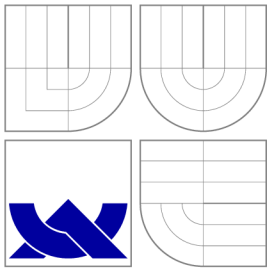
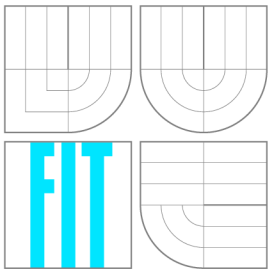


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

TESTOVÁNÍ PLATFORMY JBOSS DROOLS ZALOŽENÉ NA MODELU

MODEL-BASED TESTING OF JBOSS DROOLS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR ŠIROKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZDENĚK LETKO, Ph.D.

BRNO 2014

Abstrakt

Technika testování založeného na modelu (MBT) využívá model chování systému k automatickému generování sady testů, čímž snižuje nákladnost testování oproti konvenčnímu manuálnímu vývoji a udržbě testů. Tato práce se zaměřuje na využití zvoleného MBT nástroje OSMO při testování reálného softwarového produktu. Konkrétně se o jedná kompilátor podnikových pravidel využívaný v systému Drools, který je spoluvyvíjený společností Red Hat. V práci je popsán způsob zavedení MBT přístupu s ohledem na jeho dobré přijetí komunitou vývojářů, dále pak vytvoření modelu možných vstupů testovaného kompilátoru a zhodnocení vytvořené testovací sady. Využití MBT přístupu vedlo k odhalení pěti nahlášených a tří potencionálních a dosud nehlášených chyb v testovaném kódu. Práce na příkladu shrnuje hlavní přednosti i praktické nedostatky využití MBT technik v praxi.

Abstract

Model-based testing (MBT) is using a model of expected behavior of the system to automatically generate a set of tests. It aims at reducing the testing cost when compared to the traditional testing techniques. This work focuses on testing a real-world software system using the selected MBT tool OSMO. The tested system is responsible for compiling business rules and it is one of the main components of the Drools platform, developed by Red Hat. The work describes the introduction of MBT considering the good reception from the community of developers, then the creation of compiler input models and evaluation of the newly created test suite. The usage of the MBT resulted in detection of five reported and three potential issues in the tested code. Using the Drools compiler example, the work summarizes the main strengths and also weaknesses of practical use of MBT techniques.

Klíčová slova

testování založené na modelu, MBT, Drools, Expert, OSMO

Keywords

model-based testing, MBT, Drools, Expert, OSMO

Citace

Petr Široký: Model-Based Testing of JBoss Drools, diplomová práce, Brno, FIT VUT v Brně, 2014

Model-Based Testing of JBoss Drools

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Zdeňka Letka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Široký
May 28, 2014

Poděkování

Rád bych poděkoval odbornému vedoucímu Ing. Zdeňku Letkovi, Ph.D. a Bc. Lukáši Petrovickému za cenné rady a věcné připomínky při řešení projektu. Dále bych chtěl poděkovat svojí rodině za neutuchající podporu při řešení této práce.

© Petr Široký, 2014.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Rozšířený abstrakt

Testování je obecně považováno za jednu z velice důležitých fází softwarového cyklu. Často se testováním tráví více času než vlastním vývojem systému. V rámci technologického pokroku se rozsah aplikací zvětšuje, ty typicky implementují více funkcí a stávají se tak stále komplexnějšími. Aplikace s miliony řádků zdrojového kódu nejsou žádnou vyjímkou. Klasické přístupy k testování se stávají stále méně efektivními a více nákladnými. Je tudíž logické, že se lidé snaží vymyslet nové testovací techniky, které by s těmito problémy pomohly. Jednou z těchto technik je i modelem řízené testování (model-based testing, MBT). MBT se liší od tradičních testovacích technik především v tom, že se snaží automatizovat proces návrhu testů. Tradiční techniky jsou založeny na ručním návrhu testů. V rámci MBT se vytvoří model očekaváného chování systému, a z tohoto modelu se poté za pomoci vhodného nástroje vygeneruje sada testů. Testy je možné vygenerovat jednou a poté opakovaně spouštět nebo generovat a spouštět zároveň.

Tato práce se zabývá aplikací techniky modelem řízeného testování v prostředí platformy Drools. Drools lze považovat za rámec (framework) pro vývoj expertních systémů. Expertní systém je program, který se snaží simulovat rozhodovací činnost lidského experta při řešení složitých úloh. Znalosti jsou v Drools reprezentovány jako pravidla. Tato pravidla se zapisují ve specifickém formátu DRL (Drools Rule Language). Součástí práce je popis stávajících testů pro jednotlivé moduly Drools a také výběr vhodné části, u které by se dalo testování zlepšit pomocí techniky MBT. Jsou diskutovány dva hlavní moduly: kompilátor a samotné jádro, tedy část zodpovědná za vyhodnocování a vykonávání pravidel. Po patřičné analýze bylo rozhodnuto, že se vytvoří sada funkčních testů pro Drools kompilátor. Tento modul se stará především o překlad podnikových pravidel definovaných v DRL do interní reprezentace, kterou lze poté využít pro vyhodnocování pravidel.

Modelem řízené testování vyžaduje také specifický nástroj, pomocí kterého lze definovat jednotlivé modely a poté z nich generovat testy. Třetí kapitola obsahuje popis a porovnání pěti těchto MBT nástrojů. Nástroje jsou porovnávány především vzhledem k jejich dobré integraci s Javou, snadné použitelnosti a typu notace pro zápis modelů. Z porovnání nejlépe vyšel nástroj OSMO, který je dále použit. Jedná se o nástroj s otevřeným zdrojovým kódem, napsaný v Javě. Pro zápis modelů využívá rozšířené konečné automaty (EFSM). Přechody a podmínky přechodů těchto automatů jsou specifikovány pomocí Java metod s patřičnými OSMO anotacemi. OSMO umožňuje také dekompozici modelu na více dílčích modelů, tak aby se snadněji vyvíjeli.

Samotné testování Drools kompilátoru je popsáno v páté kapitole. V úvodu se podrobněji analyzují požadavky a předkládají se možná rešení jednotlivých problémů spojených s vývojem testování sady. Bylo nutné určit, které kontkrétní části DRL se budou v rámci práce modelovat, jaké jsou možnosti kompilace vytvořených DRL konstrukcí, jak ověřovat výstup kompilátoru a další. V části věnované návrhu a implementaci jsou popsány především dva typy implementačních tříd: datové třídy a modelové třídy. Datové třídy

slouží pouze k uchování informací o jednotlivých částech DRL a neobsahují žádnou složitou logiku. Modelové třídy naopak slouží k definici vlastních modelů. Jak již bylo řečeno, model je reprezentován konečným automatem. Modelové třídy tedy definují přechody daného automatu. Ve výsledku je tedy model reprezentován jednou nebo více Java třídami. Na konci bylo provedeno měření pokrytí kódu testy s a bez použití MBT testovací sady. Stávající pokrytí (bez MBT testů) je již vcelku vysoké, 58 % pokrytých řádků a 37 % pokrytých větví. MBT testovací sada toto pokrytí mírně navyšuje.

V rámci vývoje modelů a poté spouštění testů bylo objeveno a nahlášeno několik chyb v chování kompilátoru. Jedná se především o problémy spojené s testovaním negativního (neplatného) vstupu. Kompilátor chybně ignoruje některé neplatné konstrukce. Bylo objeveno také několik potenciálních, dosud nehlášených chyb, u kterých bude nutná další analýza a také konzulace s vývojáři.

V závěru je uvedeno několik ponaučení získaných v průběhu tvorby této práce. Prvním z nich je fakt, že modelem řízené testování opravdu dokázalo odhalit několik nových chyb a především umožnuje efektivně a systamaticky modelovat a pokrývat jednotlivé požavky na systém. Pozitivní byla i zkušenost s implementací negativního testování. Naopak OSMO, i když již je v aktuálním stavu použitelné, stále obsahuje drobné chyby a nedodělky, které mohou některé uživatele odradit od jeho použití. Z těch hlavních lze zmínit špatnou integraci s Mavenem a modelování založené na anotacích, které je sice dobře použitelné pro jednoduché případy, ovšem v případě složitejších modelů se stává těžkopádným.

Contents

1 Introduction **5**

2 Model-Based Testing **6**
2.1 Introduction to Testing . . . 6
2.1.1 Terminology . . . 6
2.1.2 Traditional Testing Processes . . . 8
2.2 What is Model-Based Testing? . . . 9
2.2.1 Model-Based Testing vs Model Checking . . . 9
2.3 The General Process of Model-Based Testing . . . 10
2.4 Benefits of Model-Based Testing . . . 11
2.5 Limitations of Model-Based Testing . . . 13

3 Selected MBT Tools Comparison **14**
3.1 Informal Requirements . . . 14
3.2 Comparison Criteria . . . 14
3.3 Tools Comparison . . . 15
3.3.1 OSMO . . . 16
3.3.2 ModelJUnit . . . 16
3.3.3 Graphwalker . . . 16
3.3.4 JTorX . . . 18
3.3.5 JSXM . . . 18
3.4 Summary and Conclusion . . . 19
3.5 Other Technologies . . . 19
3.5.1 Language: Scala . . . 19
3.5.2 Build System: Maven . . . 20

4 JBoss Drools **22**
4.1 Drools as a Part of Business Logic Integration Platform . . . 22
4.2 Drools Modules . . . 24
4.3 Drools Expert . . . 24
4.3.1 Production Memory . . . 25
4.3.2 Working Memory . . . 25
4.3.3 Inference Engine . . . 26
4.4 Current Tests in Core Modules . . . 26
4.4.1 Core Engine Tests . . . 27
4.4.2 Compiler Tests . . . 28
4.5 Identifying the Area Suitable for MBT . . . 31
4.6 Compiler Input . . . 32

4.6.1 Structure of Single DRL Rule . . . 32
4.6.2 Structure of the DRL Compilation Unit . . . 37

5 Testing the Drools Compiler **40**
5.1 Introduction . . . 40
5.2 Analysis . . . 40
5.3 Design and Implementation . . . 41
5.3.1 Decoupling Data Classes and Model Classes . . . 41
5.3.2 Data Classes . . . 42
5.3.3 Model Classes . . . 44
5.4 Test Generation and Execution . . . 45
5.5 Code Coverage with the MBT Test Suite . . . 46
5.6 Achieved Improvements . . . 46
5.6.1 Functional Test Suite . . . 46
5.6.2 Negative Testing . . . 46
5.6.3 Reported Bugs . . . 47
5.6.4 Potential Bugs . . . 49
5.7 Limitations . . . 49
5.7.1 No Offline Testing . . . 49
5.7.2 Only Single-Threaded Execution . . . 50
5.7.3 Only Basic Compilation Verification . . . 50
5.8 Lessons Learned . . . 50
5.8.1 Potential of Model-Based Testing . . . 50
5.8.2 Choosing a Suitable SUT . . . 51
5.8.3 Maturity of the OSMO . . . 51
5.8.4 Combining Scala and Maven . . . 52

6 Conclusion **53**
6.1 Future Work . . . 53

A Contents of the Compact Disc **59**

Chapter 1

Introduction

Testing is considered as a very important phase in the software development process. Often more time is actually spent on testing the program than on developing it. As the technology advances the applications have more capabilities and implement more features. Unfortunately this also leads to more complex systems. Applications with millions of lines of code are nothing unusual. Traditional testing approaches, such as capture-replay testing or script-based testing, are starting to be less and less effective and very costly, because they require manual maintenance. It seems only logical that new ways to test the software are being examined. One such technique that gained a lot of attention in past decade is called model-based testing (MBT). MBT uses a different approach to create the tests. Instead of manually developing the tests, it aims at creating a simplified model of the expected behavior of the system and then generating the tests automatically from that model.

Drools represents a practical example of such complex software. It is a rule-based production system typically used to build expert systems (programs that emulate the decision-making ability of a human expert). The model-based testing could be a viable and possibly even better alternative to current manual test design for a certain type of components. The main goal of this thesis is to examine the model-based testing in the context of Drools, decide which part of the platform would be suitable for the MBT and create the model(s) of expected behavior, generate functional test suite based on that model and asses if the method indeed meets the expectations and finds additional defects.

This thesis is divided into 5 chapters. Chapter 2 shortly introduces the traditional testing techniques, then focuses on concepts and overall process of model-based testing and also presents some benefits and limitations of this approach. Chapter 3 describes and compares several model-based testing tools, with regard to the usage in this work. At the end of the chapter, reasons for choosing one of them are presented. Chapter 4 explains what the JBoss Drools project is, discusses the current state of Drools tests and offers an area suitable for model-based testing. Chapter 5 focuses on the model-based testing of Drools compiler. It analyzes the problem more in detail, presents the design and implementation choices and compares the created test suite with the one currently used by the compiler. Last chapter 6 summarizes the entire work and presents possible future enhancements.

Chapter 2

Model-Based Testing

This chapter focuses on introducing the model-based testing and describing the general process that one typically needs to follow in order to use it effectively. Before diving into MBT, the first section contains a short introduction into testing and also defines several testing-related terms which are used later on. The Section 2.2 explains the term model-based testing more in detail. The overall process of MBT and the individual steps towards the successful application of the MBT approach are described then. The last two sections present several benefits and limitations of model-based testing.

2.1 Introduction to Testing

> *"Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems."*

This definition comes from the IEEE Sofware Engineering Body of Knowledge [38] and describes the top-level goals of testing – finding the defects and improving the product quality.

2.1.1 Terminology

Some of the testing-related terms may be widely known, others may not. Establishing a common ground and briefly explaining the terms should help the reader in grasping the concepts of traditional and also model-based testing. These terms will also be used later in the text, so it is better to have them explained together.

System Under Test (SUT) SUT refers to the application, program or generally system, that is being tested for correctness, performance, conformance to standards, etc.

Test Case Test case, or simply test, is a set of inputs, instructions how to interact with the SUT and a set of expected outputs of the SUT. It specifies how to apply the specified inputs and what outputs should the SUT produce.

Unit Test Unit test typically covers certain "unit" of code. The scope of the unit is not formally defined though. For example, in Java ecosystem one class is typically considered as a unit. However, a unit can also be a small module or a group of classes with single interface acting as a unit. Unit tests are a very useful way to obtain a certain level of

confidence that the system behaves as expected. When refactoring the code, unit tests become invaluable as they make sure the code was not severely broken during the process.

Integration Test As the name suggests, the integration test aims at verifying the wiring (integration) between different units or components. Scope of the integration test is not formally defined and they can be used to test the integration between two simple classes or a few big modules.

Functional Test Functional test focuses on a specific functionality (feature) of the system as described by the requirement(s). These tests make sure the requested functionality is in place and working as expected.

Test Suite Test suite is a set of test cases, typically with a similar purpose. System can have multiple test suites, each consisting of different kind of tests, e.g. unit test suite, functional test suite or integration test suite.

Test Result Test result is a top-level output of a test case execution. It specifies whether the test was successful or encountered some failures or errors. There are three basic types of test results: *passed*, *failed* and *error*. Passed means successful test execution (the SUT is behaving as expected). Failed test case indicates that the SUT does not behave as expected by the test, for example an assertion was not satisfied. In case the test ends with an unexpected error, such as uncaught exception, the result of the test will be error. The error may not be necessarily caused by the SUT, but possibly by a bug in the test code or in the environment. The failed and error results are very similar, the main difference is that the failed result represents an expected error and the error result an unexpected error. However, in both cases the test needs to be examined to decide if the failure is caused by the SUT or rather the test itself.

Test Policy Test policy is a high-level document describing the overall testing philosophy of the company. The test department should follow the instructions listed there and adhere to the direction it provides [16].

