the firm. In this case, departments and internships without employees and interns should be destroyed pairwise, depending on their connections to each other. In this activity, a sequence of actions following one another could be illustrated.

2.1.3.3 Add Special Internship

The next activity concerns an expansion of the firm by adding a new special internship, which has at least one special intern to be immediately hired. Some interns from the special internship should be transferred later either to a department or to a normal internship. If no department appropriate for the transfer is already present in the firm, then a new department should be originated. A newly grounded internship should refer to a suitable department. In this activity, a condition for an action to happen could be illustrated, i.e. a new internship cannot be created until there is no appropriate department for it.

2.1.3.4 Reorganize Firm

This activity is needed for a firm reorganization. Essential part of this reorganization is creation of a special internship for which interns with special skills are hired, see description in Section 2.1.3.3. Two special internships should be produced, and a new department created for the first one should be destroyed as it is not needed in this case. Here, the actions of creating a special internship and deleting a department could be reused in a different application contexts than in Section 2.1.3.2 and Section 2.1.3.3.

2.2 Software Quality Assurance

In this section, the idea of software quality assurance and existing methods for assessing quality of software artifacts are discussed. At the beginning, the notion of software quality is explained in Section 2.2.1. Then, the definition of quality assurance as a way to influence software quality through quality of the development process is provided in Section 2.2.2. Software testing as activity within the quality assurance aimed to control the quality of software products is presented in Section 2.2.3. Consequently, coverage as a means to evaluate the quality of software testing is discussed in Section 2.2.4.

2.2.1 Software Quality

According to IEEE Glossary of Software Engineering Terminology [13], defining terms in field of software engineering, *software* is computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system. The notion of *software quality* is defined as the degree to which a system, component, or process meets specified requirements. Its interpretation can be conveyed by the conception "Quality means conformance to requirements", held by the founder of modern quality assurance Philip B. Crosby 1979 [5].

Summarizing everything mentioned above, a particular computer program consisting of procedures, its documentation and necessary software data form a single software item. Requirements, i.e. expected functionality which has to be provided by this software item, are also specified. In order to determine, whether the program possesses a certain degree of quality, its delivered functionality represented by program outputs have to be compared with the expected one. A software item is considered having a quality, if it meets a specification at hand.

In this section, the notion of software quality was explained. However, ways to achieve a particular level of quality and to check whether a software product conforms to it should be discussed. In the next section, the concept of software quality assurance is presented that provides an insight of how quality can be predefined by the software development process and result from it.

2.2.2 Quality Assurance

According to IEEE Glossary of Software Engineering Terminology [13], *software quality assurance* is a planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.

The goal of software quality assurance is to provide a set of actions performed throughout the development process, in order to guarantee to some sufficient degree that result software artifacts conform to established level of quality. So, quality assurance primarily concerns and evaluates the software development process, whose quality implies a certain quality level of ultimate artifacts.

However, quality of a final product should be evaluated as well. In this case, the notion of quality control is important. *Quality control* is a set of activities designed to evaluate the quality of developed or manufactured products [13]. Thus, quality control is a separate integral part of the quality assurance activities.

In the context above, methods to evaluate the software product's quality within the quality control are required. One of the most widespread technique for this purpose is software testing that is presented in the next section.

2.2.3 Software Testing

According to IEEE Glossary of Software Engineering Terminology [13], *testing* is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items. The main goal of testing is to reveal as many errors in software artifacts as possible, and through the process of continuous correction of identified errors and retesting reach a sufficient quality level [5]. So, testing measures the quality of software in terms of found defects [2].

The illustration of artifacts participating in testing is presented in Figure 2.2. Stereotypes introduced in this diagram correspond to stereotypes from the UML Testing Profile [8].

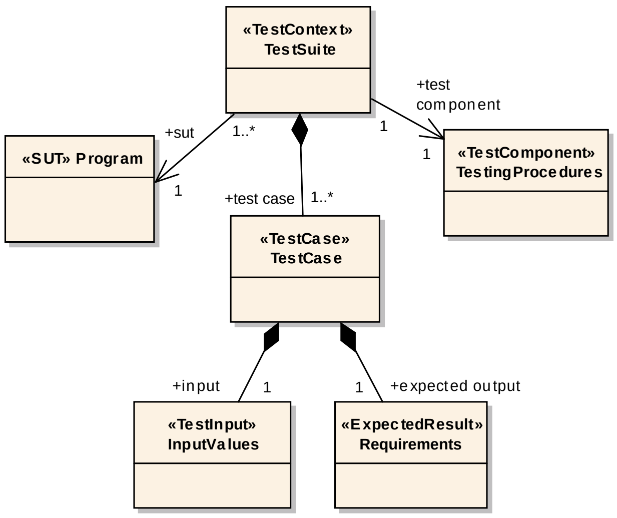


Figure 2.2: Artifacts used during testing

During testing, evaluation of a *test item*, which corresponds to a software item, takes place. Test items can be grouped in one test object, or *system under test (SUT)*. Referring to Daniel Galin in [5], before the testing process begins, a test plan and testing procedures are developed in order to define directives to conform. *Test suite* is a set of test cases selected for examination on SUT. *Test case* consists of input values and expected result developed for a particular objective, such as to exercise a certain program path or to verify compliance with a specific requirement. So, tests simulate special situations, in which the system's behavior is interesting to observe and differences between actual and expected results are detected. All definitions in this paragraph are taken from IEEE Glossary of Software Engineering Terminology [13].

Execution of a single test case for a program consists of the following actions. At the beginning, the program is executed with given input values. Afterwards, the test output has been obtained and serves as input for evaluation of results. The evaluation determines, whether the test output meets requirements stated on the program. Finally, the verdict is determined based on the previous evaluation and serves as output for the whole test case execution.

The process of error identification during software testing may involve different parts of a tested software artifact. Therefore, two classes of testing are defined:

1. *black box testing, or functionality testing*, which considers only generated

outputs comparing them with specified functional requirements and does not examine the internal structure of a test item;

2. *white box testing, or structural testing*, which inspects the internal calculation paths of a test item, where a calculation path is a sequence of computational operations created by a test case during execution.

In order to find as many errors in a tested item as possible, the largest amount of input values for black box testing or computation paths for white box testing should be tested. Thus, the quality of tests can be indicated by a percentage of executed test cases to all possible ones that is called coverage. The notion of coverage as a method to estimate the quality of software testing and its significance are explained in the next section.

2.2.4 Test Coverage

Terminology and the main idea of the notion of coverage are explained in Section 2.2.4.1. Then, a classification of white box coverage criteria is introduced in Section 2.2.4.2, in order to determine criteria relevant for this thesis. In Section 2.2.4.3, the main data structure used for coverage definition in white box testing called control-flow graph is illustrated. In Section 2.2.4.4, some concrete examples of white box coverage criteria are examined.

2.2.4.1 Idea

As mentioned above, it is not only important to evaluate the quality of a software product with the help of testing, but it is equally essential to evaluate the process of testing. For a successful testing, a test suite should consists of such test cases, which exercise SUT to the largest extent while achieving stated objectives [19].

According to Standard Glossary of Terms Used in Software Testing [1], *coverage* is the degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite. *Coverage item* is a structural element used as a basis for test coverage, e.g. lines of code or code statements. Each coverage item defines a coverage criterion corresponding to it. Measurement of achieved coverage with respect to some coverage criterion, in order to determine whether additional testing is required and if so which test cases are needed, is referenced as *coverage analysis*. Coverage analysis is realized by *coverage tool* that provides objective measures of what coverage items have been exercised by a test suite.

According to Daniel Galin [5] and IEEE Glossary of Software Engineering Terminology [13], test coverage for black box testing is the degree to which a given test or set of tests addresses specified requirements for a given SUT. The full coverage is achieved when outputs of test cases executed on the SUT meet all the requirements. Test coverage for white box testing refers to the parts of SUT's internal structure. The full white box coverage is achieved, when all possible structural elements related to the chosen coverage item are exercised during testing.

The integration of coverage analysis into the testing process is illustrated in Figure 2.3.

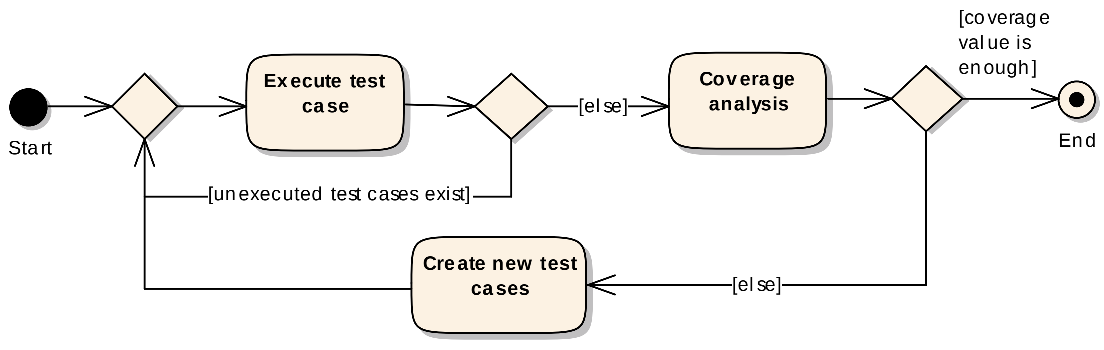


Figure 2.3: Integration of testing and coverage analysis

The coverage analysis is performed after all test cases have finished. The coverage value is calculated relatively to a predefined coverage item and a decision whether additional test cases should be designed is made. The coverage analysis either continues or the result coverage is output.

Depending on a type of testing and a chosen coverage item, different families of coverage criteria exist. Their classification is presented in the next section.

2.2.4.2 Classification

The classification of existing coverage criteria relevant for this thesis is presented in Figure 2.4. As described in [19], depending on a type of testing, structural and data coverage criteria can be distinguished. *Data coverage criteria* correspond to black box testing and define coverage on the input data space. They aim to choose a set of good values from the set of all possible input values that is usually infeasible to test as a whole. Two extreme data coverage criteria are one-value and all-values coverage. As an example, consider a variable defined in the range of real numbers from 0 to 1. One-value coverage demands to test at least one value from this range, e.g. 0,5. All-values coverage requires to test all values from this domain whose number is infinite in this case.

Structural coverage criteria refer to white box testing and deal with coverage of the control flow. They can be of control-flow-oriented and data-flow-oriented types. Control-flow-oriented coverage criteria refer primarily to structure of the program code and include statement, decision and path coverage as the most important ones for this thesis. Data-flow-oriented coverage criteria deal with the definition and use of data variables and have all-defs, all-uses and all-def-use-paths criteria as a part of their classification.

Since data coverage criteria are defined on an input data set, no additional data structures are required for coverage computation. In contrast, for structural coverage criteria and in particular for control-flow-oriented ones, an additional

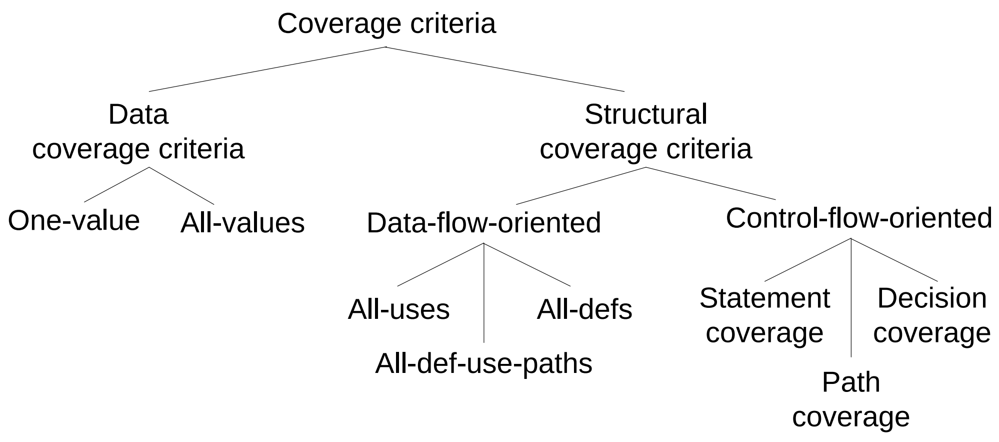


Figure 2.4: Classification of relevant coverage criteria [19]

representation of the control flow is necessary, in order to be able to operate with its structural elements in a way needed for coverage computation. Such an auxiliary data structure is called *control-flow graph* and is presented in the next section.

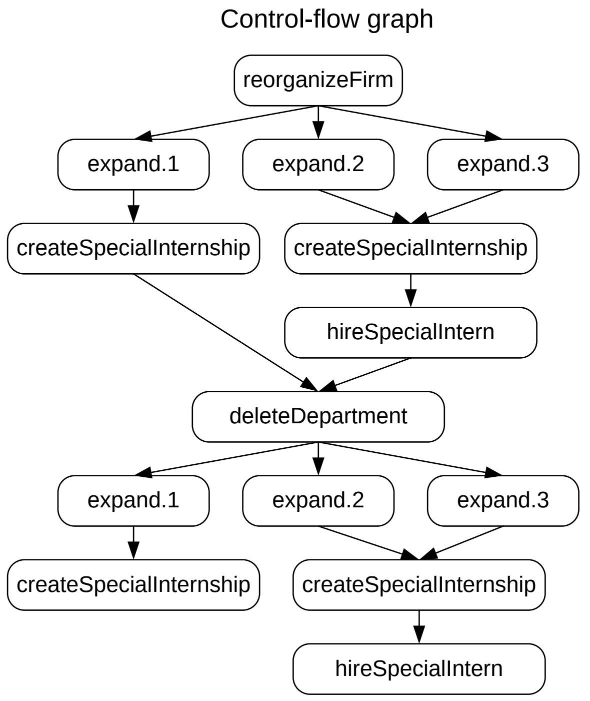
2.2.4.3 Control-Flow Graph

Control-flow graph is a data structure representing control flow of programs. *Control flow* is the sequence in which operations are performed during the execution of computer program [13]. According to Standard Glossary of Terms Used in Software Testing [1], *control-flow graph* is an abstract representation of all possible sequences of events (paths) in execution through a component or system. Thus, control-flow graph represents all possible control flow variations of a certain computer program, component or system.

As discussed by Robert Gold in [6], operations in the control flow are commands of a programming language or statements of a program written in this programming language. However, nodes of control-flow graph can be different data structures. For this thesis, the control-flow graph having program statements as nodes is of special interest, since statements are similar to rules that form a DMM semantics specification, see Section 2.3.2.

Control-flow graph has a form of a directed graph with two marked nodes: entry and exit node. Entry node corresponds to the beginning of the control flow and is a unique node having an empty preset in each control-flow graph. Exit node has an empty postset and expresses the end of the control flow either because a function has been left or because it ended. Edges of a control-flow graph represent the order of statements in the control flow. Each edge expresses the situation when a statement corresponding to the target node of this edge is executed immediately after a statement corresponding to its source node.

An example consisting of a program and a control-flow graph built for it is depicted in Figure 2.5. The program implements the activity *reorganizeFirm* from the running example, see Section 2.1.3.4. The implementation starts with the name of the public method *reorganizeFirm*, which aims to create a new special internship in the firm. It is an entry node in the invocation graph, as it starts the control flow and has no incoming edges. Then, three methods are called: *expand*, *deleteDepartment*, and *expand* again. The functionality of the method *expand* is described below.



Program

```
public void reorganizeFirm() {

    expand();
    deleteDepartment();
    expand();
}

private void  expand() {

    if (not X & not Y) {
      expand.1();
      createSpecialInternship();
    } else if (X & not Y) {
      expand.2();
      createSpecialInternship();
      hireSpecialIntern();
    } else if (X & Y) {
      expand.3();
      createSpecialInternship();
      hireSpecialIntern();
    }
}
```

Figure 2.5: Control-flow graph

The method *expand* contains three conditions determining which branch to follow during the execution. Depending on the condition whether a department and an internship suitable for transfer already exists in the firm or not, different types of expansion take place. According to the first condition, a special internship is originated, but no intern is hired, while in the next two conditions a certain expansion is first performed followed by creation of a special internship and hiring a special intern. The choice is illustrated as a fork in the control-flow graph and results in three branches originating from the enter node.

A node representing *createSpecialInternship* is inserted twice into the control-flow graph and the node *expand.1* is connected to the first one and the nodes *expand.2* and *expand.3* are connected to the second. The second node of *create-SpecialInternship* is followed by the node of *hireSpecialIntern* in the control-flow graph. Two nodes of *createSpecialInternship* cannot be merged together, since the computation path containing *hireSpecialIntern* does not correspond to the computation path of *expand.1* having no such statement. Since no further functionality is specified for *createSpecialInternship* and *hireSpecialIntern*, it ends the functional call of *expand* and so the control flow for its branches.

The control flow for *reorganizeFirm* continues with the functional call of the method *deleteDepartment*, destroying an unnecessary department. This node is inserted into the control-flow graph after the execution of the first functional call ends. Further operations within the method *deleteDepartment* are not specified, so this node is followed by the last functional call belonging to *reorganizeFirm*, which repeats the program's beginning. The control flow for *reorganizeFirm* is concluded with this functional call and its structure is repeated from the previous similar one in the control-flow graph.

Control-flow graph contains all possible paths through control flow, some of which can be covered by a test case during the test execution. The whole amount of paths covered by a test suite shows tested part of a software item.

For the definition of a single coverage criterion, a program is projected onto a control-flow graph. Then, a coverage item is chosen from the control-flow graph, defining a particular coverage criterion. Depending on the coverage item, a set of structural elements to be covered for this criterion is selected from the control-flow graph. All elements from such a set should be executed during testing to achieve the coverage of 100%. Finally, coverage analysis for the defined coverage criterion is performed on this set. Coverage is calculated as a ratio of the amount of tested elements from the set to the amount of all elements in it.

Definitions of some coverage criteria relevant for this thesis and their computation are examined in the next section.

2.2.4.4 Coverage Criteria

As illustrated in Section 2.2.4.2, coverage criteria can be of data or structural type. The weakest criterion among structural ones is statement coverage. *Statement coverage* demands a test suite to execute every statement in a program at least once. It corresponds to coverage of control-flow graph's nodes. A formula for statement coverage is: $coverage_{statement} = \frac{amount\ of\ executed\ statements}{amount\ of\ all\ statements}$.

This criterion is considered to be weak, since it does not test some control flow structures for common errors, which cannot be revealed even when the statement coverage of 100% is obtained. An example for such structures could be a loop that will be tested only with one iteration, or empty branches in if-then-else conditions.

A structural coverage criterion that handles all possible cases exhaustively is *all-paths coverage*. It requires testing every path through the control-flow graph. This is a very strong criterion, since all possible sequences of execution events have to be executed during testing. However, all-paths coverage is often impossible to realize, because the computation can be hampered by the presence of loops causing an infinite amount of paths. If no loops are present, all-paths coverage implies a check of all combinations of decision outcomes. The computation of all paths might be sometimes unpractical because of a huge amount of conditional branches. A formula for all-paths coverage is: $coverage_{all-paths} = \frac{amount\ of\ tested\ paths}{amount\ of\ all\ paths}$.

A structural coverage criterion situated between two extremes above is decision coverage. *Decision coverage* ensures that each decision outcome is tested at least once. This criterion implies statement coverage, since decision coverage checks each branch resulting from a decision that implies executing every statement at least once. Decision coverage can detect more erroneous structures than statement coverage, e.g. empty else-statements. However, loops still remain unchecked in contrast to all-paths coverage. A formula for decision coverage is: $coverage_{decision} = \frac{amount\ of\ tested\ decision\ outcomes}{amount\ of\ all\ decision\ outcomes}$.

The coverage criteria above cannot always reach a maximum of 100%, because of the presence of unreachable statements, impossible decision outcomes, and unsatisfiable paths. This is a common problem in the field of testing for which several approaches facilitating its handling exist. For unreachable statements, static analysis for "dead" code can be applied. A technique of finding infeasible paths and abandoning them during testing is described by Gupta et al. in [10].

A data coverage criterion similar to all-paths coverage and testing a problem domain exhaustively is *all-values coverage*. It forces to test every value from an input set. This criterion leads to combinatorial complexity for many input domains, i.e. tends to be either too complicated in computation or impossible to realize at all. An example could be a variable of type integer, for which testing the whole domain of integer numbers requires too much computational effort.

The definitions above are based on Standard Glossary of Terms Used in Software Testing [1], Certified Tester Foundation Level Syllabus [2], IEEE Glossary of Software Engineering Terminology [13], and the work of Utting et al. [19].

Section 2.2 provides an overview of concepts regarding software quality, quality assurance and quality control. Up to now these concepts were discussed in application to software in general. In the next section, software items primarily relevant for this thesis, i.e. DMM semantics specifications, and the approach for their development are presented.

2.3 Dynamic Meta Modeling

In this section, an introduction to DMM, the approach to specify semantics for visual modeling languages formally, is provided. Section 2.3.1 gives an overview of the DMM approach. General constructs of the language for semantics specification with DMM are presented in Section 2.3.2. Then the mechanism to control the execution process in DMM semantics specifications, i.e. invocation mechanism, is illustrated in Section 2.3.3. After specifying the semantics with the mentioned constructs, a method for computation of the actual behavior based on the developed semantics specification is explained in Section 2.3.4.

2.3.1 Overview

A description of Dynamic Meta Modeling (DMM) is provided in the dissertation of Jan Hendrik Hausmann [11]. DMM is an approach for formal specification of dynamic semantics for visual modeling languages. The syntax of these languages must be represented in the form of a meta model as determined in the OMG modeling and metadata specifications. All considered models must be in compliance with the defined meta model and can be seen as sentences of a language specified with it. The goal of such formal specification is to enable automatic analysis of models' behavior and to facilitate understandability of their precise semantics [4].

An overview of the DMM approach is presented in Figure 2.6.

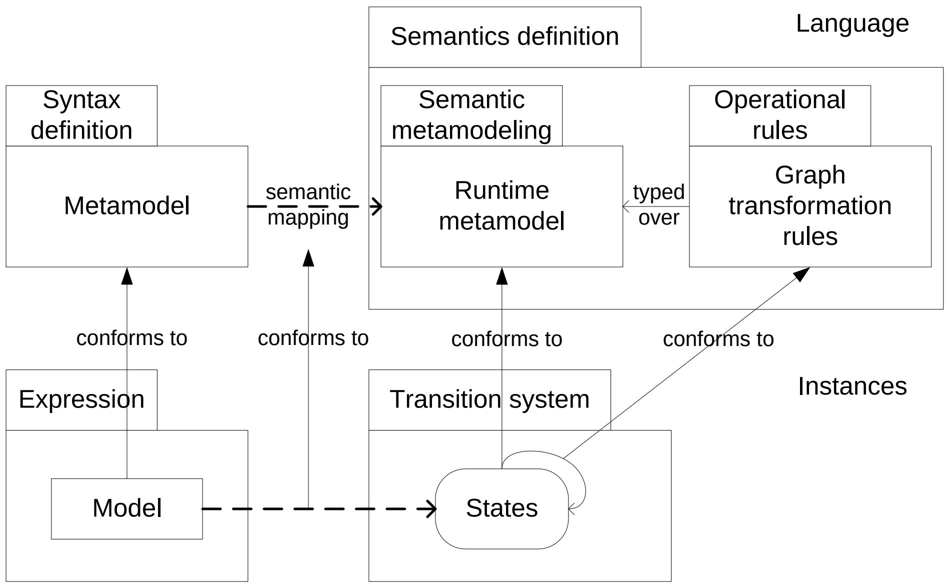


Figure 2.6: Architecture of the DMM approach

A DMM semantics specification consists of two components: a runtime meta model, which expresses states of execution, and operational rules, which describe behavior by operating with instances of the runtime meta model. The runtime meta model is an extension of the language's syntactic meta model with concepts referring to behavior at runtime. Such concepts may be a token and its location, information about the active state and other constructs expressing states of execution in different modeling languages. If a language has no runtime concepts, syntactic and runtime meta models coincide. Operational rules represent the dynamic semantics and describe how instance of the runtime meta model change over time. Instances of the runtime meta model are mapped to graphs typed over the runtime meta model which are used in the operational rules.

In order to formalize the behavior of the running example with the help of DMM, the syntactic meta model and informal description of its semantics, provided in Section 2.1.2 and Section 2.1.3, are given as input for a DMM semantics specification. The runtime meta model coincides with the syntactic meta model, as no special execution constructs are needed in this case, and states of execution can be represented by instances of the syntactic meta model. Out of the informal semantics, operational rules are elaborated and presented in Example 2.1.

Example 2.1 *Operational rules representing semantics for the running example formally are illustrated in Figure 2.7. Structure of the rules and their types will be explained in the next section.*

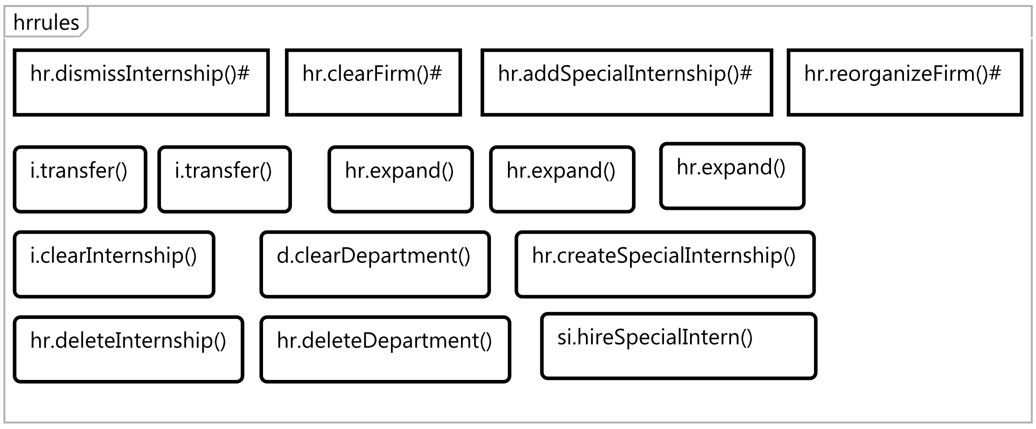


Figure 2.7: Ruleset for the running example

In this section, the main idea of DMM was discussed. Concrete syntax of the DMM language and its general constructs will be presented in the next section.

