Master's thesis, 30 credits

Expert System for Error Analysis

Rule Based Reasoning Applied on Log Information and Dump Reports from Mobile Phones

Daniel Gustavsson
Daniel Molin

LUND UNIVERSITY

June 15, 2010

Abstract

We have developed an expert system called ADLA, which stands for Automatic Dump and Log Analyzer. It uses rules to perform an analysis on test reports, log files and dump files that are generated when ST-Ericsson runs tests on their mobile platforms. In addition to the requested knowledge already written as rules, more can be added to further increase the competence of the system.

The analysis is completely automatic and is started by the test tool after the test has been performed. The result is generated as a graph of rules and facts, used to reach the conclusions. ADLA has a graphical part, which displays the graph and allows the user to interact with it. The advantage with this is that one can see how the conclusions were reached, and that many details of the facts and rules are directly available to the user.

Preface

ADLA is the name of the expert system we have built. It is separated into two parts where the first handles the reasoning and the second displays the graphical representation of the result. The whole development process is described in chapters 4 through 6. Below we go through the full outline of this report and give a short summary of our respective contributions.

Report Outline

The problem we tried to solve for ST-Ericsson is presented together with our goals for the project in chapter 1. Our research about expert systems and its components is shown in chapter 2. In chapter 3, an overview of the systems surrounding ADLA is found, which tells how the test tool JATT communicates with ADLA and how the rule engine Drools is integrated.

Chapter 4 reflects the work with constructing ADLA's reasoning part. This includes how the system gets its input, how the data is handled internally and how a result is produced. The result is saved to a graph that the visual part of ADLA reads. This part is described in chapter 5 and includes the graphical user interface and the framework JUNG that is used for drawing. The main problems and solutions are gone through in chapter 6. This chapter can be skipped if the reader is not interested in the details.

Chapter 7 summarizes the features in ADLA and what happens when it runs. Our conclusion, containing evaluation of goals together with suggestions for future work, is presented in chapter 8. The appendix Manual and Guidelines is intended as an introduction manual for ST-Ericsson and describes some typical use cases and helpful tips.

Contributions

This part is a requirement from LTH and summarizes overall our respective focuses. Daniel Gustavsson has focused on the graph structure, error handling and researching JUNG. Daniel Molin has focused on graphical features, rule design and researching Drools.

Contents

Abbreviations

ADLA	Automatic Dump and Log Analyzer, this is the system we have developed.
CTS	Compatibility Test Suite. Android CTS is an open-source test harness that tests compliance with the Android compatibility definition.
DRL	Drools Rule Language, format used for the rules. DRL files are the files containing the rules.
JATT	Java Automatic Test Tool, computer software developed by ST-Ericsson for testing mobile phones.
JDTS	Java Device Test Suite, test suite from Sun which ST-Ericsson uses to perform Java ME quality tests on mobile phones.
JSR	Java Specification Request, describes a specification for the Java platform.
JUNG	Java Universal Network/Graph, a framework we use to visualize graphs.
OPA	Open Platform API, a proprietary mobile platform API from ST-Ericsson.
TCK	Technology Compatibility Kit, suite of tests which checks compliance with a JSR.
XSLT	Extensible Stylesheet Language Transformations, used for transforming XML documents into other XML documents, e.g. XHTML.

Chapter 1

Introduction

1.1 Background

This project was made at Lund University, Faculty of Engineering (LTH), department of Computer Science. It involves developing an expert system for ST-Ericsson, intended for error analysis. Examiner is associate professor Jacek Malec at the department of Computer Science. The work was carried out at ST-Ericsson in Lund, Sweden.

1.1.1 ST-Ericsson

ST-Ericsson develops mobile platforms used in many of today's mobile phones. The headquarters is located in Geneva, Switzerland and they are established in more than 20 countries. ST-Ericsson was formed in 2009 as a joint venture between STMicroelectronics and Ericsson, and has about 8,000 employees.

Our contact at ST-Ericsson, Pernilla Lundström, handled the first introduction for the project. During the work, Pascal Collberg and Magnus Karlsson were our supervisors.

1.1.2 Problem Description

As a step in development, the mobile platform needs to be tested. For this purpose ST-Ericsson runs different test suites to verify the platform. These are automatically run by their test software JATT. The result consists of numerous files containing data from the tests. Because JATT only performs a basic analysis on the files, they often need to be read manually to identify the problem if a test failed. Depending on the problem, this can be more or less complex and sometimes require a great deal of expertise. ST-Ericsson requested an expert system to automatically detect the most common problems and thereby decrease the work required from the users. This is especially helpful to new co-workers because many novice questions can be avoided. With an expert system, the experts can gradually update the knowledge base with new problems, to increase its area of competence.