Code Coverage Code coverage, or sometimes called test coverage, is a metric used to describing the degree to which the source code is tested (covered) by a test suite. It is usually measured as a percentage of code exercised by the specified test suite. Code coverage uses several *coverage criteria*, the basic ones are [28]:

- function coverage – measures percentage of functions (subroutines) actually called
- statement coverage – measures percentage of executed statements (lines)
- branch coverage – measures percentage of executed branches of each control structure (e.g. `if` and `switch`)
- condition coverage – measures percentage of boolean statements evaluated to both `true` and `false`

Positive and Negative Testing Positive testing exercises the SUT in terms of valid input(s). It is trying to verify that the output of the SUT based on te valid input is correct. Negative testing focuses on the invalid input(s). Malformed input should not cause the SUT to fail unexpectedly. The SUT should rather cope with the error or generate meaningful error and stop gracefully.

Online and Offline Testing Online testing refers to a process when the generation and execution of test cases is done at the same time. In offline testing, the test cases are generated first, and only after that they are executed.

2.1.2 Traditional Testing Processes

This section briefly describes the traditional testing processes widely used in industry. It starts with the manual testing and then proceeds with two more advanced techniques, namely capture/replay testing and script-based testing.

Manual Testing

Manual testing is the oldest style of testing, but still widely used. Tester or test designer needs to manually design the test cases, based on informal requirements, and then manually execute them. The tester goes over each test case, manually performs all the required steps, compares the output of the SUT with the expected one and records the test result [45].

The biggest disadvantage of the manual testing is the execution time. It often takes a big amount of time to execute even the basic test cases. This becomes very costly over time, as the number of test cases increases and new versions of the SUT need to be tested.

Capture/replay Testing

Capture/replay testing tries to solve the problem with a test re-execution. It still requires the test cases to be manually designed, but in theory it enables the testers to capture inputs, outputs and interactions with SUT once, and then re-execute (replay) them later. However, the reality is often much more complicated and the capture/replay testing is thus considered very fragile [45]. Even a very small change in SUT API can cause a big number of tests to fail. Failed tests then need to be manually re-executed and captured. The problem lies in the lack of abstraction in the recorded test cases. The tests capture the low-level inputs and outputs of the SUT rather than higher level, more abstract, view.

Script-based Testing

Script-based testing focuses on creating executable test scripts, which may consist of one or more test cases. It solves the problem of test re-execution by automating it. The tests still need to be designed manually. Every time the tests need to be executed, it can be done very easily (no human interaction required) just by running the test scripts again. However, that increases the test maintenance, because the scripts need to be updated not only when the requirements change, but also whenever some implementation details change (e.g. API that is used to interact with the SUT). As the number of test scripts increases, the details of one SUT interface are typically spread out over multiple tests and the test scripts maintenance becomes very costly [45].

2.2 What is Model-Based Testing?

The goal of model-based testing is to automate not just the test execution, but also the test design. This is one of the main differences from the traditional testing techniques. The MBT can be shortly defined as following [45]:

> *"Model-based testing is the automation of the design of black-box tests".*

With traditional techniques, the tester needs to manually create the black-box tests. In case of model-based testing, instead of developing the tests, a model of the SUT behavior (or its environment) is created, and then it is used to automatically generate the test cases, using a suitable MBT tool.

Over the last decade the term model-based testing has also greatly evolved and nowadays it is used for different test generation techniques. Following four MBT approaches are considered the dominant [45]:

1. *Using a domain model to generate test input data.* The model represents information about possible values of input data. Testing all the possible inputs is not practical, only subset of those values needs to be selected. That is the purpose of the model—cleverly selecting and combining the subsets of input data.

2. *Using a behavioral model to generate test cases with oracles.* Behavioral model specifies not only the inputs of the SUT, but also its outputs (or is at least able to check whether the output is correct). The model describes the expected behavior of the SUT—how the SUT is supposed to react to the inputs. Building these models is more challenging than creating the input data models. However, advantage of this approach is that it covers the whole process of test generation and thus the process can be fully automated.

3. *Using an environment model to generate test cases.* In this case, the model is used to describe the expected environment of the SUT. Using these environment models, one can generate sequences of calls to the SUT. However, the model does not specify the expected outputs of the SUT, so determining if the test passed or not might be very difficult.

4. *Generating a test scripts from abstract tests.* This technique is a bit different from the previous ones. It aims at generating test scripts from given abstract test case descriptions (e.g. UML sequence diagrams). In this case, the model is meant as information about the API (Application Programming Interface) of SUT and a way how to transform high-level calls into executable tests.

This work uses a combination of the first two. The models will be used to guide the generation of the possible inputs of the SUT, while at the same time also checking that the SUT is correctly reacting to the presented input.

2.2.1 Model-Based Testing vs Model Checking

Model-based testing should *not* be confused with model checking [10]. The main difference is the fact that model checking is a formal verification method, that can be used to prove the system does not contain certain types of errors (deadlocks, race conditions, etc). On the other hand model-based testing is still a testing method. It is built on a formal basis, but it can not be used to prove the system is correct. As a famous saying goes:

"Program testing can be used to show the presence of bugs, but never to show their absence!"

— Edsger W. Dijkstra, Notes On Structured Programming

2.3 The General Process of Model-Based Testing

The Figure 2.1 shows the overall process of model-based testing. As the first step the model is created. After that, test selection criteria and test case specifications are defined. Test generator then uses these to generate a set of test cases (test suite). The last step involves executing the tests and capturing the results. More detailed description of each of the steps follows [46]:

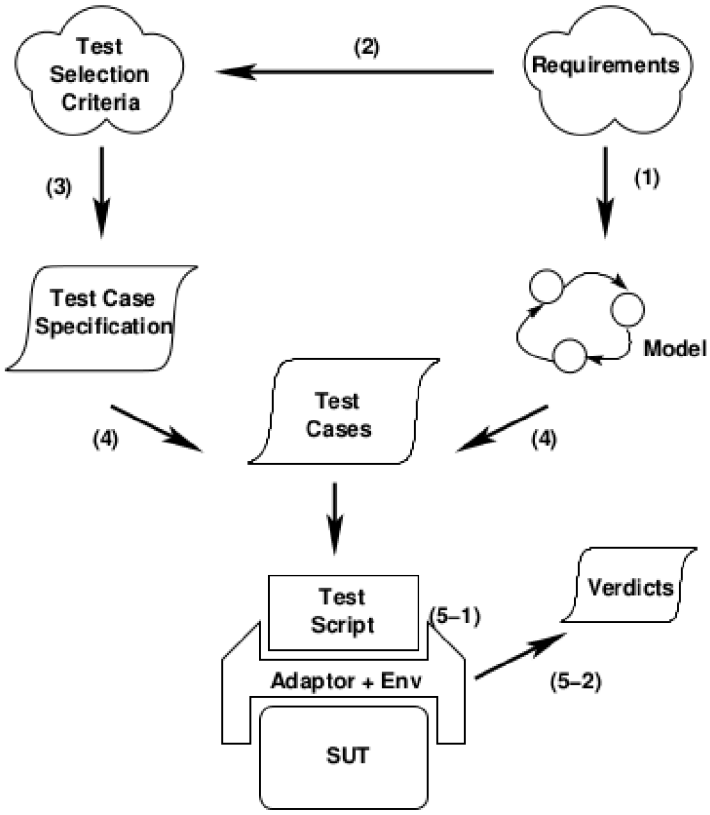


Figure 2.1: The Process of Model-Based Testing. Taken from [46].

Step 1. Creating a model of the SUT (or its environment) is one of the most important phases in MBT. Informal requirements and specification documents are typically used to build such model.

In certain cases the test model can also be used as a design model. However, the test generation model and the development (design) model need to be independent enough to ensure that the potential defects in development model are not propagated into generated tests [2]. Because of this, the test models are usually created directly from requirements and specification documents. Sometimes it may also be practical to reuse few key aspects of the development model and base the test model on them. In both cases, it is necessary to

validate the model against the informal requirements. Very useful side effect of the model validation is that it often catches requirements errors.

If the model is expected to be useful for the purposes of MBT, it has to be more abstract (simpler) than the SUT itself. If that would not be the case, validating the model would take same effort as validating the SUT itself, which is not desirable. In that case it would certainly be better to directly validate the SUT and not bother with creating and validating the model. At the same time, the model also has to be precise enough if one wants to automatically generate meaningful tests from it. If the model is not precise enough, the tests still has to be designed manually and the added value of the model is very small.

Step 2. In order to produce "good" and quality test cases (test suite), the automatic test generation needs aid in form of the test selection criteria. 'Good' test suite in this context is one that meets the test policy defined for the SUT. Methods like TMap® [22] or International Software Testing Qualifications Board (ISTQB) guidelines [18], widely used in industry, suggest to create clear test policy for all development projects.

Different aspects can be used to define test selection criteria: functionality of the system (requirements-based test selection criteria), structure of the test model (state coverage, transition coverage), data coverage heuristics (pairwise, boundary value) or pure randomness.

Step 3. *Test case specifications* are created by transforming the test selection criteria to make them more suitable for use with automatic test case generator. The goal of the transformation is to formalize the notion of test selection criteria so that the generator is capable of deriving tests from them.

Step 4. Test generation can start after both the model and the test case specifications are created. Usually there is more than one test case that satisfies particular test case specification. In such cases the test generator just picks one of those tests. Significant effort may be spent on optimizing the test suite so that it contains smaller number of tests that cover a lot of test case specifications.

Step 5. The last step is the actual execution of generated test cases. It can be either manual (e.g. person executes the tests) or automatic. The automatic test execution uses two components which are together called *test execution environment*. Such components are *adaptor* and test results (verdicts) recorder. The adaptor, as the name suggest, is used to adapt (bridge) the concrete SUT interface with the abstract test data. As noted before, the SUT and the model are using different level of abstraction — that is why the adaptor is needed. The generation and execution of the tests may be done in one place (online testing) or the tests may be generated beforehand and executed later (offline testing).

2.4 Benefits of Model-Based Testing

Model-based testing provides various benefits for people and organizations using it. This section briefly discusses four such benefits: SUT fault detection, reduced testing cost and time, improved test quality and requirements defect detection. The Practical Model-based Testing [45] contains a detailed description of all of these benefits, please refer there for more information.

SUT Fault Detection

Detecting faults in the SUT is the primary goal of testing. Experience shows that the model-based testing is quite good at that. Studies at IBM [14] and Microsoft [43] report that the model-based testing is typically able to find at least the same number of faults as the manually created test suites. In one of the Microsoft cases, the number of found faults was even 10 times higher. In smart card industry, the model-based testing is being deployed on daily basis. Various reports say that the number of found defects has always been higher than or equal to the number of defects found by the manual process [8, 9, 5].

The model-based testing is of course no silver bullet. The quality of test suite and the number of detected faults greatly depends on the skill of people designing the models. We can not say that the model-based testing always detects more faults than the manual process, or vice versa.

Reduced Testing Cost and Time

Model-based testing can reduce the time and effort spent on testing. Number of case studies has been published to support this claim. They show that the cost of the model-based testing is usually lover when compared the manual test case design [8, 9, 12].

Besides the test design, the model-based testing can also reduce time needed to analyze test failures. There are basically three areas where it can help [45]: (1) as the test cases are systematically generated the report failures follow consistent style, which may help in the analysis if one already knows the style, (2) some tools are capable of generation of the shortest path that leads to the failure, this may greatly simplify the analysis, (3) one can inspect not only the generated test case, but also the corresponding abstract one, leading to better understanding of the failure.

Improved Test Quality

Manually designed test cases reflect the skills and experience of the test engineer who created them. Even if the engineers follow the same guidelines, the resulting tests may be of different quality. The rationale for design decisions is not usually captured and process is not repeatable. The model-based testing tools, based on algorithms and heuristics, make the process systematic and repeatable. For example, the coverage of the model can then be used to measure the quality of the test suite.

Requirements Defect Detection

One of the less expected benefits of the model-based testing is the ability to catch requirements errors early in the process. The models are usually created from informal requirements captured in natural language. The models use lower-level constructs and their construction usually brings additional questions like “What constraints are put on this input?” or “Can this attribute contain a negative value?”. Answering these questions may uncover inconsistencies in the requirements. The model is basically an abstract prototype of the SUT, so the ability to find requirement errors should not be surprising [7].

2.5 Limitations of Model-Based Testing

Model-based testing is not perfect of course. Several known limitations are introduced in this section. They are only briefly described as detailed list can be found in [45].

Since model-based testing is still a testing method, instead of purely formal approach, the basic limitation is that it can not surely find *all* the differences between the implementation and the model, thus can not prove the non-existence of a bugs in the SUT. That is, however, general limitation of all testing methods.

Model-based testing also puts different requirements on tester's skills, comparing to manual test design. Besides the application domain, the tester (or better model designer) also needs to be able to abstract the problem correctly and design the models. The learning curve is usually steep and may require additional training costs.

Another limitation comes from the fact that MBT is mainly used just for functional testing. Although it is sometimes being used for stress testing the web-based applications, the majority of use cases are connected with the functional testing. Among other things, this leads to the state where the testers need to use other approaches for non-functional tests, and thus need to master the traditional techniques together with model-based testing.

Some "pain" points can also manifest themselves after one has overcome the initial problems and started to use the MBT [45]:

- **Outdated requirements**: Software projects usually evolve over time. If no one updates the requirements, they become obsolete and out of date. If the model designer creates a model based on incomplete or invalid requirements, the model-based testing will then find lots of SUT "errors", which are in fact not defects.

- **Inappropriate use of model-based testing**: There may be parts of the SUT that are very difficult to model. Creating a few manually designed tests may be much quicker. It takes some time and experience to be able to tell which SUT parts are suitable for MBT and which should rather be tested with a different approach.

- **Time to analyze failed tests**: Whenever some of the tests fail, the tester needs to examine the failure to determine the root cause. The cause may be in the SUT, the adaptor code or the model itself. In case of generated test cases with complex sequences of steps, this may become harder than with traditional techniques where the manually designed sequences are usually shorter and easier to understand.

- **Useless metrics**: Most people, and primarily managers, are using number of designed test cases as a metric for a test development progress. However, the MBT enables easy generation of thousands of test cases, so such metric becomes useless. One needs to move towards other metrics describing the test progress, such as code coverage, requirements coverage or model coverage [40].

Chapter 3

Selected MBT Tools Comparison

This chapter introduces and compares five model-based testing tools. The first section lists the informal requirements that need to be considered when choosing the right tool for the planned Drools compiler testing, described later. The next section transforms those requirements into comparison criteria and describes each of them. The last section contains the conclusion based on the criteria and short description of each tool.

3.1 Informal Requirements

In order to select a suitable tool for our purpose the requirements need to be specified. This section contains the basic informal requirements which are later transformed into comparison criteria. The first requirement arises, among other reasons, from the fact that the thesis is being created in cooperation with the Red Hat. Red Hat is an open source company and thus it tends to prefer open sourced tools over the proprietary ones. The tool should be open sourced, with sources available free of charge. The availability of the source code also helps in understanding the tool and possible error diagnosis.

The next requirement comes from the fact that the SUT is implemented in Java. The tool should be able to easily integrate with Java and possibly other JVM languages. This requirement is very important as the SUT is also implemented in Java.

The whole MBT process should be mainly focused on people with good knowledge of Java and minimum information about the MBT itself, different modeling techniques, etc. The model and test generation will most probably be maintained by people who develop/test the Drools platform and for them it is much easier to create a Java class and code some part of the model there, instead of learning some not very well known modeling environment. Easy to use tool is more probable to get adopted comparing to some complex one, with several dozens of steps needed to create even the simplest model.

3.2 Comparison Criteria

When comparing two or more tools (or basically any two objects) and then selecting one that fits the requirements, one should define the comparison criteria. Following paragraphs describe the criteria that have been synthesized from the above mentioned requirements.