2.3.2 General Constructs

Operational rules comprising DMM semantics specifications, see Figure 2.6, are implemented as graph transformation rules (GTRs). A single GTR expresses alternation of an initial graph in some way which results in a new graph. Each GTR has left-hand side and right-hand side, which correspond to precondition and postcondition for it. Left-hand side defines conditions under which a GTR can apply by specifying a subgraph to be changed during the application of this GTR. A particular matching of the GTR's precondition on the state graph is called *matching context*. Right-hand side contains a specification of a post-state resulting from GTR application, i.e. a subgraph which should substitute the matching context.

Host graph is a graph to be changed by a GTR. A certain GTR can apply on a host graph if matching between the left-hand side of the GTS and the host graph occurs, i.e. the graph on the left-hand side is a subgraph of the host graph. Afterwards, the matching context is modified to the GTR's postcondition.

Syntax of the DMM language used for description of operational rules is defined by the ruleset meta model. An excerpt from the ruleset meta model regarding GTRs is presented in Figure 2.8.

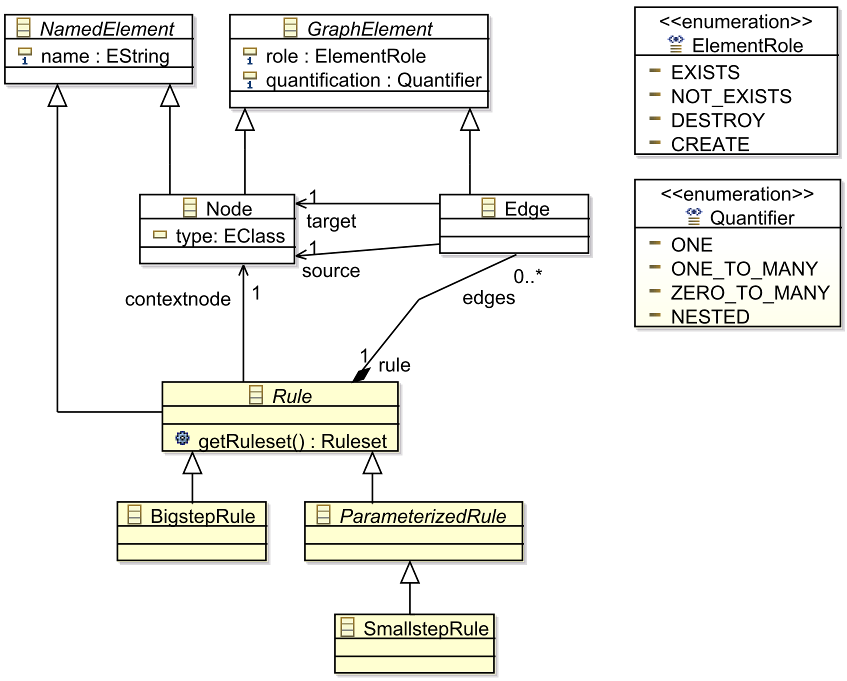


Figure 2.8: Excerpt from the ruleset meta model regarding GTRs

Graph transformation rules (GTRs) are represented by the abstract class *Rule*. Each rule belongs to a certain set of rules, called *ruleset*, that can be accessed through the function *getRuleset*. Such ruleset is presented in Example 2.1.

All rule in a ruleset inherit from the abstract class *Rule*. Types of rules relevant for this thesis are represented by the classes *BigstepRule* and *SmallstepRule*, because they influence the execution while the others do not change the current graph state anyhow. *BigstepRule* describes rules which apply whenever their left-hand side matches with the host graph. *SmallstepRule* describes rules that can apply only if they are invoked by other rules. The concept of invocations will be discussed in Section 2.3.3. Each type can be distinguished according to name convention for the rules in DMM: names of bigstep rules end with '#' and names of smallstep rules have no additional signs. The type *SmallstepRule* also inherits from an intermediary abstract class *ParameterizedRule*, allowing them have parameters relevant for invocations.

The main reason for distinguishing several types of rules in DMM semantics specifications is a separation of different logical parts for better readability and understandability maintenance of rulesets. As it can be concluded from the names of the rule types, bigstep rules model bigger steps of execution and their matching cannot be controlled by developer. On the other hand, smallstep rules model smaller pieces of logic, whose order and place of execution can be set by developer with the help of invocations, see Section 2.3.3.

A rule may have several implementations, so every rule is characterized by two names: *name* and *uniqueName* obtained from the classes *NamedElement* and *Rule* correspondingly. The attribute *name* refers to the name of a rule similar for all implementations of this rule. The attribute *uniqueName* is a name discriminating each instance of a rule and describing concrete rules used during the execution. A rule having two implementations in the DMM semantics specification describing behavior for the running example is provided in Example 2.2.

Example 2.2 *The rule modeling the transfer of interns during the internship dismissal, see Section 2.1.3.1, has two implementations depicted in Figure 2.9 and Figure 2.10. The first rule aims to delete one intern from the internship and create a new employee corresponding to this intern in the given department. The second rule is meant as a stop for the transfer and applies, when no more intern are left in the internship and multiple employees representing the transferred interns exist in the given department.*

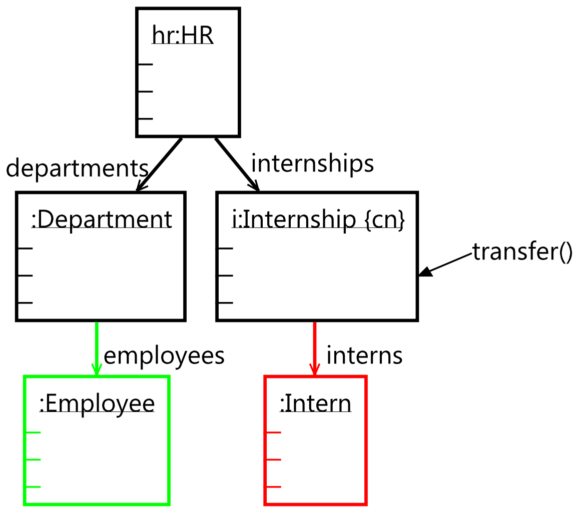


Figure 2.9: Smallstep rule transfer.1

Each GTR operates with graphs, whose elements are represented by instances of the class *GraphElement*, see Figure 2.8. White color highlights elements relating to it. This class has two inheriting classes: *Node* representing nodes of GTRs and *Edge* representing connections between two distinct nodes. An edge references exactly one source and one target node. Every node or edge can belong to exactly one rule. A rule consists of at least one node and none or several edges depending on the amount of nodes and existing connections among them. As shown in Example 2.2, the rule *transfer.1* consists of five nodes with four edges between them.

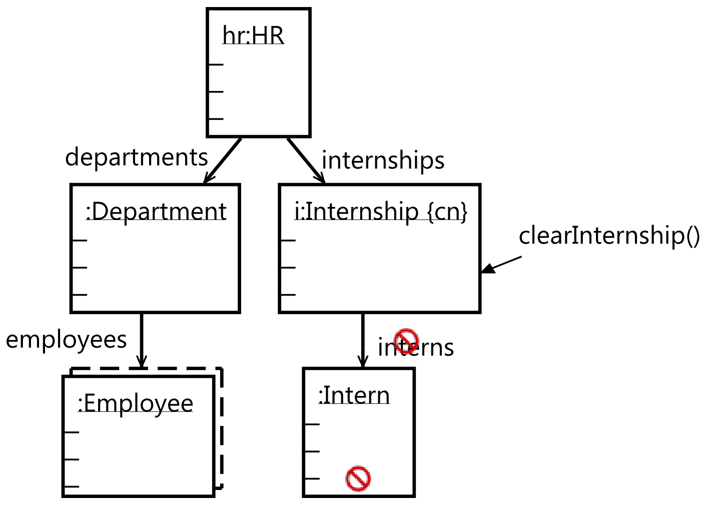


Figure 2.10: Smallstep rule transfer.2

A node is also characterized by name inherited from the abstract class *NamedElement*. Each GTR is bound to a specially marked node on the left-hand side referred to as *context node*. Context node is also told to own the GTR's behavior and must be set always in its implementation. In both rules in Example 2.2, the node named *i* of type *Internship* is chosen as a context node, that is represented by the sign {cn}.

For all elements of GTRs, side of a GTR to which a considered graph element belongs to is assigned through the attribute *role* of the class *GraphElement*. According to enumeration *ElementRole*, there are three different roles serving this purpose: {exists} for elements on the left-hand side which should be preserved after the rule application, {delete} for elements on the left-hand side which should be deleted during the rule application, and {create} for new elements on the right-hand side. In the Example 2.2, the left-hand side of the rule *transfer.1* consists of three node with the role *exist* depicted as black and of one node with the role *destroy* depicted as red. A new element which should be created on the right-hand side has the role *create* is marked with green.

In some situation, a certain data structure is not allowed to appear on the left-hand side of a GTR. For this purpose, DMM also allows to specify *negative application conditions (NACs)*, which denote the absence of a certain data structure in order for matching to succeed. A rule containing NACs is the rule *transfer.2* presented in Example 2.2. Here, the condition that no intern should exist in order for the rule to match is modeled with the help of NAC illustrated as a stop sign.

Sometime more than one instance of the same type needs to be processed on the same fashion. For this purpose, DMM has *universally quantified structures (UQSs)*, which enable successful matching for all elements in the host graph corresponding to the left-hand side. UQSs can be defined by setting the attribute *quantification* of the class *GraphElement*. Values resulting in UQSs are given by the enumeration *Quantifier*, see Figure 2.8, and equal to *one_to_many* or *zero_to_many*. The rule *transfer.2* in Example 2.2 contains UQSs as well. In order for the rule to apply, presence of a department with one or several employees is required. As the number of employees depends on a particular model, they are designed as UQS, which means that the same matching occur for all instances of class *Employee*.

In this section, main constituents of DMM semantics specifications - rules with their properties - were discussed. In the next section, a mechanism to control the execution of rules in DMM and constructs related to it are explained.

2.3.3 Invocation Mechanism

In this section, *rule invocations* as a structural mechanism in DMM allowing distribution of the logic over multiple rules is illustrated. An excerpt from the ruleset meta model regarding the invocation mechanism is presented in Figure 2.11.

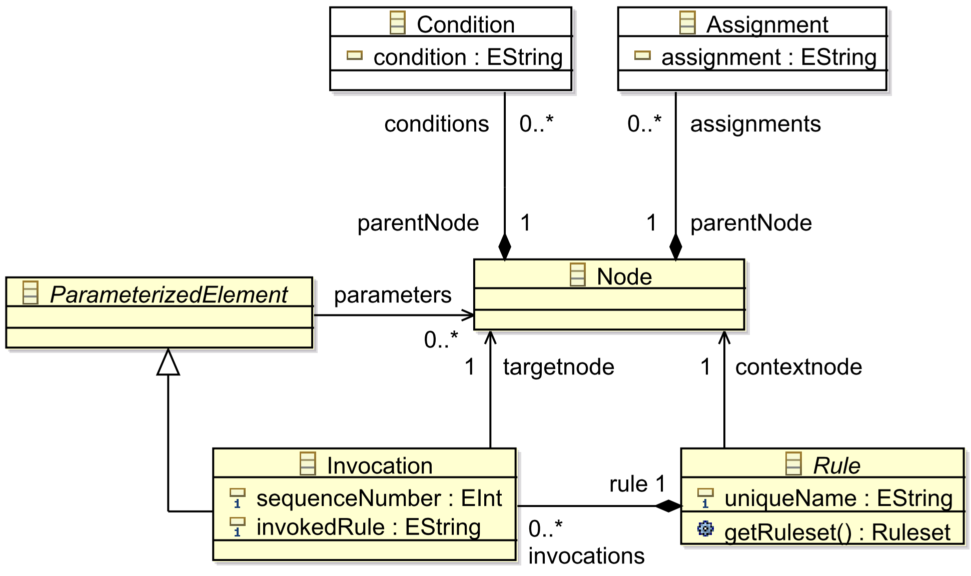


Figure 2.11: Excerpt from the ruleset meta model regarding the invocation

As shown in this figure, each rule may have no or multiple invocations. Each invocation is an instance of the type *Invocation* that has two attributes: *invokedRule* specifies a rule invoked by the rule owning the invocation and *sequenceNumber* defines the number of this invocation in the row of other invocations belonging to this rule. When a rule defines several invocations, sequence numbers fix the order of their execution and influence the kind of succeeding rules during the execution. If no sequence numbers are specified, the order of invocations is determined by the tool performing the execution, see Section 2.3.4. A rule illustrating concepts explained above is presented in Example 2.3.

Example 2.3 *Bigstep rule performing the activity of firm's reorganization, see Section 2.1.3.4, is presented in Figure 2.12. The rule consists of three invocations with the order fixed by sequence numbers. First, the invocation* expand *is proceeded, then comes* deleteDepartment*, and* expand *is called for the second time.*

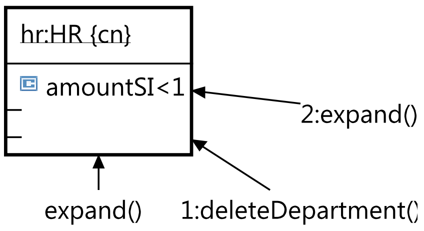


Figure 2.12: Bigstep rule reorganizeFirm

Invocation is always defined on a certain node called its *target node*. This node restricts the possible matching of an invoked rule on the host graph to some predefined elements. These predefined elements are also given through the abstract class *ParameterizedElement* which references none or multiple nodes as a context for a certain invocation. When an invocation is called, the type of its target node should comply to the context node type of the specified invoked rule. Context and target nodes are like objects in object-oriented programming, while rules and invocations are like functional calls. In Example 2.3, the node *hr* of type *HR* serves as a target node for all three presented invocations, and is indicated by arrows pointing at it.

Another mechanism for restriction of matching context is specification of assignments and conditions. These are defined on graph nodes, and each belongs to a single node. Conditions have a form of mathematical expressions, and a corresponding GTR can match only when all its conditions hold. In Example 2.3, the rule can apply only if no special internships currently exist in the firm. This is programmed by the condition that the counter *amountSI* defined for the class *HR* should be less than 1. Assignments aim to change values of attributes. Example 2.4 demonstrates the usage of assignments for the running example.

Example 2.4 *The smallstep rule initiating a new special internship is shown in Figure 2.13. As far as the internship is created, the amount of special internships of the firm should be increased by 1. This is done through the assignment of the attribute* amountSI *in the class* HR*.*

Invocations may also fail, when an invoked rule cannot match on a given host graph. In this case, a special rule *dmm_specification_failure* matches indicating

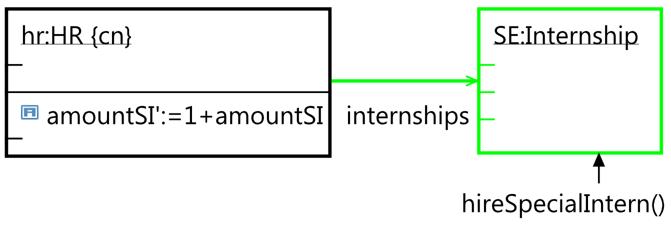


Figure 2.13: Smallstep rule createSpecialInternship

that an error occurred during the execution. With the help of this rule, the developer can track a point where the specification failure is located and its reason.

As already explained in Section 2.3.1, bigstep rules match whenever they can, so their names cannot be given as values for the attribute *invokedRule* in the class *Invocation*. Thus, only smallstep rules can be invoked. Which rule owns an invocation does not matter, so it can be both bigstep and smallstep rules. A single bigstep rule with all following smallstep rules derived from the specified invocations is considered as a single functional unit. A bigstep rule finishes its execution, when no more unprocessed invocations are left in it. No another bigstep rule is allowed to interrupt the execution of the current one. Thus, a control mechanism is formed by bigstep rules matching whenever suitable and smallstep rules executing the behavior as a developer programmed it.

Two previous sections provide an insight of structure of rules representing DMM semantics and a way they can be controlled with the help of the invocation mechanism. These concepts describe the behavior for the whole modeling language. The form of actual behavior for sentences of this language and a way to compute it are discussed in Section 2.3.4.

2.3.4 Computing GTS

Each model, i.e. instance of some syntactic meta model, represents a sentence in a language described by this meta model and has a form of a graph typed over it. Each model has a certain behavior described by the corresponding DMM semantics specification. A concrete behavior of such a model is computed as graph transition system (GTS) based on DMM semantics specification. GTS has a form of labeled transition system.

GTS is a sequence of execution events corresponding to rules from the ruleset applied to the model during the execution. GTS contains all sequences of rules, which can be derived starting with a model by using the invocation mechanism and arbitrary matching of bigstep rules. The computation process stops if no more bigstep rules can match on the current graph state or DMM specification failure in a smallstep rule occurred.

The toolset GROOVE computes GTSs in DMM. An example of a GTS computed for a model specified in the language of the running example is presented in Example 2.5. For more information about the GROOVE toolset see [15].

Example 2.5 *A GTS computed by the toolset GROOVE based on DMM semantics specification for the running example is illustrated in Figure 2.14. The input model consists of one department having no employees and one internship having one intern.*

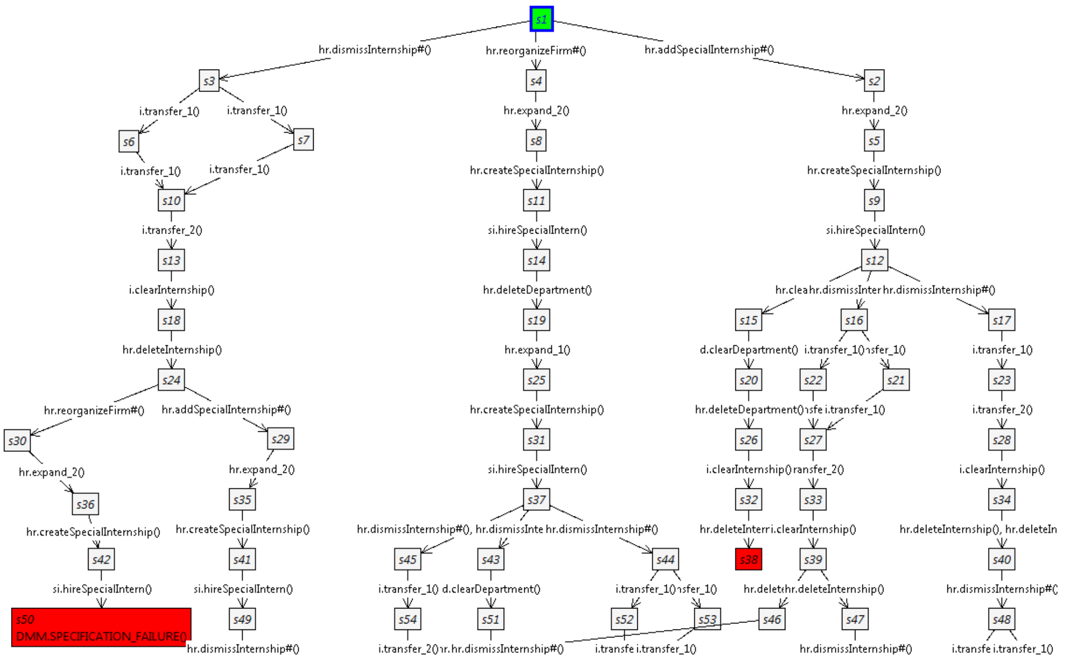


Figure 2.14: Graph transition system computed by GROOVE

The GTS consists of an initial state corresponding to the input model. Labels on transitions refer to the names of rules applicable to their source state. As shown in the graphic, three bigstep rules are applicable to the graph resulted from the input model. Each of these bigstep rules are followed by different smallstep rules depending on specified invocations. Red states correspond to the end of a computation path. One path ends with the DMM specification failure, pointing on a failed matching.

GTSs can be analyzed by applying model checking techniques. This is the point where automatic analysis of models is enabled, that was the main purpose of the behavior specification using DMM. In order to carry out the analysis, model checking requires two input parameters: GTS and properties for which the actual behavior should be checked.

In this section, the process of formal semantics specification with the help of DMM was introduced. However, as explained in Section 2.2, quality of designed artifacts should be assessed as well, in order to know to which extent the created formal semantics satisfies initial requirements. Test-Driven Semantics Specification (TDSS), the approach to assist in quality assurance of DMM semantics specifications including the quality control of created artifacts, is presented in the next section.

2.4 Test-Driven Semantics Specification

The process of creating high quality DMM semantics specifications with the help of continuous testing in TDSS is explained in Section 2.4.1. Methods to evaluate the quality of executed tests are discussed in Section 2.4.2.

2.4.1 High Quality Semantics Specifications

Quality evaluation of DMM semantics specifications is important, since it delivers assessment of a degree, to which a created formal behavior specification reflects the informal semantics that it describes. It is especially significant, since a semantics specification is helpful only when it is correct to some extent, otherwise the results are unreliable.

In order to evaluate the quality of a DMM semantics specification, initial requirements should be provided and developed artifacts should be compared for compliance with them. Requirements are any artifacts against which a formal semantics can be evaluated. For DMM, it might be UML specification in case when the semantics of UML diagrams is to be modeled [4]. For the running example, the informal description of modeled activities provided in Section 2.1 can serve as initial requirements.

Quality assurance explained in Section 2.2.2 aims at providing adequate confidence that a software item conforms to established technical requirements, while testing checks whether a software item is correct to some extent, see Section 2.2.3. Analogously, TDSS takes the quality of DMM semantics specification into consideration and checks their correctness with respect to stated requirements. It is inspired by the test-driven development approach and described thoroughly by Soltenborn et al. in [17].

Artifacts used in TDSS and their interrelations are presented in Figure 2.15. These artifacts can be compared to those illustrated in Figure 2.2 for testing in software engineering. Stereotypes introduced in this diagram correspond to stereotypes from the UML Testing Profile [8]. Test context *SemanticsTest* corresponds to *TestSuite* in software engineering and interacts with DMM semantics specification serving as SUT. Test cases consist of example models serving as test input. Expected result for test cases in TDSS is traces of execution events expressing expected behavior of example models.

TDSS testing process is illustrated in Figure 2.16. TDSS starts with creation of models corresponding to language elements which should be exercised by testing. Then, discussing models' semantics, execution events are identified and formalized

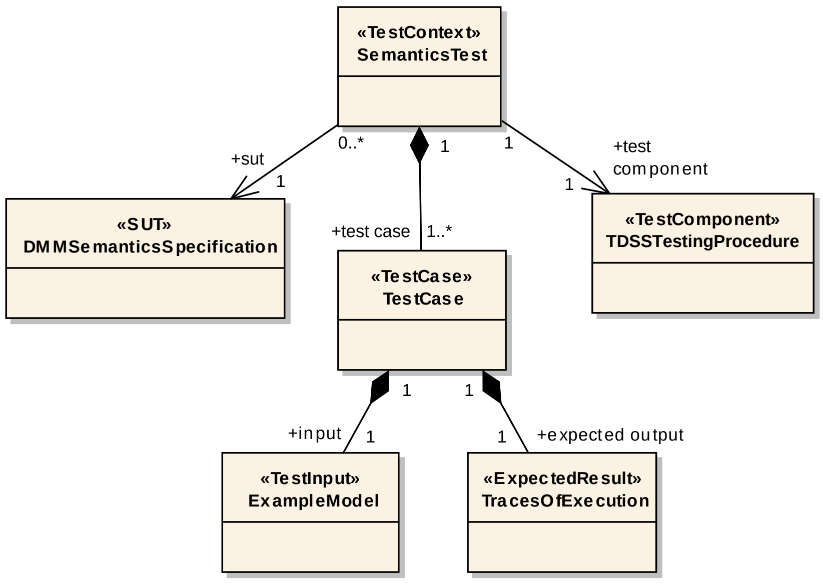


Figure 2.15: Artifacts used in TDSS

using traces of execution events. This notation allows describing the behavior precisely but at the same time informally. However, traces of execution events cannot be used by a model checker for the automated analysis on a transition system, so they are translated into some temporal logic dialect, e.g. CTL formulas. Test cases are created out of constructs specified above.

After all language elements are covered by designed test cases, their execution begins. For all test cases, based on the DMM semantics specification at hand and the model from the current test case, a GTS is computed. Then, developed CTL formulas are verified for being contained in the computed GTSs. If the verification fails, the corresponding test case fails as well and the DMM semantics specification needs to be corrected by the developer. The execution continues until all test cases have passed.

The process of testing a DMM semantics specifications on a set of models containing important or interesting language structures and correcting specification failures depending on test verdicts, gives a hope to yield a specification with a higher quality level than the one created without following TDSS. However, in order to estimate how good a DMM semantics specification elaborated with TDSS is, the question of how good a given DMM semantics specification has been tested in TDSS arises. In order to be able to answer this question, the quality of executed tests has to be discussed.

2.4.2 Quality of Tests

The quality of testing DMM semantics specifications can influence the quality of these specifications, as successful testing detects errors and so improves the quality.

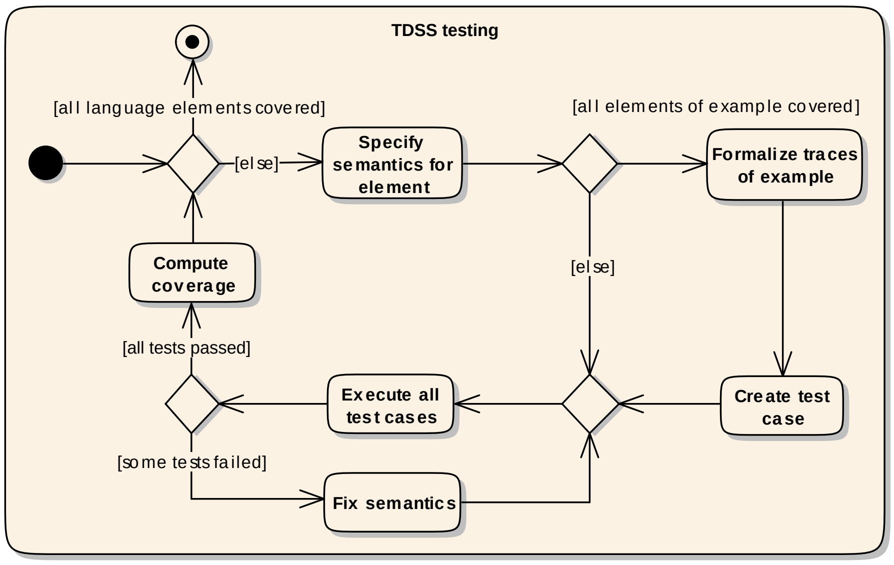


Figure 2.16: TDSS testing process with coverage analysis

The type of testing used in TDSS is white box testing, since the developer has an access to internal structure of the specification's implementation, i.e. ruleset.