1.2 Aims and Goals

The list below summarizes the requirements from the initial project description and a complementary meeting with ST-Ericsson. In the meeting it was also discussed that the system could be either an integral part of JATT or a stand-alone system. Its output should be in XML format to be able to automatically transform it into a web page later, using XSLT.

- Design simple input interface between expert system and outside world.
- Implement expert system for memory leak detection and dump report analysis.
- With help from Java developers design output format for the expert system.
- It must be easy to add new knowledge to the expert system, e.g. new dump failure reasons.
- It must be possible to add new areas of competence to the system, e.g. device log analysis.
- The results from the expert system shall be displayed on a web page.
- Should be able to read the result from test suites.
- Should be able to read reg-dumps.
- Should be able to explain the error codes from OPA.

1.3 Target Audiences

ADLA's main target audience is the developers who need to run tests on the platform in mobile phones with JATT, one of ST-Ericsson's test tools. The people who develop JATT itself are also included because they are the ones who will be writing rules and further develop the expert system, henceforth called rule developers. The users that do not make any changes in the system are simply called users.

At the end of the project, we were informed about another user group, which runs JATT as a part of another test tool. These users are only interested in if the test job was successful or not, unlike the developers who also want to know why a test failed.

1.4 Boundaries

One clear limitation is that the idea of writing our own rule engine (a major component of an expert system) was discarded. Instead, we explored the possibility of using a third-party rule engine and adapt it to our needs.

Initially, ADLA only needs to be able to handle benchmark data from the simulator Moses and selected data from previous JATT runs.

We can assume that the formats of the input files will not change. This includes for example, all test suite reports and the JATT log.

In a meeting with ST-Ericsson, we chose which test suites to include in the project and to prioritize them. The list shows the test suites we limited ourselves to, and each number represents its priority:

- Benchmark (1)
- TCK (1)
- JDTS (1)
- Android CTS (2)
- Atlet (3)
- LabView (4)

Chapter 2

Information Gathering and Preparation

This chapter describes the most important parts of the information gathering process, before and during the project. This theoretical research is the basis for the entire work. Also, the possibility of using a third-party rule engine is explored.

2.1 What is an Expert System?

Expert systems is a branch of artificial intelligence. According to Professor Edward Feigenbaum of Stanford University [Giarratono and Riley, 1998, pp. 1-2] an expert system is:

> an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solutions.

Giarratono and Riley continue by listing some advantages of using an expert system [pp. 4-5]. For this project, the most important ones are increased availability, multiple expertise, and explanation. The first one matters to us because one of our goals was to decrease the workload on the experts by making their expertise on error analysis more available to their co-workers. With the ability to gather expertise from several experts in the same system, the knowledge may exceed that of a single expert. By also combining it with solutions for common errors, new co-workers can get up to speed more quickly. Because the expert system can be made to explain the reasoning that led to a conclusion, it can be seen as a source of education. The authors also mean that a system capable of explaining itself increases the confidence that the correct decision has been made by the system.

The knowledge in an expert system is usually represented by a set of rules. The rules operate on facts, which are kept in the working memory while the

expert system is running. Every rule has two parts, the condition part and the consequence part. This can be thought of as an if-then-statement where the consequence is executed *if* the condition is satisfied.

The inference engine uses a pattern matcher to decide which rules are satisfied by the facts and thereby eligible for execution. Because many rules can be satisfied simultaneously the inference engine produces a prioritized list, called the Agenda, on which conflict resolution is applied to decide which rule gets to execute. When a rule is running, it may modify the working memory by inserting, updating and removing facts, and thereby change which rules are satisfied by the pattern matcher. All these parts, shown in figure 2.1, of the expert system are usually called the rule engine.