License. License is an important issue that needs to considered whenever one wants to use any new software system. Tools with the open source license and thus available source code have in this specific case a big advantage over the proprietary ones.

User community and project activity. Very important aspect of the overall maturity of the tool is also the size and activity of user community. E-mail lists and user forums are very useful way to get the basic support and the frequency of posts there also helps in understanding the rough number of active users. In case of the open sourced tools it is also useful to check the source code changes to see if the project is still actively developed and alive.

Level of integration with Java. This criterion arises from the fact that the SUT is implemented in Java. Good integration of the tool with Java makes the entire MBT process easier and faster.

Model representation. The model of the expected SUT behavior can be expressed by different notations. In theory all of the notations can be used to model wide range of systems. However, there often exists one notation that is more appropriate than the others for certain kinds of systems. The finite state machines and their extensions should be the preferred ones as they are well known and handle the input modeling well.

Support for online/offline testing. Online testing is very useful in the model development to try the current implementation. The tests will fail immediately after encountering an error which decreases the overall execution time and thus the round-trip time from model update to test execution.

The main advantage of the offline testing is the fact that the tests can be generated once and possibly executed multiple times. For example, the MBT tool can generate thousands of tests and they can be repeatedly executed as a part of the full continues integration build.

Parallel test execution. The big advantage of model-based testing is that it can generate very large test suite, because once the model is created adding new tests cases is just matter of running the MBT tool (possibly with different settings to ensure different tests are produced). Once there are thousands of tests their execution usually becomes rather long, unless the tests can be easily executed in parallel.

Selection of test cases. Different tools may provide different means to select the appropriate test cases. For example, in case of the FSM/EFSM model it can be the state coverage or transition coverage. One of the popular criteria is also based on pure randomness, e.g. randomly stepping through the model until some predefined number of steps is reached. Advantage of that approach comes from usage of the (pseudo)randomness — the sequence of steps is different for each execution, so in theory more states are explored and thus bigger number of errors can be detected.

Report/output. The test report serves as an overview of the executed tests. It lists the number of successful and failed tests. Good test report is also essential for test failure analysis.

Counterexample. Counterexample is an sequence of steps that leads to the specified failure/error. Ability of the tool to create the counterexample makes the test failure analysis easier, and provides executable example that can be used to reproduce the issue directly.

3.3 Tools Comparison

For comparison we pick five tools based on the analysis of various model-based testing tools lists and surveys [1, 24, 6]. These tools are OSMO [20], ModelJUnit [34], Graphwalker [31], JTorx [4] and JSXM [13]. The Table 3.1 contains a summary of the different criteria for each of the tools. The tools are briefly described in own subsections. Summary of the

comparison is provided in the Section 3.4, which also lists the reasons for choosing one of the tools for further use.

3.3.1 OSMO

OSMO stands for Open Source Modeling Objects. It is a set of open source tools and libraries for modeling software systems and applying the models. The main component is the OSMO Tester which is model-based testing tool.

OSMO uses Java as a modeling language. That enables easier adoption among developers and testers as Java is one of the most popular languages. Model is represented as an EFSM, using basic Java methods with annotations that tell the OSMO how to process the methods. Following two annotations are the main ones. They are used even for the simplest models. Full list of the supported annotations can be found in OSMO User Guide [20].

- `@TestStep`. Test step is a transition from one state of the system to another. Annotation takes the test step name as a parameter. The method marked by this annotation must not take any parameters.
- `@Guard`. Guard (or condition) says whether the specified test step can be performed in certain state. The test step name is specified as part of the annotation. The method marked by this annotation must not take any parameters and must return boolean.

OSMO enables both online and offline testing. The online testing is supported directly by the framework so it is easy to use. The offline testing is supported by creating an adaptor code, which uses templates filled with specific data during the test generation. As a result, set of JUnit test cases can be created. These tests are then executed in parallel.

3.3.2 ModelJUnit

ModelJUnit is an open source MBT framework created by Mark Utting and others. It was designed as an extension to JUnit.

ModelJUnit is very similar to OSMO both in terms of modeling and test generation. The (E)FSM model is also written in Java using specific annotations as method markers. However, there are some differences. ModelJUnit uses `@Action` instead of `@TestStep` to specify the transition methods. The guard methods are using simple naming convention instead of annotations. The method named `myTransitionGuard()` must return boolean and guards the transition named `myTransition`.

Great source of more information about the ModelJUnit are chapters 5.2 and 5.3 in Practical Model-based Testing [45]. There is also an older (2007), but still very useful talk[1] by Mark Utting about ModelJUnit and model-based testing in general.

3.3.3 Graphwalker

Graphwalker is an open source tool that enables test sequences generation from FSMs and EFSMs. It uses graph-based modeling to specify the expected behavior and then wires the model with the adaptor to actually generate and execute the tests.

A standard Graphwalker workflow consist of three quite independent steps [37]:

[1] See http://www.youtube.com/watch?v=g4Uo2pyrWCg

Criteria/Tool	**OSMO**	**ModelJUnit**	**Graphwalker**	**JTorX**	**JSXM**
Licence	Open source - GNU LGPL	Open source - GNU GPL	Open source - MIT	Open source - custom, BSD-like	Could not find (sources not available)
Community, activity	Mostly devs, but active, several commits per month	No messages in mailing list; commits from time to time	Mailing list not found;	Last commit in Sep 2012; no mailing list or forum	Mailing list not found; n/a for last commit
Java integration	Trivial, out-of-the-box	Trivial, out-of-the-box	Easy; adaptor needs to be created	Easy;	Easy; able to generate JUnit tests
Model representation	FSM, EFSM; annotated Java methods	FSM, EFSM; annotated Java methods	FSM, EFSM; GraphML + Java to interact with SUT	LTS; Models in Aldebaran, GraphML or GraphViz	Special type of EFSM - Stream X-Machines (SXMs)
Online/offline testing	Both	Both	Both	Both	Offline only
Parallel test execution	Yes, for JUnit tests	Yes	Not out-of-the-box	Not out-of-the-box	Yes; generates JUnit tests
Test case selection	State, transition coverage; random	State, transition coverage; random	Different coverage and stop criteria	Randomness; Test purposes (automaton or LTS)	Based on testing theory of SXM
Report/output	Coverage report	Coverage metrics	Coverage metric; visualization of transition graph traversal	Coverage metrics	Standard JUnit report
Counter-example	Not out-of-the-box	Not out-of-the box	Not out-of-the box	Not out-of-the box	Not out-of-the-box
Comment	Easy to use, flexible	Similar to OSMO; less active	Graphical modeling + coding in Java	Mostly for educational purposes	More steps needed (model → XML tests → exec. tests)

Table 3.1: MBT tools comparison based on specified criteria.

1. *The model design.* Grapwalker is using GraphML to specify the states and transitions of the (E)FSM. GraphML is XML based format, so writing the model by hand is not recommended as it is quite difficult. There are various graphical designers that can used to specify the model, for example editor yEd[2].

2. *Test automation code.* This is the programming part. Using Java, one needs to develop the classes that will interact with the SUT and check that the output of the SUT conforms to the expected one.

3. *Test execution.* Different strategies and stop criteria can be used to traverse through the model and thus generate different types of tests, e.g. smoke tests, functional tests or stability tests.

3.3.4 JTorX

JTorx is a successor to the TorX. It aims at easier deployment through the improved installation, configuration and usage. It uses Labeled Transition System (LTS) to specify the expected SUT behavior. JTorx supports interaction with systems that communicate using standard input and output or TCP. For other systems adaptor needs to be created.

A typical JTorx configuration, shown in the Figure 3.1, contains at least following components. An `Explorer` is used to access the state space of the `Model`. A `Primer` provides access to the suspension automaton (see [44]) of the `Model`. The `Driver` drives the entire process, optionally taking control commands from users. `Driver` is also responsible for saving the test log and creating the final report (verdict). The `Adapter` provides ways to communicate with the SUT.

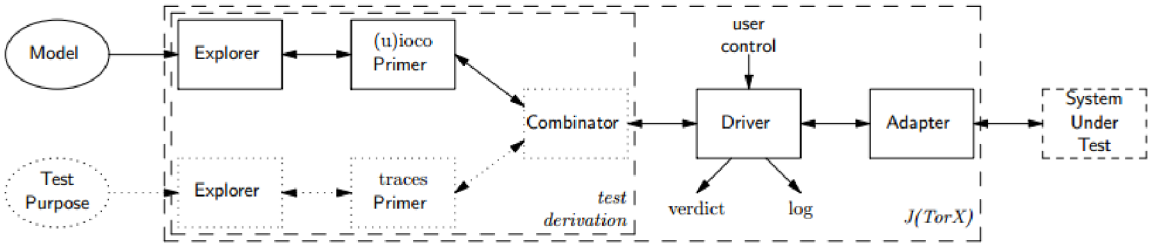


Figure 3.1: Tool components of a typical JTorX configuration. Items `TP`, `Explorer`, `Primer` and `Combinator` in the dotted boxes are only present in a guided test run. Taken from [4].

3.3.5 JSXM

JSXM is a model-based testing tool implemented in Java. It uses special type of extended finite state machines, Stream X-machine (SXM), to express the model. The generated tests are represented as XML files, thus independent of the technology or programming language of the implementation [13]. However, the tests are not directly executable and so called *test transformers* are needed. Test transformer is capable of translating the independent representation into executable form. JSXM comes with transformer into JUnit tests. The creation of own transformers, for different languages and frameworks, is also supported.

[2]See http://www.yworks.com/en/products_yed_about.html

3.4 Summary and Conclusion

The presented tools are similar in many ways and yet different in others. They are all implemented in Java, which makes them typically easy to integrate with the Java-based SUT. The tools are primarily developed as a part of academic research. The source code is usually publicly available, so there is a possibility to extend the tool and contribute the required features. The notation used for modeling the behavior is mostly (E)FSM. Some of the tools are also using other notations based on (E)FSM, such as SXM. OSMO and ModelJUnit rely purely on Java to define the model and execute the tests. The JSXM, JTorX and Graphwalker use XML to store the intermediate result of test generation (the abstract tests).

The JTorx does not seem to be maintained as a last commit in the source repository is in September 2012. This would make it harder to solve potential problems as the community basically does not exist. JSXM and Graphwalker were ruled out, because of their usage of abstract tests. They need one more step on the way from the model definition to executable tests which makes the model design more error prone. In the end, the choice was constrained to OSMO and ModelJUnit. In regard to the comparison criteria, they both have the same features and limitations. Choosing one of them was primarily matter of personal experience. The OSMO is easier to setup and use. It also contains great amount of documentation which helps later with the advanced features. There are also features (e.g. data generators) that are not present in ModelJUnit. The OSMO was chosen as a tool that will be used for the model-based testing.

3.5 Other Technologies

Besides the OSMO, there are also other areas that need to be considered before beginning with the implementation. Choosing the specific technologies is often matter of knowledge, experience and personal opinion of the person or people implementing the solution. That of course should not be the only requirement.

Following two section discuss the language that will be used to implement the designed solution and chosen build system which will be used to build, test and run the MBT test suite. The reasons for selecting each of them are briefly introduced together with the description.

3.5.1 Language: Scala

Due to the nature of the Drools project, described in Chapter 4, and the OSMO MBT tool – both are implemented in Java and in case of the OSMO the users are encouraged to implement the model also in Java, the implementation language choice was constrained to the so-called JVM (Java Virtual Machine) languages. The obvious best known one is Java. Other JVM languages can be used in place of Java as they usually integrate with it very well. Java has its issues and is often considered verbose and boiler-platy (requiring a lot of boiler-plate code for even simple tasks). The other JVM languages are trying to address those issues. To name just a few of them: Scala [25], Groovy [21], Kotlin [33] or Ceylon [27]. Scala was chosen for various reasons, described in the following paragraphs.

Scala is a mature, statically typed JVM language introduced in 2003. The name is an abbreviation of words SCAlable LAnguage and it reflects the fact that it can be used for great variety of projects with different sizes (scales with the project size). From simple

one class projects or even scripts to large business critical applications will millions lines of code.

Scala is also considered a hybrid or multi-paradigm language as it fully supports both functional and object oriented paradigms. It tries to bring the best from each and combine it into single language. For example, everything in Scala is an object, there are no primitive types like in Java. On the other hand, Scala encourages the developers to use short pure functions (with no side effects) and immutable data structures as those are the building blocks of functional programming.

In comparison with Java, Scala offers very concise syntax and features that eliminate the boiler plate nature of Java. For examplem the concept of `case classes` removes the need to define the standard methods `toString()`, `equals()` and `hashcode()` as those are automatically created under the hood, based on the properties of the class.

Other JVM languages offer similar features like Scala. Groovy is very popular, probably even more than Scala, but unfortunately it is a dynamic (sometimes called scripting) language and that is considered by many people in Java community as major disadvantage and the reasons for that are very convincing. The biggest disadvantage seems to be the fact that compiler can not catch nearly as many bugs during compilation as in case of statically typed languages. Although there are other languages like Ceylon or Kotlin that are statically typed, they are not as widely used and mature as Scala. That usually results in smaller support from various tools and libraries (e.g. IDEs, build tools, frameworks), which makes the usage of the language harder.

To sum things up, Scala was chosen due to the following reasons: (1) concise syntax and non-verbosity, (2) static nature and great integration with the Java, (3) support for both OO and functional paradigms and (4) overall faster development time (in comparison with Java).

3.5.2 Build System: Maven

Every non-trivial software project needs some build system to enable easy compilation, testing, packaging (if necessary) and preferably also dependency management. In the Java and Scala ecosystems there are several well known build systems – namely Ant [23], Maven [11], Gradle [19] and SBT [35]. Each of them has its advantages and disadvantages, which will not be discussed. At the end, the Maven was chosen, because it is already used by Drools and it is also the de facto standard build tool in Java world.

Maven is a build system or more precisely entire project management system that supports great variety of languages (Java, Scala, Groovy, etc), supports and already contains many useful plugins and uses standard XML language to describe the project. The last point is very important, because it enables people already familiar with the Maven to easily understand other project's build process (assuming the other project is also using Maven).

Central point of the entire Maven-based project is a Project Object Model (POM), typically saved in the file `pom.xml`. The POM describes the project properties like the location of the sources, location of the tests, ways how to run the tests or name of the resulting artifact. Maven uses the *convention over configuration* principle with reasonable default values. Of course almost all of the configuration options can be overridden in case it is needed.

In order to use and easily test the Scala code the project needs to use and define two Scala-related plugins. The most important one is the `maven-scala-plugin`[3] which

[3] http://scala-tools.org/mvnsites/maven-scala-plugin/

enables mixed compilation of the Java/Scala sources. The second plugin is merely used to execute the tests, be it unit tests or the entire compiler model-based test suite. It is called `scalatest-maven-plugin` and provides the bridge between the Maven and the ScalaTest testing framework.

Since Scala was chosen as an implementation language, the SBT (Simple Build Tool) was also considered as that's the default build tool for Scala. However, contrary to the name, the learning curve is pretty steep and using the SBT would put another requirement on the people interested in the project. Maven should be this case better alternative as almost every Java developer knows at least basics. The Drools project uses the Maven too, so people familiar with Drools should not have troubles understanding the build process of the compiler model-based testing module.

Chapter 4

JBoss Drools

This chapter introduces the project JBoss Drools (referred to as Drools below). The first section talks about the bigger picture of Business Logic Integration Platform and the role of Drools there. The second section explains how the Drools source is code divided into modules. In third section the main component (Expert) is described more in detail. The last two sections discuss the current state of the tests in Drools and also pick some interesting parts for model-based testing.