According to Section 2.2.4, coverage is a method to evaluate results of white box testing. It expresses a ratio between structural elements of a SUT, which were exercised during testing, and all possible structural elements defined for testing with regard to a certain coverage criterion. Figure 2.3 describes the integration of testing and coverage analysis in software engineering. Coverage analysis measures achieved coverage after test suite's execution and determines, whether additional testing is necessary.

The extension of TDSS with coverage analysis is presented in Figure 2.16. The coverage analysis in TDSS is performed after all test cases have successfully passed. It consists of calculation of the coverage value regarding a chosen coverage criterion, which serves as output of the coverage analysis. If the coverage value is not sufficient, then additional test cases should be designed in order to execute untested parts of the specification. If the coverage value satisfies the predefined level of quality, i.e. all language elements are covered, TDSS with coverage analysis finishes. The overall results of this process are semantics specifications with a possibly higher level of quality. It is also possible to compare several semantics specifications relatively to each other with respect to a similar criterion.

This chapter provided important foundations of approaches, which serve as an inspiration for later parts of this thesis. Detailed requirements for this thesis and their motivation are presented in the next chapter.

3 Requirements for the Thesis

In this chapter, detailed requirements for this thesis will be formulated. At the beginning, a motivation for new coverage criteria in order to evaluate testing of DMM semantics specifications more thoroughly is presented in Section 3.1. Then, requirements based on which new coverage criteria for testing DMM semantics specifications will be developed are provided in Section 3.2.

3.1 Motivation for New Coverage Criteria

Inspired from the coverage criteria in software engineering introduced in Section 2.2.4.4, two coverage criteria for testing DMM semantics specification are already formulated. The first coverage criterion roughly analogous to statement coverage is explained in Section 3.1.1. The second coverage criterion roughly analogous to all-values coverage is introduced in Section 3.1.2. Limitations associated with these criteria are discussed as well and serve as a motivation for the definition of further coverage criteria, which should overcome them.

3.1.1 Rule Coverage

Rule coverage is a coverage criterion corresponding to statement coverage, which ensures that each statement of a tested software artifact is executed at least once. In DMM, the software artifact exercised during testing is the ruleset. So, rule coverage demands each rule of a considered ruleset being used at least once during the semantics test execution, i.e. applies at least in one matching context. Thus, rule coverage tests several matching contexts, which all together use each rule from the ruleset at least once.

In order to achieve a rule coverage of 100% in TDSS expanded with coverage analysis, such set of example models should be developed, whose models together use all rules from the DMM semantics specification during the execution of the semantics test. The formula for rule coverage is: $coverage_{rule} = \frac{amount\ of\ used\ rules}{amount\ of\ all\ rules\ in\ ruleset}$.

Analogous to the estimation of statement coverage's power, rule coverage is also considered as weak. As it is explained in Section 2.1.1, such constructs as one rule used several times at different places or recursively executed actions may appear in semantics specifications. In the case of rule coverage analysis, exercising one single matching context picked out of all contexts in which a particular rule occurs is enough to cover it. Thus, some matching contexts are not being checked for correctness.

3.1.2 All-Instances Coverage

The next existing coverage criterion for DMM corresponds to all-values coverage for black box testing in software engineering. This criterion tests values from the input domain exhaustively. According to Figure 1.1, for DMM semantics specifications this means to test all possible models that can be created based on the syntactic meta model of the considered modeling language. The formula for all-instance coverage is: $coverage_{all-instances} = \frac{amount\ of\ tested\ example\ models}{amount\ of\ all\ possible\ example\ models}$.

All-instances coverage performs testing of all matching contexts relevant for a considered modeling language. Even if not the whole DMM specification has been tested, it means that some parts of it are never used by any input. But for all possible inputs, the specification has been checked for correctness.

However, a complete testing of all elements from the language's input domain is highly unpractical, since the amount of possible models that can be created in the language is usually infinite. The simple example for such a situation is an association end with the cardinality which allows many instances of a type but does not specify an upper bound. Such construct results in an infinite number of models. Thus, the computation is impossible in this case.

In order to overcome the gap between the extreme techniques in software engineering, some intermediate criteria were designed. Examples are decision coverage from structural coverage criteria and all-boundaries or random-value coverage from data coverage criteria [19]. These intermediate criteria require more test cases as the simple ones but still remain computationally feasible. The same intuition will be used in this thesis for the definition of new coverage criteria for DMM. The concrete requirements for the new coverage criteria are given in the next section.

3.2 Requirements for New Coverage Criteria

In testing DMM semantics specifications, some intermediate coverage criteria in between of the two introduced extremes of rule coverage and all-values coverage are desirable. Their main goal is to deliver more insight into the quality of semantics specifications but simultaneously to require feasible computation complexity. Depending on the structure of the semantics specification at hand, a suitable coverage criterion should be chosen.

On the one hand, new coverage criteria should be more expressive than rule coverage, which also serves as a minimum quality measurement of DMM semantics specifications. It implies that they should test rules in more different matching contexts than rule coverage. An example could be a smallstep rule deleting a department for the running example that is used in several activities, i.e. at different execution points. For the coverage value of 100% according to rule coverage, testing this rule only once is enough, but other its applications still remain unchecked. So, new coverage criteria should provide more information about the correctness of rule's applications in different contexts. New coverage criteria should be formally defined and their interrelations among each other with respect to increasing expressiveness should be checked through the possibility to built a hierarchy among them.

On the other hand, new coverage criteria should be computable in comparison to all-values coverage. They should result in a finite amount of cases to check and a finite amount of structural elements to cover as well. An example of computational feasibility is rule coverage, which requires to check each rule in the given ruleset, whose amount is predefined and finite. In order to prove that the new coverage criteria are computable, a coverage tool should be developed and applied on DMM semantics specifications of example modeling languages.

In this chapter, the motivation and requirements for new coverage criteria were presented. Their concrete realizations will be shown in the next chapter.

4 Coverage Criteria in DMM

In this chapter, new coverage criteria for testing DMM semantics specifications will be introduced, formally defined, applied to the running example, and further discussed. A high level introduction to data structures and algorithms needed for the new coverage criteria are presented in Section 4.1. Then, a formalization of data structures and mathematical concepts common for all new coverage criteria are provided in Section 4.2. Based on concepts described in the previous sections, two intermediate families of coverage criteria - rule coverage and edge coverage criteria, their commonalities, peculiarities, and application cases are described in Section 4.3 and Section 4.4 correspondingly. The strongest coverage criterion, all-paths coverage, its idea, application and properties are discussed in Section 4.5. Finally, connections among the introduced coverage criteria according to their expressiveness are shown as a hierarchy illustrated in Section 4.6.

4.1 Introduction

In this section, the basis and inspiration for data structures required by the new coverage criteria are presented, see Section 4.1.1. Then, the general idea of the algorithm for coverage analysis in DMM and integration of the presented data structures in it are explained in Section 4.1.2.

4.1.1 Inspiration

Control-flow-oriented coverage criteria serve as inspiration for the definition of new coverage criteria in DMM, since the implementation of DMM semantics specifications is available that leads to the direction of structural coverage criteria and no actual data flow is present in GTRs that makes control-flow-oriented ones to be of interest. These criteria require the control-flow graph, whose structure is obtained by analyzing the internal structure of a software item and built from executable elements following each other in a predefined order. A software item in this case is a program written in a certain programming language, and executable elements are statements comprising this program. The order of statements in the control-flow graph is determined by their order in the program.

In DMM, a software item consisting of executable elements with a certain order can be found as well, where executable elements are rules of a given ruleset, see Section 2.3.2. The invocation mechanism explained in Section 2.3.3 determines the control flow in DMM. According to it, bigstep rules can match whenever it is possible depending on the current graph state and are not able to interrupt the execution of other bigstep rules. Smallstep rules can apply only if they are called either by a bigstep or by a smallstep rule and if they can match on the current graph state. Thus, the possible order of rules in the control flow is predefined by a DMM semantics specification, where invoked rules follow a rule containing them. If multiple invocations are present in a rule, their order is decided using sequence numbers. Thus, the sequential order of rules resembles the sequential order of statements.

Additionally, each programming language has mechanisms to express conditions resulting into several branches of execution and loops making some pieces of a program repeatable depending on a given condition. Forks and loops are reflected in the control-flow graph as well. Forks are depicted as several edges outgoing from a single node. Loops are sketched as an edge pointing backwards to some node that has been already seen during the execution.

Similar to program structure, forks and loops can be found in a ruleset as well. A fork corresponds to a case, when a rule has several implementations. In this case, a branch that will be pursued during the execution is determined by the current graph state, i.e. matching context. A recursion reflects a loop, i.e. a case when a certain rule invokes itself or a rule that has been already executed in the same matching context before. If a rule invoking itself invokes some other rules as well, all these rules become a part of the loop. After a fork or a loop finishes, the sequential order of the rules is restored.

As shown above, a structure similar to a control-flow graph can be created with elements existing in DMM. Nodes of the new graph are bigstep and smallstep rules. The order of nodes is determined by derivation sequences of invoked rules, which are sequences of invocations computed for a single bigstep rule and consisting of smallstep rules invoked further. Forks and loops can be represented as nodes with several outgoing edges and edges referencing backwards. This data structure is called *invocation graph*. The similarity between the control structure in code testing and DMM are also illustrated in Table 4.1 on Page 33.

Table 4.1: Analogy between the control structure in code and DMM testing

Elements	Control-flow graph	Invocation graph
Node n	Statement, e.g. x = 10	Rule, e.g. init.1()
Edge $e : n1 \rightarrow n2$	$n2$ directly follows $n1$ in the control flow	$n2$ follows $n1$ in the derivation sequence of invoked rules

Since bigstep rules cannot invoke each other, and only smallstep rules can form a certain invocation structure, an invocation graph is constructed for each bigstep rule separately. The example of invocation graph for the bigstep rule *reorganizeFirm* implementing the behavior described in Section 2.1.3.4 is illustrated in Figure 4.1. The derivation sequence of rules invoked by this rule can be seen on the right-hand side of the figure. Based on it, the invocation graph is constructed and presented on the left-hand side of the figure. This invocation graph can be compared with the control-flow graph shown in Figure 2.5.

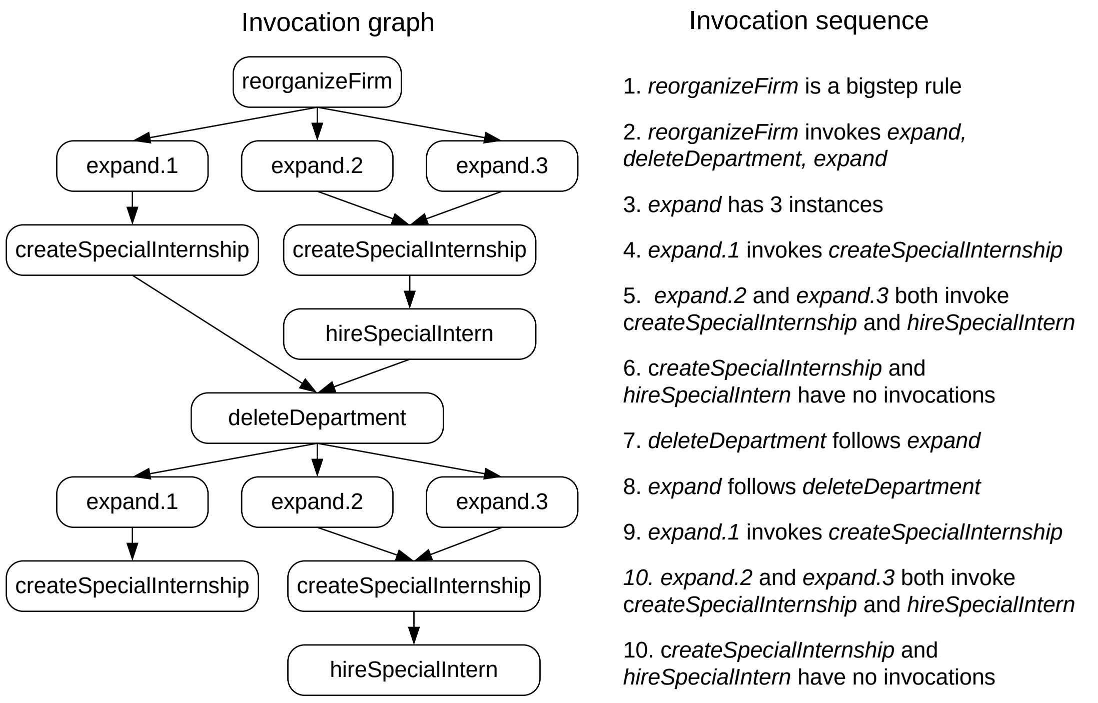

Invocation graph
reorganizeFirm
expand.1
expand.2
expand.3
createSpecialInternship
createSpecialInternship
hireSpecialIntern
deleteDepartment
expand.1
expand.2
expand.3
createSpecialInternship
createSpecialInternship
hireSpecialIntern
Invocation sequence
1. reorganizeFirm is a bigstep rule
2. reorganizeFirm invokes expand, deleteDepartment, expand
3. expand has 3 instances
4. expand.1 invokes createSpecialInternship
5. expand.2 and expand.3 both invoke createSpecialInternship and hireSpecialIntern
6. createSpecialInternship and hireSpecialIntern have no invocations
7. deleteDepartment follows expand
8. expand follows deleteDepartment
9. expand.1 invokes createSpecialInternship
10. expand.2 and expand.3 both invoke createSpecialInternship and hireSpecialIntern
10. createSpecialInternship and hireSpecialIntern have no invocations

Figure 4.1: Invocation graph

The bigstep rule *reorganizeFirm*, see Example 2.3, serves as a root for the invocation graph. This rule has three invocations *expand*, *deleteDepartment*, and *expand* again, which are executed one after another regarding their sequence numbers (0, 1 and 2 correspondingly). The first invocation *expand* has three smallstep rules implementing it. In the invocation graph, it corresponds to a node with the name *reorganizeFirm* having three outgoing edges to nodes, each representing one instance of the rule *expand*: *expand.1*, *expand.2*, or *expand.3*. The rule *expand.1* invokes the smallstep rule *createSpecialInternship*, while rules *expand.2* and *expand.3* invoke rules *createSpecialInternship* and *hireSpecialIntern* sequentially. So, the node *expand.1* is connected with the node *createSpecialInternship.* The rules *expand.2* and *expand.3* point to the another node *createSpecialInternship* followed by *hireSpecialIntern*, as their individual subgraph are identical, and so do sets of executional paths starting from these rules.

The rules *createSpecialInternship* and *hireSpecialIntern* are the last in this series as they have no invocations themselves. In the invocation graph, it results in an edge from the node *createSpecialInternship* followed the node *expand.1* and from the node *hireSpecialIntern* belonging to the branches of *expand.2* and *expand.3* to the node corresponding to *deleteDepartment*, i.e. next invocation of the rule *reorganizeFirm.* The rule *deleteDepartment* has no more invocations, so the last invocation of the rule *reorganizeFirm* is processed. The last invocation *expand* completely repeats the logic of the first invocation. However, its existing representation in the invocation graph cannot be referenced, since this invocation ends the execution of the bigstep rule and its end rules serve as leaves for the whole invocation graph. But the first invocation is followed by two more ones, so referencing it would not end the execution process.

Analogously to the control-flow graph containing all paths through the control flow, the invocation graph contains all combinations of sequences in which rules of a ruleset appear. The problem of infeasible paths arises here as well, since these sequences are computed based solely on the information about invocations that does not take into account all properties of rules, e.g. conditions, assignments, or implemented logic. An example of an infeasible path is a recursive call of a rule, which has a condition determining its end after three loops but which will result in an infinite amount of paths in the invocation graph.

In this section, a data structure similar to control-flow graph called invocation graph necessary for the definition of new coverage criteria in DMM has been illustrated. Its role and application in the algorithm for the coverage analysis in testing DMM semantics specifications will be explained in the next section.

4.1.2 Intuition for the Algorithm

In order to compute data structures for coverage analysis in DMM, a transformation of the ruleset in the form of invocation graphs is necessary. For each bigstep rule, an invocation graph can be built. For this technique, only bigstep rules having *at least one invocation* are considered, since each of bigstep rules having no invocations form only one computation path consisting of this bigstep rule itself, and so the application of the rule coverage is enough to test all computation paths. As a result, a particular ruleset is mapped onto a set of invocation graphs.

Analogously to the control-flow graph, having a set of invocation graphs and a definition of a coverage criterion, elements that have to be covered to achieve 100% coverage regarding this criterion are selected from the invocation graphs. For the rule coverage, see Section 3.1.1, this set is a set of nodes from all invocation graphs corresponding to the rules having distinct unique names, i.e. a rule is considered being covered if it was used at least once no matter where it is located. A set of elements covered during the execution is a subset of elements to cover appearing in GTSs computed for example models from the semantics test, see Section 2.3.4. Coverage equals to the ratio between these two sets.

The algorithm of coverage analysis regarding a single coverage criterion is presented in Figure 4.2. This diagram can be enhanced by the information from Figure 2.16. The algorithm begins with computation of a set of invocation graphs from a given ruleset. It is performed once at the beginning and the set is used for the whole coverage analysis with several coverage criteria.

Then, the definition of the coverage criterion starts. It consists of a choice of coverage item and selection of elements from the invocation graphs, which satisfy

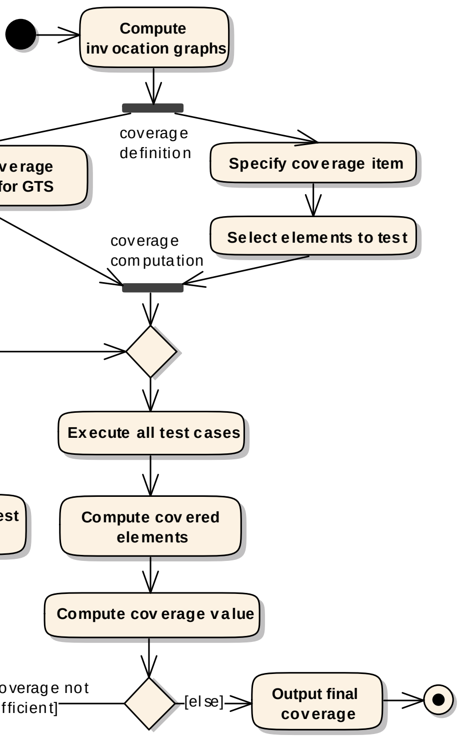


Figure 4.2: Algorithm for coverage analysis in TDSS

the definition of the coverage item. In parallel, a specification of conditions, under which selected elements are considered covered by a GTS takes place.

After the coverage criterion has been specified, the execution of semantics test with the integrated coverage analysis begins. After a set of test cases specified by the developer is executed, covered elements from the computed GTS are identified. Then, the coverage value is computed. If the reached coverage does not satisfy a predefined level, new test cases are created and the process of coverage computation continues, otherwise the final coverage value is output and the whole algorithm finishes.

So, the algorithm of coverage analysis in testing DMM semantics specifications is general for all coverage criteria. The only difference from one criterion to another is in a chosen coverage item and conditions determining which elements are covered by a computed GTS. Thus, a formalization common to all new coverage criteria and individual characteristics of each criterion are outlined in the next sections.

4.2 Common Formalization

The formal definition of coverage criteria in DMM requires the formalization of the required data structures as well. Data structures common for coverage analysis in DMM are described formally in this section. The invocation graph serving as a basis for the coverage analysis is explained in Section 4.2.1. Further necessary data structures defined on the basis of invocation graphs are illustrated in Section 4.2.2.

4.2.1 Invocation Graph

Firstly, the algorithm for building an invocation graph will be explained in the form of pseudo code in Section 4.2.1.1. Invocation graphs constructed for the running example are depicted in Section 4.2.1.2 and will be used later for illustration of developed coverage criteria. The formalization process of data structures necessary for the formal definition of the invocation graph is described in Section 4.2.1.3. According to the described process, data structures in the form of data sets derived from the ruleset meta model are presented in Section 4.2.1.4. The formal definition of the invocation graph is provided is Section 4.2.1.5.

4.2.1.1 Computation

The algorithm for computation of a single invocation graph for the activity *Compute invocation graphs* in Figure 4.2 is described in this section in the form of pseudo code. Its first part is presented in Figure 4.3.

```
computeGraph(Rule bigstepRule) : Graph

invocationGraph : Graph
leafRules : Set of Rule
invocation : Invocation
invocations, followingInvocations : List of Invocation

IF bigstepRule has invocations THEN

  ADD bigstepRule to invocationGraph
  COMPUTE invocations of bigstepRule

  ADD bigstepRule to leafRules
  SET followingInvocations to invocations

  FOR each invocation from invocations
    SET followingInvocations to followingInvocations without invocation
    CALL computeInvocationStructure(leafRules, bigstepRule, invocation, followingInvocations)
    SET leafRules to result of computeInvocationStructure
  ENDFOR

  RETURN invocationGraph

ENDIF
```

Figure 4.3: Algorithm for invocation graph computation, part 1

The algorithm takes a bigstep rule as input and produces an invocation graph as output. The following data types are also used in the algorithm: *Rule* refers to rules, *Invocation* describes invocations, and *Graph* represents a directed graph with nodes of type *Rule*. The computation starts with the check, whether the input bigstep rule has invocations, since a bigstep rule is processed further only if it has invocations. Then, the rule is inserted into a newly initialized invocation graph. A set of invocations for this rule is computed, in order to determine further invocation structure.

Two additional data structures are also necessary for the computation. These are a set of rules currently serving as leaves of the invocation graph (*leafRules*) and a set of invocations following a currently handled rule (*followingInvocations*). The set *leafRules* delivers the result of going one step deeper into the invocation structure and recursively achieves a list of rules that have no more invocations and so conclude the current derivation sequence. These rules serve as source nodes for the edges connecting them with the starting rules of a succeeding invocation. The set *followingInvocations* for a rule is defined as a list of invocations, belonging to the same invoking rule and having larger sequence numbers than the current one, united with invocations following the invoking rule. For the bigstep rule *reorganizeFirm* from the running example, described in Example 2.3 and Figure 4.1, the set *followingInvocations* for the invocation of *expand* with the sequence number 0 consists of the following invocations *deleteDepartment*, *expand*, and *createSpecialInternship* invoked by this rule. This set is required for merging of identical subgraphs in the invocation graph that facilitates the coverage analysis a lot since equal matching contexts are aggregated.

The two sets mentioned above are initialized: the set *leafRules* first contains only the current bigstep rules, as no invoked rules have been computed up to know, and the set *followingInvocations* consists of all invocations of the current rule. Then, processing of invocations begins. For each invocation from the set of invocations computed for the given bigstep rule, the set of following invocations is modified by deleting the current invocation from it that preserves invocations only with larger sequence numbers. This set will be enhanced by rules immediately invoked by the rules corresponding to handled invocations. At this point this information is not available.

Then a special procedure that computes the invocation structure for the bigstep rule is called. Its input parameters are the set *leafRules*, the bigstep rule, the current invocation, whose invocation structure will be computed, and the set of following invocations, that is needed if merging occurs. As a result, the set *leafRules* is changed to the result of the computation of invocation structure.

The second part of the algorithm describing the computation of invocation structure is presented in Figure 4.4.

In order to compute the invocation structure for a rule given as input, each rule from the set *leafRules* is processed. Then, invoked rules corresponding to this invocation are found and for each rule a check takes place, whether a node describing a current invoked rule already exists in the invocation graph. If such a node is present, then the possibility for merge is examined. Merge happens when the set of following invocations for the new and the existing nodes are equal. If so, then a new edge from the input rule referencing the existing node is created. Otherwise, a new node and associated data structures are built. So, a node for

```
computeInvocationStructure(Set of Rule leafRules, Invocation invocation, List of Invocation
followingInvocations) : Set of Rule

currentRule, invokedRule: Rule
invokedRules, newLeafRules, nextLeafRules : Set of Rule
nextInvocation : Invocation
invocations, nextFollowingInvocations : Set of Invocation

FOR each currentRule from leafRules

  COMPUTE invokedRules of the invocation

  FOR each invokedRule from invokedRules

    IF invokedRule already exist in invocationGraph THEN

      ADD first invocation of invokedRule to followingInvocations
      IF followingInvocations equal to following invocations of exising node THEN
        ADD edge : currentRule → existing node of invokedRule
      ENDIF

    ELSE

      ADD invokedRule to invocationGraph
      ADD edge : currentRule → invokedRule
      SET following invocations of invokedRule

      COMPUTE invocations of invokedRule
      ADD invokedRule to nextLeafRules
      SET nextFollowingInvocations to invocations
      ADD following invocations of invokedRule to nextFollowingInvocations

      FOR each nextInvocation from invocations
        SET nextFollowingInvocations to nextFollowingInvocations without current invocation
        CALL computeInvocationStructure(nextLeafRules, invokedRule, nextInvocation,
        nextFollowingInvocations)
        SET nextLeafRules to result of computeInvocationStructure
      ENDFOR

      ADD nextLeafRules to newLeafRules

    ENDIF

  ENDFOR

ENDFOR

RETURN newLeafRules
```

Figure 4.4: Algorithm for invocation graph computation, part 2

the invoked rule is added to the invocation graph, an edge connecting the current rule from the *leafRule* and the invoked rule is created, following invocations for the invoked rule are set.