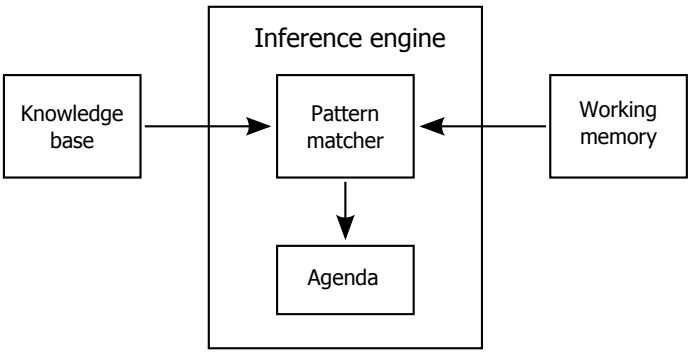


Figure 2.1: The parts of an expert system referred to as the rule engine.

Apart from the rule engine, an expert system can, according to [Giarratono and Riley, 1998, pp. 23-24], also contain an explanation facility, which explains the reasoning of the system to the user. There may also be a knowledge acquisition facility, which provides a way to enter knowledge into the system without coding it explicitly.

The inference engine can use different algorithms for its pattern matcher. The simplest method is to just go through the rules sequentially and look for matches. This is of course very slow so more efficient algorithms have been developed, for example Rete, Treat and OPS5. Every algorithm has its strength and weaknesses. As we will see later, Rete and its variants are among the most popular.

2.2 Forward and Backward Chaining

An expert system can operate in two different ways: forward chaining, which is data driven, and backward chaining, which is goal driven. If the system is data driven one just adds the data to the working memory and sees what result the system will produce. For goal driven systems, one has to have a list of hypothesis for the system to prove. The only goals and sub-goals the system can confirm are the ones in the hypothesis list. This means that backward chaining systems are more suitable for problem checking where one has a theory about what is wrong and forward chaining when one does not know what outcome to expect.

Ultimately, the data decides which technique to use as [Giarratono and Riley, 1998, p. 147] explains by describing the data as trees:

> A good application for forward chaining occurs if the tree is wide and not very deep. This is because forward chaining facilitates a breadth-first search. That is, forward chaining is good if the search for conclusions proceeds level by level. In contrast, backward chaining facilitates a depth-first search. A good tree for depth-first search is narrow and deep.

Some expert systems use both forward and backward chaining at the same time. These kinds of systems have both forward and backward engines, with specific rules for each one. By using hybrid rules that use both engines, better performance can be achieved.

In our case, it is important to get all the sub-results and it is also impractical to maintain a list of hypotheses. In addition, we anticipate our tree-structure to be more wide than deep because the number of steps needed to reach a conclusion is expected to be few. Because of this, the focus will be on forward-chaining systems and hybrids.

2.3 Third-Party Rule Engine

In the early stages of the project we agreed with ST-Ericsson that it would be too time-consuming to develop an entire expert system from scratch, especially the rule engine. There are many benefits of using a third-party rule engine, for example, they are well tested, they use well-known and fast algorithms, and they are continually improved. Developing a similar application would consume most of the project time and affect both the amount of knowledge and the number of features in the final system. These were the main reasons for choosing a third-party rule engine.

We thought it was desirable that the system should be based on open-source. The reasons were for example, to get an insight into the system and the source code, be able to freely extend the program, and not being dependent on proprietary software. We looked at a large number of rule engines, and systems with similar capabilities, to find a suitable option. Because we had not yet decided if ADLA should be a part of JATT or a stand alone system, we preferred systems written in Java to simplify a potential integration. After the first elimination round the systems in table 2.1 looked the most promising.

To weed out unsuitable systems we formulated a number of criteria. As already discussed in section [2.2 Forward and Backward Chaining], a forward chaining system was more likely to fit our needs. As a consequence, Euler and InfoSapient were eliminated. The reason they were on the list at all, was that we did not want to remove them until we were sure to find alternatives.

Name	License	Impl. language	Rule language	Chaining	Algorithm
CLIPS[a]	Public domain	C	Lisp-like	Forward	Rete
Drools[b]	ASL (Apache)	Java	Drools Rule Language	Forward	ReteOO
Euler[c]	W3C Software License	Java, C#, Prolog, etc.	?	Backward	?
InfoSapient[d]	CPL 1.0	Java	?	Backward	?
Jena[e]	HP (BSD style)	Java	RDF as XML	Both	Multiple
Jeops[f]	LGPL	Java	?	Forward	Modified Rete
Jess[g]	Proprietary	Java	Jess Rule Language	Both	Enhanced Rete
SweetRules[h]	LGPL	Java	Multiple	Both	Multiple

Table 2.1: A selection of third-party rule engines we have tried.