4.1 Drools as a Part of Business Logic Integration Platform

Drools is a part of so called *Business Logic Integration Platform* (BLIP) [29]. This platform aims at providing unified and integrated way to use rules, workflow and event processing. Following behavioral modeling techniques are supported by the platform:

- Business Rules Management
- Business Process Management (BPM)
- Complex Event Processing (CEP)

Drools itself is responsible for the business rules and complex event processing. The BPM part is implemented by other project, called jBPM [41]. Recently (November 2013) these two projects has been unified under the name *KIE* (Knowledge Is Everything). The names Drools and jBPM are still used as they are well known and established. The term KIE is meant as an umbrella for those projects. The biggest reason for this unification was to simplify the integration between those projects. They are now using the same KIE API which makes the integration easier and usually also much faster in terms of development time.

The entire KIE platform is fully open sourced, written in pure Java and developed under the JBoss.org community header. The KIE basically consist of several main projects as shown in the Figure 4.1. Following paragraphs briefly describe the purpose and features of each project.

Drools. Formally, the Drools project consists of two main components — Expert and Fusion. Expert is the rule-based expert system and Fusion provides the CEP capabilities. However they are both integrated together in way that makes it impossible to use only Expert or only Fusion. More information about Expert is in following section 4.3.

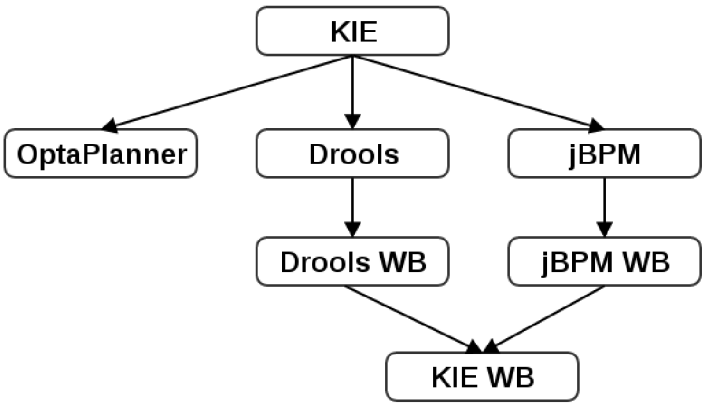


Figure 4.1: Structure of the main top-level KIE projects.

Drools WorkBench. Drools WorkBench (Drools WB) is a rich web-based application that aims at providing complete solution for business assets authoring lifecycle. It features several editors for creation of different types of assets, supports test design and execution over the created assets and as the last step, deployment into the specified Maven repositories as plain JAR files. These files are then picked up by the applications that need to use those specified assets.

Drools WB also enables authoring for both business users and technical users. Business users can use the graphical wizards and editors that does not require deep technical knowledge and for developers there is a plain text editor or possibility to push the changes directly into the underlying data store (typically one or more Git repositories).

jBPM. The term jBPM is often used in context of one or more different (but related) projects. It usually refers to the core engine responsible for execution of the business processes. Other applications, such as jBPM Designer or jBPM Console NG (see below) are not considered the core engine part, although they are also important.

The jBPM engine basically receives a standard BPMN2 [26] business process definition as input, validates it and then executes each process node according to the BPMN2 specification. It also implements advanced features like transaction handling, persistence or pluggable human task service. However, the strongest advantage is probably the flexibility and lightweight nature of the engine as that enables easier embedding inside other applications.

jBPM WorkBench. jBPM WorkBench (in the developer community known as jBPM Console NG) serves as an graphical interface for the management of business processes. It features several screens for deployment and starting/managing the processes and tasks (complete, cancel, reassign,...). Business process authoring is also embedded and the actual BPMN2 editor can be also used in a standalone mode.

KIE WorkBench. KIE WorkBench combines tooling for both Drools and jBPM. It simply glues together the features of the Drools WorkBench and the jBPM WorkBench. Then the end user has all needed editors and screens in one application in case he is using both the Drools and jBPM.

OptaPlanner. OptaPlanner is a planning and business resources optimization tool. It uses metaheuristic algorithms to solve planning problems with limited set of resources, for

example Traveling Salesman Problem[1] or employee shift rostering. Originally it was a part of the Drools, but with the arrival of KIE, it has become a top-level project. Although it integrates with Drools easily, it does not have any direct dependency on it.

4.2 Drools Modules

The Drools platform consists of several more or less independent modules. Each module is basically a Maven [11] project that contains own source code tree. Following lines briefly describe the most important ones as they will be referred to later in this chapter. The list is by no means complete. However, the other modules are not that widely known and they typically provide optional extensions. They will not be described here. The complete list of modules can be found on Github[2], where the Drools source code is hosted.

drools-core. This is the core module that implements the actual pattern matching algorithm and rule processing (Expert). It also provides the CEP capabilities (Fusion). Following section 4.3 introduces the rules processing engine more in detail.

drools-compiler. Compiler is responsible for transforming the rule definitions into objects that can later be used to create the knowledge base. After compilation, each rule is basically represented as a standard Java class. This module depends on `drools-core` and in theory it can be considered as optional. If all of the resources have been already compiled and saved e.g. into binary file, the compiler is not needed in runtime. However, typical Drools scenarios consist of the resources compilation at runtime, and those directly depend on the compiler to do that.

drools-decisiontables. Decision tables are one of the other ways to define rules. Instead of writing the rules by hand, an Excel spreadsheet is created and filled with the data needed to generate the rules. This way, the technical people can develop the Excel template and business analysts can just insert actual values that will be used in rule conditions and actions. This module is optional.

drools-persistence-jpa. Module primarily used to persist the state of the computation into relational database, using standard Java Persistence API (JPA) [32]. This module is strictly optional as the computation can also occur purely in-memory which means the persistence is not needed.

4.3 Drools Expert

Drools Expert is one of the building blocks of the entire KIE platform. It is a rule-based production system, typically used to create expert systems. *Expert system* [17] is a computer program that is trying to emulate the decision-making process of human expert. One of the first successful forms of Artificial Intelligence software were the expert systems.

Expert consists of three main components: a production memory, a working memory and an inference engine. Figure 4.2 shows the overall architecture. Production memory is used to store the knowledge the system has. Working memory contains the facts the system is aware of. Inference engine then uses the knowledge and the known facts to derive new facts. Following subsections introduce these components more in detail.

[1] See http://www.tsp.gatech.edu/index.html
[2] See https://github.com/droolsjbpm/drools

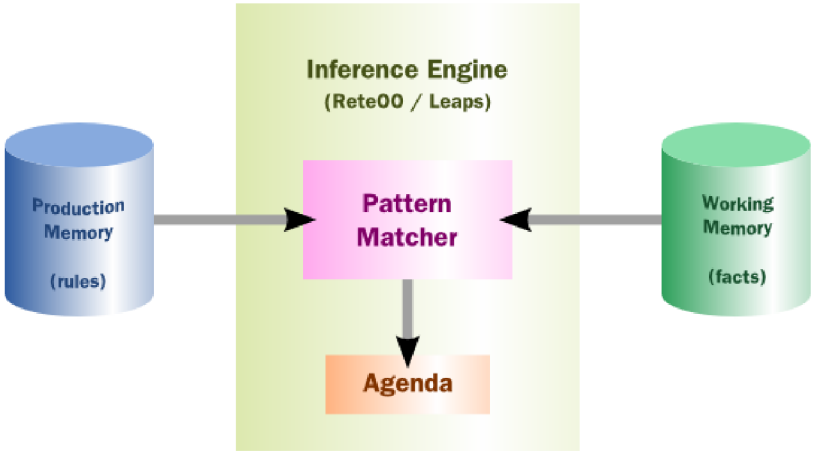


Figure 4.2: Structure of the Expert rule engine. Taken from [30].

4.3.1 Production Memory

Production memory (also known as Knowledge Base) contains the knowledge. As the Expert is rule-based engine, the knowledge is stored as a set of *rules*. Each rule has a form of `when-then` (or sometimes called `if-then`) expressions. Drools is using own notation to specify the rules: Drools Rule Language (DRL) [30]. Various DRL construct are described later in chapter 5.

4.3.2 Working Memory

Working memory holds the facts the engine knows about. The fact can for example be a structure (or object) that holds information about person (name, surname, etc). Facts are being matched against the rules (more about the matching and inference in subsection 4.3.3). The fact types can either be declared as POJOs (Plain Old Java Objects) or inline, directly in DRL file.

Drools (and the whole KIE platform) uses the concept *session* to access the working memory. Two implementations of session exist:

- *Stateless.* Stateless session can be roughly compared to a single function call. The function receives parameters, does some computation and returns a result. In this case the parameters are fact objects, the computation is reasoning about the facts (firing rules) and result is the newly derived facts. Typical use cases for stateless session are validation, filtering or routing.

- *Stateful.* Stateful session typically exists for a longer time. It enables iterative changes to the working memory during the lifetime. As opposed to stateless session the facts can be inserted in multiple steps and after each step the rules can be evaluated. The stateful session needs to be disposed after the usage or the facts in working will be held in memory indefinitely and that will lead to memory leaks. Typical use cases for stateful session are continues monitoring or diagnostics. The stateful session also needs to be used for complex event processing.

4.3.3 Inference Engine

The inference engine is the core component of the rule engine. It is responsible for matching incoming facts onto declared rules and once the rules are fully matched they can be executed (fired).

Two main methods of reasoning are forward chaining and backward chaining. Expert implements both of these techniques.

Forward chaining. Forward chaining is considered as a *data-driven* technique. Data (the facts) are being inserted into the working memory which results in one or more rules being matched and scheduled for execution. The result of the execution is called conclusion [30].

Backward chaining. Backward chaining is so called *goal-driven*. The reasoning starts with an conclusion which needs to be satisfied. If the engine can not satisfy the conclusion, it searches for other satisfiable conclusions, called subgoals, that can help satisfy an unknown part of the current goal. Once the original conclusion is satisfied or there are no more subgoals, the process ends [30]. For example the well-known programming language Prolog [42] uses a backward chaining.

Pattern Matching

Pattern matching is quite general term which describes the act of matching sequence of tokens to some predefined pattern. In rule engines, it is used to match the facts onto the rule conditions. Once all of conditions have been matched the rule is eligible to be executed. There are few well-known patter matching algorithms for rule engines, for example Leaps [3] or Rete [15].

Agenda

Agenda is a structure that holds the rule activations and is responsible for executing them. Once there are two or more activations in agenda, they are said to be in conflict. Determining the order, in which the activations are processed, is called conflict resolution. One of the simplest conflict resolution strategies is based on the concept of *salience*. Salience can be viewed as an priority of the rule. Higher salience means higher priority. The default salience is 0 (if the rule does not specify one). The activations are sorted according to the salience and then executed. When there are two or more activations with identical salience, the order of execution among them is based on the order of rule definitions. Definitions declared sooner would have preference. This change came in Drools 6, in Drools 5 the order was arbitrary in case of identical salience. Expert also supports other conflict resolution strategies. For example based on activation groups or rule flow. See Drools documentation [30] for more details.

4.4 Current Tests in Core Modules

The entire Drools platform already contains a great number of different tests in different modules and describing all of them would not be feasible. We will focus on examining the current state of tests in two core modules, `drools-core` and `drools-compiler`. These two modules are the most important ones as they are the building stones of the entire platform. Other modules like `drools-decisiontables` are also important, but they merely extend

the capabilities of the core modules and are considered optional. The exact version of Drools used for the analysis is `6.1.0.Beta2`.

Following sections also include code coverage statistics. The code coverage has been measured using Cobertura [28], version 2.0.3. Cobertura is an open source Java tool used to calculate the percentage of code accessed by the tests, in other words the code coverage. It uses the Java bytecode instrumentation to setup the tested classes. The instrumented code is then executed during the test run. Data about the exact lines and branches accessed are captured, and later can be used to generate report. By default Cobertura creates a HTML report showing the statistics for line and branch coverage.

4.4.1 Core Engine Tests

Unit Tests

The engine unit tests are located in the `drools-core` module. The Figure 4.3 shows summarized code coverage statistics for all classes in that module. The total line coverage is 17% and branch coverage is 16%. Detailed coverage report for all packages in the `drools-core` module is not presented here and it is quite long. Instead, the Figure 4.4 shows only the report for classes inside a single package, `org.drools.core.phreak`. This package has been chosen, because it contains the implementation classes for the new PHREAK [39] rule algorithm, introduced in Drools 6. As the overall statistics show, the coverage is quite low with only 13% line coverage and 10% branch coverage. Lots of the classes does not even have any unit test coverage at all. Of course the analysis was done using only the tests in the `drools-core` module. Integration tests included in the `drools-compiler` module exercise the engine much more. See the section 4.4.2 for some information about the integration tests for engine and compiler modules.

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	1937	17% 12592/72797	16% 4786/29495	1.97

Figure 4.3: Code coverage summary in the `drools-core` module, v6.1.0.Beta2. All tests in the module were executed.

Performance Tests

There is also about two dozens of performance tests that exercise the rule engine. These tests usually measure, how long it takes the engine to insert facts into the working memory, and then evaluate and execute the rules. The benchmarks are located outside the primary Drools source tree, in repository called `droolsjbpm-integration` [3]. This work focuses on testing the correctness of the Drools rather than its performance, so these tests will not be described. As there is no other documentation, please refer to the source code for detailed information.

[3] See https://github.com/droolsjbpm/droolsjbpm-integration/tree/master/drools-benchmark

Coverage Report - org.drools.core.phreak

Package	# Classes	Line Coverage	Branch Coverage	Complexity
org.drools.core.phreak	31	13% 477/3446	10% 200/1948	5.016

Classes in this Package	Line Coverage	Branch Coverage	Complexity
AddRemoveRule	2% 15/587	2% 7/326	5.658
LeftTupleEntry	0% 0/12	N/A N/A	1
PhreakAccumulateNode	0% 1/324	0% 0/201	7.6
PhreakBranchNode	0% 1/109	0% 0/62	9
PhreakEvalNode	1% 1/62	0% 0/33	5.25
PhreakExistsNode	0% 1/237	0% 0/142	11
PhreakFromNode	0% 1/143	0% 0/71	7.167
PhreakJoinNode	45% 82/182	26% 34/127	7.4
PhreakNotNode	1% 4/237	2% 3/138	9.5
PhreakQueryNode	1% 1/88	0% 0/32	4.4
PhreakQueryTerminalNode	1% 1/66	0% 0/38	4.333
PhreakRuleTerminalNode	46% 58/124	21% 20/92	7.375
PhreakTimerNode	2% 3/147	0% 0/70	2.613
PhreakTimerNode$1	0% 0/4	N/A N/A	2.613
PhreakTimerNode$Executor	0% 0/24	0% 0/6	2.613
PhreakTimerNode$ExecutorHolder	0% 0/2	N/A N/A	2.613
PhreakTimerNode$Scheduler	N/A N/A	N/A N/A	2.613
PhreakTimerNode$TimerNodeJob	0% 0/21	0% 0/8	2.613
PhreakTimerNode$TimerNodeJob$1	0% 0/4	0% 0/2	2.613
PhreakTimerNode$TimerNodeJobContext	0% 0/19	N/A N/A	2.613
PhreakTimerNode$TimerNodeTimerInputMarshaller	0% 0/10	0% 0/2	2.613
PhreakTimerNode$TimerNodeTimerOutputMarshaller	0% 0/4	N/A N/A	2.613
RightTupleEntry	0% 0/12	N/A N/A	1
RuleAgendaItem	27% 6/22	0% 0/4	1.083
RuleExecutor	40% 73/179	26% 33/126	4.16
RuleExecutor$SalienceComparator	0% 0/17	0% 0/8	4.16
RuleNetworkEvaluator	22% 107/467	14% 35/243	6.955
SegmentPropagator	0% 0/45	0% 0/34	6.667
SegmentUtilities	44% 122/272	37% 68/183	5.818
StackEntry	0% 0/26	N/A N/A	1
TupleEntry	N/A N/A	N/A N/A	1

Figure 4.4: Code coverage statistics for classes in package `org.drools.core.phreak`, v6.1.0.Beta2. All tests in the module were executed.