Afterwards, the procedure similar to that done for the bigstep rule and described in Figure 4.3 is performed for the invoked rule. At the beginning, invocations of this rule are computed. Then a new set *nextLeafRule* is initialized with the invoked rule added. A new set of following invocations is computed out of the set of invocations belonging to the invoked rule plus the set of invocations following the invoked rule itself. For each invocation from the set of invoked rule's invocations, the computation of invocation structure is repeated recursively. The recursion stops when no more unprocessed invocations exist.

As a result of the computation of invocation structure, the whole invocation structure for the bigstep rule is being constructed gradually and constitutes the corresponding invocation graph. The concepts explained above are demonstrated in the next section by invocation graphs constructed for the semantics specification of the running example.

4.2.1.2 Example

A set of invocation graphs for four bigstep rules from the ruleset shown in Example 2.1 is presented in Figure 4.5. Design of the invocation graphs is explained below.

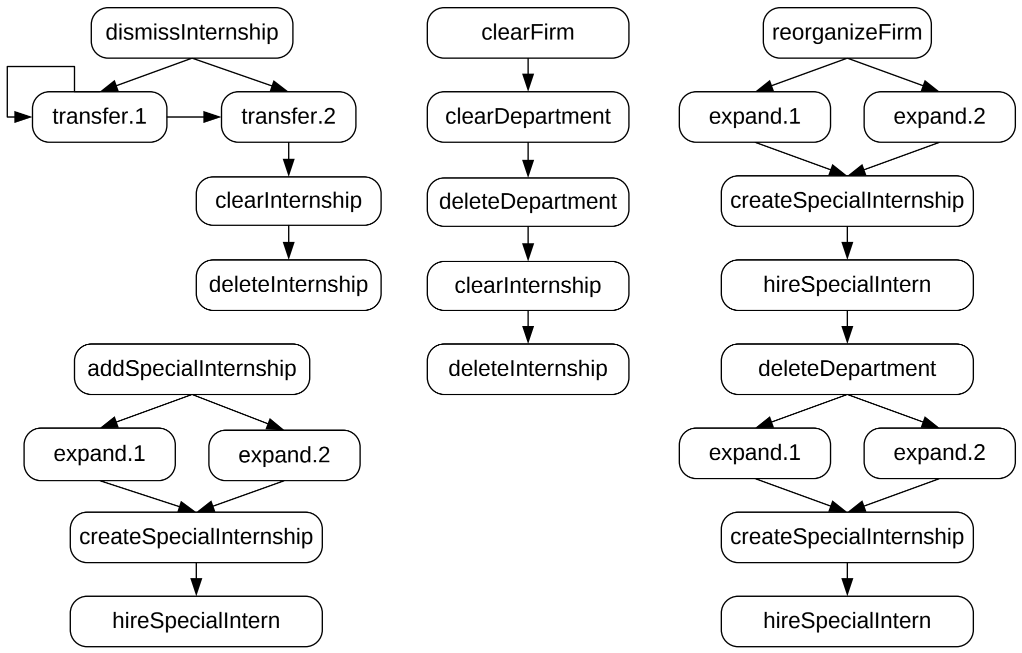


Figure 4.5: Set of invocation graphs for the running example

4.2.1.2.1 Dismiss Internship The invocation graph for the bigstep rule *dismiss-Internship* starts with a fork representing the invocation of the rule *transfer*, which is implemented by the two rules *transfer.1* and *transfer.2* resulting in two different branches in the control flow. The rule *transfer.1* invokes the rule *transfer* again. This invocation corresponds to a recursive call of the rule *transfer.1* or the end of the recursion through the rule *transfer.2*. So, *transfer.1* can point directly to *transfer.2* enabling the merge of several execution paths. The rule *transfer.2* has one further invocation *clearInternship*, which follows it in the invocation graph. The rule *clearInternship* invokes the rule *deleteInternship*, which concludes the invocation sequence for this bigstep rule, since it does not invoke any more rules.

4.2.1.2.2 Clear Firm

The invocation graph for the bigstep rule *clearFirm* consists of two invocation structures, one for each of its two invocations *clearDepartment* and *clearInternship*. Their order is not fixed through sequence numbers, however only one combination is modeled, where the invocation structure of *clearDepartment* is succeeded by the invocation structure of *clearInternship*, based on the alphabetical order. The invocation structure of *clearDepartment* consists of one rule *deleteDepartment*, which is placed immediately after it. Analogously, the rule *deleteInternship* is put after *clearInternship*. As neither *deleteDepartment* nor *deleteInternship* invoke further rules, *deleteDepartment* is connected with *clearInternship*, and the execution of the bigstep rule stops after *deleteInternship*.

4.2.1.2.3 Add Special Intern

The invocation graph for the bigstep rule *addSpecialIntern* has one fork for two implementations of the rule *expand*. Here *expand* is simplified by having two implementations instead of three, as illustrated in Figure 4.1. Both instances of the rule *expand* invoke the same rules *createSpecialInternship* and *hireSpecialIntern*, so both branches of *expand* can be merged. The rule *hireSpecialIntern* ends the whole execution, as it has no further invocations.

4.2.1.2.4 Reorganize Firm

The invocation graph for the bigstep rule *reorganizeFirm* is presented in Figure 4.1. This invocation graph is simplified as well, leaving out the branch of *expand.1* and preserving only aggregated branches of *expand.2* and *expand.3*.

4.2.1.3 Formalization Process

For the formal specification of the data structure needed for the coverage analysis, a connection between the ruleset meta model defining constituents of DMM semantics specification formally and invocation graphs is necessary. Thus, nodes and edges of invocation graphs are derived from the instances of this meta model, i.e. concrete rulesets. This means that a set of rules and a set of invocations have to be extracted from any given ruleset in order to have building elements for an invocation graph.

To gather objects for building invocation graphs, operations to collect instances of the necessary classes from the ruleset meta model are required. Such operations are provided by the Object Constraint Language (OCL) that has a number of types and commands allowing the manipulation of such collections [20]. With the help of OCL, sets of elements can be accessed and filtered according to the given conditions.

Having computed the sets of necessary objects, the invocation graph can be formulated in mathematical notation. The derived sets will also serve as a part of mathematical definitions of the data structures and functions containing conditions for selection and coverage of graph elements.

4.2.1.4 OCL Sets

For the following definitions, a single ruleset, i.e. one instance of the class *Ruleset*, is considered. The set of all rules contained by the given ruleset *allRules* can be gathered with the help of the operation *getRules* in the class *Ruleset*. Then, the set of all bigstep rules possessing an invocation graph is computed from the set *allRules*. These are objects of the class *BigstepRule*, which have at least one invocation, because only they produce an invocation structure. Invocations of a rule can be collected by accessing the association end *invocations* of the class *Rule*, see Figure 2.11.

Definition 4.1 *B* is a set of bigstep rules having invocation graphs in a ruleset: $B := allRules \rightarrow collect(r \mid r.oclIsTypeOf(BigstepRule) \ \wedge \ r.invocations.notEmpty())$.

The set of smallstep rules, which are objects of the class *SmallstepRule*, is computed from the set *allRules* as well.

Definition 4.2 *S* is a set of smallstep rules in a ruleset: $S := allRules \rightarrow collect(r \mid r.oclIsTypeOf(SmallstepRule))$.

Invocations determine the order in which nodes of the invocation graph are connected with each other. When several implementations of the invoked rule exist, a single invocation corresponds to the invocation of several concrete rules characterized by their unique names. So, an additional relation expressing invocations for them is needed. Such a relation in natural language is: a rule r_1 invokes another rule r_2, when there exists an invocation in the rule r_1, whose invoked rule's name is equal to the name of r_2 and the context node of r_2 is of type compatible to the target node of this invocation.

According to Figure 2.11, invocations are represented by the class *Invocation* that has an association *targetnode* pointing to a target node of an invocation. The class *Rule* has an association *contextnode* pointing to a context node. Target and context nodes are represented by the class *Node*, their types can be accessed through the attribute *type*.

Definition 4.3 *invokes(*r_1*,* r_2*)* is a relation between two rules, where r_1 invokes r_2: $invokes(r_1, r_2) \ :\Leftrightarrow \exists i \in r_1.invocations \ and \ i.invokedRule = r_2.name \ and \ r_2.contextNode.type.oclIsKindOf(i.targetNode.type)$. The domain of this relation is: $invokes(r_1, r_2) \subseteq rules \times S$.

In this section, auxiliary data sets needed for the definition of data structures for coverage analysis in DMM have been derived from the ruleset meta model. So, an invocation graph can be formally described in the next section.

4.2.1.5 Definition of Invocation Graph

Invocation graph corresponds to exactly one bigstep rule serving as a root and comprises all smallstep rules that can be derived from this bigstep rule using invocation mechanism. That means that for each smallstep rule, there exists an invocation path from the bigstep rule, i.e. the root of the graph, to this smallstep rule, i.e. a node in the graph. So, the set of nodes of a single invocation graph is a subset of the union B and S, see Definitions 4.1, 4.2. Edges of the invocation graph connect rule following each other directly during execution and are characterized by their names formed from the unique names of rules

Definition 4.4 *Invocation graph* for the i-th bigstep rule $bigstepRule_i$ from the set B is a graph $G^i_{invBR} = (V_i, E_i)$, *where the set of nodes* $V_i \subseteq B \cup S$, *the set of edges* $E_i \subseteq V_i \times V_i$, *and* $(v_1, v_2) \in E_i \;:\Leftrightarrow\; invokes(v_1, v_2)$.

Then, the actual behavior in the form of GTS is formalized as well. GTS is a graph with transitions containing labels of rules from the set $B \cup S$ which were used during the testing of a given example model, see Section 2.14.

Definition 4.5 *Graph transition system* is a graph: $GTS = (V_{GTS}, E_{GTS})$, *where* V_{GTS} *is a set of graph states,* $E_{GTS} \subseteq V_{GTS} \times V_{GTS}$ *is a set of graph transitions, and* $\forall e \in E \; \exists r \in rules : \; e.label = r.uniqueName$.

In this section, two graphs serving as a basis for the coverage analysis in DMM were formally defined. Further data structures derived from them and used for the coverage computation directly are presented in the next section.

4.2.2 Additional Data Structure

The next data structure necessary for the coverage computation is a set of elements to cover in order to achieve the coverage of 100% with respect to a certain coverage criterion. The coverage criterion is determined through the selection of a coverage item. To compute this structure, all invocation graphs corresponding to a given ruleset are traversed and a set of elements satisfying the definition of the coverage item is filtered.

The distinction in coverage item is expressed through the relation *equals* determining when two elements are considered to be equal. A structural element belong to $elements_{cover}$ if no other element equal to this one already exists in this set. This relation is defined individually by each coverage criterion.

Definition 4.6 Let N be the amount of invocation graphs built for a given ruleset, then $elements_{cover}$ for a certain coverage criterion are elements from the invocation graphs selected as follows:

1. $elements_{cover} := \oslash$;

```
2. $\forall e \in \cup_{i=1}^{N} G_{invBR}^{i}$ :
   $if\ (\forall e_2 \in elements_{cover} : \neg equals(e, e_2))$
   $elements_{cover} = elements_{cover} \cup e$
   $endif.$
```

According to the algorithm for coverage analysis in DMM, conditions when $elements_{cover}$ are considered to be covered during the semantics test execution are specified as well. Analogously to the function *equals*, each coverage criterion overwrites the function *covered* determining conditions for coverage with respect to GTS.

Definition 4.7 $elements_{covered}$ for a certain coverage criterion is a set of elements from $elements_{cover}$ covered by the given GTS and computed as:

```
1. $elements_{covered} := \emptyset$;
2. $\forall e \in elements_{cover}$ :
   $if\ (covered(e, GTS))$
   $elements_{covered} := elements_{covered} \cup e$
   $endif.$
```

Based on the sets derived above, the coverage value as a ratio of the size of the set $elements_{covered}$ to the size of the set $elements_{cover}$ is computed.

Definition 4.8 *Coverage value* is a ratio: $coverage = \frac{|elements_{covered}|}{|elements_{cover}|}$.

This computation finishes the coverage analysis and the computed value is output as the final result. In the following sections, new coverage criteria in DMM are presented and these sets are further refined for each coverage criterion individually.

4.3 Rule Coverage Criteria

In this section, coverage criteria concerning rules in a ruleset are illustrated. Common structures for this coverage family are distinguished by a structural element used, which is in this case nodes of the invocation graphs. The concrete rule coverage criteria are described in Sections 4.3.1, 4.3.2, and 4.3.3 according to the following pattern. Firstly, the main idea is explained. Secondly, based on the idea, the own formal specification of relations *equals* and *covered* is provided. Thirdly, the application of the criterion on the running example is illustrated. Summarizing the introduced points, the discussion about limitations of the criterion concludes the description.

4.3.1 Rule Coverage

In this section, characteristics discriminated the simplest coverage criterion, rule coverage, are described. This criterion serves as a minimal measure of quality for a DMM semantics specification.

4.3.1.1 Idea

Rule coverage is a coverage criterion demanding that each rule from the ruleset is exercised at least once during testing. From the GTRs' point of view, it ensures that at least one matching context of each rule is checked, since one execution path containing this rule is enough to cover it that implies testing of at least one matching context. Concerning invocation graphs, the set of elements to cover contains rules from the set of invocation graphs having distinct unique names. The rule's location within this set does not matter.

Recalling the set of invocation graphs for the running example illustrated in Figure 4.5, rules to test in order to achieve 100% of the rule coverage are highlighted in Figure 4.6. This is one possible set of elements to cover in the set of invocation graphs. Other nodes of the invocation graphs having the same names can be selected instead.

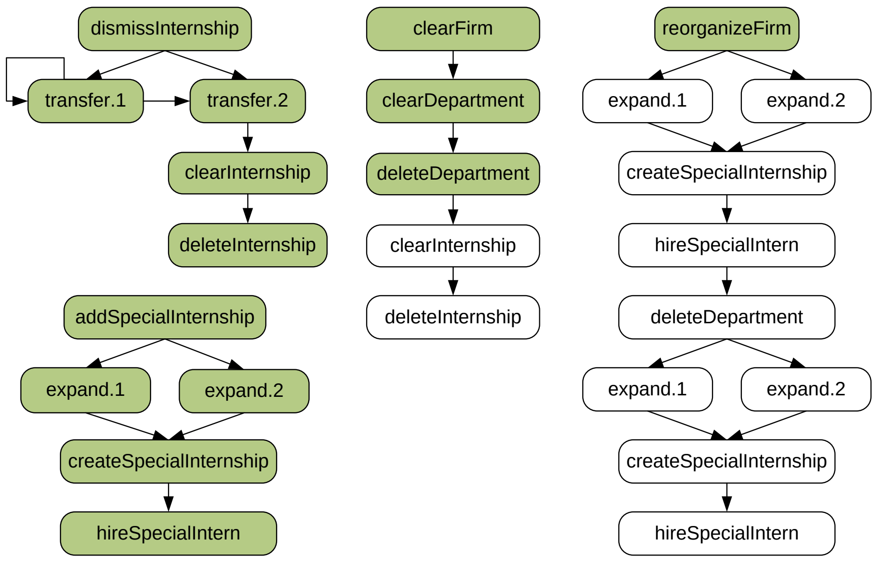


Figure 4.6: Elements to cover for rule coverage

A rule is covered by one of computed GTSs, if there exist a label coinciding with its unique name.

4.3.1.2 Formal Definition

The relation *equals* for the rule coverage considers two nodes as the same entity, when their unique names coincide throughout all the set of invocation graphs.

Definition 4.9 $equals_{RC}(v_1, v_2)$ is a relation defining the condition when two nodes are considered equal in the rule coverage : $equals_{RC}(v_1, v_2) := (v_1.uniqueName = v_2.uniqueName)$.

The relation *covered* for the rule coverage considers a rule covered by a given GTS, when there exists a transition in GTS, whose label and the unique name of the rule coincide.

Definition 4.10 $covered_{RC}(v, GTS)$ is a relation defining the condition, when the rule is covered by the given GTS in the rule coverage : $covered_{RC}(v, GTS) := (\exists t \in E_{GTS} \ \wedge \ t.label = v.uniqueName)$.

4.3.1.3 Example

The goal is to check the DMM specification for the running example performing TDSS with the rule coverage analysis. For this purpose test cases are created, in order to exploit some part of the specification, which also contains some errors. Errors are either the incorrect behavior that has been modeled or a matching that has failed. For the rule coverage, the 14 rules depicted in Figure 4.6 should be used each at least once to achieve the coverage of 100%.

After the execution of a first test case, parts of the semantics specification tested are shown in Figure 4.7. Green nodes are rules that should be tested for the rule coverage of 100%. Orange traces are execution paths through the semantics specification that have been checked. Red elements designated with red lightnings are faults in the semantics specification that have not been detected by testing so far. Green lightnings indicate errors that have been found during the testing.

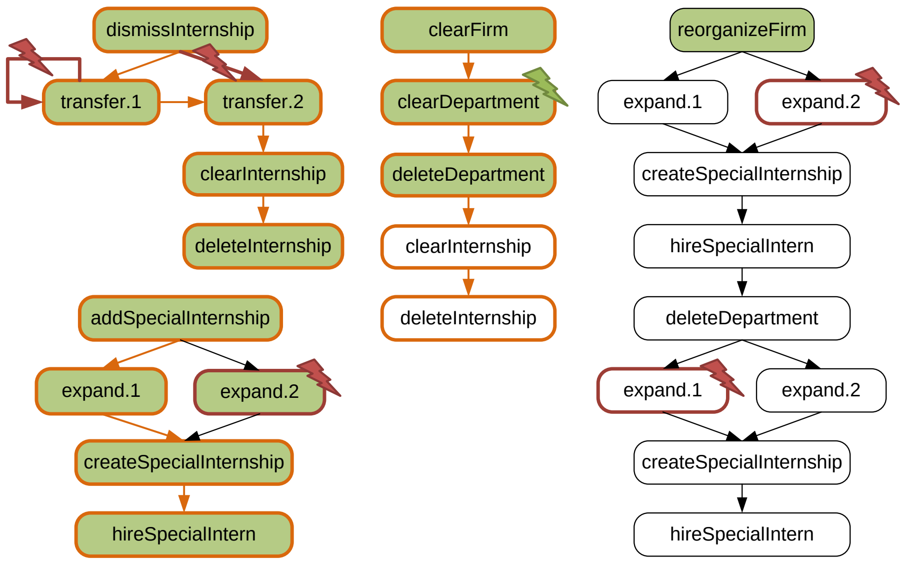


Figure 4.7: Rule coverage analysis with coverage < 100%

As it can be concluded from the picture above, the rule coverage of 100% was not reached since the bigstep rule *reorganizeFirm* and the smallstep rule *expand.2* were never used during the semantics test. An error in the smallstep rule *clearDepartment* has been revealed and successfully corrected. But a lot of errors still remain unrevealed, e.g. problems with rules *expand.1* and *expand.2* appearing in contexts, other than the tested ones.

The execution of the second test case yielded the coverage of 100% and is depicted in Figure 4.8. The rule *expand.2*, which was first marked in the invocation graph for the bigstep rule *addSpecialInternship*, was covered in another invocation graph, that is legitimate regarding to the rule coverage. However, despite no new errors have been disclosed, the maximum of coverage value is achieved.

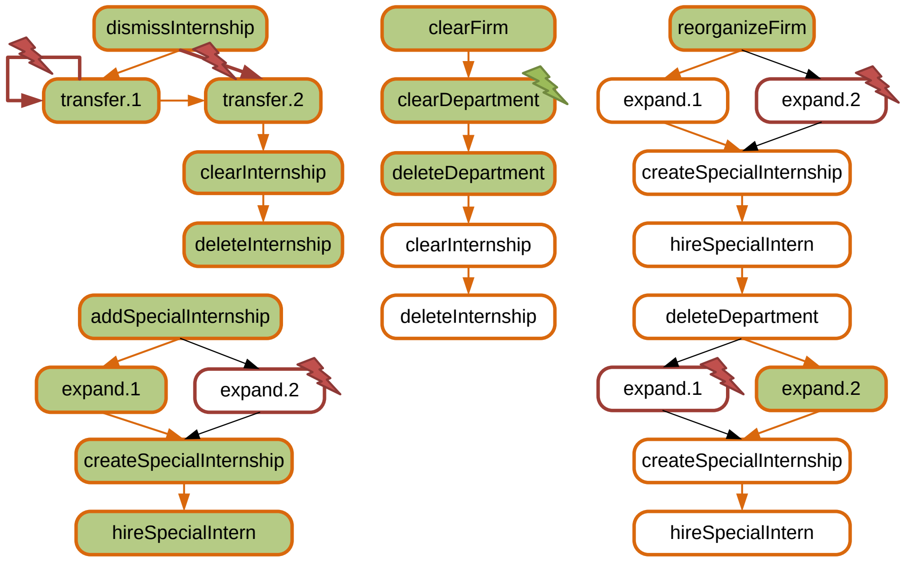


Figure 4.8: Rule coverage of 100%

4.3.1.4 Discussion

Demanding the check of all rules from the invocation graphs with different unique names helps to detect some error in the DMM semantics specification. However, since a single rule is only tested in one matching context, some matching contexts remain untested. So, this criterion is concluded to be weak, and a definition of more powerful rule coverage criteria that checks more matching contexts is desirable.

4.3.2 Rule Coverage Plus

In this section, properties of a more powerful criterion as the rule coverage, called rule coverage plus, are explained. It considers rules not only within the whole set of invocation graphs, but also within each invocation graph independently.

4.3.2.1 Idea

The idea of the rule coverage plus is to apply the rule coverage to each invocation graph separately. Thus, the rule coverage plus is a coverage criterion requiring that all rules with unique names in every invocation graph are used at least once during testing. It results in testing more matching contexts for the same rule, since at least one matching context for a rule is checked within each invocation graph where it appears. Rules to test in order to reach 100% of the rule coverage plus for the running example are illustrated in Figure 4.9. This is a possible selection of nodes within the invocation graphs, so each node within a single invocation graph can be replaced by another one with the same unique name in this set.

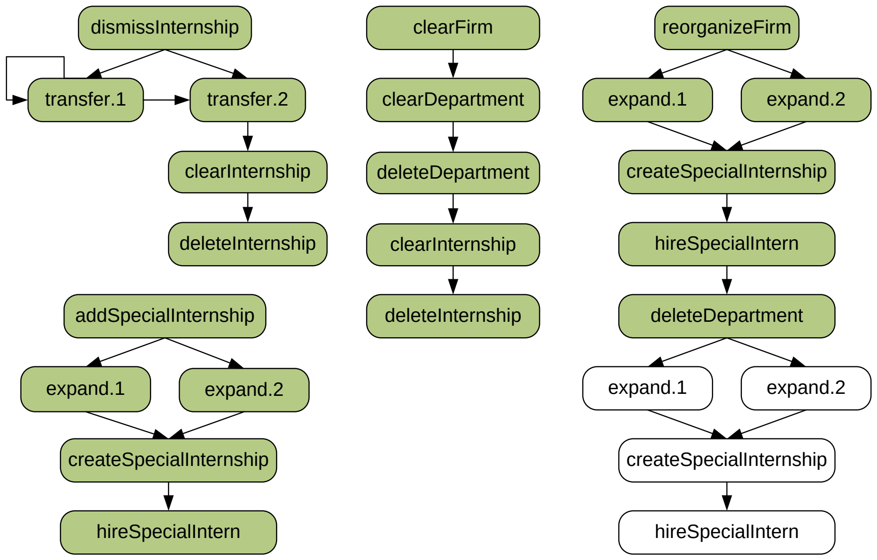


Figure 4.9: Elements to cover for rule coverage plus

The size of the set is larger than for the rule coverage and equals to 21. So, more elements need to be covered in order to reach the maximum coverage value. According to this coverage criterion, multiple rules with the same name within the same invocation graph are considered as a single element for covering by a GTS. A rule is covered by a GTS, if a label with its unique name appears after a label of the bigstep rule which this rule belongs to and no other bigstep rule intervene in between.

4.3.2.2 Formal Definition

The relation *equals* for the rule coverage plus considers two nodes as the same entity, when their unique names coincide within the same invocation graph. So, rules with the same unique name appearing in diverse invocation graphs are considered distinct.

Definition 4.11 $equals_{RCP}(v_1, v_2)$ is a relation defining the condition when two nodes are considered equal in the rule coverage plus : $equals_{RCP}(v_1, v_2) := (v_1, v_2 \in V_i \ \wedge \ v_1.uniqueName = v_2.uniqueName)$.

The relation *covered* for the rule coverage plus considers a rule covered by a given GTS, when there exists a transition in the GTS, which covers this rule according to Definition 4.10 of the rule coverage and belongs to the same bigstep rule as the invocation graph which this rule is a part of. So, it is not enough anymore to consider only rule's name; the rule's context regarding the bigstep rules should be taken into account as well.

The relation *follows(t_1, t_2)* expresses that a transition t_2 comes after the transition t_1 in a GTS and no bigstep rule intervenes between them. This relation is necessary for determining which bigstep rule an observed smallstep rule belongs to, since labels of smallstep rules invoked by a certain bigstep rule are situated between a label of their own bigstep rule and a label of the next bigstep rule. The relation *follows* can be formulated as: there should exist a path from t_1 to t_2 in GTS such, that no element of this path belongs to the set B of bigstep rules possessing invocation graphs.

Definition 4.12 *follows(t_1, t_2)* is a relation defining that the transition t_2 follows the transition t_1 in GTS without interruption by a bigstep rule: $follows(t_1, t_2) := (\exists t_1 ... t_i ... t_2 \in E_{GTS} \ \wedge \ \nexists t_j \in t_1 ... t_j ... t_2 \ \wedge \ t_j \in B)$.

Now, the relation *covered* can be formulated. The relation *follows* used is presented in Definition 4.12.

Definition 4.13 $covered_{RCP}(v, GTS)$ is a relation defining the condition when the rule is covered by the given GTS in the rule coverage plus: $covered_{RCP}(v, GTS)$ $:= (\exists t_1, t_2 \in E_{GTS} \ \wedge \ v \in V_i \ \wedge \ t_1.label = bigstepRule_i.uniqueName \ \wedge \ follows(t_1, t_2) \ \wedge \ t_2.label = v.uniqueName)$.