[a] http://clipsrules.sourceforge.net/WhatIsCLIPS.html (verified 2010-06-02)
[b] http://downloads.jboss.com/drools/docs/5.0.1.26597.FINAL/drools-expert/html_single/index.html (verified 2010-06-02)
[c] http://www.agfa.com/w3c/euler (verified 2010-06-02)
[d] http://info-sapient.sourceforge.net/White_Paper/BusinessProcessRules.pdf (verified 2010-06-02)
[e] http://jena.sourceforge.net (verified 2010-06-02)
[f] http://www.cin.ufpe.br/~jeops/manual (verified 2010-06-02)
[g] http://www.jessrules.com (verified 2010-06-02)
[h] http://sweetrules.semwebcentral.org (verified 2010-06-02)

When we tried to run SweetRules we discovered that it required a large amount of third-party software (about 10 different). Not only that, but many of them were way outdated and no longer actively developed, which made us exclude it from the list.

Despite the fact that CLIPS probably is the most well known system, we had a hard time trying to write rules because the syntax is Lisp-like. Even though it is possible for us to learn, it takes time. Because all the rule developers would have had to go through the same time-consuming learning process, we wanted a more familiar syntax. Because Jess syntax is the same as that of CLIPS, it presents the same problem to us. In addition, Jess costs money and does not have any striking advantages to justify a price tag (in our opinion), even though the source code is provided, which is rare for proprietary software.

Of the three remaining systems, Drools felt like the better choice compared to the other two, Jena and Jeops. It has a good manual and the rule syntax is easy to learn if one already knows Java. Drools also integrates with Eclipse, which is an advantage because this is what ST-Ericsson uses to develop JATT. For more information about Drools, see section [3.2 Drools].

2.4 JSR-94: Java Rule Engine API

JSR-94 is a standard to provide basic rule engine operations through a Java API. One of its goals is to simplify the process when one wants to change the rule engine in an application. Unfortunately, because the standard does not specify the rule language one would still have to rewrite all the rules, which is a major part of an expert system. In addition, it is very limiting as the whole API of the chosen rule engine cannot be used, but only the basic functionality specified by JSR-94. [Toussaint, 2003, p. 9]

In theory it would be an advantage, in our case, to use the standard because it would be easier for ST-Ericsson to change the rule engine in the future, if a better option has been developed. However, in practice the drawbacks outweigh the advantages; the work needed to change the rule engine would be more or less the same. Therefore, we would rather use the extra functionality the rule engines provide, than the limited set specified by the standard.

2.5 Regular Expressions

Regular expressions can be used in many ways, and are described by [Goyvaerts and Levithan, 2009, p. 1] as:

> [...] a specific kind of text pattern that you can use with many modern applications and programming languages. You can use them to verify whether input fits into the text pattern, to find text that matches the pattern within a larger body of text, to replace text matching the pattern with other text or rearranged bits of the matched text, to split a block of text into a list of subtexts, and to shoot yourself in the foot.

As the authors indicate regular expressions are a very powerful tool for finding and manipulating data, but can also be a great source of trouble if one is not careful when writing patterns. Standard Java provides support through the classes in the `java.util.regex` package and uses a Perl-style of regular expressions, which is the most popular style [Goyvaerts and Levithan, 2009, p. 3]. In Java one uses the `Pattern` class to compile regular expression. The pattern is then used to create a `Matcher` that can match `Strings` against the pattern. The syntax for the patterns can be found on the Pattern page in the Java Platform API Specification[1].

One example of how we use regular expressions in ADLA is to extract stack prints of Java exceptions from different log files. Below is a slightly modified example of an exception, specially triggered for this occasion. We have modified the package names for the JUNG-classes to make them fit on the page.