4.4.2 Compiler Tests

The `drools-compiler` module contains at least three different kinds of tests. Besides the unit tests, which would be expected to be side-by-side with the code they are testing, there are also functional tests and even integration tests. All of the tests are located in a single source tree, so distinguishing them is not an easy task. The integration tests, testing the integration with the core engine, reside mainly in its own package called `org.drools.compiler.integrationtests`. However, there seems to be no distinction between where the unit and functional tests are placed. One needs to dive into the source to actually understand the purpose of the test.

Code coverage statistics for the `drools-compiler` module and its packages are presented in the Figure 4.5. All tests in the module (unit, functional and integration ones)

were included in the test run measuring the coverage. The first line of the report shows that the overall line coverage is 57% and branch coverage is 38%. These numbers are already quite high. The Rest of the report shows the same statistics for every package in the compiler module. Some packages, such as `org.drools.compiler.compiler`[4] or `org.drools.compiler.lang.api.impl`, have a very high coverage, 70% or more for both lines and branches. Please note that few classes were excluded from the coverage report as Cobertura was not able to instrument them, saying they contain too large methods[5]. Those classes are `DSLMapLexer`, `DRL5Lexer` and `DRL6Lexer`.

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	660	57%	22504/39197	37%	9415/24988	4.488
org.drools.compiler.builder.impl	2	78%	39/50	57%	8/14	1.478
org.drools.compiler.cdi	12	82%	359/434	77%	162/210	2.38
org.drools.compiler.commons.jci.compilers	34	50%	301/600	39%	81/203	1.939
org.drools.compiler.commons.jci.problems	2	N/A	N/A	N/A	N/A	1
org.drools.compiler.commons.jci.stores	1	N/A	N/A	N/A	N/A	1
org.drools.compiler.compiler	82	73%	2662/3644	70%	1307/1842	3
org.drools.compiler.compiler.io	5	N/A	N/A	N/A	N/A	1
org.drools.compiler.compiler.io.memory	4	67%	255/378	50%	105/208	3.066
org.drools.compiler.compiler.xml	3	75%	189/251	61%	32/52	1.659
org.drools.compiler.compiler.xml.rules	25	91%	419/459	64%	74/114	1.74
org.drools.compiler.kie.builder.impl	67	56%	1893/3370	50%	699/1378	2.047
org.drools.compiler.kie.builder.impl.event	3	71%	5/7	N/A	N/A	1
org.drools.compiler.kie.util	10	48%	89/182	50%	41/82	3.696
org.drools.compiler.kproject	1	64%	36/56	41%	15/36	2.211
org.drools.compiler.kproject.models	22	79%	430/544	62%	82/132	1.49
org.drools.compiler.kproject.xml	7	52%	46/88	30%	17/56	3.333
org.drools.compiler.lang	56	47%	6131/12824	31%	3093/9769	10.066
org.drools.compiler.lang.api	33	66%	2/3	N/A	N/A	1
org.drools.compiler.lang.api.impl	27	90%	351/388	100%	6/6	1.019
org.drools.compiler.lang.descr	91	63%	1216/1928	48%	302/629	1.58
org.drools.compiler.rule.builder	33	86%	1365/1571	79%	701/884	3.973
org.drools.compiler.rule.builder.dialect	5	91%	427/467	90%	231/256	5.606
org.drools.compiler.rule.builder.dialect.asm	38	84%	427/506	89%	34/38	1.413
org.drools.compiler.rule.builder.dialect.java	25	85%	650/762	82%	146/176	2.051
org.drools.compiler.rule.builder.dialect.java.parser	47	45%	4414/9749	23%	2035/8599	11.801
org.drools.compiler.rule.builder.dialect.mvel	24	85%	792/929	79%	230/290	2.694
org.drools.compiler.rule.builder.util	1	85%	6/7	100%	14/14	7

Figure 4.5: Code coverage statistics for the `drools-compiler` module, v6.1.0.Beta2. All tests in the module were executed.

Functional Tests

Most of the functional tests only verify that compiler does not generate any compilation errors for a given DRL definitions. They are not going deeper, checking that the actual

[4]The package name may seem like a typo in the report, with two `compiler` strings in it, but it is the result of package renaming. It should be fixed on the Drools side.

[5]See http://sourceforge.net/p/cobertura/mailman/message/31149799/ for more information

compiled data structures have correct and expected attributes. An example of a typical compiler test is shown on Figure 4.6. The test consists of a rule definition, code that compiles it and finally a check that verifies no errors were encountered during the compilation and thus the compilation was successful. In more detail, the first line just marks the method with an annotation to tell the testing framework that this methods needs to be executed. The second line declares the test method header. The third line starts the rule definition, specified as a plain string. The rule definition ends at line 8. Line 10 contains a call to a common method `compileDrlString(String str)` which transforms the rule definition into an internal representation and return the results of the compilation. Please note that this particular method does not exist in the test code itself. In reality it is often more complicated, but abstracting those complications into a single method call clearly shows the structure of the test and at the same time it does not hide any important information. The next line (11) extracts the errors from returned results. Lines 12 and 13 contain a check to see if no errors were reported. If that is not the case, e.g. the compiler thinks the definition is corrupted or some other error occurred, the entire test is being marked as failed, returning the list of encountered errors together with the failure message.

```
1.    @Test
2.    public void testRuleWithSalienceGetsCompiled() {
3.      String rule =
4.        ‘‘rule ’rule-with-salience’\n’’ +
5.          ‘‘salience 123\n’’ +
6.        ‘‘when\n’’ +
7.        ‘‘then\n’’ +
8.        ‘‘end\n’’;
9.
10.     Results results = compileDrlString(rule);
11.     List<Message> errors = results.getMessages(Message.Level.ERROR);
12.     if (!errors.isEmpty()) {
13:       fail(‘‘Unexpected compilation errors!\n’’ + errors);
14.     }
15.   }
```

Figure 4.6: Example of a typical Drools compiler functional test.

Integration Tests

The `drools-compiler` also contains a big number of tests that basically exercise the compiler as a side-effect. Those tests are primarily used to verify the correct behavior of the rule engine. However, the engine needs to be initialized somehow, and the easiest way is to use the plain DRL constructs, which are then compiled into internal representation the engine can understand. From the nature of these tests, they can be called integration tests, as they exercise both the compiler and the core engine at the same time. There is over 1200 such integration tests. Each test basically consists of three main steps: (1) the knowledge base is created from one or more DRL constructs, (2) knowledge session is obtained, optionally some facts are inserted, and the rules are fired, (3) one or more assumptions about SUT behavior are verified using the assert statements.

4.5 Identifying the Area Suitable for MBT

As the code coverage on Figure 4.3 suggests, the classes in package `org.drools.core.phreak` need more unit tests. At first it appeared that they could be created using the model-based testing. However, after some initial analysis of the requirements and directly the source code, several concerns were raised, as if the model-based testing is the right tool for the job:

- **Poorly specified requirements.** The informal requirements are not systematically defined. Some pieces can be found on community pages, some in blog entries, but the overall document describing the requirements is missing. This makes it difficult to create the model without looking into implementation details. This point is not specific to MBT, it would apply also for the manual test design.
- **Complexity of the algorithm.** The underlying pattern matching algorithm, called PHREAK, is quite complex. It is build on the Rete and ReteOO algorithms and adds custom enhancements. One would need to know all of the predecessors to completely understand it. Creating a model of such algorithm would be very difficult for someone with no prior knowledge of how it actually works. Again, this point applies not only to model-based testing, but also to traditional manual test design process.
- **Insufficient documentation.** The new algorithm is still quite new and the documentation [30] lacks a detailed description of the general design decisions and covers the implementation details only partially (applies to December 2013, the documentation will be most probably updated in near future).
- **The quality of source code.** The lack of requirements and proper documentation brings one to the need to study the source code itself. Unfortunately, some parts of the code exhibit bad practices like extremely long method bodies (several hundreds of lines), too many method parameters (10 and more) or missing comments on proper places. This makes the understanding of the code very hard and time consuming.
- **Stateless nature of the implementation classes.** The implementation classes of the new algorithm are basically stateless, consisting only of methods that operate on their inputs. This is usually not a problem, however in case of model-based testing, the created models would have to specify the SUT's environment (the different method inputs) instead of the SUT behavior. The SUT does not have any state that could be captured by the model. It is certainly possible to create such environment model, but the experience shows that it is much harder [45].

Based on the concerns above, it has been decided not to continue in this direction. In order to create the unit tests one needs detailed knowledge of the underlying algorithm, which unfortunately was not the case. Learning it together with the principles of model-based testing would not be feasible due to time constraints.

The second option, for the model-based testing usage, are the Drools compiler tests Analysis of the current tests showed some weaknesses that could be mitigated by creating functional test suite using the model-based testing:

- **Small number of actual functional tests.** Compiler contains a lot of test cases that exercise the integration with the core engine. It however lacks a comprehensive functional test suite that would primarily exercise and test the compiler and not the engine as a whole.

- **Functional coverage is scattered.** The compiler tests are often created ad-hoc, for example based on the reported issues. The test cases are not designed in way that would systematically cover all the different constructs supported by the compiler. With the help of model-based testing, it should be possible to define and model the individual constructs, and then verify they are all covered by the generated test cases.
- **Small number of negative tests.** Most of the current compiler tests are focusing on the positive input, meaning they are testing that the compiler is able to compile specified construct without any errors. Only small number of tests is actually exercising the compiler from the other direction – making sure it correctly handles malformed and invalid constructs and generates proper error messages.

To sum things up, the model-based testing was found to be not suitable for the core engine unit tests. The compiler functional testing, on the other hand, could benefit from the usage of model-based testing in the areas of centralized functional coverage and negative testing. It has been decided to pursue this idea and create functional test suite with the help of model-based testing.

4.6 Compiler Input

This section focuses mainly on describing the syntax of the Drools compiler input. Introducing the structure and syntax of the DRL rule and the compilation unit is essential for understanding how to create the models which will represent the compiler input.

4.6.1 Structure of Single DRL Rule

Typical DRL rule consists of the rule *name* and three main sections: *attributes*, *conditions*, also called Left Hand Side (LHS) and *actions*, also called Right Hand Side (RHS). The rule name is mandatory and it poses as a unique identifier of the rule. None of the main sections is strictly mandatory. The rule can have only the name and no attributes, conditions or actions. However, for the rule to be useful in some way, it needs to contain at least some actions typically also conditions. The attributes are useful way to alter the behavior of the rule and the engine itself, but they are often not necessary.

The most recent version of DRL (6) supports also additional advanced constructs like *conditional named consequences*, which enable multiple labeled consequences to be defined for single rule. Execution of a particular consequence is then driven based on matched conditions. These constructs are not yet widely used, will not be described here and will not be considered during the testing. More information about them can be found in Drools documentation [30].

Figure 4.7 shows the basic rule structure written in the DRL. The next paragraphs describe each of the rule sections more in detail.

Rule Attributes

Rule attributes act as a metadata about the rule. They influence the rule behavior and instruct the rule engine to treat the rule in a slightly different way based on the concrete attribute and its value. Following list shows all of the attributes supported by Drools. Each list item consists of the name of the attribute, type of its value (e.g. string), brief

```
rule ‘‘Rule name’’
    // attributes
when
    // conditions
then
    // actions
end
```

Figure 4.7: Structure of single DRL rule.

description of the semantics and example of the usage. More detailed information about the attributes and their semantics can be found in Drools documentation [30].

- `no-loop` (<no-value>). Prevents the possible infinite rule re-activation in case the consequence updates the fact(s) the conditions depend on. Only one rule activation will be fired for given fact.

  Example: `no-loop` – marks the rule as one with the no-loop property.

- `lock-on-active` (boolean). Disables (locks) the additional activations of all rules within the same ruleflow or agenda group having this flag set.

  Example: `lock-on-active true` – marks the rule as a one with the lock-on-active property.

- `salience` (integer). Salience acts as a priority of the rule and it is used for conflict resolution when firing rules. Default value (if none specified) is zero. Larger value means greater priority of the rule and thus earlier rule evaluation in case of conflict.

  Example: `salience 42` – sets the salience rule attribute to 42

- `agenda-group` (string). Agenda groups allow partitioning of the agenda and thus provide more execution control. Only rules in specified agenda group that has the focus are eligible to be fired.

  Example: `agenda-group ‘‘processing’’` – marks the rule as a member of “processing” agenda group

- `auto-focus` (boolean). This attribute is used together with agenda-group. When the rule is ready to be fired and its agenda-group does not have focus, the auto-focus will make sure it gains the focus automatically, so that the rule can be actually fired.

  Example: `auto-focus false` – disables the auto focus for particular rule

- `ruleflow-group` (string). Ruleflow [30] is another way to control the execution of the rules. When used, only the rules from single ruleflow-group can be activated.

  Example: `ruleflow-group ‘‘validation’’` – marks the rule as a member of “validation” ruleflow group.

- `activation-group` (string). Rules belonging to the same activation group, identified by the name, can only be fired exclusively (just one rule from such group). The first rule fired will cancel all pending activations of other rules in the same activation group.

Example: `activation-group ‘‘exclusive’’` – marks the rule as a member of “exclusive” activation group.

- `dialect` (string). Code expressions in LHS and RHS of the rule may be specified in different language dialects. This attribute tells the compiler which dialect is used. It can be specified on global (package) level and overridden by individual rules. Currently there are two supported dialects “java” and “mvel”.

  Example: `dialect ‘‘mvel’’` – sets the dialect for the particular rule to “mvel”.

- `date-effective` (string representation of the date). The rule may only be fired after the specified date.

  Example: `date-effective ‘‘01-feb-2013’’` – marks the rule as being eligible to fire only after the 1st of February, 2013.

- `date-expires` (string representation of the date). The rule may only be fired before the specified date.

  Example: `date-expires ‘‘31-dec-2014’’` – marks the rule as being eligible to fire only before the 31st of December, 2014.

- `enabled` (boolean). Simple flag that specifies if the rule is actually enabled and thus eligible to be fired.

  Example: `enabled false` – disables the rule. Such rule will not be fired, even if the conditions are met.

- `duration` (long). Specifies the amount of time in milliseconds to wait for the rule to be fired after it has been fully matched. The rule will be fired after the delay if it is still eligible to.

  Example: `duration 1000` – will cause the rule to wait one second between the time the conditions are met and the time the actions are executed (if the conditions are still met after one second).

- `timer` (string – timer expression). Timers enable repetitive rule executions. Two timer expression formats are supported: the interval based and cron based.

  Example: `timer (int: 30s)` or `timer (cron: * 0/2 * * * ?)` – first one tells the engine to execute the rule actions every 30 seconds, in case the conditions are met. The second one instructs the engine to execute the rule every 2 hours, at midnight, 2am, 4am and so on.

- `calendars` (string). Calendars let users control when the rule can be fired. They use the Quartz format and are fully customizable. At runtime, the string name needs to be bound to the actual implementation.

  Example: `calendars ‘‘weekends’’` – marks the rule as one that is being influenced by the calendar named “weekends”.

Rule Conditions (LHS)

The LHS of the rule contains a list of conditional elements. DRL supports few dozens of different conditional constructs. Following paragraphs describe the most common ones as those will be later modeled. Refer to the Drools documentation [30] for detailed info about the rest of them.

Pattern Pattern is one of the most important conditional elements. It can potentially match any fact inserted into working memory. Pattern has optional binding and contains zero or more constraints. Constraint is basically an expression that needs to return `true` or `false`. Constraint can not be used on its own, it needs to be part of the pattern. As the name suggests, it constraints the possible set of facts that match the particular pattern. The Figure 4.8 shows an example of two patterns. Line 2 contains pattern with binding and no constraints. Pattern on line 3 has two constraints and no binding.

```
      ...
1.    when
2.        $m : Address()
3.        Person(age > 18, address == $m)
4.    then
      ...
```

Figure 4.8: Example of rule patterns usage.