4.3.2.3 Example

The goal is to perform the rule coverage plus analysis on the DMM specification for the running example. The set of elements to cover for the rule coverage plus is depicted in Figure 4.9. The rule coverage of 100% is reached in a way depicted in Figure 4.8. A projection of this view onto the rule coverage plus perspective is illustrated in Figure 4.10.

The rule coverage plus is less than 100%, as the smallstep rule *expand.2* in the invocation graph for the bigstep rule *addSpecialInternship* has not been tested. So, new test cases are required in order to achieve the full rule coverage plus. The execution of the new test case delivered the coverage value of 100% and is illustrated in Figure 4.11.

As a result of the coverage analysis, a new error has been found in the DMM semantics specification: the rule *expand.2* in the invocation graph for the bigstep rule *addSpecialInternship* has been corrected by the developer. However, two faulty rules *expand.1* and *expand.2* in the invocation graph for the bigstep rule *reorganizeFirm* are still not tested, since the maximum coverage is achieved by exercising an execution path that does not involve them.

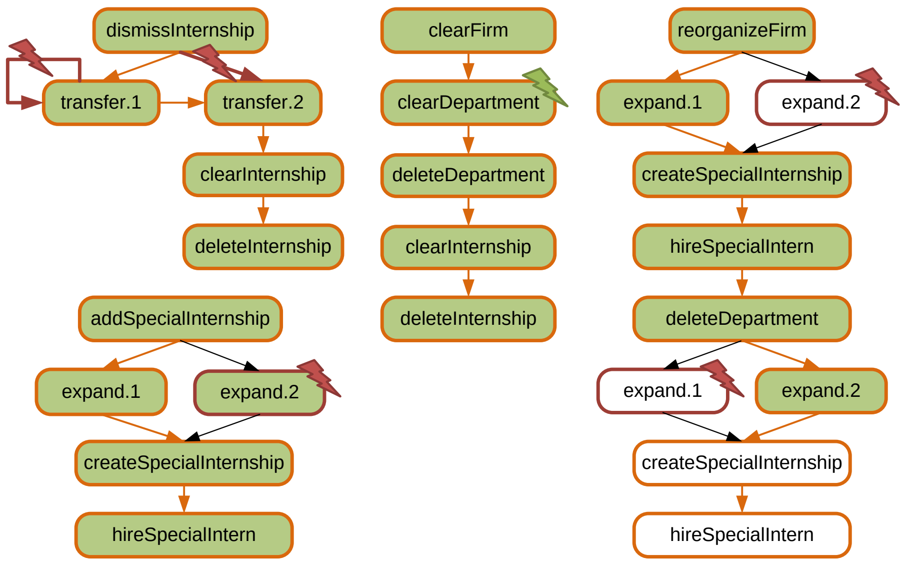


Figure 4.10: Projection of 100% rule coverage onto rule coverage plus

4.3.2.4 Discussion

The rule coverage plus ensures that each rule applies correctly at least in one matching context in each invocation graph where it appears. Thereby the rule coverage plus tests more contexts and assists in finding more failures in the DMM semantics specification, as the previous rule coverage. With the help of the rule coverage plus, the rule coverage is guaranteed for each invocation graph providing higher expressiveness of this criterion.

However, important matching contexts still remain untested and some incorrect behavior was not detected as well. So, this criterion is concluded to be more powerful as the rule coverage, but a definition of a more powerful rule coverage criterion would be helpful.

4.3.3 Rule Coverage Plus Plus

In this section, the last rule coverage criterion, called rule coverage plus plus, is elaborated. It ensures that all nodes from the set of invocation graphs are covered during testing.

4.3.3.1 Idea

The main idea of the rule coverage plus plus is to cover all nodes of the invocation graphs. Nodes with the same unique name within one invocation graph are considered being distinct for this coverage criterion. The rule coverage plus plus demands to check those execution paths through the invocation graphs, which include each node at least once. Since the merging of identical subgraphs during the invocation graph's construction assembles the equal matching contexts in one

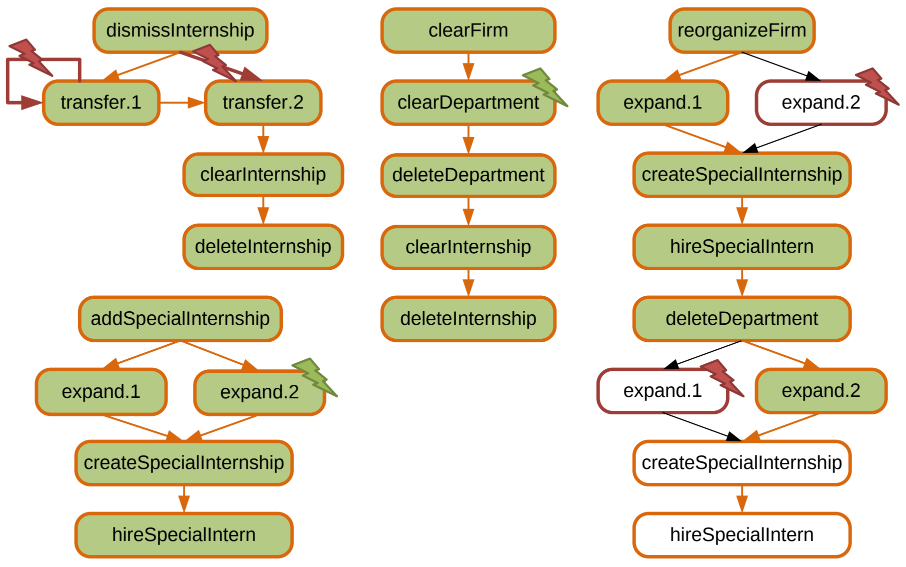


Figure 4.11: Rule coverage plus of 100%

and leaves different ones untouched, more distinct execution paths are demanded comparing with the previous rule coverage plus.

The illustration of parts of the DMM semantics specification to cover in order to achieve 100% of the rule coverage plus plus for the running example is presented in Figure 4.12.

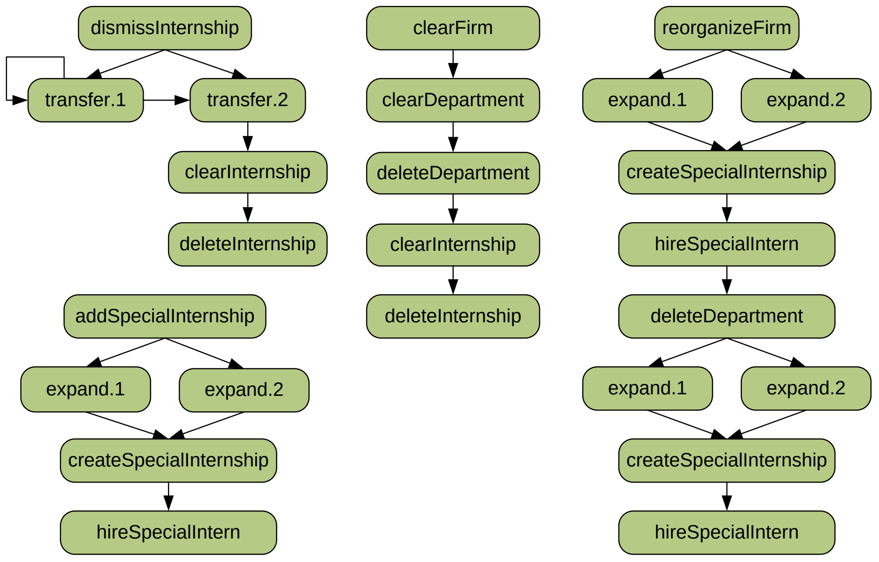


Figure 4.12: Elements to cover for rule coverage plus plus

Here, rules with the same name have to be tested several times but in different contexts, e.g. rule *expand.1* and *expand.2* once after the different bigstep rules *addSpecialInternship* and *reorganizeFirm*, and once within the same bigstep rule but after the different rules *reorganizeFirm* and *deleteDepartment*. Even when a rule follows the same one at different execution points, e.g. in the invocation graph for the bigstep rule *reorganizeFirm*, the smallstep rule *createSpecialInternship* always goes after the same two rules, the matching contexts at these points still can differ, since the graph states resulting from application of the previous rules may be different. So even in such a case, more parts of the semantics specification than by the previous rule coverage criteria are tested.

For covering, to be able to distinguish nodes which rules with the same unique name in the GTS represent, the context in which a rule appears should be taken into account as well. For example, in the invocation graph for the bigstep rule *reorganizeFirm*, when a rule *expand.1* appears, one needs to know what exact node to mark as covered: the one following the rule *reorganizeFirm* or the one following the rule *deleteDepartment*. Looking at the sets of previous rules, it is possible to determine the difference. For the cases like *createSpecialInternship*, when the previous rules for both instances are identical, both nodes are marked as covered.

4.3.3.2 Formal Definition

The relation *equals* for the rule coverage plus plus considers two nodes equal, when they are the same entity.

Definition 4.14 $equals_{RCPP}(v_1, v_2)$ is a relation defining the condition when two nodes are considered equal in the rule coverage plus plus : $equals_{RCPP}(v_1, v_2) := (v_1 = v_2)$.

The relation *covered* for the rule coverage plus plus considers a rule covered by a given GTS, when there exists a transition in the GTS, which covers this rule according to Definition 4.13 of the rule coverage plus and the set of incoming labels for the rule's label in GTS is a subset of the set of incoming edges of the rule. The set of incoming rules for a rule r in an invocation graph is a set of source nodes of edges which have this rule r as a target node.

Definition 4.15 *incomingRules* is a set of rules directly preceding the rule r_c in the invocation graph: $incomingRules(r_c) \subset V_i$, where $incomingRules(r_c) := V_i \rightarrow collect(r \mid \exists e \in E_i \ \wedge \ e.source = r \ \wedge \ e.target = r_c)$.

Incoming labels are labels of transitions, which directly precede a given transition in the GTS. The set of incoming transitions for a transition t in GTS is a set of transitions which end in the same state as the transition t starts.

Definition 4.16 *incomingLabels* is a set of labels of transitions directly preceding the transition t_c in the given GTS: $incomingLabels(t_c) \subset E_{GTS}$, where $incomingLabels(t_c) := E_{GTS} \rightarrow collect(t \mid t.target = t_c.source)$.

Now, the formal definition for the relation *covered* can be formulated.

Definition 4.17 $covered_{RCPP}(v, GTS)$ is a relation defining the condition when the rule is covered by the given GTS in the rule coverage plus plus : $covered_{RCPP}(v, GTS)$ $:= (v \in V_i \wedge \exists t_1, t_2 \in E_{GTS} \wedge t_1.label = bigstepRule_i.uniqueName \wedge follows(t_1, t_2) \wedge t_2.label = v.uniqueName \wedge incomingRules(v) = incomingLabels(t_2))$.

4.3.3.3 Example

The set of elements to cover for the rule coverage plus plus is depicted in Figure 4.12. The 100% of rule coverage plus is also achieved in a way depicted in Figure 4.11. The projection of the 100% rule coverage plus onto the elements to cover for the rule coverage plus plus is presented in Figure 4.13.

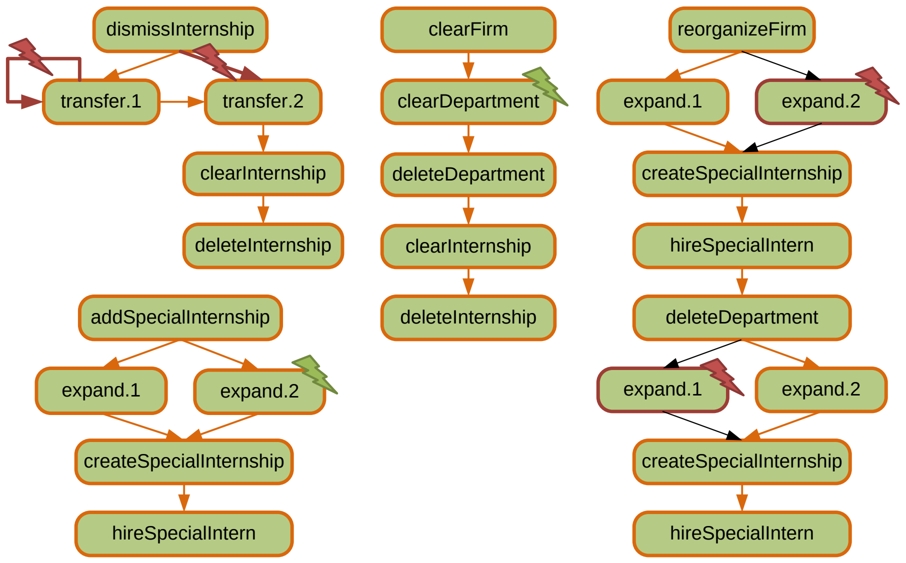


Figure 4.13: Projection of 100% rule coverage plus onto rule coverage plus plus

As shown in the graphic, the rule coverage plus plus is less than 100%, since the smallstep rules *expand.1* and *expand.2* in the invocation graph for the bigstep rule *reorganizeFirm* has not been covered. So, new test cases are created, and the full rule coverage plus plus for the running example is achieved and illustrated in Figure 4.14.

During the coverage analysis, both faulty rules *expand.1* and *expand.2* in the invocation graph for the bigstep rule *reorganizeFirm* have been detected in the DMM semantics specification. However, some incorrect behavior still remained. The application of the rule *transfer.2* immediately after the bigstep rule *dismiss-Internship* and the recursive application of the rule *transfer.1* do not deliver right results. The reason is that the rule coverage plus plus demands the coverage of execution paths,, that

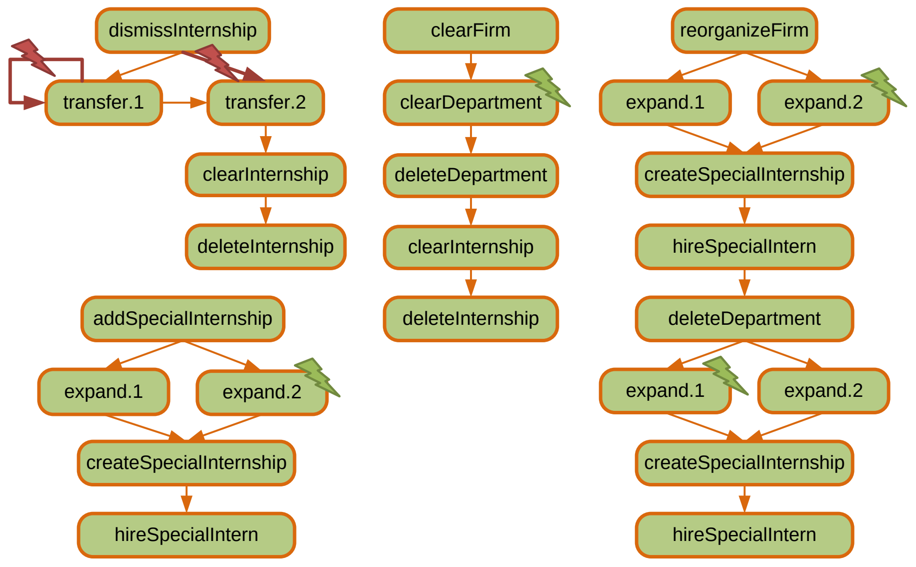


Figure 4.14: Rule coverage plus plus of 100%

4.3.3.4 Discussion

The rule coverage plus plus ensures the coverage of execution paths, which use all nodes in the invocation graphs at least once. Thereby it assists in finding more failures in the DMM semantics specification, as the previous rule coverage criteria, since all rules are tested in more matching contexts than by the rule coverage plus. Thus the rule coverage plus plus is considered as a more powerful rule coverage criterion.

However, some errors in the DMM semantics specification still remain unrevealed. The reason is that the rule coverage plus plus leaves out the execution paths formed from the already tested nodes in another way, i.e. cases when already tested rules follow each other in different combinations during the execution. Such incorrect behavior can be detected, when connections among rules would be taken into account as well. For this purpose edge coverage criteria will be introduced in the next section.

4.4 Edge Coverage Criteria

In this section, the edge coverage criteria concerning connections among rules in a ruleset are illustrated. Common structures for this coverage family are defined on the edges of invocation graphs. Three concrete edge coverage criteria are described in Sections 4.4.1, 4.4.2, and 4.4.3 with regard to the pattern introduced for the rule coverage criteria in Section 4.3.

4.4.1 Edge Coverage

In this section, the simplest edge coverage criterion, called edge coverage, is described. This criterion guarantees that every possible pair represented by edges in the set of invocation graphs is exercised during testing.

4.4.1.1 Idea

Edge coverage is a coverage criterion demanding that all distinct edges from the invocation graphs for a given ruleset are exercised at least once during testing. Each edge represents a pair of rules, following one another in an invocation graph and occurring in a certain matching context. Similar to rules distinguished by their unique names unequivocally, an edge is characterized unambiguously by its name. This name is built from unique names of rules, connected by this edge, and has a form: $unique\_name\_of\_the\_first\_rule \rightarrow unique\_name\_of\_the\_second\_rule$. Examples of edge labels for the running example are: $dismissInternship \rightarrow transfer.1$, $transfer.1 \rightarrow transfer.1$, etc. So, on the basis of invocation graphs, the edge coverage ensures that all edges having different names are used at least once during semantics test execution, i.e. they are a part of at least one execution path generated during the test execution.

In order to illustrate the edge coverage criteria more conveniently, the semantics of the running example will be modified, resulting in a larger amount of similar edges within the invocation graphs. For this modification, the bigstep rule *addSpecialInternship* is dismissed, so is the corresponding invocation graph, and replaced with a new bigstep rule named *reorganizeFirm2*. The behavior of this rule is similar to the bigstep rule *reorganizeFirm*. The only difference is that *reorganizeFirm2* has four invocations ordered by sequence numbers: *deleteDepartment* (1), *expand* (2), *clearDepartment* (3), and *expand* (4).

The modified set of invocation graphs for the running example with highlighted edges to test for achieving 100% of the edge coverage is depicted in Figure 4.15.

As for the rule coverage, this set is only a possible collection of elements corresponding to the names of 19 edges required for testing in this case. Edges with the same name no matter in which invocation graph are considered as the same entity for the coverage analysis.

4.4.1.2 Formal Definition

The relation *equals* for the edge coverage considers two edges to be equal, when their names coincide throughout the set of invocation graphs. Edges with the same name appearing in diverse invocation graphs are considered equal.

Definition 4.18 $equals_{EC}(e_1, e_2)$ is a relation defining the condition when two edges are considered equal in the edge coverage : $equals_{EC}(e_1, e_2) := (e_1.name = e_2.name)$.

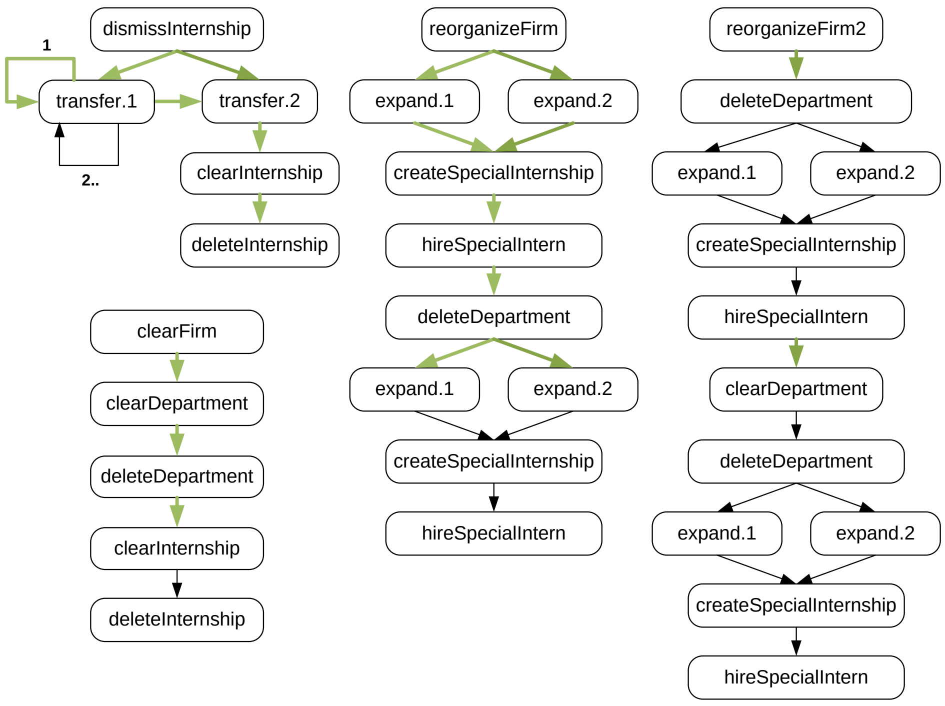


Figure 4.15: Elements to cover for edge coverage

The relation *covered* for the edge coverage considers an edge covered by a given GTS, when there exist two transitions in GTS, so that they directly follow each other, their labels coincide with the names of the rules forming the edge, and their order agrees with the direction of the edge (first source rule, than target rule).

Two transitions in a GTS follow each other directly, when they share a state, i.e. a source state of one transition is a target state of another one. Moreover, the label of one transition has to be the same as the edge's source rule, and the label of another - as the edge's target rule. For the edge to be covered by these transitions, their order should preserve the order in which rules appear in the edge, i.e. the transition corresponding to the source rule should come before the transition corresponding to the target rule of the edge.

Definition 4.19 $covered_{EC}(e, GTS)$ is a relation defining the condition when an edge is covered by a given GTS in the edge coverage : $covered_{EC}(e, GTS) := (\exists\, t_1, t_2 \in E_{GTS} \wedge t_1.target = t_2.source \wedge t_1.label = e.source.uniqueName \wedge t_2.label = e.target.uniqueName.$

In order to mark an edge as covered, its name should be derived at least once from the computed GTSs.

4.4.1.3 Example

The idea of coverage analysis remains the same as for rule coverage criteria. After the execution of the first test case, parts of the semantics specification tested are shown in Figure 4.16. The coloring used is similar to that for the rule coverage analysis.

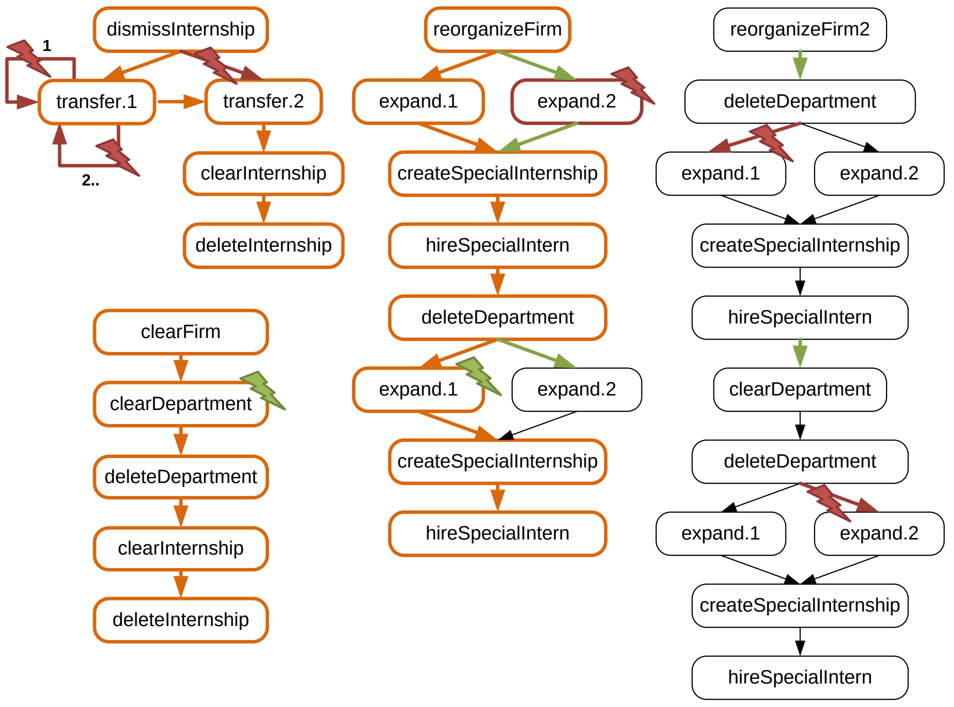


Figure 4.16: Edge coverage analysis with coverage < 100%

According to the graphic, the edge coverage of 100% was not reached so far, since some green edges were never used during the semantics test. Faulty edges $dismissInternship \rightarrow transfer.1$ and $transfer.1 \rightarrow transfer.1$ should also be tested for the maximum of the coverage value. So, the coverage analysis continues.

Errors in the smallstep rules *clearDepartment* and *expand.1* were revealed during the exercising of edges $clearFirm \rightarrow clearDepartment$ and $deleteDepartment \rightarrow expand.1$. The erroneous edges $transfer.1 \rightarrow transfer.1$ are situated within the recursion in the 1-st and later recursive loops. Errors in edges $deleteDepartment \rightarrow expand.1$ and $deleteDepartment \rightarrow expand.2$ in the invocation graph for the bigstep rule *reorganizeFirm2* signify some matching errors in the context of exact this rule.

The execution of a second test case yielded the coverage of 100% and is depicted in Figure 4.17. Errors in the bigstep rules *dismissInternship* and *reorganizeFirm* were found. The maximum of coverage value is achieved, despite the error in the recursive call still remains, since it occurs after the first loop through the recursion that was only tested. Errors in the bigstep rule *reorganizeFirm2* have not been disclosed as well.