The example exception

```
Exception in thread "AWT-EventQueue-0" java.lang.RuntimeException: java.lang.Exception
  at com.stericsson.jatt.adla.map.MapVertex.toString(MapVertex.java:70)
  at jung.decorators.ToStringLabeller.transform(ToStringLabeller.java:33)
  at jung.decorators.ToStringLabeller.transform(ToStringLabeller.java:27)
  at jung.renderers.BasicRenderer.renderVertexLabel(BasicRenderer.java:75)
  at jung.renderers.BasicRenderer.render(BasicRenderer.java:60)
  at jung.BasicVisualizationServer.renderGraph(BasicVisualizationServer.java:367)
  at jung.BasicVisualizationServer.paintComponent(BasicVisualizationServer.java:321)
  at javax.swing.JComponent.paint(Unknown Source)
  at javax.swing.JComponent.paintToOffscreen(Unknown Source)
  at javax.swing.BufferStrategyPaintManager.paint(Unknown Source)
  at javax.swing.RepaintManager.paint(Unknown Source)
  at javax.swing.JComponent._paintImmediately(Unknown Source)
  at javax.swing.JComponent.paintImmediately(Unknown Source)
  at javax.swing.RepaintManager.paintDirtyRegions(Unknown Source)
  at javax.swing.RepaintManager.paintDirtyRegions(Unknown Source)
  at javax.swing.RepaintManager.seqPaintDirtyRegions(Unknown Source)
  at javax.swing.SystemEventQueueUtilities$ComponentWorkRequest.run(Unknown Source)
  at java.awt.event.InvocationEvent.dispatch(Unknown Source)
  at java.awt.EventQueue.dispatchEvent(Unknown Source)
  at java.awt.EventDispatchThread.pumpOneEventForFilters(Unknown Source)
  at java.awt.EventDispatchThread.pumpEventsForFilter(Unknown Source)
  at java.awt.EventDispatchThread.pumpEventsForHierarchy(Unknown Source)
  at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
  at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
  at java.awt.EventDispatchThread.run(Unknown Source)
Caused by: java.lang.Exception
  at com.stericsson.jatt.adla.map.MapVertex.toString(MapVertex.java:68)
  ... 24 more
```

The patterns used

```
Pattern p1 = Pattern.compile(".*java\\.\\S+\\.\\S*Exception.*");

Pattern p2 = Pattern.compile(
  "((\\t\\.{3} \\d+ more)|" + // ... 3 more
  "(\\tat )|" +               // at
  "(Caused by\\: )).*");      // Caused by:
```

[1] http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html (verified 2010-06-02)

When extracting an exception like the one in the example, we begin by looking for a match with the pattern `p1` that will match the first row in each exception. After that, we use pattern `p2` on the rows following until it no longer matches. This is done by calling `p2.matcher(line).matches()` for every row.

The first pattern looks for a row that contains the word `java`, followed by a dot and then a sequence of non-whitespace characters, and after that another dot followed by a word ending with `Exception`. For this to work it requires that the exception class is located in the `java` package hierarchy, and that the class name ends with `Exception`, which is the case for all standard Java exceptions.

The second pattern is a bit more complicated because it has three parts separated by or-operators to be able to match any of the three other types of rows. The first part matches the last row in the example that starts with three dots, the second matches all rows starting with `at`, and the last part is for the `Caused by` row. `\\t` indicates that the line must start with a tab. `\\.{3}` is short for `\\.\\.\\.`, which is simply the three dots. `\\d+` means one or more digits.

Chapter 3

Architectural Overview

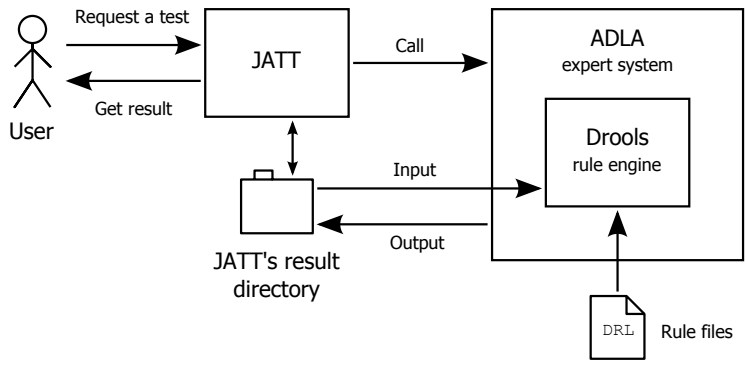


Figure 3.1: Overview of ADLA and surrounding systems.

ADLA is the name of the system we have constructed and stands for Automatic Dump and Log Analyzer. It is an expert system that analyzes the files output from JATT to find errors.

Figure 3.1 shows how the user requests a test of a mobile phone to be performed by JATT. After JATT has executed the test and output the result to its result directory, it calls ADLA. ADLA is supposed to analyze the files in the result directory to identify errors. The result of the analysis is put in the directory and will be available to the user via JATT's web page. A more detailed explanation of JATT is found below, in section [3.1 JATT - Java Automated Test Tool].