Special DRL Operators DRL supports also several custom operators: `matches`, `not matches`, `contains`, `not contains`, `memberOf`, `not memberOf`, `soundslike`, `str`, `in` and `not in`. They can be used at the place of standards operators.

Temporal Operators Temporal operators are used to model the temporal relations among facts (events). They are used in the complex event processing part of the Drools platform. There is 13 temporal operators supported: `after`, `before`, `coincides`, `during`, `finishes`, `finishedby`, `includes`, `meets`, `metby`, `overlaps`, `overlappedby`, `starts` and `startedby`. The syntax is the same for all the operators. Figure 4.9 shows simple example of rule condition that uses the temporal operator `after`. The rule excerpt captures a condition where the event `$eventB` needs to occur after the `$eventA`. Semantics of the operators is described in the Temporal Operators section of Drools documentation [36].

```
...
when
    $eventA : EventA()
    $eventB : EventB(this after $eventA)
then
...
```

Figure 4.9: Example of rule condition using the temporal operator.

Other Conditional Elements There are other common conditional elements that can be roughly divided into two groups, basic and advanced ones. The basic elements are `and`, `or`, `not` and `exists`. Each of them takes other conditional element(s) as parameter(s) and their semantics is very simple. The `and`, `or` and `not` serve as boolean operators and make sure both conditions are met (`and`), at least one of them is met (`or`) or the condition is not met (`not`). The `exists` condition takes pattern as parameter and returns `true` if at least one fact in the working memory matches that pattern definition (in other words, there exists a fact that can be matched against the pattern).

Besides the basic elements described above, DRL supports also few advanced ones. Their semantics is described below:

- `forall.` The `forall` conditional element returns `true` when all facts that match the first pattern, match also the other patterns.
- `from.` The `from` enables users to specify arbitrary data source for facts. It allows reasoning about data outside of the working memory.
- `collect. collect` allows reasoning about a collection of facts, obtained either from working memory or another data source.
- `accumulate.` The `accumulate` conditional element is basically a more advanced version of `collect`. It can achieve the same results and on the top of that it enables the execution of custom actions for each of the specified elements. It also returns an object as result of the computation.

Rule Actions (RHS)

RHS specifies the actions that needs to be executed once the rule is fully matched. Those actions usually modify facts inside the working memory, causing other rules to be activated. It is recommended that the consequence is kept short and simple and does not contain imperative-like statements.

The RHS can basically contain arbitrary Java code plus one special `modify` statement (introduced later). There is also a special global variable injected into the RHS. It is called `drools` and lets users interact with the rule engine from inside the consequence. The `drools` object is of type `KnowledgeHelper` and contains set of convenient methods, which can be used to manipulate the working memory. The most common ones are: `update()`, `insert()`, `insertLogical()`, `delete()`, `halt()`, `getWorkingMemory()`, `setFocus()`, `getRule()` and `getTuple()`. Even though the names are often descriptive enough and for the purpose of testing the compiler it should be enough to know the syntax, it is certainly useful to introduce at least basic semantics of the methods. Following list contains each of the method with brief description of how they affect the rule engine and the parameters they take as input (if any).

- `drools.update(object, factHandle)` lets the engine know that the `object` has been updated and rules involving that particular object may need to be re-evaluated.
- `drools.update(object)` behaves exactly like the one above, but the `factHandle` is automatically retrieved by the engine.
- `drools.insert(new SomeObject())` places the newly created object into the working memory.
- `drools.insertLogical(new SomeObject())` behaves similarly as plain `insert()`, but once there are no facts to support the truth of the current rule, the engine will automatically remove the inserted object.
- `drools.delete(object)` removes the object from working memory. Again, this may cause the dependent rules to be re-evaluated.

- `drools.halt()` immediately terminates the rule execution, even if there are other rules in the agenda waiting to be executed.
- `drools.getWorkingMemory()` return the `WorkingMemory` object that represents the current contents of the working memory.
- `drools.setFocus(String agendaName)` sets the focus to the specified agenda group.
- `drools.getRule()` returns the rule currently being executed. Useful for logging and debugging.
- `drools.getTuple()` returns the tuple with currently executing rule and the current rule activation. Useful for logging and debugging.

The first four methods (`update`, `insert`, `insertLogical` and `delete`) are so common that the DRL contain macros that let users to call them directly without the `drools` prefix. Fo example, the remove macro will substitute the call to the `remove()` for a `drools.remove()`.

The `modify` Statement The `modify` statement is an extension to the DRL that enables structured fact updates. The same result can be achieved also with the `update` operations, but with more boiler-plate code around. Figure 4.10 shows the general syntax schema of the modify statement. The `<fact-expression` must refer to a fact object reference. The `expression` list contains a setter calls to the given object.

An example of an actual modify statement, in the context of the entire rule definition, is shown in the Figure 4.11. Line 3 specifies a condition that checks if working memory contains a person named "Darth Vader". Overall the rule makes sure the address and mission for this person are set. The `modify` statement itself starts at line 5, specifying the fact reference to work with. Lines 6 and 7 then call the setters of the person object, specifying concrete values for address and mission fields. The statements ends on line 8 with a curly brace.

```
modify ( <fact-expression> ) {
  <expression> [, <expression ]*
}
```

Figure 4.10: Syntax schema of a modify statement [30].

4.6.2 Structure of the DRL Compilation Unit

DRL compilation unit, probably widely known as *rules file*, may contain multiple rules, queries and functions as well as other constructs like imports or globals. It is typically stored in a file (thus the name rules file) with extension `.drl`. The structure of a typical DRL compilation unit is shown in Figure 4.12. Following lines briefly describe each of the construct with focus on the syntax. The semantics is discussed only briefly as it is not that important for compiler tests.

package name specifies a name of the current package. Packages are used as a name space for rules, functions and other constructs and have the same semantics as Java packages.

```
1.   rule ‘‘Update Lord Vader’’
2.   when
3.       $person : Person( name == ‘‘Darth Vader’’)
4.   then
5.      modify($person) {
6.          setAddress(‘‘Death Star’’),
7.          setMission(‘‘Destroy the rebels!’’)
8.      }
9.   end
```

Figure 4.11: Example of a modify statement in the context of entire rule.

```
package <package-name>;

<imports>

<declared-types>

<globals>

<functions>

<queries>

<rules>
```

Figure 4.12: Structure of DRL compilation unit (rules file).

imports bring the specified type/class into the current name space so that it can be referenced by the simple name instead of fully qualified one. Again the semantics is same as for standard Java imports.

declared types are a way to define data structures directly in the DRL. They can also be used to enhance already created POJO (Plain Old Java Object) classes with custom Drools annotations.

globals are global variables that can be accessed from every rule. They can be for example used to specify common logging object.

functions allow to define shared code directly in DRL, instead of using Java classes. The syntax schema of the DRL function is shown on Figure 4.13. The syntax is exactly the same as one would expect in the static Java method. The `function` keyword is followed by the return type and function name. Parameters are specified inside the parentheses, separated by a comma. Function body may contain any Java code valid in the place of function definition.

queries enable support for backward chaining inside the rule engine. They are also able to search the working memory for facts that match the stated conditions. The Figure 4.14 shows a syntax schema for the query. The parameters are in a typical type + name format. The body of the query consists of the same elements that can be used in the LHS of the rule.

```
function <return-type> <name> ( [<paramter>]* ) {
  <expression> [, <expression ]*
}
```

Figure 4.13: Syntax schema of a DRL function [30].

```
query ‘‘<name>’’ ([<paramter>]*)
    [<condtion>]*
end
```

Figure 4.14: Syntax schema of DRL query.

rules represent list of DRL rule definitions. The rule structure has already been introduced in subsection 4.6.1.

Chapter 5

Testing the Drools Compiler

This chapter describes the crucial part of the entire work — the model-based testing of the Drools compiler. First section serves as an introduction, briefly mentioning what steps need to be completed and how should they be connected to make the entire testing process successful. Section 5.2 analyzes the problem more in detail, explaining what parts of the compiler will be tested and how to divide the testing process into smaller tasks. Design and implementation section sheds a light on the data representation and model creation. Test generation and execution is described in Section 5.4. Sections 5.6 and 5.7 present the improvements introduced by this work and also the limitations of current solution. The last section summarizes the lessons learned during the testing process.

5.1 Introduction

Proper testing of the compiler is very important as it is the part of the system that the users will come in contact the first. They define a rule and then expect the compiler to successfully compile it, or return a meaningful error if the input is not valid. Together with the engine, the compiler module is the core part of the entire Drools rules engine.

The testing process will consist of several steps. The first is the analysis of different sub-tasks and possible solutions for them. Then comes the design and implementation of the model and other utility classes. Once the model is ready, the test suite needs to be generated and executed.

5.2 Analysis

The Drools compiler supports many different constructs and input formats. This work focuses on the most widely known and used ones, described in detail in the Section 4.6. Those constructs are a rule definition and a DRL compilation unit. Apart from the rule definitions, the compilation unit may include also imports, globals, declared types, functions and queries. Additional input formats such as decision tables and DSL will no be modeled.

Data Representation It should be beneficial to represent the individual DRL constructs as different implementation classes. The classes could be either mutable or immutable, both have their advantages and disadvantages. The complicated logic should not be stored together with these data classes, but should rather be separated into own classes. This way the logic will not coupled with the simple data-only classes.

DRL compilation There are two possible cases how the compiler can react to the provided input:

- When the input is valid the compiler needs to successfully compile it and create the expected data objects representing the input in a way the rule engine can understand them.
- In case the input is invalid for some reason, the compiler should return meaningful error message(s) describing the compilation errors. The compiler should not crash, or even worse silently accept the malformed input.

Model Decomposition Using the OSMO specific features (mostly annotations) the expected behavior needs to be modeled. Implementing the entire model in a single class is not recommended [20]. The resulting class would be too big and the maintenance would be very painful. Instead, the overall model should be decomposed into smaller ones. Those will be responsible only for a certain part of the compiler input, such as rule attributes or rule conditions. Using the OSMO configuration and setup code, the pieces will be put together just before the test generation and execution.

Negative Testing The first prerequisite for the negative testing is getting the invalid input. Such input can be malformed in multiple places. There are basically two negative input scenarios that differ in the number of malformed constructs. The generated input may contain either single or multiple invalid constructs. We need to consider both cases, because the compiler often does not scan the rest of the input if it encounters an unrecoverable error. It returns the current error message and stops. Considering both scenarios makes sure that all invalid constructs get eventually processed by the compiler and also that the compiler can handle multiple recoverable input errors.

5.3 Design and Implementation

Good software design plays one of the key roles on the path to successfully create a quality software. Design should focus on decoupling the different components of the system and providing clear interfaces between them. Following the decoupling principle also enables easier enhancements in the future as the changes are limited only to affected component/set of components.

5.3.1 Decoupling Data Classes and Model Classes

Before going into the details about how the components should be decoupled, the terms *data class* and *model class* are introduced. For the purposes of this work the data class is a kind of class that is used to store data objects only. These classes are not supposed to do any advanced computation over the data. In Java world such classes are commonly called POJOs. The data classes should be also easily testable as they do not depend on any complicated logic.

The term model class is used in conjunction with a class that contains the tool-dependent constructs (e.g. OSMO annotations), that actually enable the modeling of the expected behavior. Such classes of course typically use the data classes to store the needed information. The may use them indirectly using *model state* which enables even higher level

of decoupling as it abstracts the data classes and provides methods for manipulating the mutable state of the model.

The constructed rule definition and compilation unit are the ideal candidates for the data classes. They are introduced in more in the following subsection. The model classes are then described in subsection 5.3.3.

5.3.2 Data Classes

The data classes are merely used to store the information, without any complex operations (methods) involving the stored data. There are two other names that are often used to classify the classes with a similar purpose: model classes and domain classes. However, in this thesis those names are used in different contexts. Since the work is about model-based testing, the term model class is used to mark the actual model of the expected behavior. The term domain class is used to describe the fact types (classes) that are used inside the rule definitions and over which the rule engine operates. Typical domain objects used for testing are Person, Address or Message.

It is often very convenient and even recommended to make the data classes *immutable*. Immutability means that the objects (instances) can not be changed after they are created. Instead of manipulating the internal state of the object, the methods that are supposed to change some property are returning new instance with the specified property changed. Such methods typically start with the prefix `with` as they will return new instance based on the current one, but with the specified property changed. The main advantage of the immutability is that it greatly simplifies the reasoning about the program code as the created objects will always stay the same after they are instantiated. One of the disadvantages is the fact that with every change there needs to be a new instance created which may in some cases cause performance issues. However, in case of the Drools compiler tests this should not be a problem as there are other tasks like test generation that will be much more computationally intensive and will hide the small overhead of the immutable classes. All data classes used for the Drools compiler testing are immutable.

The data classes are designed to represent certain parts of the rule or other DRL constructs. Each class needs to implement the `toDrlString()` method which converts the data encapsulated by the class into string representation that the compiler understands.

`RuleDef` and Related Classes

The `RuleDef` class can be considered as one the dominant data class in the entire project. It represents the actual rule definition. It is not meant as a fully featured rule with the executable actions and representation ready to be consumed by the rule engine. The fully featured rule is a result of the compilation process. Drools uses the class named `Rule` to represent the executable rules so using the name `RuleDef` makes it easier to distinguish between the two.

The class diagram in Figure 5.1 shows the simplified view of the `RuleDef` class and also other data classes directly related. The `RuleDef` holds the rule name and uses three additional classes to store the data about the rule definition: `RuleAttributes`, `RuleConditions` and `RuleActions`.