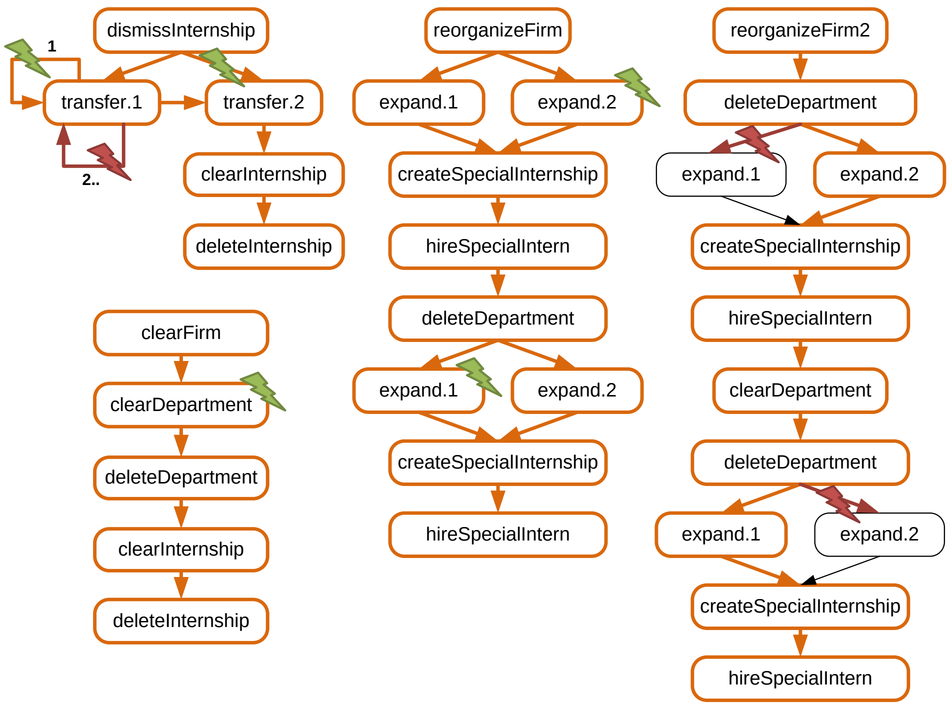


Figure 4.17: Edge coverage analysis with coverage 100%

4.4.1.4 Discussion

Demanding the use of all edges from the set of invocation graphs with distinct names involves testing of matching contexts, which were not exercised by the rule coverage criteria, and helps to detect new errors in the DMM semantics specification. These matching contexts concern different possible order sequences of the same nodes of the invocation graphs.

Considering nodes having multiple incoming edges in the invocation graph, in order to fulfill the requirements of the rule coverage criteria to use each rule at least once, only one edge from those can be chosen each time during testing. In contrast, the edge coverage considers such multiple edges as distinct objects and forces to test all of them increasing the number of tested matching contexts.

However, a single pair of rules represented by an edge is only tested in one matching context leaving the application of this pair on another possible graph states unchecked. Thus, this criterion is concluded to be the weakest among the edge coverage ones, and a definition of more powerful edge coverage criteria is desirable.

4.4.2 Edge Coverage Plus

In this section, properties of a more powerful criterion than the edge coverage, called edge coverage plus, are explained. It ensures that all edges with distinct names in a single invocation graph are exercised during testing.

4.4.2.1 Idea

The idea of the edge coverage plus is to apply the edge coverage to each invocation graph independently, thus requiring all distinct edges in every invocation graph to be exercised at least once during testing. So, the edge coverage plus regards edges having the same name but located in different invocation graphs as different elements to cover. Each of these edges should be at least once a part of execution paths generated during the semantics test and belonging to the same bigstep rule as the invocation graph of this edge. Edges to test in order to reach 100% of the edge coverage plus for the running example are illustrated in Figure 4.18.

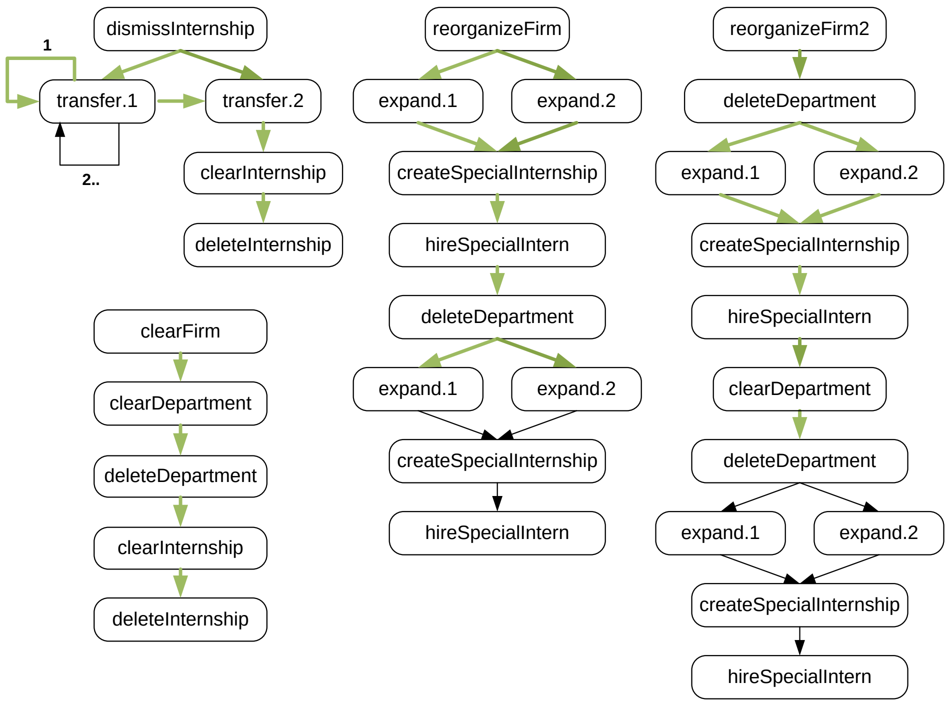


Figure 4.18: Elements to cover for edge coverage plus

For the edge coverage plus, the size of the set is larger than for the edge coverage and equals to 26. So, more pairs of rules need to be tested in order to reach the maximum of coverage value. The additional testing is motivated by the reasoning, that even if a pair of rules works correctly in one matching context, it does not exclude the possibility of erroneous behavior in another matching context, since graph states, on which the rules apply, may vary according to the logic of the rules applied before.

4.4.2.2 Formal Definition

The relation *equals* for the edge coverage plus considers two edges as the same entity, when their names coincide within the same invocation graph. So, edges with the same name appearing in diverse invocation graphs are considered distinct.

Definition 4.20 $equals_{ECP}(e_1, e_2)$ is a relation defining the condition when two edges are considered equal in the edge coverage plus : $equals_{ECP}(e_1, e_2) := (e_1.name = e_2.name \ \wedge \ e_1, e_2 \in E_i)$.

The relation *covered* for the edge coverage plus considers an edge covered by a given GTS, when there exist two transitions in the GTS, which cover this edge according to Definition 4.19 of the edge coverage and belong to the same bigstep rule as the invocation graph which this edge is a part of. So, the transitions covering the edge should lie after the corresponding bigstep rule but before the next following one that is expressed by the relation $follows(t_i, t_j)$ in Definition 4.12. Thereby, the context of a bigstep rule is taken into account.

Definition 4.21 $covered_{ECP}(e, GTS)$ is a relation defining the condition when an edge is covered by a given GTS in edge coverage plus : $covered_{ECP}(e, GTS) := (\exists t_1, t_2, \ t_b \in E_{GTS} \ \wedge \ t_1.target = t_2.source \ \wedge \ t_1.label = e.source.uniqueName \ \wedge \ t_2.label = e.target.uniqueName \wedge e \in E_i \wedge t_b.label = bigstepRule_i \wedge follows(t_b, t_1))$.

4.4.2.3 Example

The edge coverage of 100% is reached in a way depicted in Figure 4.17. The projection of this view onto the edge coverage plus perspective is illustrated in Figure 4.19.

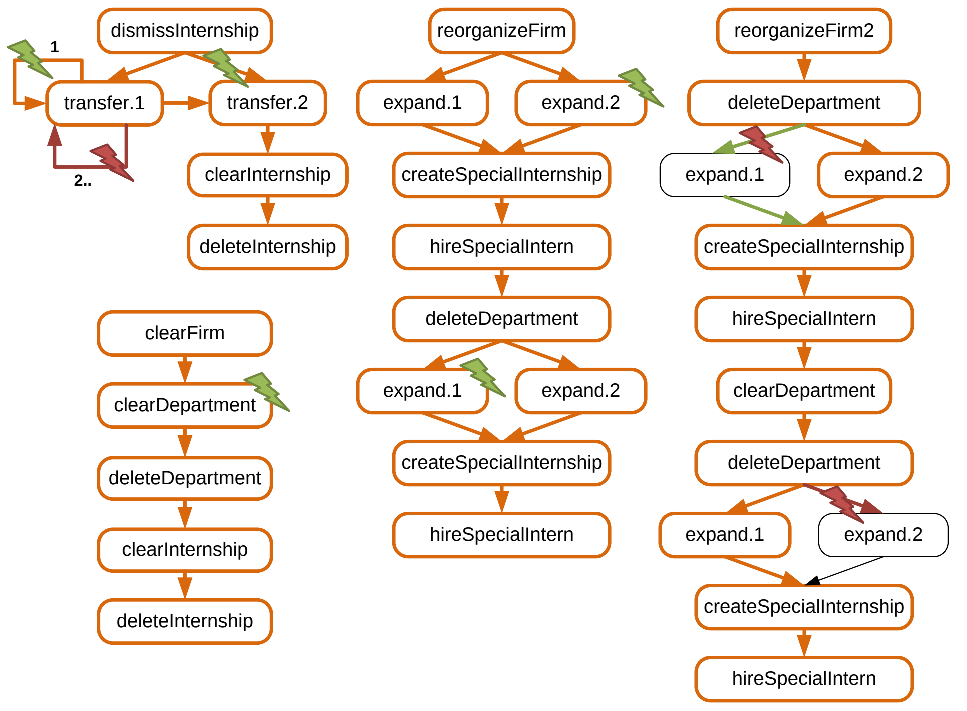


Figure 4.19: Projection of 100% edge coverage onto edge coverage plus

The edge coverage plus is less than 100%, since some edges in the invocation graph for the bigstep rule *reorganizeFirm2* marked green still need to be covered. So, a new example model is required in order to achieve the complete edge coverage plus. The delivered coverage value of 100% and is illustrated in Figure 4.20.

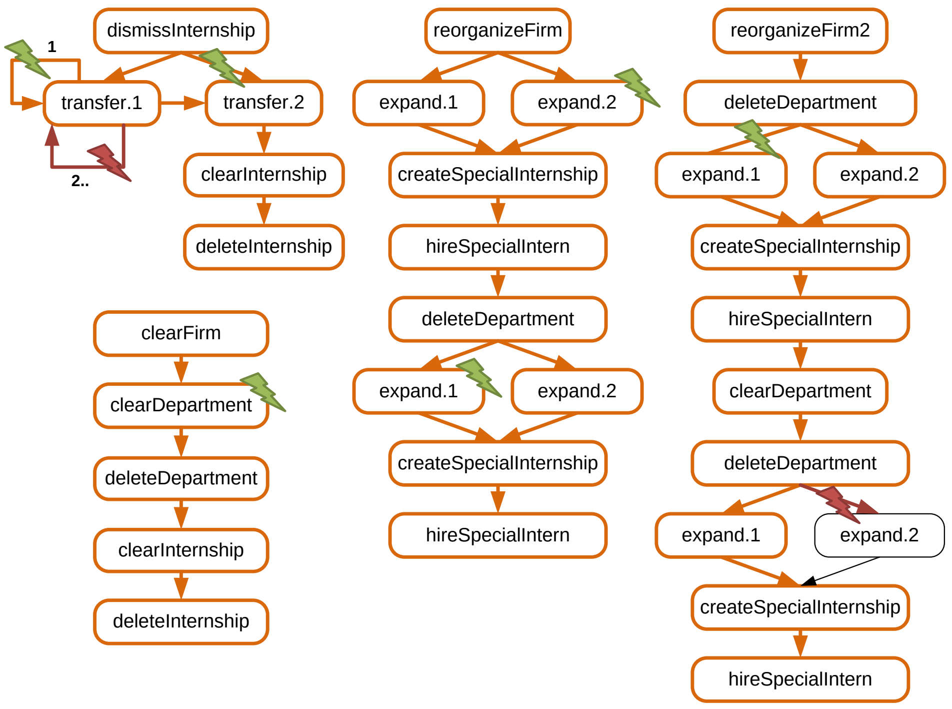


Figure 4.20: Edge coverage plus of 100%

The new test case was designed in such a way, that it generates an execution path through the branch of the first *expand.1* in the invocation graph for the bigstep rule *reorganizeFirm2*, revealing another error in the semantics specification. However, one incorrect rule application between rules *deleteDepartment* and *expand.2* in the invocation graph for the bigstep rule *reorganizeFirm2* is still not discovered, since the maximum coverage is achieved by exercising an execution path that does not involve them.

4.4.2.4 Discussion

The edge coverage plus ensures that each pair of rules represented by edges applies correctly at least in one matching context in each invocation graph. Thereby the edge coverage plus exercises more matching contexts and assists in finding more failures in DMM semantics specifications than the previous edge coverage.

However, some incorrect behavior still was not detected, since not all of the execution paths were involved by the matching contexts used during testing. Thus, this criterion is concluded to be more powerful as the edge coverage, but a definition of a more powerful edge coverage criterion would still be helpful.

4.4.3 Edge Coverage Plus Plus

In this section, the most expressive edge coverage criterion, called edge coverage plus plus, is elaborated. It requires all edges from the set of invocation graphs be covered during testing.

4.4.3.1 Idea

The main idea of the edge coverage plus plus is to test all pairs of rules appearing in the form of edges in the set of invocation graphs. All the collected pairs show matching contexts for a rule with respect to rules directly preceding it. Covering all edges in the set of invocation graphs checks the same pair of rules on different state graphs determined by the application of preceding bigstep and smallstep rules.

The illustration of parts of the DMM semantics specification to cover in order to achieve 100% of the edge coverage plus plus for the running example is presented in Figure 4.21.

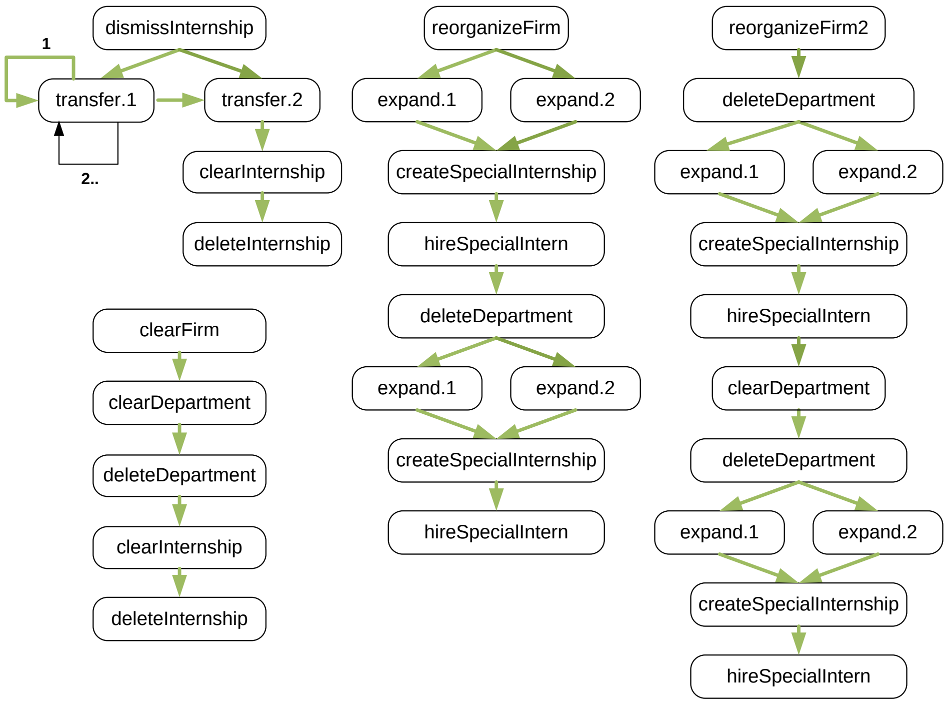


Figure 4.21: Elements to cover for edge coverage plus plus

Here, all edges of the invocation graphs should be exercised during the semantics test. The equally named edges within one invocation graph have to be tested several times, e.g. edge *deleteDepartment* → *expand*.1 once after the bigstep rules *reorganizeFirm2* and once after the smallstep rule *deleteDepartment*. Even when an edge follows the same rule at different execution points, e.g. in the invocation graph for the bigstep rule *reorganizeFirm2* the edge *expand*.1 →

createSpecialInternship always goes after the same rule *deleteDepartment*, the graph states at these points still can differ. So even in such a case, more parts of the semantics specification than by the previous edge coverage criteria are tested.

In order to be able to distinguish edges with the same name during the GTS coverage, the context in which an edge appears should be taken into account as well. For example, in the invocation graph for the bigstep rule *reorganizeFirm*, when the edge $expand.1 \rightarrow createSpecialInternship$ appears, one needs to know what exact edge to mark as covered: the one following the rule *reorganizeFirm* or the one following the rule *deleteDepartment*. Looking at the sets of previous edges, it is possible to determine the difference. For the cases like $expand.1 \rightarrow createSpecialInternship$ in the invocation graph for *reorganizeFirm2*, when the previous edges for both instances are identical, both nodes are marked as covered.

4.4.3.2 Formal Definition

The relation *equals* for the edge coverage plus plus considers two edges equal, when they are the same entity. Different edges with the same name within one invocation graph are considered being distinct for this coverage criterion.

Definition 4.22 $equals_{ECPP}(e_1, e_2)$ is a relation defining the condition when two edges are considered equal in the edge coverage plus plus : $equals_{ECPP}(e_1, e_2) := (e_1 = e_2)$.

The relation *covered* for the edge coverage plus plus considers an edge covered by a given GTS, when there exists two transition in the GTS, which cover this edge according to Definition 4.21 of the edge coverage plus and the set of labels of transitions forming the incoming edges for the considered edge in the GTS is a subset of the set of incoming edges for this edge in the invocation graph.

The set of incoming edges for an edges in an invocation graph is a set of edges which have the source node of this edge as their target node.

Definition 4.23 *incomingEdgesIG* is a set of edges preceding the edges e_c in an invocation graph: $incomingEdgesIG(e_c) \subset E_i$, where $incomingEdgesIG(e_c) := E_i \rightarrow collect(e \mid e.target = e_c.source)$.

Incoming edges in a GTS are labels of transitions, which precede a transition, representing the source node of a given edge, and their pairs form edges from the set *incomingEdgesIG* for this edge. A set of incoming transitions for a transition t in GTS is a set of transitions which end in the same state as the transition t starts.

Definition 4.24 *incomingEdgesGTS* is a set of labels of transitions forming edges which preceding a given edge e_c in a GTS: $incomingEdgesGTS(e_c) \subset GTS$, where $incomingEdgesGTS(e_c) := \{t_1.label \rightarrow t_2.label \mid t_1, t_2, t_3, t_4 \in E_{GTS} \ \wedge \ e_c = t3.label \rightarrow t4.label \ \wedge \ t_1.target = t_2.source \ \wedge \ t_2.target = t_3.source \ \}$.

Now, the formal definition for the relation *covered* can be formulated. The relation *follows* used is presented in Definition 4.12.

Definition 4.25 $covered_{ECPP}(e, GTS)$ is a relation defining the condition, when the edge is covered by the given GTS in the edge coverage plus plus : $covered_{ECPP}$ $(e, GTS) := (\exists t_1,\ t_2,\ t_b \in E_{GTS}\ \wedge\ t_1.target = t_2.source\ \wedge\ t_1.label = e.source.uniqueName\ \wedge\ t_2.label = e.target.uniqueName\ \wedge\ e \in E_i\ \wedge\ t_b.label = bigstepRule_i \wedge follows(t_b, t_1) \wedge incomingEdgesIG(e) = incomingEdgesGTS(e))$.

4.4.3.3 Example

The edge coverage plus of 100% is achieved in a way depicted in Figure 4.17. The projection of the 100% edge coverage plus onto the elements to cover for edge coverage plus plus is presented in Figure 4.22.

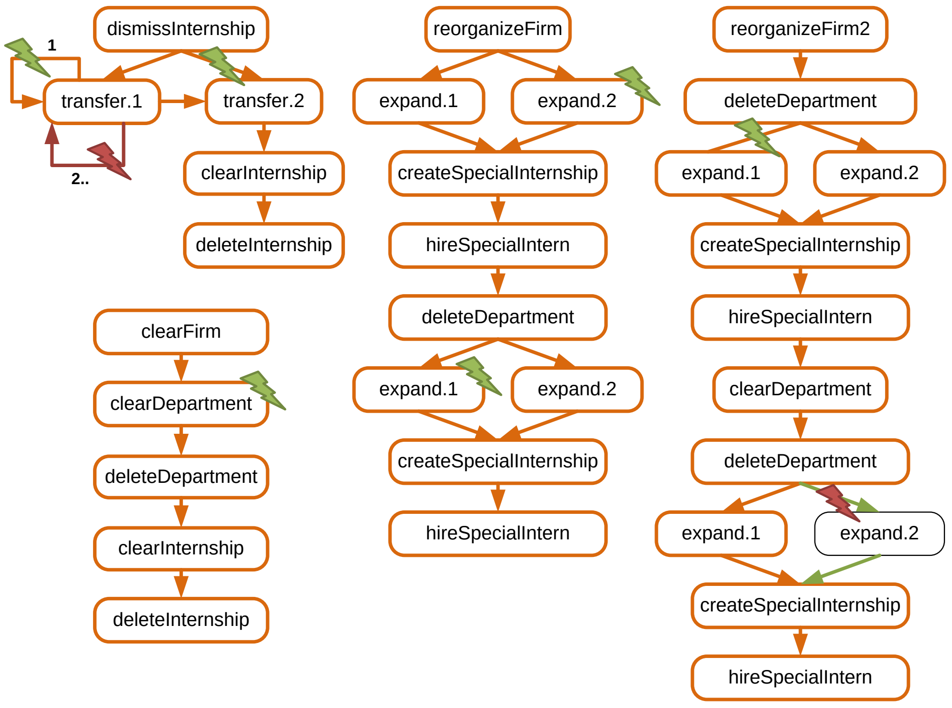


Figure 4.22: Projection of 100% edge coverage plus onto edge coverage plus plus

As shown in the graphic, the edge coverage plus plus is less than 100%, since some edges in the invocation graph for the bigstep rule *reorganizeFirm2* marked green still need to be covered. So, a new example model is created in order to achieve maximum of the edge coverage plus plus, which is illustrated in Figure 4.23.

During the coverage analysis, both faulty matching contexts *deleteDepartment* $\rightarrow$ *expand.2* in the invocation graph for the bigstep rule *reorganizeFirm2* has been detected in the DMM semantics specification. However, some incorrect behavior

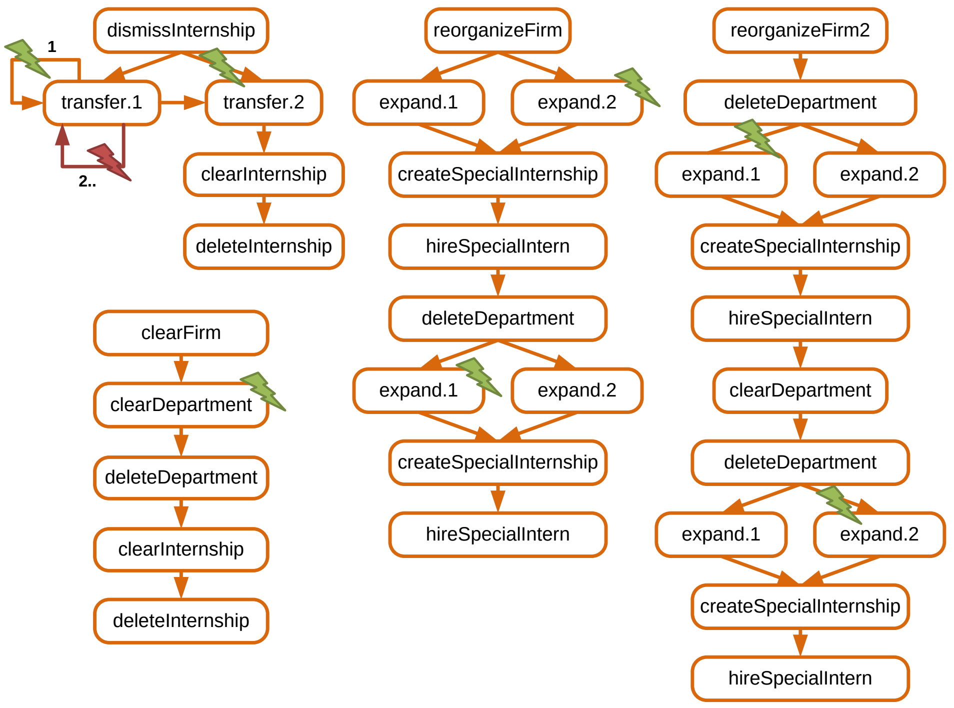


Figure 4.23: Edge coverage plus plus of 100%

still remains. The recursive application of the rule *transfer.1* does not deliver correct results after a certain recursive loop. This error will not be found, since the maximal edge coverage plus plus involves the execution of all edges once, that leaves out the execution of paths containing repetitions of the same edges.

4.4.3.4 Discussion

Edge coverage plus plus ensures that all edges from the invocation graphs are tested. Thereby it tests more occurrences of rule pairs than edge coverage plus and so assists in finding more failures in DMM semantics specifications, as the previous edge coverage criteria. Thus, edge coverage plus plus is considered as the most powerful criterion from this coverage family.

However, some errors in the DMM semantics specification still remain unrevealed. They concern special execution paths, when already tested edges could follow each other in a recursive fashion during the execution. Such incorrect behavior can be detected, when more execution paths would be taken into account. For this purpose all-paths coverage criterion will be introduced in the next section.

4.5 All-Paths Coverage

In this section a coverage criterion concerning executional paths, which can be derived from a given ruleset, is explained. This criterion is considered to be the most expressive one among all presented above. It is described according to the pattern used for the rule and edge coverage criteria.