While building ADLA we decided to use the third-party rule engine Drools for the reasoning part. This means Drools will be integrated into ADLA to provide a rule engine. Drools uses rules from the rule files for its reasoning on the input. In section [3.2 Drools] later in this chapter, we describe how Drools is adapted to fit our needs.

3.1 JATT - Java Automated Test Tool

JATT is a tool that automates testing of the Java platform in mobile phones. The software is developed by ST-Ericsson and consists of, a database to store test jobs and test results, and a web page for scheduling test jobs and viewing results. A number of JATT clients, with attached mobile phones, poll the database for test jobs and execute them. JATT is also accessible through other test tools where JATT's is only one of many sub-results. This means that for some users the entire result from JATT will be represented by a green or red indicator, which translates into success or failure. These users will delegate the troubleshooting to the developer team in case of failure. Because the fault is not always a problem some delegations will be unnecessary, i.e. some minor faults that can safely be ignored still cause the test to fail.

Each test job consists of a number of tasks that describe how the test is performed. A typical test job starts by flashing the phone with software and booting it up. Then the test is copied and executed, and finally the results are collected. Depending on which kind of test the phone is supposed to perform the tasks will be different. The content of the test job is chosen on the JATT web page when the test is submitted to the database. After waiting for its turn the job is retrieved by a JATT client, which executes it. The result is then stored in the database and is available through the web page.

There are different types of test jobs that run either a benchmark application or another test suite. Each test suite is a collection of tests, which checks some part of the Java implementation in the mobile phone. For example, there are several TCK (Technology Compatibility Kit) suites, each of which checks compliance with a specific JSR (Java Specification Request), e.g. functions and API calls. Another suite is called JDTS[1] from Sun, which according to their web page is used to "[...] evaluate, validate, and verify the quality of implementations of the Connected Limited Device Configuration (CLDC) and the Mobile Information Device Profile (MIDP) on a particular device". The results from these suites are reports that list all the performed tests and whether they were successful or not.

For benchmarks the result is performance values, e.g. the number of frames per second the phone can render a specific scene. To run the benchmark applications during the development of ADLA, we used a simulator called Moses. The other test suites would have required an actual mobile phone, which we did not have the possibility to use. Instead, data from earlier saved runs was used for testing these suites.

3.1.1 Result Directory

Together with the reports from test suites and benchmarks, the result from a test job also consists of various logs and in some cases dump files. All of these files in JATT's result directory are packed by JATT in a zip-file, which is available on the web page. The directory's content largely depends on what test job JATT has executed and which of the tasks were successful. The contents of

[1] Java Device Test Suite, http://java.sun.com/products/javadevice (verified 2010-05-18)

these files are the main input to ADLA. Below we describe some of the more important ones.

Request.xml

This XML file contains all the tasks for the test job that will be executed in sequential order by JATT. Each task has attributes, e.g. paths to test programs and a time out value. The time out specifies the maximum amount of time the task is allowed to run, if it is exceeded the task is terminated by JATT to avoid deadlocks.

Depending on the tasks specified, different tests or activities are performed on the phone. Some common tasks include flashing the phone with a specific firmware, installing a MIDlet, collecting test data and uninstalling the MIDlet. A MIDlet is a type of Java application for mobile phones.

JATT log

This text file is the complete log from JATT. It includes information about what tasks have been run, all exceptions thrown in JATT and various other messages about what occurred during the run.

Core Dump

This dump file is produced by the phone in case of a serious software crash. The file is about 150 MB large and the only thing directly readable is a small text header. The rest of the file consists of memory dumps printed in hex. Because of this, the program chkArm is used to produce a more readable dump report from the core dump.

Phone log

As the name suggests, this log file is produced on the mobile phone. One file for every CPU is created and it contains debug data of everything the processor does.

Benchmark result file

There are special kinds of MIDlets that test the mobiles performance i.e. benchmarks. The results are printed to a simple XML file as the name of the sub-test and a value. The value could for example be the number of frames per second (fps) when rendering a complex scene on the phone, or the number of triangles drawn in a second.

Reports

In addition to the files mentioned above there are various test suites that produce different types of reports. Some test suite, called TCK, tests a specific API to ensure compliance with the specification for the mobile platform. Others like JDTS tests quality and benchmarks test performance. The reports are not standardized, which means an individual parser must be designed for each one of them.