`RuleAttributes` The rule attributes are stored as an immutable list of `RuleAttribute` instances. Using the wrapping class to hold the actual data, instead of storing the list

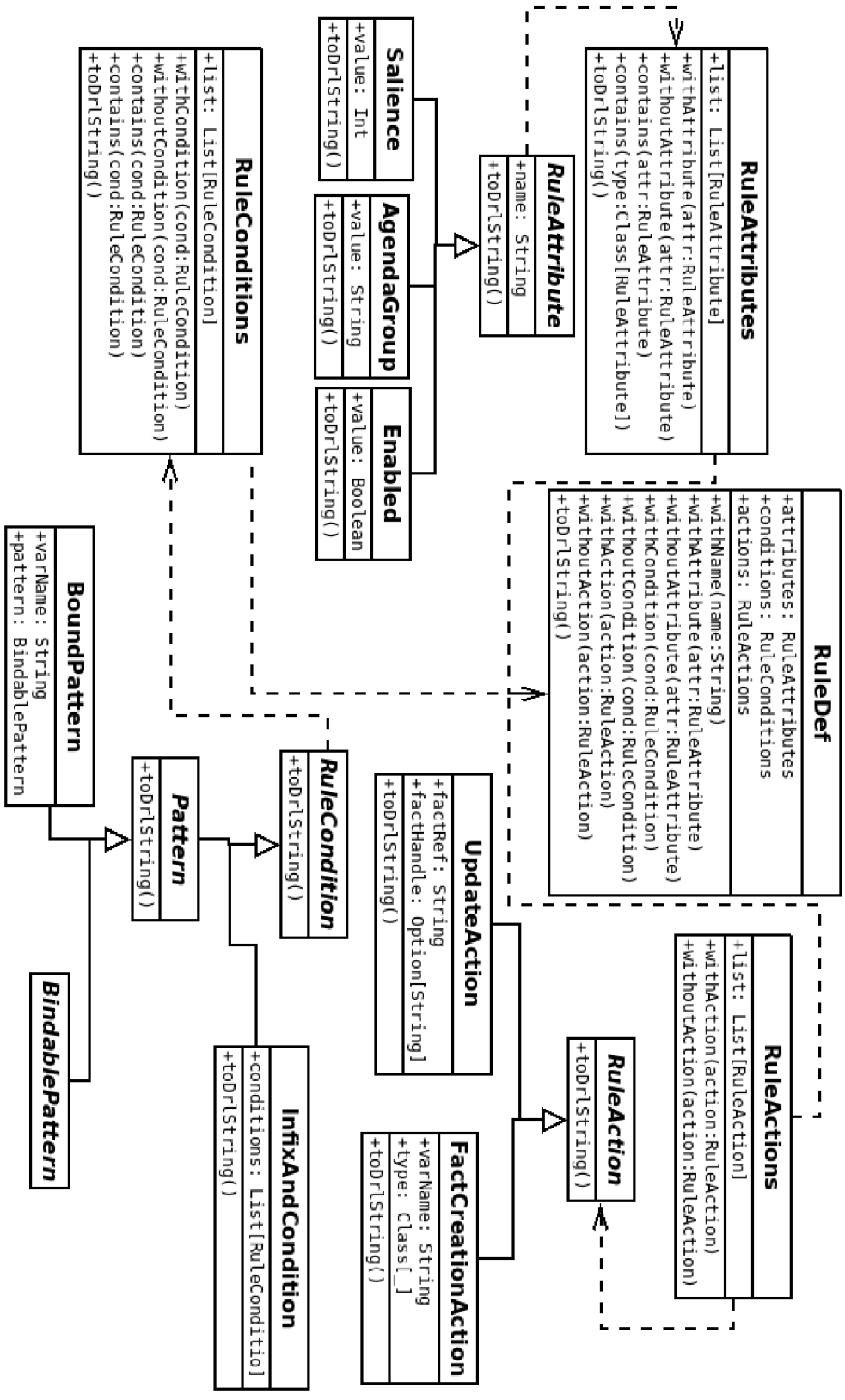


Figure 5.1: Simplified class diagram showing the RuleDef class and related classes.

directly in the `RuleDef` class, makes it easy to place methods operating over the attributes together with the list instead of storing them in the `RuleDef` or some external classes. It also makes the internal representation less exposed as the implementation details are hidden in that class.

The `RuleAttribute` is a common interface that all rule attributes need to implement. The class diagram in Figure 5.1 shows three concrete implementations: `Salience`, `AgendaGroup` and `Enabled`. These classes directly represent the specified attributes. The diagram shows only those three classes, but in fact there is 14 rule attributes that needs to stored this way. Showing them all would only clutter the diagram. Using the class hierarchy to capture the different attributes enables easier pattern matching over the instances and should also improve the readability of the source code as compared to storing the attributes as raw data, for example in some kind of map pointing from attribute name to its value.

`RuleConditions` The `RuleConditions` class represents a list of individual rule conditions. It also contains a set of basic operations over them, such as `contains()`, `withCondition()` or `withoutCondition()`. `RuleCondition` is again just a common interface for the classes that want to act a rule condition. The class diagram in Figure 5.1 shows two concrete conditions `BoundPattern` and `InfixAndCondition`. The `BoundPattern` is a result of binding a `Pattern` to a variable. The pattern is more thoroughly described in section 4.6.1.

`RuleActions` Similarly as with attributes and conditions, the `RuleActions` class serves as a container for the defined rule actions. Actions share a common interface `RuleAction`. They can be divided into three groups: the rule engine helper methods, arbitrary Java code and the `modify` statement. The Figure 5.1 shows examples of the rule action data classes. The implementation of course contains many more of such classes, mirroring the actions described in Section 4.6.1.

`DrlCompilationUnit`

The structure of the compilation unit has been briefly described in Section 4.6.2. Class `DrlCompilationUnit` is merely used to store the underlying data for the compilation unit, such as package name, imports, globals, declared types, functions, queries and rule definitions. The package name is stored directly as a string in the class. The other structures are represented by own data classes.

5.3.3 Model Classes

The purpose of the model classes is to drive the generation of the test cases. They are directly connected to the OSMO as they use the tool specific constructs to define parts of the model. In our case the model classes assemble the compiler's input and verify that the compiler has generated correct outputs or in case of the malformed input valid error messages. In other words, they represent the expected behavior of the compiler. Following list shows the four main model classes created and used for the purpose of the compiler testing:

- `RuleDefModel`. Serves as a container for other model classes that comprise the rule definition: `RuleAttributesModel`, `RuleConditionsModel` and `RuleActionsModel`.

- `InvalidRuleDefModel`. This class models the invalid rule definition. It contains different steps that deliberately corrupt the rule definition in order to determine if the compiler will correctly report errors during the compilation.
- `DrlCompilationUnitModel`. Similarly as with rule definition model, this class serves mainly as a holder for specific model classes like `DrlFunctionModel` or `DrlQueryModel`. Besides that, it directly contains logic to model the package name and imports. Moving those into own model classes would not be much beneficial as they are quite trivial to model.
- `InvalidDrlCompilationUnitModel`. Contains a logic (model) that corrupts the compilation unit on purpose to see if the compiler is capable of identifying such malformed input.

Storing the Model State

Model typically needs to store its state in some way. The naive solution would be to embed the variables describing the state directly into the model class. However, that quickly becomes unmaintainable as the models become bigger. The model then needs to be split into multiple classes and the state needs to be shared. The preferred solution is to abstract the state away into a specific class / set of classes and share the state between the model classes. The state classes are mirroring the model classes, so for example there is a `RuleDefModelState` for a rule definition model and `DrlCompilationUnitModelState` for a compilation unit model.

Model classes that describe the valid and invalid constructs share the same state classes. They have been designed to be flexible enough to be usable for both cases.

5.4 Test Generation and Execution

Test execution and mainly test generation are the tasks where OSMO, as a tool, plays a key role. OSMO needs several inputs in order to begin with the test generation: the actual model classes, the traversal algorithm and conditions that specify when to stop the test case generation.

The Figure 5.2 shows an example of OSMO configuration and test generation code, similarly as used by the compiler MBT tests. The first line creates a new `OSMOTester` object which is the central point of interaction with OSMO. The second line specifies a model factory which is capable of creating the specific model objects. Usage of the model factories simplifies the configuration code, because part of it can be moved to the factory. Line 3 specifies an algorithm that will be used to traverse the FSM (constructed from the models). Lines 4 and 5 define a conditions for terminating a single test and the entire suite. In this case they are based on number of steps performed (for test end) and number of tests generated (for test suite end). The last line (6) instructs the OSMO to begin the test generation. The parameter `System.nanoTime()` servers as a randomized seed.

There are four classes in package `org.drools.compiler.mbt` that contain the OSMO setup code and configuration. Two of them are responsible for generating tests with valid constructs (rule and compilation unit) only and the other two use models for both valid and invalid constructs, to test the compiler's reaction to malformed and invalid input.

By default, when one executes goal `mvn test`, only the unit tests for data and model classes are executed. This is a very convenient way to run the unit tests, to see if recent

```
1.    val tester = new OSMOTester()
2.    tester.setModelFactory(new SimpleRuleDefModelFactory)
3.    tester.setAlgorithm(new BalancingAlgorithm)
4.    tester.setTestEndCondition(new Length(100))
5.    tester.setSuiteEndCondition(new Length(150))
6.    tester.generate(System.nanoTime())
```

Figure 5.2: Example of OSMO configuration and test generation code.

changes have broken some existing functionality or not. The model-based testing is triggered with the help of Maven profile. The POM declares a profile called `mbt`. Running the MBT tests is very easy, all that is needed is to tell the Maven to use the profile: `mvn test -Pmbt`.

5.5 Code Coverage with the MBT Test Suite

The code coverage of the compiler code has been measured again, together with the generated model-based tests, to see if the new MBT test suite increases the coverage and how much in case it does. The Figure 5.3 shows the overall coverage report. Both the line and branch coverage have been slightly increased. The line coverage is 58% which is one percent more, and branch coverage is 39% which is two percent more. The package with the highest increase is `org.drools.compiler.rule.builder.dialect.java.parser` which is responsible mainly for parsing the rule actions.

The increase in code coverage may seem low. However, the compiler already had more than two thousand tests and high coverage. The common constructs tested by the MBT suite already had a manual tests for them in most cases. The new test suite brings the advantages of model-based testing: the test process is centralized around the models are the covered DRL constructs are easily traceable. This makes the MBT test suite very systematic in terms of the actual coverage of the DRL constructs.

5.6 Achieved Improvements

This Section summarizes the added value and improvements introduced by this work. It discusses the creation of functional test suite, introduction of systematic negative testing and mentions the reported bugs and issues classified as potential bugs.

5.6.1 Functional Test Suite

Creating the functional test suite was one of the goals of the entire process. The test suite was successfully generated and already covers the widely used DRL constructs. The nature of the models also enables easy addition of other constructs in future. The measurement proved that the test suite already slightly increases both the line coverage and branch coverage of the compiler code.

5.6.2 Negative Testing

One can argue that the negative testing is almost as important as positive testing. The current Drools compiler test suite contains only a limited number of negative tests, and

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	660	58%	23021/39197	39%	9753/24988	4.488
org.drools.compiler.builder.impl	2	78%	39/50	57%	8/14	1.478
org.drools.compiler.cdi	12	82%	359/434	77%	162/210	2.38
org.drools.compiler.commons.jci.compilers	34	74%	445/600	72%	148/203	1.939
org.drools.compiler.commons.jci.problems	2	N/A	N/A	N/A	N/A	1
org.drools.compiler.commons.jci.stores	1	N/A	N/A	N/A	N/A	1
org.drools.compiler.compiler	82	73%	2672/3644	71%	1315/1842	3
org.drools.compiler.compiler.io	5	N/A	N/A	N/A	N/A	1
org.drools.compiler.compiler.io.memory	4	67%	255/378	50%	105/208	3.066
org.drools.compiler.compiler.xml	3	75%	189/251	61%	32/52	1.659
org.drools.compiler.compiler.xml.rules	25	91%	419/459	64%	74/114	1.74
org.drools.compiler.kie.builder.impl	67	56%	1897/3370	51%	704/1378	2.047
org.drools.compiler.kie.builder.impl.event	3	71%	5/7	N/A	N/A	1
org.drools.compiler.kie.util	10	48%	89/182	50%	41/82	3.696
org.drools.compiler.kproject	1	64%	36/56	41%	15/36	2.211
org.drools.compiler.kproject.models	22	79%	430/544	62%	82/132	1.49
org.drools.compiler.kproject.xml	7	52%	46/88	30%	17/56	3.333
org.drools.compiler.lang	56	47%	6149/12824	31%	3104/9769	10.066
org.drools.compiler.lang.api	33	66%	2/3	N/A	N/A	1
org.drools.compiler.lang.api.impl	27	90%	351/388	100%	6/6	1.019
org.drools.compiler.lang.descr	91	63%	1220/1928	48%	304/629	1.58
org.drools.compiler.rule.builder	33	87%	1370/1571	79%	704/884	3.973
org.drools.compiler.rule.builder.dialect	5	93%	438/467	90%	232/256	5.606
org.drools.compiler.rule.builder.dialect.asm	38	84%	427/506	89%	34/38	1.413
org.drools.compiler.rule.builder.dialect.java	25	85%	653/762	84%	149/176	2.051
org.drools.compiler.rule.builder.dialect.java.parser	47	48%	4732/9749	26%	2273/8599	11.801
org.drools.compiler.rule.builder.dialect.mvel	24	85%	792/929	79%	230/290	2.694
org.drools.compiler.rule.builder.util	1	85%	6/7	100%	14/14	7

Figure 5.3: Code coverage statistics for the `drools-compiler` module, v6.1.0.Beta2, including the model-based test suite.

they are scattered all over the test suite so one can not easily say what is and what is not covered by them.

One of the outputs of the work are models that describe the malformed DRL constructs. They introduce a generic and systematic way to define invalid rule attributes, conditions, actions or any other construct from the DRL compilation unit. These models set a good base for negative testing. They already contain several tens of transitions (steps) and the classes are ready for additional extensions. Several issues related to the negative testing were already found and reported. They are described in more detail in the Section 5.6.3.

5.6.3 Reported Bugs

During the test development (model creation) and test execution, few issues have been found and reported as a bugs in the Drools compiler.

Duplicated Attributes Not Reported as Error

Declaring the same rule attribute multiple times does not make any sense and only makes the rule definition ambiguous, as it is not clear which one will be actually used. The compiler should report this as an error. The issue is reported in Red Hat Bugzilla under ID 1092084[1] and has already been fixed by the development team.

Multiline Rule Actions Broken

The issue manifests itself if one declares single rule action over multiple lines. It is reported in Red Hat Bugzilla under ID 1092502[2]. By default the compiler will place a semicolon at the end of each line, which then makes the multiline rule actions invalid. Example of such action is shown in the Figure 5.4. The presented example is of course trivial and probably no one would ever use it like this. However, there are practical cases where dividing the single action into multiple lines makes sense. For example, the method calls to builder class, used to build complex objects, are recommend to be placed one per line, as it greatly improves the readability.

```
...
then
  System.
    out.
    println(‘‘Rule executed!’’);
end
```

Figure 5.4: Example of a single rule action spanning multiple lines.

Hex and Oct Literals Not Recognized by `mvel` Dialect

The rule may use one of the two dialects: `mvel` and `java`. In case the rule specifies `mvel`, and the consequence contains hexadecimal or octal number literals, the compiler will return error saying it can not process the number literal. Those literals should be supported even for the `mvel` dialect. The issue is reported in Red Hat Bugzilla under ID 1098825[3].

Duplicated Annotations on Declared Type Not Reported As an Error

One can specify different metadata on declared type, using the annotations such as `@role` or `@expires`. As of now, the compiler will not raise an error if one annotation is declared multiple times. This makes the rule definition ambiguous. The issue is reported in Red Hat Bugzilla under ID 1099896[4].

Attributes `date-effective` and `date-expires` accept invalid day in month

Rule attributes `date-effective` and `date-expires` take as an parameter a date literal, for example `28-May-2014`. However, if the date contains an invalid day in month, the compiler

[1] https://bugzilla.redhat.com/show_bug.cgi?id=1092084
[2] https://bugzilla.redhat.com/show_bug.cgi?id=1092502
[3] https://bugzilla.redhat.com/show_bug.cgi?id=1098825
[4] https://bugzilla.redhat.com/show_bug.cgi?id=1099896

won't raise an error and the rule gets compiled successfully. This is of course not desirable as then the rule execution would probably fail at runtime. Compiler should detect these kind of errors and properly report them. The issue is reported in Red Hat Bugzilla under ID 1100254[5].

5.6.4 Potential Bugs

This section briefly mentions issues found during the testing, which may or may not be real bugs. In near future, these issues will be discussed with the developers to determine their status and if they should be fixed.

Multiple Nested `exists` Conditions

The `exists` serves as way to determine if certain condition is satisfied within the entire working memory. Using a nested `exists`, for example `exists (exists (String()))`, does not make much sense. The construct is syntactically correct of course. The question is if the compiler should be smart enough to do the advanced semantic analysis and let user know that such condition is useless, or simply let the users worry about the correct usage of the conditions and do not report anything.

Unknown Annotations on Declared Types

The declared types may optionally contain one or more annotations, such as `@role` or `@duration`. The compiler and engine understands predefined set of the annotations. However, if one specifies an annotation that is not known, for example by a mistake, the compiler will not report any errors. There may be a reason for this behavior. The compiler could ignore the unknown annotations in believe that some external annotation processor would understand them. It needs to be discussed in detail with developers to find out if this is indeed expected behavior or simply a bug.