4.5.1 Idea

All-paths coverage is a coverage criterion demanding that all distinct paths in the invocation graphs for a given ruleset are exercised at least once during testing. All paths are all possible routes through the invocation graph from its root to all its leaves. Similar to other elements distinguished by their names unequivocally, a path is characterized unambiguously by a list of labels of rules composing it.

In order to visually illustrate this criterion, the set of invocation graphs used for edge coverage analysis is taken. All paths to test for achieving 100% of all-paths coverage are formed from the edges highlighted in Figure 4.24.

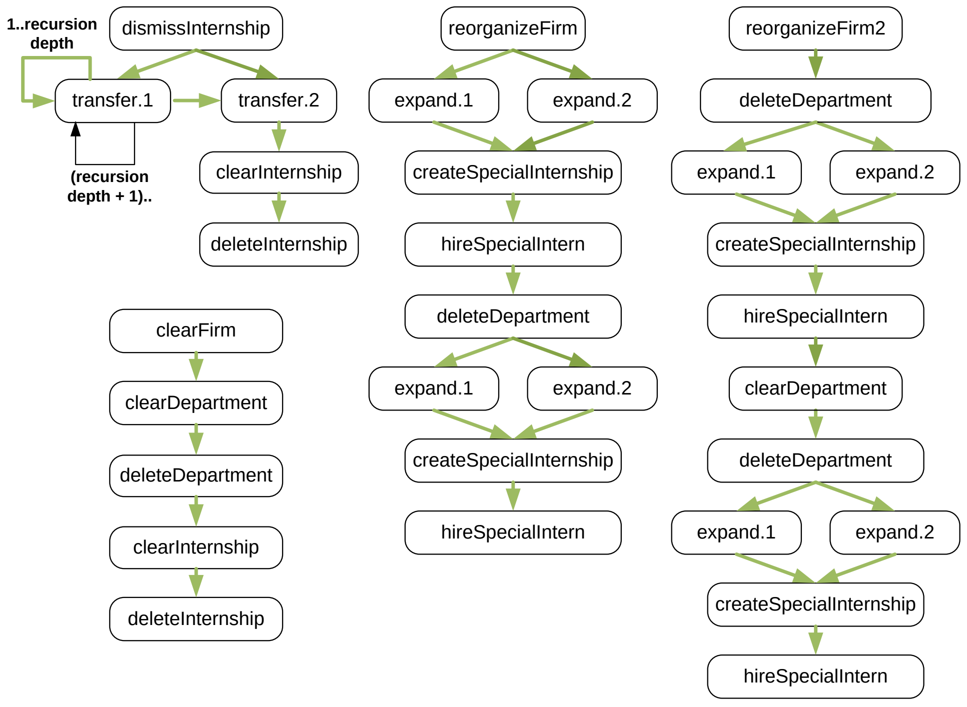


Figure 4.24: Elements to cover for all-paths coverage

All-paths coverage is aimed to handle errors and test application of rules within recursive calls, since neither the rule coverage plus plus nor edge coverage plus plus check execution paths containing a repetition of one or several graph elements. As it is also known from the coverage criteria in software engineering, the presence of loops causes infinite number of paths. The same problem arises in invocation graphs as well.

In order to limit the number of paths, a special parameter, called *recursion depth*, is introduced. It contains information about how many times the control flow should go through a repeated structure. Considering a recursion with recursion depth N on a single edge, it will result in N paths containing this edge and each with i repetitions of it, where $i = 1, N$. This parameter can be set by the developer for every round of all-paths coverage analysis.

Thus, on the basis of invocation graphs, all-paths coverage ensures that all paths down to a specified recursion depth are used at least once during semantics test execution, i.e. they appear at least once in the computed GTSs.

4.5.2 Formal Definition

The coverage item for the all-paths coverage is a single path. A path is a list of rules, following one another in an invocation graph, where the starting rule is a bigstep rule, i.e. root of the invocation graph, and the ending rule is a smallstep rule that has no more invocations, i.e. leaf of the invocation graph. The definition of a path belonging to the i-th invocation graph is provided below.

Definition 4.26 $path_i$ is a path in the invocation graph G^i_{invBR} : $path_i := \{ (r_1.uniqueName, ..., r_N.uniqueName) \mid r_1, ..., r_N \in V_i \ \wedge \ r_1 = bigstepRule_i \ \wedge \ \nexists e \in E_i : e.source = r_N \ \wedge \ \forall j = (1, N-1) : \exists e \in E_i \ \wedge \ e.source = r_j \ \wedge \ e.target = r_{j+1} \}$.

The common structures defined for the coverage criteria in Section 4.2.2 are valid for the all-paths coverage as well. They use the path from Definition 4.26 as structural elements. So, the relations *equals* and *covered* are left to define.

The relation *equals* for the all-paths coverage considers two paths equal, when their lists of labels coincide, i.e. they have equal sizes and consist of the same labels following each other in the same order. Only paths from to the same invocation graph can possibly be equal, as otherwise they would have different starting labels.

Definition 4.27 $equals_{AllPaths}(path_{i1}, path_{i2})$ is a relation defining the condition, when two paths from the same invocation graph are considered equal in the all-paths coverage : $equals_{AllPaths}(path_{i1}, path_{i2}) := (path_{i1} = (r_{11}.uniqueName, ..., r_{1N}.uniqueName) \ \wedge \ path_{i2} = (r_{21}.uniqueName, ..., r_{2N}.uniqueName) \ \wedge \ \forall j = (1, N) : \ r_{1j}.uniqueName = r_{2j}.uniqueName)$.

The relation *covered* for all-paths coverage considers a path covered by a given GTS, when there can be found a sequence of transitions in the GTS, directly following one another, which has the same length as the path, and labels of its transitions coincide with unique names of the rules from the path comparing in consecutive order.

Definition 4.28 $covered_{AllPaths}(p, GTS)$ is a relation defining the condition, when a path $p = (p_1, ..., p_N)$ is covered by the given GTS in the all-paths coverage: $covered_{AllPaths}(p, GTS) := (\ \exists t_1, ..., t_N \in E_{GTS} \ \wedge \ (\forall i = (1, N-1) : \ t_i.target = t_{i+1}.source \ \wedge \ t_i.label = p_i) \ \wedge \ (t_N.label = p_N) \ )$.

4.5.3 Example

In this section, all-paths coverage analysis on the running example is shown. The coverage values of 100% according to edge coverage plus plus is achieved in a way depicted in Figure 4.23. The projection of the full edge coverage plus plus onto the elements to cover for the all-paths coverage is presented in Figure 4.25.

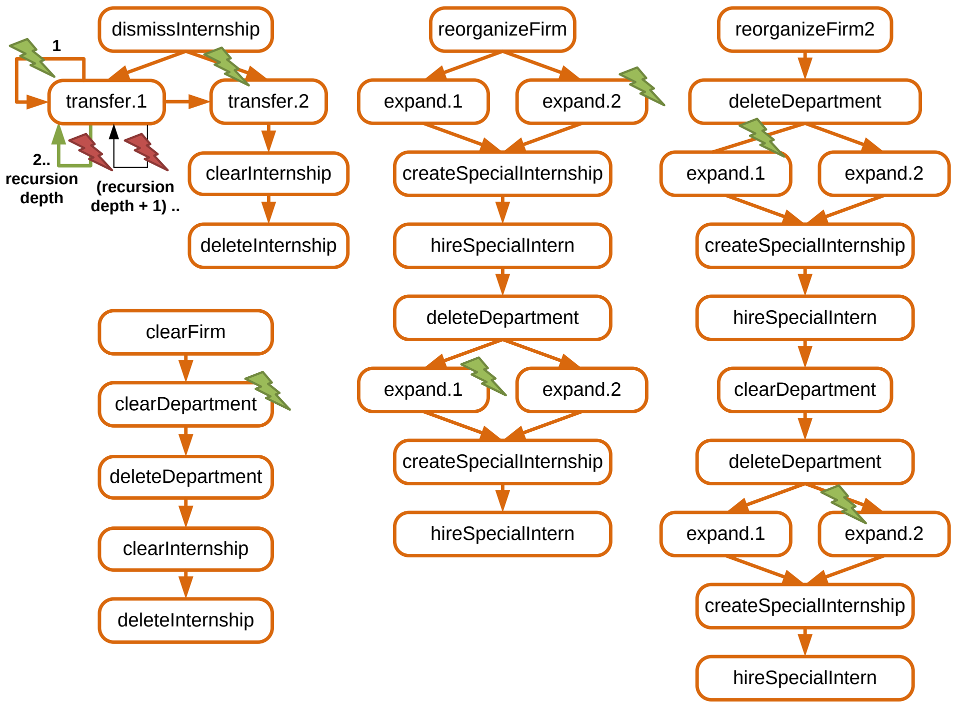


Figure 4.25: Projection of 100% edge coverage plus plus onto all-paths coverage

As it can be concluded from the graphic, the all-paths coverage is less than 100%, since edges within the recursion in the invocation graph for the bigstep rule *dismissInternship* marked green still need to be covered. So, several new example models are designed, in order to exercise the recursion with different amount of repetitions. The complete all-paths coverage is achieved, when ($recursion\ depth$) number of paths containing an increasing number of the edges $transfer.1 \rightarrow transfer.1$ are tested during the semantics test execution.

The all-paths coverage of 100% for the running example is illustrated in Figure 4.26. During the coverage analysis, some incorrect behavior in the recursive execution of the rule *transfer.1* in the invocation graph for the bigstep rule *dismissInternship* has been detected in the DMM semantics specification. However, some errors still can remain unrevealed, since the maximal all-paths coverage ensures the execution of all paths with repetitions until the recursion depth that leaves out the execution paths having more recursive loops than the limit set by the recursion depth.

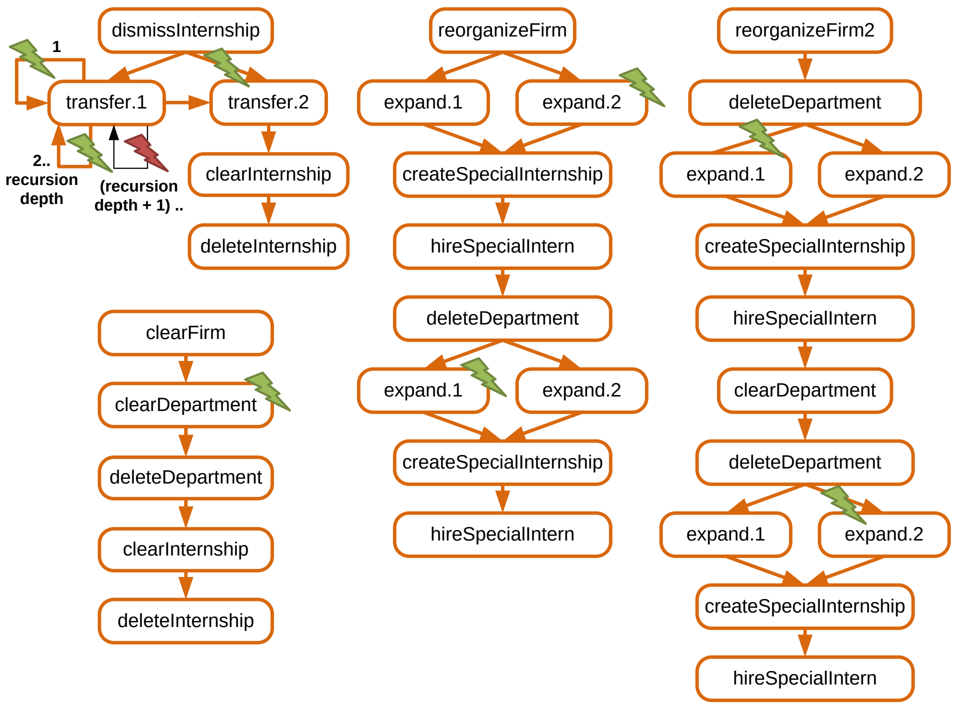


Figure 4.26: All-paths coverage of 100%

4.5.4 Discussion

The all-paths coverage is a coverage criterion expressing which part of all possible invocation sequences computed from a ruleset have been covered during the test execution. Thereby, this criterion tests all possible matching contexts for each rule exhaustively and so assists in finding more failures in DMM semantics specifications than the previous rule and edge coverage criteria, since it involves paths which were not engaged in testing before. Thus, the all-paths coverage is considered as the most powerful coverage criterion in DMM.

However, there are some problems involved with this criterion. Because of limited computational resources and an infinite number of paths produced by loops, some of paths are excluded from the set to test through the recursion depth. The 100% of the all-paths coverage is achieved, despite the erroneous behavior contained in these excluded paths will not be discovered during testing.

4.6 Hierarchy of Coverage Criteria

In the previous sections, coverage criteria for testing DMM semantics specification were introduced. In this section, interrelations among these criteria are discussed. The hierarchy of the coverage criteria and a principle for its design are given in Section 4.6.1. The interpretation of the created hierarchy with respect to the introduced principle is provided in Section 4.6.2. Then, the hierarchy is discussed further in Section 4.6.3.

4.6.1 Hierarchy

The main motivation to create a hierarchy of the coverage criteria in DMM is the ability to judge about the power of each criterion individually. This is helpful for the developer to know, which criterion would be the most appropriate for a certain DMM semantics specification. The hierarchy of coverage criteria is presented in Figure 4.27, where arrows designate hierarchical dependencies among criteria.

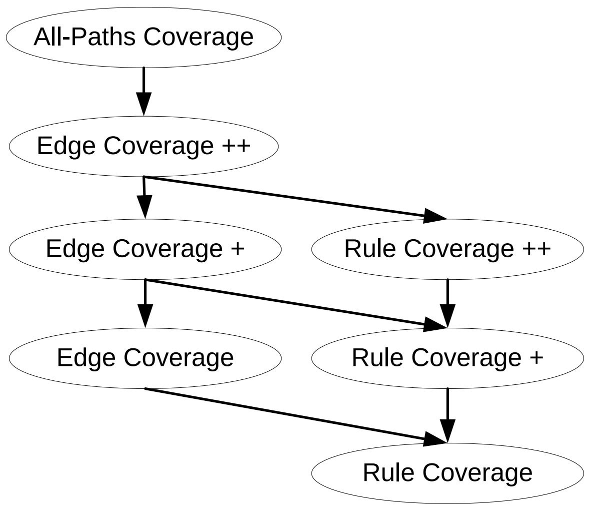


Figure 4.27: Hierarchy of coverage criteria according to expressiveness

The principle of the hierarchical dependencies is the following: if a coverage criterion located on a higher level of hierarchy achieves a complete coverage, it implies a complete coverage for all criteria placed lower than this one.

4.6.2 Interpretation

The reflection of the principle introduced above in the created hierarchy is explained in this section.

Each criterion of the same type created later expands the set of structural elements to cover of the previous one. As a consequence, each higher level coverage criterion exercises all matching contexts tested by the lower level coverage criteria, since structural elements in invocation graphs represents distinct matching contexts due to the merging. The concrete reasoning for each hierarchy relations presented in Figure 4.27 is the following:

1. *All-Paths Coverage* is on the top of the hierarchy, since it tests all paths exhaustively or in the case of a loop up to the defined recursion depth that includes matching contexts and structural elements tested by all other coverage criteria;

2. *Edge Coverage++* precedes *Edge Coverage+*, since it tests all edges appearing in all invocation graphs that implies testing of all edges with unique names in each invocation graph required by edge coverage plus;
3. *Edge Coverage++* precedes *Rule Coverage++*, since it tests all edges appearing in all invocation graphs that implies testing all rules required by rule coverage plus plus as a part of exercised edges;
4. *Edge Coverage+* precedes *Edge Coverage*, since it ensures edge coverage for every invocation graph;
5. *Edge Coverage+* precedes *Rule Coverage+*, since it tests all edges with unique names appearing in every invocation graph that implies testing all rules with unique names required by rule coverage plus as a part of exercised edges. There is also no need to think about whether the same entities were picked out, as all elements in the set of elements to cover are characterized by their names making the conformity of names determinative;
6. *Rule Coverage++* precedes *Rule Coverage+*, since it tests all rules appearing in all invocation graphs that implies testing of all rules with unique names in each invocation graph required by rule coverage plus;
7. *Edge Coverage* precedes *Rule Coverage*, since it tests all edges with unique names appearing in the ruleset that implies testing all rules with unique names required by rule coverage as a part of exercised edges. The conformity of name holding for the *Edge Coverage+* and *Rule Coverage+* remains valid here as well;
8. *Rule Coverage+* precedes *Rule Coverage*, since it ensures rule coverage for each invocation graph.

In the hierarchy, edge coverage plus and rule coverage plus plus are not connected by any arrow, since edge coverage plus does not test all rules in each invocations graph, but only those belonging to edges with different names. And the rule coverage criteria do not check all edges, required even by the simplest edge coverage. Exactly in the similar manner, edge coverage does not imply rule coverage plus, and another way around.

4.6.3 Discussion

The introduced hierarchy aims to assist the developer in estimation of the expressive power of a particular coverage criterion. The developer should start the coverage analysis with all-paths coverage. This criterion provides the most expressive coverage analysis, because it tests either all possible execution paths exhaustively or in the case of loops checks them to an extent defined by the developer through the recursion depth. Having obtained the full all-paths coverage, the developer achieves the complete coverage of all other coverage criteria as well.

The hierarchy built according to an increasing amount of tested matching contexts implies an increasing amount of errors that can be revealed, since checking more matching contexts enables finding more errors. The rule/edge coverage plus can discover errors in matching contexts of a rule/edge with the same name in different invocation graphs, which were not checked by the rule/edge coverage. The rule/edge coverage plus plus can find errors in different matching contexts of a rule/edge appearing in the same invocation graph multiple times.

Regarding the relations between rule- and edge-types, each edge coverage criterion finds errors, which can be disclosed by a corresponding rule coverage criterion. This is guaranteed by the fact, that each edge coverage criterion exercise execution paths going through all edges considered different according to it, while a corresponding rule coverage criterion exercises some of them containing rules necessary for testing. So, errors hidden in additionally checked edges can be discovered.

Concerning the relations between edge coverage and rule coverage plus, or edge coverage plus and rule coverage plus plus, since their tested matching contexts are not in any dependency, so they reveal different types of errors as well.

4.7 Conclusion

In this chapter, coverage criteria and their usage in the coverage analysis for testing DMM semantics specifications are introduced and formally defined.

At the beginning, the invocation graph as a data structure representing a DMM semantics specification for the definition of coverage criteria is presented. Its form, the intuition for its usage in the definition of coverage criteria, and its common formalization are illustrated. The formalization includes the algorithm for its computation, the derivation of data sets and relations necessary for its definition, and the formal definition of the invocation graph itself. Then, additional data structures used for the definition of coverage criteria are presented based on the invocation graph structure.

Coverage analysis is performed with respect to some coverage criteria, so three different families of coverage criteria are developed for the coverage analysis in TDSS: rule coverage, edge coverage and all-path coverage criteria. The rule coverage family requires rules corresponding to nodes of the invocation graphs selected according to certain conditions appear in particular execution paths in the generated GTSs. Three concrete rule coverage criteria are created, each delivering more information about the correctness of DMM semantics specification regarding exercised rules than the previous one.

The edge coverage family demands the same as the rule coverage family but for pairs of rules corresponding to edges of the invocation graphs. It consists of three concrete edge coverage criteria as well, each of which is more expressive with regard to tested edges as the previous one. The all-paths coverage family consists of one criterion, which ensures that all paths computed for the invocation graphs are exercised during testing, restricting the paths resulting from loops by the parameter of recursion depth.

In conclusion, a hierarchy among the developed coverage criteria is depicted, in order to arrange them according to the property of expressiveness. There hierarchical relations reflect the fact that a complete coverage of a certain coverage criterion implies a complete coverage with respect to all coverage criteria situated lower than this one. According to this hierarchy, the developer should always start with the all-paths coverage, since it is the most powerful criterion among all.

But the problem regarding the computational complexity of the developed coverage criteria arises. The recommended all-paths coverage even with the lowest recursion depth can still demand too much computational effort. The rule coverage is always feasible, but it delivers the least amount of information about the correctness of the DMM specification. So, computational complexity of the coverage criteria in between should be assessed for the developer to be able to choose the suitable one for the semantics specification at hand.

For this purpose, the implemented coverage tool is introduced in the next section, and its evaluation on a set of examples is provided.

5 DMM Coverage Tool

In this chapter, the implementation of the introduced approaches is presented, which is integrated within the implementation of the DMM workbench realizing the DMM approach and its features. It begins with the explanation of the DMM coverage tool's design and rationale provided in Section 5.1. The coverage analysis with the tool and its technical characteristics reflecting the computational complexity of the coverage criteria based on application for several semantics specifications are discussed in Section 5.2.

5.1 Design

The DMM coverage tool consists of two general parts: the code executing the coverage analysis and the code implementing the coverage criteria. The design of components responsible for the coverage analysis is presented in Figure 5.1.

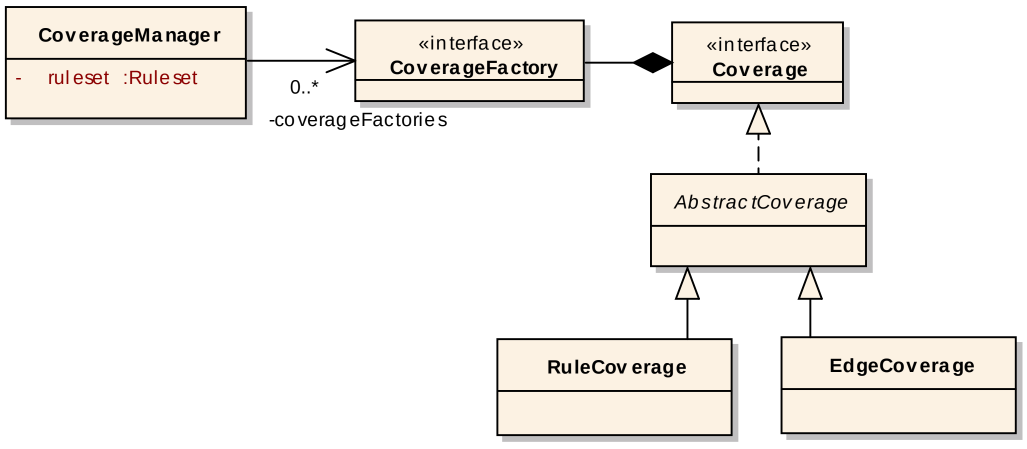


Figure 5.1: Design of components executing coverage analysis

The coverage analysis is executed by the class *CoverageManager*, which collects all coverage criteria necessary for the current round of coverage analysis, a ruleset to test, a test case, and a GTS computed for it. Each coverage criterion inherits from the class *AbstractCoverage*, which realizes all methods from the interface *Coverage* needed to implement for a single criterion and to report about it. The interface *Coverage* contains the interface *CoverageFactory*, which assists in creating executable coverage criteria that can be used during the coverage analysis. Each coverage criterion is realized by a class implementing this interface and making a particular criterion ready for execution. As examples, rule and edge coverage criteria are illustrated in this graphic, i.e. classes *RuleCoverage* and *EdgeCoverage*. Classes implementing other coverage criteria look exactly the same.

The rationale of this design is a possibility to adjust the coverage analysis to a particular situation with the help of *CoverageManager*. It serves as a central unit to determine the coverage criteria and test cases to use. Moreover, all methods necessary for the definition of each coverage criteria are generalized in the interface *Coverage*. It facilitates expansion of the tool with further coverage criteria and maintenance in the case of changing the existing ones. This design is also believed to provide better understandability of the source code.

The design for edge coverage criteria is shown in Figure 5.2 and serves as an example of the design of rule coverage and all-paths coverage criteria as well.

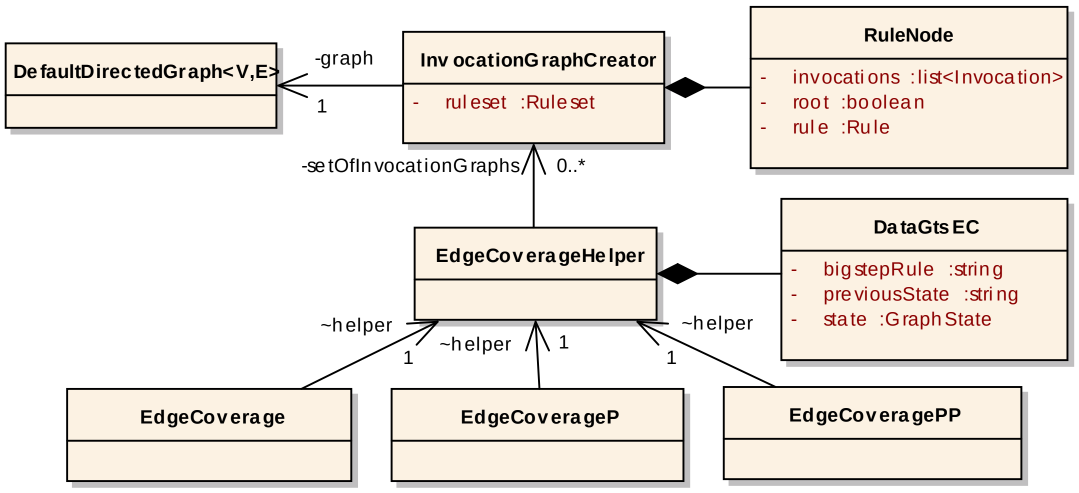


Figure 5.2: Design of components for edge coverage criteria

Edge coverage criteria are represented by classes *EdgeCoverage*, *EdgeCoverageP*, and *EdgeCoveragePP* correspondingly. The functionality common for all edge coverage criteria is extracted in the helper class *EdgeCoverageHelper*. The class *EdgeCoverageHelper* calls the functionality of the class *InvocationGraphCreator* mapping the given ruleset to a set of invocation graphs, which are instances of the class *DefaultDirectedGraph*. The class *DefaultDirectedGraph* has an auxiliary class *RuleNode*, which represents nodes of the invocation graph with all necessary information regarding it, like a corresponding rule, whether it is a root in the invocation graph, and a set of rule's invocations. The class *EdgeCoverageHelper* contains the class *DataGtsEC*, which assists in covering the structures computed based on the invocation graph by storing the information derived from the input GTS.