3.1.2 Connection with ADLA

ADLA can be run in two different ways, which means it has two entry points. The first way is as a stand-alone application using the method `RunExpertsystem.main()` and the second is as an integrated part of JATT, using `Expertsystem.execute()`. Both entry points set up some variables like, `TEST_JOB_NAME` and `TEST_JOB_DIR` before calling `Expertsystem.launch()`, which initializes Drools and starts reasoning. The difference is that when running from JATT the object `testJobContext` provides the mentioned variables as opposed to running stand-alone where they have to be manually specified.

3.2 Drools

Drools is an open-source rule engine written in Java by the JBoss Community (JBoss by Red Hat). The purpose of this section is to describe Drools and its features, to make it easier to understand how it is used by ADLA. It is based on the Drools manual [JBoss Community, 2009], which is recommended for further reading.

The inference engine in Drools uses a modified version of the Rete algorithm, called ReteOO. According to the manual, this is "an enhanced and optimized implementation of the Rete algorithm for object oriented systems".

3.2.1 Information Flow in Drools

Drools is delivered in a number of JAR files containing classes to be used directly in the application, or it can also be built from the source code. Drools can be used in many different ways depending on the application. The following reflects how it is used in each run of ADLA, step by step.

The first thing to do is to set up a `KnowledgeBase`, which is described in the manual as a repository for all knowledge definitions. The rules are inserted from a `KnowledgeBuilder` that is used to compile them. Drools also has the capability to compile them only once, instead of at each run, to save time when initializing Drools. If so, one has to remember to compile the rules when they are changed, which is why we do not use it. The `KnowledgeBuilder` can handle many different kinds of data formats, where the DRL files we are using is one example. The next thing is to create a `StatefulKnowledgeSession` from the `KnowledgeBase`. When it has been created, simply call `fireAllRules()` to start reasoning.

The rule engine will now begin executing the rules in the `KnowledgeBase`. First the rules and the facts will be matched by the inference engine to determine which rules can be executed. Then the class `Agenda` will choose what order to execute them. All facts are kept in the `WorkingMemory` and each change of them triggers another match of the rules and facts. When no more rules can be executed (or if an Exception is thrown) `fireAllRules()` will return.

3.2.2 Rule Files and Rule Syntax

The rules are written in DRL files, which are just regular text files. It is also possible to input rules in other formats, e.g. XML, but we think it is easier to read the DRL format. Every DRL file can contain any number of rules so several related rules can be grouped together. Each rule starts with the keyword `rule` followed by the name of the rule and ends with the keyword `end`. Between, the two keywords `when` and `then` are found.

The `when`-part is the condition part, which describes what objects and conditions must be met for the rule to execute. For example, if the rule requires a `Banana` object that has its `String` attribute `status` set to `"ripe"`, it is written as: `Banana(status == "ripe")`. It is important to realize that all objects used by Drools are just ordinary Java objects without any sort of modifications. To use the banana object in the consequence part it has to be assigned to a variable: `$b : Banana(status == "ripe")`. Note that none of the lines in this part ends with semicolon. If no conditions are given, the rule will always be executed and the keyword `when` can be left out.

When the rule gets executed the code in the consequence part, below the `then` keyword, is run. This part consists of regular Java code with a few additions, like the keywords `insert`, `update` and `remove`. For example, to eat the ripe banana from the previous example the rule can be written as:

```
1 rule "Eat ripe banana"
2   when
3     $b : Banana(status == "ripe")
4   then
5     $b.setStatus("eaten");
6     update($b);
7 end
```

3.2.3 Using Drools in Eclipse

If one uses Eclipse (http://www.eclipse.org (verified 2010-06-10)) to develop applications, we recommend downloading Drools directly to Eclipse 3.4. This adds support for Drools perspective, a number of views and the Drools runtime. This also adds syntax highlighting of rules in DRL files, and the ability to create Drools projects. With Drools projects one gets a number of benefits, like syntax checking while editing DRL files and the ability to debug rules in the same way as regular Java code. The new views lets one inspect for example the working memory and the agenda, while debugging.

To install Drools in Eclipse, go to the "Software Updates" dialog in Eclipse, choose the "Available software" tab and click "Add site". Insert the link to "Drools 5.0 Eclipse Workbench for 3.4"[2], choose the component "JBoss Drools Core" and hit install. After rebooting Eclipse, create a new "Drools Project" and when asked about "Drools Runtime" click on "Configure Workspace Set-

[2]Currently http://downloads.jboss.com/drools/updatesite3.4 (verified 2010-05-12)

tings". Click "Add" and "Create a new Drools 5 Runtime" and choose a new location for the Drools Runtime. To instead convert an ordinary Java project, simply right-click on it and choose "Convert to Drools Project".