Generics Not Supported in Function Parameters

The DRL function should be an easy way to define utility method without the need to create class. It should support the same constructs as a static Java method. Currently, the compiler can not correctly parse the function parameters if they are using generics, e.g. `List<String>`. Unless there is a good reason for this behavior, it seems to be a bug.

5.7 Limitations

Even though the generated test suite already covers a number of uses cases and helped to uncover several bugs, there are certain limitations, that would be useful to eliminate in the future. These limitations are described next.

5.7.1 No Offline Testing

As of now, the tests can be executed only by the means of online testing. The test generation and execution are both implemented together in the OSMO model classes. The offline

[5] https://bugzilla.redhat.com/show_bug.cgi?id=1100254

testing support is missing. Both have their advantages and disadvantages, so having the option to choose from both would definitely be a step forward.

5.7.2 Only Single-Threaded Execution

The test generation and execution currently runs in a single thread, utilizing only one CPU core. The OSMO does not provide any out-of-the-box means to run the generation in parallel. Using more than just one thread would decrease the overall testing time. One solution would be to use the offline testing – let the OSMO generate the tests using only one thread and then execute the tests in parallel.

5.7.3 Only Basic Compilation Verification

Verifying that the compiler actually creates a valid internal representation of the specified DRL construct proved to be much harder to implement than anticipated. Right now the tests will only make sure the compiler did not return error messages for valid DRL constructs, and of course returned error messages in case the DRL construct is deliberately made invalid. In case of the valid rule definitions the values of individual attributes are also checked. The internal data structures representing the rule conditions and actions are not currently being verified against the expected rule definitions.

5.8 Lessons Learned

This section summarizes and presents lessons learned through out the entire testing process. They reflect a personal opinion gained from applying the model-based testing and using the OSMO MBT tool.

5.8.1 Potential of Model-Based Testing

Based on the results in this work, the model-based testing proved to be a useful technique. The code coverage of the Drools compiler has been slightly increased and several issues were reported as defects in the compiler.

The existing compiler test suite already has around two thousands of tests. It has been developed and maintained by few Drools developers for several years (it is hard to say exactly). It would be very hard to estimate, how much effort was spent on creating and maintaining the tests, so the comparison with model-based functional test suite, in terms of effort, would not be beneficial. However, even with the limited time for the model-based testing, several issues were reported and others marked as potential defects. As the DRL features related to the issues have been present in the compiler for a long time, it is safe to assume that the bugs were introduced also a long time ago. Without the functional model-based test suite it would be hard to systematically locate the constructs which are missing the coverage.

Use cases such as the presented Drools compiler functional testing can surely benefit from the usage of model-based testing. Designing the models, instead of manually creating the tests, of course needs some getting used to. However, after the initial learning period, the model design becomes straightforward. The results also show that model-based testing is able to find additional defects, and if properly applied it can also decrease the effort needed for testing.

5.8.2 Choosing a Suitable SUT

Before diving into the entire model-based testing process, one should carefully choose the SUT. There are cases (e.g. installer testing) where other testing techniques may be much more appropriate than the model-based testing. As noted in the Practical Model-Based Testing [45], the ability to decide when to use model-based testing and when not to, is learned over time mostly through experience.

Nowadays, the model-based testing is mainly used to create functional test suites. As discussed in the Chapter 4, the unit tests could be created using the model-based testing, but it would require significantly more effort. The SUT requirements should be also clearly specified, which often may not the case for the small units of code. They are using lower level APIs that may be hard to map to the actual requirements. On the other hand, it should be easy enough to map the requirements to the specific functional tests. From the own experience and several surveys mentioned in the Practical Model-Based Testing [45], the functional tests are still the preferred area for model-based testing adoption.

For model-based testing to be effective the SUT also needs to have a certain level of complexity. Using the MBT for trivial applications (e.g. `max` function) brings the initial setup and configuration overhead, and because the models describe trivial behavior they bring only a little value. Model-based testing is best used for larger applications with non-trivial logic.

5.8.3 Maturity of the OSMO

Even though the OSMO is actively developed and maintained, and already contains a lot of features, from our experience it is still not ready for a widespread, mainstream usage. Following few paragraphs summarize the problems and complications encountered during the Drools compiler testing.

The OSMO artifacts (jar files) are only accessibly as part of a `zip` distribution and they are not deployed to any public Maven repository (e.g. Maven Central[6]). Declaring the OSMO as a dependency in a Maven project becomes difficult. One needs to manually copy (install) the artifacts into the local Maven repository. There are two options how to do that. The first one is to manually compile the OSMO from sources and copy (install) the created artifacts. The second one is to copy the jar files bundled inside the `zip` distribution. Both tasks are possibly error prone and should not be necessary. Average Java programmer would not expect to do these additional steps in order to use the tool.

Another area which could benefit from further enhancements is the annotation based model definition. Most Java programmers already know the annotations and how to use them, so it is easy for them to get started. However, with the increased number of test steps (transitions) the model creation may become cumbersome. One needs to create two methods with annotations for each test step and its condition, and link them together using a string name. Unfortunately this also leads to issues during refactoring. The change of the name in one of the annotations is not propagated to the other. The result is a broken link. OSMO will report runtime error early in the test run, but it is still an inconvenience from the model designer point of view.

Based on personal experience, OSMO is still the best tool from the ones that were considered and compared. If one is able to overlook or work around the mentioned complications it can still be very much usable.

[6] http://search.maven.org/

5.8.4 Combining Scala and Maven

The last point is not directly related to the model-based testing, but rather to the other tools and technologies that were used during the testing process. Event though the basic integration between Scala and Maven works correctly, some additional issues were identified. The Scala compiler is considerably slower than the Java one, so compiling the code usually takes more time. In purely Scala world (using the SBT as a build tool), this disadvantage is countered by the so-called incremental compilation. SBT is able to start a compilation server on background and recompile only pieces of code that actually changed. However, the Maven plugin for Scala does not have such feature. It basically compiles the entire source tree again, even if only one class changes. This results in a increased compilation time.

Based on the experience gained though out the entire testing process, the usage of Maven together with Scala was not the best choice. The SBT (native Scala build tool) would most probably be a better alternative, even thought is also has its drawbacks.

Chapter 6

Conclusion

The goal of this thesis was to study the model-based testing in context of Drools and use it to generate a test suite for certain suitable part of the Drools. After the model-based testing was introduced, five tools that support the MBT were described and compared. One of them, OSMO, was chosen as the one that will be used to aid the testing process. After the analysis of the current Drools tests, the Drools compiler has been chosen as a module for which to create the functional test suite with the help of model-based testing. Using OSMO, several models were created, capturing different features of the compiler. The models were then used to generate the functional test suite. The tests were executed together with the current compiler tests to determine if the code coverage has been increased. Even though the coverage was already quite high, the functional test suite slightly increased both line and branch coverage.

During the model design and test execution, several Drools compiler issues were found and reported. They were mostly related to the negative input, showing that the compiler did not handle malformed constructs correctly in some cases and accepted them without generating any error. Besides the reported issues, there were identified also three potential issues. Those will need deeper analysis and discussion with developers to determine if they are indeed defects or rather features. The test suite is planned to be used, maintained and expanded also in the future.

The current test generation process also has some limitations. The part responsible for checking the compiler output takes into account only the messages generated by the compiler and basic rule characteristics. The actual compiled rule conditions and actions are not checked against the original rule definition. The test generation and execution is currently only single-threaded, which makes the overall testing time longer. Lastly, the current implementation supports only online testing.

6.1 Future Work

The limitations set a base for the future work. Improving the compilation verification by deeply analyzing the compiler output and comparing it to the generated rule definition is probably the most important one. Another useful feature would be the offline testing. Being able to generate hundreds of tests once and then execute them periodically would decrease the overall time needed to run the tests.

There are also other options how to improve the current implementation. The models were created only for a DRL rule and a compilation unit. Drools supports also other kinds

of input formats such as decision tables, user created DSLs or even score cards. The future tasks could focus on creating models of these alternative inputs and generating test suite also for them.

The last suggested improvement is not directly connected to the Drools, but rather to the OSMO. The current annotation based model definition is simple and well known, but it has some drawbacks. Since the step name needs to be declared as string, we are loosing the type safety. Also, one needs to create a method for every guard and test step, which may become cumbersome. Creating a custom OSMO DSL, for example implemented in Scala, should simplify the model creation even more, and at the same time help with the boiler-plate nature of the annotation based solution.

Bibliography

[1] Bernhard Aichernig, Willibald Krenn, Henrik Eriksson, and Jonny Vinter. D 1.2 - state of the art survey - part a: Model-based test case generation. Technical report, 2008.

[2] Pretschner Alexander and Phillips J. *Model-based testing of Reactive Systems (Lecture Notes in Compureter Science, vol. 3472)*. Springer, 2005.

[3] Don Batory. *The LEAPS Algorithm*. Technical report, Austin, TX, USA, 1994.

[4] Axel Belinfante. *JTorX: a Tool for On-Line Model-Driven Test Derivation and Execution*. URL: http://eprints.eemcs.utwente.nl/17751/01/main.pdf. Accessed [2014-01-12].

[5] Eddy Bernard, Bruno Legeard, Xavier Luck, and Fabien Peureux. Generation of test sequences from formal specifications: Gsm 11-11 standard case study. *Softw. Pract. Exper.*, 34(10):915–948, August 2004.

[6] Robert V. Binder. *Open Source Tools for Model-Based Testing* [online]. URL: http://robertvbinder.com/open-source-tools-for-model-based-testing/. Accessed [2014-05-18].

[7] Barry W. Boehm, Terence E. Gray, and Thomas Seewaldt. Prototyping vs. specifying: A multi-project experiment. In *Proceedings of the 7th International Conference on Software Engineering*, ICSE '84, pages 473–484, Piscataway, NJ, USA, 1984. IEEE Press.

[8] Fabrice Bouquet and Bruno Legeard. Reification of executable test scripts in formal specification-based test generation: The java card transaction mechanism case study. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 778–795. Springer Berlin Heidelberg, 2003.

[9] Fabrice Bouquet, Bruno Legeard, Fabien Peureux, and Eric Torreborre. Mastering test generation from smart card software formal models. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 70–85, Berlin, Heidelberg, 2005. Springer-Verlag.

[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[11] Sonatype Company. *Maven: The Definitive Guide, 1st Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2008.

[12] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.

[13] Dimitris Dranidis. *JSXM Manual*. URL: http://www.jsxm.org/files/JSXM%20Manual_v1.2.0.pdf. Accessed [2014-01-12].

[14] Eitan Farchi, Alan Hartman, and Shlomit Pinter. Using a model-based test generator to test for standard conformance. *IBM Syst. J.*, 41(1):89–110, January 2002.

[15] Charles L. Forgy and Susan J. Shepard. Rete: A fast match algorithm. *AI Expert*, 2(1):34–40, January 1987.

[16] Amir Ghahrai. *Test Policy Document* [online]. URL: http://www.testingexcellence.com/test-policy-document/. Accessed [2014-05-13].

[17] Joseph C. Giarratano and Gary D. Riley. *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 2005.

[18] Dorothy Graham, Erik Van Veenendaal, Isabel Evans, and Rex Black. *Foundations of Software Testing: ISTQB Certification*. Intl Thomson Business Pr, 2008.

[19] Hubert Klein Ikkink. *Gradle Effective Implementation Guide*. Packt Publishing, 1st edition, 2012.

[20] Teemu Kanstrén. *OSMO User Guide* [online]. URL: https://osmo.googlecode.com/files/osmo-guide.pdf. Accessed [2014-01-11].

[21] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[22] Tim Koomen, Leo van der Aalst, Bart Broekman, and Michiel Vroon. *TMap Next, for Result-driven Testing*. UTN Publishers, 2006.

[23] Steve Loughran and Erik Hatcher. *Ant in Action: Java Development with Ant, Second Edition*. Manning Publications Co., Greenwich, CT, USA, 2007.

[24] Zoltán Micskei. *Model-based testing overview, tools and projects* [online]. URL: http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html. Accessed [2014-05-18].

[25] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition*. Artima Incorporation, USA, 2nd edition, 2011.

[26] Team of authors. *Business Process Model And Notation (BPMN), version 2.0* [online]. URL: http://www.omg.org/spec/BPMN/2.0/. Accessed [2014-05-13].

[27] Team of authors. *Ceylon JVM language* [online]. URL: http://ceylon-lang.org/. Accessed [2014-05-10].

[28] Team of authors. *Cobertura pages* [online]. URL: http://cobertura.github.io/cobertura. Accessed [2014-04-25].

[29] Team of authors. *Drools – Business Logic Integration Platfrom pages* [online]. URL: http://jboss.org/drools. Accessed [2014-04-11].

[30] Team of authors. *Drools 6.1.0.Beta2 Documentation* [online]. URL: http://docs.jboss.org/drools/release/6.1.0.Beta2/drools-docs/html_single/. Accessed [2014-05-11].

[31] Team of authors. *Graphwalker Homepage* [online]. URL: http://graphwalker.org/. Accessed [2014-05-18].

[32] Team of authors. *Java Persistence API pages* [online]. URL: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html. Accessed [2014-04-25].

[33] Team of authors. *Kotlin JVM language* [online]. URL: http://kotlin.jetbrains.org/. Accessed [2014-05-10].

[34] Team of authors. *ModelJUnit Homepage* [online]. URL: http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/. Accessed [2014-05-18].

[35] Team of authors. *SBT Documentation* [online]. URL: http://www.scala-sbt.org/. Accessed [2014-05-18].

[36] Team of authors. *Temporal Operators – Drools 6.1.0.Beta2 Documentation* [online]. URL: http://docs.jboss.org/drools/release/6.1.0.Beta2/drools-docs/html_single/#d0e10425. Accessed [2014-05-11].

[37] Team of authors. *Workflow - This is how you would work using GraphWalker* [online]. URL: http://graphwalker.org/documentation/workflow-this-is-how-you-would-work-using-graphwalker. Accessed [2014-01-12].

[38] Mark Proctor. *IEEE Sofware Engineering Body of Knowledge* [online]. URL: http://www.swebok.org. Accessed [2014-05-10].

[39] Mark Proctor. *R.I.P. RETE time to get PHREAKY* [online]. URL: http://blog.athico.com/2013/11/rip-rete-time-to-get-phreaky.html. Accessed [2014-05-10].

[40] Harry Robinson. Obstacles and opportunities for model-based testing in an industrial software development. *First European Conference on Model-Driven Software Engineering*, 2003. [online]. URL: http://www.harryrobinson.net/ObstaclesAndOpportunities.pdf.

[41] Mauricio Salatino. *jBPM Developer Guide*. Packt Publishing, 2010.

[42] Leon Sterling and Ehud Shapiro. *The Art of Prolog (2Nd Ed.): Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 1994.

[43] Keith Stobie. Model based testing in practice at microsoft. *Electron. Notes Theor. Comput. Sci.*, 111:5–12, January 2005.

[44] Jan Tretmans. *Formal Methods and Testing.* chapter Model Based Testing with Labelled Transition Systems, pages 1–38. Springer-Verlag, Berlin, Heidelberg, 2008.

[45] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann, 2006.

[46] Mark Utting, Alexandar Pretschener, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, pages 297–312, 2011.

Appendix A

Contents of the Compact Disc

- **latex-report/** — directory with the LaTeX sources of this thesis
- **code-coverage-reports/** — directory with the detailed HTML based coverage reports for `drools-core` and `drools-compiler` modules
- **drools-compiler-mbt/** — directory with the created MBT tests for `drools-compiler` (Maven project with models, data classes, etc)
- **drools/** — directory with the source code of the Drools platform, version 6.1.0.Beta2
- **maven-repo/** — local Maven repository with all the artifacts required to run the tests
- **README.txt** — text file with instructions how to execute the MBT test suite and how to measure the code coverage statistics