The advantages of this design are, firstly, the functionality concerning the construction of the invocation graph is consolidated in one class and its results are reused for all coverage criteria participating in the coverage analysis. Secondly, all necessary information about a single node of an invocation graph is stored in the class *RuleNode* and can be used in different stages of execution without the need to traverse the invocation graph once more. During the coverage, the given GTS is also traversed only once saving all needed information with the help of the class *DataGtsEC*. The computation for all three edge coverage criteria is implemented in the general fashion and differs only in parts distinct for each criterion individually.

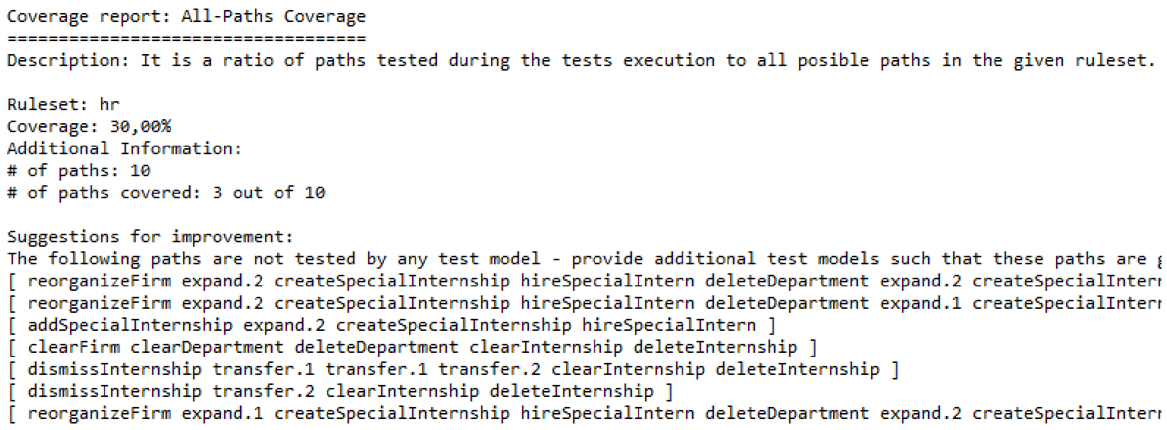
In general, the design is aimed to implement the concept as general as possible and facilitate expansion, maintenance, and understandability of the source code.

5.2 Example Application

In this section, the application of the DMM coverage tool on several examples is presented. Firstly, the coverage analysis performed with the tool is described in Section 5.2.1. Secondly, the computational complexity of the developed coverage criteria computed with the DMM coverage tool is discussed in Section 5.2.2.

5.2.1 Coverage analysis

In order to execute the coverage analysis with the DMM coverage tool, a DMM semantics specification in the form of a ruleset and a set of example model with the specified expected behavior should be given. As a result, a coverage report containing the result of the coverage analysis is formed. An excerpt from the coverage report for the running example concerning the all-paths coverage is depicted in Figure 5.3.

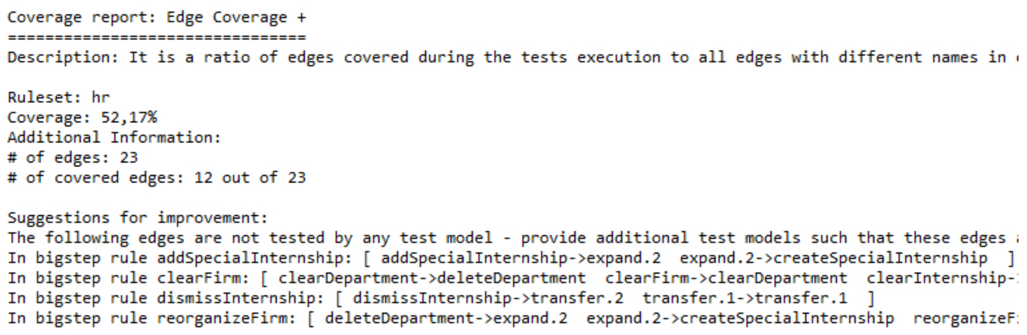


```
Coverage report: All-Paths Coverage
===================================
Description: It is a ratio of paths tested during the tests execution to all posible paths in the given ruleset.

Ruleset: hr
Coverage: 30,00%
Additional Information:
# of paths: 10
# of paths covered: 3 out of 10

Suggestions for improvement:
The following paths are not tested by any test model - provide additional test models such that these paths are
[ reorganizeFirm expand.2 createSpecialInternship hireSpecialIntern deleteDepartment expand.2 createSpecialInter
[ reorganizeFirm expand.2 createSpecialInternship hireSpecialIntern deleteDepartment expand.1 createSpecialInter
[ addSpecialInternship expand.2 createSpecialInternship hireSpecialIntern ]
[ clearFirm clearDepartment deleteDepartment clearInternship deleteInternship ]
[ dismissInternship transfer.1 transfer.1 transfer.2 clearInternship deleteInternship ]
[ dismissInternship transfer.2 clearInternship deleteInternship ]
[ reorganizeFirm expand.1 createSpecialInternship hireSpecialIntern deleteDepartment expand.2 createSpecialInter
```

Figure 5.3: Coverage report: all-paths coverage

For each coverage criterion participating in the coverage analysis, it consists of the coverage value in percent, the overall amount of structural elements to cover and the amount of covered ones. If the coverage of 100% has not been achieved, recommendations which part of the semantics specifications have to be tested further, in order to improve the coverage value, are given. In this graphic, paths which were not exercised during the testing process are listed. The developer should consider these, when creating new test cases to improve coverage.

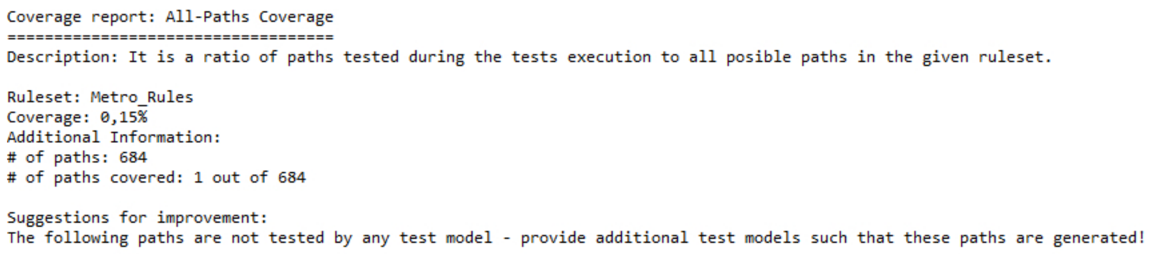
The excerpt from the coverage report concerning edge coverage plus is presented in Figure 5.4. In this part, recommendations are given for each bigstep rule individually.



```
Coverage report: Edge Coverage +
================================
Description: It is a ratio of edges covered during the tests execution to all edges with different names in

Ruleset: hr
Coverage: 52,17%
Additional Information:
# of edges: 23
# of covered edges: 12 out of 23

Suggestions for improvement:
The following edges are not tested by any test model - provide additional test models such that these edges
In bigstep rule addSpecialInternship: [ addSpecialInternship->expand.2  expand.2->createSpecialInternship  ]
In bigstep rule clearFirm: [ clearDepartment->deleteDepartment  clearFirm->clearDepartment  clearInternship-
In bigstep rule dismissInternship: [ dismissInternship->transfer.2  transfer.1->transfer.1  ]
In bigstep rule reorganizeFirm: [ deleteDepartment->expand.2  expand.2->createSpecialInternship  reorganizeF
```

Figure 5.4: Coverage report: edge coverage plus

Some lists of recommendation for increasing the coverage ratio can be rather long. Consider a language modeling a metro system with the ruleset consisting of the same amount of rules as that for the running example, the amount of paths to cover with respect to all-paths coverage equals 684 and the amount of covered path equals 1 for this language, see Figure 5.5. The recommendation consists of a list of 683 paths, which are impossible to take into account by the developer at once. So, the developer should choose several paths and create test cases exercising them.



```
Coverage report: All-Paths Coverage
===================================
Description: It is a ratio of paths tested during the tests execution to all posible paths in the given ruleset.

Ruleset: Metro_Rules
Coverage: 0,15%
Additional Information:
# of paths: 684
# of paths covered: 1 out of 684

Suggestions for improvement:
The following paths are not tested by any test model - provide additional test models such that these paths are generated!
```

Figure 5.5: Coverage report for metro semantics

Another observation concerns the fact that some rules, which are covered for the rule coverage plus, are uncovered for the rule coverage plus plus. Such situation may happen, when a particular rule has several incoming edges in the invocation graph, and so all these contexts must be tested for its coverage with respect to rule coverage plus plus. In the case, when only some edges from the incoming ones appear in the GTS leaving the other edges untested, the rule is covered by rule coverage plus but remains uncovered by rule coverage plus plus. Analogously for the edge coverage family, edge coverage plus plus can have edges covered by edge coverage plus in the list of uncovered elements, since they have several incoming edges, not all of which were checked during the testing.

5.2.2 Technical Characteristics

In the previous section, a testing of the DMM coverage tool on two examples, the running example and the language modeling metro system, were introduced. These examples were rather simple for modeling a real problem domain. In contrast, the UML semantics, like UML activities, UML state machines, or UML interactions, represents languages complicated and large enough for modeling the real world. For more information see UML Specification [9] and the UML refenrence manual [16]. The behavior of UML activities with the help of DMM is described in the dissertation of Hendrik Hausmann [11].

For the assessment of computational complexity, two semantics specifications used in the previous section are considered. The average run time of the coverage analysis for all developed coverage criteria with the DMM coverage tool is presented in Table 5.1 on Page 78. For each example semantics, coverage analysis is performed on two different example models from each language.

Table 5.1: Average run time for coverage analysis with DMM coverage tool (msec)

Exam-ple	Inv. graph	All-paths	Edge++	Edge+	Edge	Rule++	Rule+	Rule
Run. ex.	197	14 56	30 87	9 13	8 21	19 60	13 17	6 4
Metro ex.	183	477 193	1302 73	77 8	103 15	1446 20	109 17	3 6

It can be concluded from the table that the easiest criterion to compute is rule coverage that proves the assumption about its simplicity made before. Depending on example models used, the fluctuations in coverage values differ. However, the overhead needed for the computation of rule coverage plus plus and edge coverage plus plus is usually significantly larger than the overhead required by edge coverage, edge coverage plus, and rule coverage plus.

In the case of a semantics specification having a small amount of paths like in the running example, all-paths coverage remains feasible to compute. However, for the semantics having much more paths like in the metro example, its computational effort depends on an example model used. As shown in the table for the metro example, the all-paths coverage value for the first example model is several times more than for edge coverage plus or rule coverage plus, but it is still feasible in comparison with rule coverage plus plus and edge coverage plus plus. In this case, the time for coverage in rule coverage plus plus and edge coverage plus plus is more than the time for computing all paths. However, for the second model, the overhead for all-paths coverage is the largest, since the time for computation of all paths is much more than the time for the coverage itself.

As a result of the coverage analysis for two example semantics shown above, there is a tendency for rule coverage plus plus and edge coverage plus plus to require more computational time than the other criteria, even all-paths coverage. For the metro example, the all-paths coverage was computed with the recursion depth of 1. The all-paths coverage with the recursion depth of 2 is infeasible for this language. So, for semantics specifications represented by invocation graphs containing conditions and recursions, all-paths coverage could be infeasible at all. Thus, the computation time depends mostly on the semantics specification at hand, and cannot be exactly predicted in advance.

This evaluation could be helpful for a developer, who would like to test the created DMM semantics specification, in order to choose the most suitable coverage criteria, which would deliver the most information about the correctness of this specification, see hierarchy in Figure 4.27, and still would possess an affordable computational complexity.

The general advise for finding this compromise would be to start with the edge coverage plus, since this criterion implies three other criteria, which are edge coverage, rule coverage plus and rule coverage, and requires relatively few computational efforts. If it is computable, rule coverage plus plus or edge coverage plus plus can be tried, depending on which one is computable. It is better to start with edge coverage plus plus, since it implies rule coverage plus plus. If no criteria of mentioned above are feasible, further advice would be to perform both edge coverage and rule coverage plus, since they handle different structural elements and reveal different kinds of errors. In the worst case, execute the rule coverage as a minimum guarantee of quality.

6 Outlook: Critical Pair Analysis

This chapter suggests the application of critical pair analysis (CPA) as an outlook for this thesis, in order to achieve more sophisticated coverage analysis in DMM. The problem that can be handled with the help of CPA is introduced in Section 6.1. The idea of CPA is explained in Section 6.2, in order to illustrated how this approach could be used further for the DMM coverage analysis. A possible way to facilitate the stated problem by applying CPA is proposed in Section 6.3.

6.1 Problem

The DMM coverage analysis developed in this thesis concerns separate parts of a DMM semantics specification, i.e. it checks the correctness of logic within bigstep rules individually. However, several bigstep rules can apply on a given example model in different sequences, since they match on a current graph state whenever possible. Connections between bigstep rules are not considered during the coverage analysis. In order to check the correctness of a DMM semantics specification as a whole, it is desirable to test all possible sequences of bigstep rules, which can occur during the execution of this semantics specification.

The trivial solution for this problem could be testing of all permutations of all bigstep rules. However, bigstep rules do not follow each other in arbitrary order, as their order is determined by the possibility of matching of one bigstep rule after the executed invocation sequence belonging to another bigstep rule. Even if every bigstep rule from the ruleset can follow all other bigstep rules, it could not be computationally feasible to test all the possible combinations, since their amount combinatorially explodes when the number of bigstep rules gets larger.

So, it would be helpful to limit the amount of all permutations to the amount of feasible or interesting ones. The infeasible permutations can never be obtained by testing, because the suitable matching cannot be generated during the execution. So, these should be excluded from the set of all permutations to test. Interesting permutations are those, in which bigstep rules influence each other. For example, if one bigstep rule must not apply after another bigstep rule, this situation is interesting to check for correctness.

So, CPA as a possible approach to handle the problem of testing interrelations among bigstep rules is presented in the next section.

6.2 Idea

According to Ehrig et al. in [3], Hausmann et al. in [12], and Mens et al. in [14], CPA is a technique to detect conflicts and dependencies among rules. A critical pair is a pair of GTRs, which are in conflict with each other, meaning there exists an application of one GTR, which disables an application of the second GTR, and vice versa. A dependency is a situation, when two rules cannot be be exchanged without affecting the overall result of the sequence.

There are three main types of conflicts, which can be detected between two GTRs r_1 and r_2 by CPA:

1. *delete-use conflict*, when r_1 deletes some graph object, that is required by r_2 for matching;
2. *produce-forbid conflict*, when r_1 produces some graph objects, that together with existing ones form a NAC contained in r_2;
3. *change-attribute conflict*, when r_1 changes attributes, that influence the matching of r_2.

There is also a notion of parallel independence between two GTRs. It implies that the applications of these GTRs in arbitrary order produces the same result. When a GTR can apply before but not after another GTR, then asymetrical conflict between these GTRs takes place. Symmetrical conflict refers to the case, when rules disable each other mutually.

A tool implementing CPA is called Attributed Graph Grammars (AGG) and described in [3]. It is also accessible online under the AGG Homebase [18]. The AGG tool can be exploited as follows. Firstly, a meta model in the form of a type graph should be specified. Secondly, using this meta model, a set of GTRs are defined. NACs and multiplicities can be specified within the AGG as well. Thirdly, the conflicts described above are searched in these graph transformations.

In order to find conflicting GTRs, minimal critical graphs application to which cause conflicts between rules are computed. These minimal graph are intersections of the left-hand sides of these GTRs. Conflict means that there is at least one item in such a graph, which is deleted or changed somehow by one of these GTRs, and both of them are applicable to this graph.

The result of this search is a matrix mapping the GTRs to themselves and marking conflicting pairs and specifying types of conflicts occurred. After the conflicts have been calculated, different ways of solving them exist. Firstly, parallel applications of rules resulting in conflicts can be solved by setting priorities by the developer. Symmetric conflicts can be resolved by fixing the order of conflicting rules. The asymmetric conflicts can be worked out by performing additional actions enabling the application of the following rule.

The idea of possible use of the CPA approach and its program realization for the DMM coverage analysis is discussed in the next section.

6.3 Discussion

Section 2.3.2 showed that the logic of a single bigstep rule is spread onto a set of smallstep rules belonging to its invocation sequence. In order to compute the absolute dependencies among bigstep rules with the help of CPA, changes of the initial host graph made by the invoked smallstep rules should be propagated up. Having the result of the invocation sequence's execution, its conflicts and dependences with other bigstep rules can be computed. However, this method may demand much computational efforts, so a limitation of the computation only on bigstep rules is proposed.

So, CPA can be applied to discover dependencies and conflicts among bigstep rules only, in order to figure out on the level of bigstep rules whether some problems are already present in the general logic of the semantics specification. As also explained in [12], detected conflicts or dependencies do not immediately mean errors in the specification. They could be an intentional part of the specified behavior, since rules can be dependent according to the modeled behavior as well.

Then, a graph containing all bigstep rules as nodes and connections between rules showing allowed sequencing could be built. Such graph would represent all sequences of bigstep rules, which could match one after another during the execution and, therefore, should be tested. A coverage analysis on that graph could be used as a metrics for this testing process. The more advanced application of CPA could consider the behavior of smallstep rules, so computing all dependencies in the whole ruleset.

In order to integrate CPA into the DMM coverage analysis, a mapping from a syntactical meta model in DMM to a graph representing a meta model in AGG should be implemented. Elements of a ruleset should also be transformed into the format of GTRs in AGG. After these steps, CPA features available in AGG can be exploited. The developer uses the feedback about the analysis results for correction of the DMM semantics specification if necessary.

So, application of this approach would reveal additional dependencies among structural elements of a DMM semantics specification, thereby assisting in testing more matching contexts and finding more possible errors in the specified behavior.

7 Conclusion

In this chapter, the main results of this thesis are summarized and some perspectives for future development of the topic are proposed. The essence and benefits of the thesis are described in Section 7.1. Open questions left and perspectives for this work follows in Section 7.2.

7.1 Summary

In this thesis, quality of DMM semantics specifications as a correspondence between this formal specification and the semantics specified informally is discussed. The TDSS approach assists in developing high quality DMM semantics specifications through continuous testing on a set of example models, which exercise particular parts of a DMM semantics specification revealing incorrect behavior in them. A solution for an improvement of the quality of DMM semantics specifications through an improvement of the quality of testing these specifications was proposed. This solution consists in an expansion of the TDSS testing with a coverage analysis for it. The coverage analysis delivers an extent, to which a DMM semantics specification is checked for correctness, that delivers a particular confidence about its quality.

In order to perform the coverage analysis, a new data structure expressing the control flow in DMM semantics specifications called invocation graph has been introduced. Based on the invocation graph, new coverage criteria have been specified through a selection of structural elements representing a set of elements to cover and a specification of conditions, when these elements are covered during the coverage analysis. According to this procedure, three families of coverage criteria in DMM, which are coverage regarding rules, edges, and paths in the invocation graph, have been developed. Each of them has been described by its idea, formal definition, application example, and relation to the other criteria.

Coverage criteria within the rule or edge coverage family have been defined with respect to increasing expressiveness. This means that each further criterion exercises a DMM semantics specification to a larger extent than the previous ones. Coverage criteria from the different families test different parts of the specification with the possibility to reveal more incorrect behavior. The all-path coverage tests all possible execution paths exhaustively limiting the infinite number of paths resulting from the loops by the amount of rounds through it. In order to formalize the relations among the developed coverage criteria, their hierarchy with respect to the property of expressiveness has been created.

Then, the coverage analysis with respect to the presented coverage criteria has been realized by a DMM coverage tool. With the help of this tool, an approximate computational complexity for each coverage criterion has been evaluated by application on a set of example DMM semantics specifications.

The technique elaborated in this thesis is a way to facilitate quality assurance for DMM semantics specifications. The developer has a means to check a certain level of correctness of these semantics specifications determined by the used coverage criterion and find erroneous behavior in them. The developer has a set of coverage criteria in disposal, from which the most suitable one, delivering a compromise between the expressiveness and computational complexity, can be chosen for testing the semantics specification at hand.

In general, this technique is applicable, in order to assure to some degree quality of formal semantics specifications created with the help of DMM and TDSS. The specifications are checked for correctness with the help of tests, quality of which are estimated by the coverage analysis regarding the coverage criteria discussed above. So, in the form of coverage analysis in DMM, the developer has a means to obtain better tests for checking correctness of DMM semantics specifications. Through a higher quality of testing, a larger extent of semantics specifications is exercised resulting in improvement of their quality, which contribute to the quality of models designed in these languages as well.

7.2 Perspective

The first problem with the proposed coverage analysis comes from the software engineering and concerns unreachable rules or edges and infeasible paths in the invocation graph. In this case, the coverage value of 100% cannot be always reached no matter of the example models used. This problem is caused by the fact that invocation graphs are computed based solely on the static information contained in the ruleset and with no consideration of actual executional behavior. For the rule or edge coverage criteria, it means that a certain matching context can never occur. This case cannot be detected from the structure of the ruleset, and so no matter which test cases are used, no coverage criteria can achieved the coverage value of 100%.

The problem related to the all-paths coverage regards irrelevant paths, which are paths that can never be executed. In DMM, assume that a developer would like to test more paths by increasing the recursion depth and so adding more paths into the set of elements to test by iterating more through the loops. Since the information about the recursion stop implicitly or explicitly programmed into a ruleset is not being propagated to the invocation graphs, it is never known whether additionally generated paths still may be generated by any example model in this language.

An example to illustrate this problem could be a recursion modeled in a way that it stops after 2 loops. If a developer has enough computation power, the testing for the recursion depth 3 and 4 would seem to be beneficial. However, the coverage would sink and complete coverage could never be achieved. In a simple case, the developer could analyze the semantics specification and understand the reason for such behavior. In a more complicated case, when it is impossible to decide whether a path is irrelevant or not, example models which could generate it have to be searched.

The process of finding models, which exercise particular parts of a DMM semantics specification, may be hard or even impossible for the developer to realize manually. So, a method to find a correspondence between an example model and a particular part of a DMM semantics specification could be helpful. Moreover, the problem of deciding whether some path cannot be reached during testing either since a suitable example model cannot be found or since the path is impossible to execute at all is also relevant for the topic of this thesis.

The topic of generating test data, satisfying special predefined conditions, is discussed for test coverage in software engineering as well. One technique for generation of an input exercising a selected branch for branch coverage is presented by Gupta et al. in [10]. The idea of this approach is to dynamically refine an initially chosen input, so that a selected path or element in the program is executed. It also helps either to track the path by adjusting the input or to identify the path as infeasible. For testing in DMM, it could mean starting with some initial model, then refine it in a way that an aimed path is generated.

Another approach for automatic test data generation is proposed in [7]. It is based on constraint solving techniques and uses constraint systems to detect feasible paths and generate test data, global constraints to detect some of the non-feasible paths, and partial consistency techniques to reduce the domains of possible values. This approach could be adjusted for the coverage improvement in the context of this thesis as well.

Further improvement of the quality assurance in testing DMM semantics specifications through an application of the critical pair analysis was also suggested. This approach could assist in testing by checking the dependencies within a DMM semantics specification. The idea of its application is described in detail in Chapter 6.

Bibliography

[1] Standard Glossary of Terms Used in Software Testing, Version 2.1. Technical report, Glossary Working Party, International Software Testing Qualifications Board (ISTQB), April, 2010.

[2] International Software Testing Qualifications Board. CTFL (Certified Tester Foundation Level) Syllabus . Technical report, International Software Testing Qualifications Board (ISTQB), 2011.

[3] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science.* EATCS. Springer, March 2006.

[4] Gregor Engels, Christian Soltenborn, and Heike Wehrheim. Analysis of UML Activities Using Dynamic Meta Modeling. In M. M. Bosangue and E. Broch Johnsen, editors, *Proceedings of the Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006)*, volume 4468 of *LNCS*, pages 76–90, Oslo (Norway), June 2007. Springer (Berlin/Heidelberg).

[5] Daniel Galin. *Software Quality Assurance.* Pearson Education, 2004.

[6] Robert Gold. Control Flow Graphs and Code Coverage. *Applied Mathematics and Computer Science*, 20(4):739–749, 2010.

[7] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. *SIGSOFT Softw. Eng. Notes*, 23:53–62, March 1998.

[8] Object Management Group. UML Testing Profile, Version 1.0. Technical report, Object Management Group, 05-07-2007.

[9] Object Management Group. OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3. Technical Report formal/2010-05-03, 2010.

[10] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Generating Test Data For Branch Coverage. In *Proceedings of the International Conference on Automated Software Engineering*, pages 219–227. IEEE, 2000.

[11] Jan Hendrik Hausmann. *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages.* PhD thesis, University of Paderborn, 2005.

[12] Jan Hendrik Hausmann, Reiko Heckel, and Gabriele Taentzer. Detecting Conflicting Functional Requirements in a Use Case Driven Approach: A Static Analysis Technique Based on Graph Transformation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), Orlando, FL (USA)*, pages 105–155, New York, NY (USA), Mai 2002. ACM Press.

[13] IEEE. IEEE Glossary of Software Engineering Terminology, IEEE Standard 610.12. Technical report, IEEE, 1990.

[14] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electronic Notes in Theoretical Computer Science* , 127(3):113–128, 2005. Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra 2004).

[15] Arend Rensink, Iovka Boneva, Harmen Kastenberg, and Tom Staijen. *User Manual for the GROOVE Tool Set.* Department of Computer Science, University of Twente, The Netherlands, August 25, 2010.

[16] James E. Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley-Longman, 1999.

[17] Christian Soltenborn and Gregor Engels. Towards Test-Driven Semantics Specification. In A. Schürr and B. Selic, editors, *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, volume 5795 of *LNCS*, pages 378–392, Denver, Colorado (USA), 2009. Springer (Berlin/Heidelberg).

[18] The AGG Team. The AGG Homebase, September 2011.

[19] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA (USA), 2006.

[20] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA (USA), 2 edition, 2003.