3.2.4 Drools Logger and Audit View

Drools has a built-in logger, `KnowledgeRuntimeLogger`, which takes the `StatefulKnowledgeSession` as an argument. Numerous events, e.g. for rules and objects, are logged to an XML file. The events for rules include creations, cancellations and executions of activations. For objects the events are insert, update and remove.

The output log file is very hard to read, which is why Drools includes the Audit view in Eclipse. Once one has located the button to open the log, in the upper right corner of the view, it shows all the events in chronological order. The events are displayed as a tree structure with descriptive icons for each type. If an object or activation is selected, its creation event will be highlighted, as shown in figure 3.2.

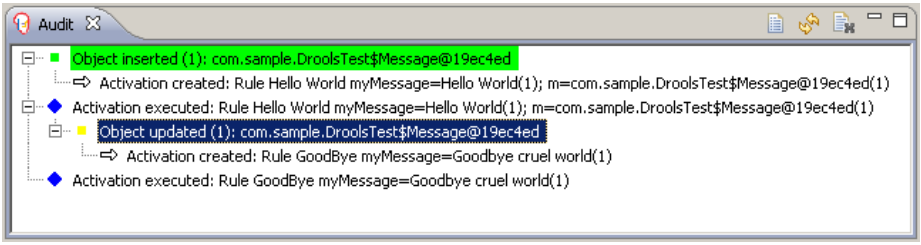


Figure 3.2: Shows the Audit view for a small example. The green highlight shows the origin of the object in the selected event.

3.3 Use Cases

To ensure a suitable architectural design we have come up with a few typical use cases that the user of the system will want to perform. Through the whole development process we have tried to update the steps in each one of them to see that they still are easy to perform. The steps for each one are described in [A.1 Use Cases].

- Set Up ADLA to Run Through JATT
- Running ADLA stand-alone
- Adding a new rule
- Adding a new area of competence
- Removing a rule

Chapter 4

Constructing ADLA

This chapter describes how the reasoning part of ADLA is built. This includes many of the necessary features built around and on top of Drools to extend the functionality of ADLA beyond Drools' standard capabilities. The chapter is roughly organized according to the information flow and includes how information in ADLA is input, stored, tracked and output.

4.1 Input

The purpose of the input is to form facts that can take part in reaching conclusions in the system. These conclusions can represent for example answers to questions or various decisions, but can also be a part in further reasoning. The input to ADLA is essential; if the quality of the input is poor, it will be reflected in the output. In practice, this usually means that too little or only insignificant information was present in the system when the reasoning was performed. This could lead to the system reaching the wrong conclusion, in the same way as a human expert would.

ADLA is designed to get its input in two ways. The first way is that JATT can send objects as arguments when it calls ADLA as mentioned in subsection [3.1.2 Connection with ADLA]. The benefit of this is that no files have to parsed; everything is already neatly contained in Java-objects and can be used as facts almost directly. The only processing needed is to encapsulate them in objects that ADLA can handle. The disadvantage with this method is that ADLA must be called from JATT and cannot be started separately. The other way for ADLA to get input is to read the files in JATT's result directory, described in subsection [3.1.1 Result Directory]. This can be done either as a start-up step, or later when the input is needed. This method is necessary with most of the facts because they are not available directly from JATT. In addition, this makes it possible to run ADLA separately using data from any previous JATT run, which helps to ensure identical data is used when developing rules.

At the moment, ADLA almost exclusively gets its input from JATT's result directory. The reason for this is that the simulator we use while running JATT

only supports benchmark tests. For other test suites like TCK and JDTS, data from previous JATT runs that were conducted on real hardware, must be used. This data must come from the result directory since the data from JATT is only available when the test job is running.

If the input is gathered from JATT's result directory, it is collected as text and must be parsed to the correct data format to be useful in the rules. When the data is parsed, it is stored in nodes, which are used as facts in the rule engine. Because only small parts inside the files are useful, regular expressions are used to extract interesting parts (see section [2.5 Regular Expressions]). As long as the file formats stay the same, any data can be collected. One drawback is that if the format changes one does not always notice, which could lead to a lack of facts and bad conclusions. As stated in section [1.4 Boundaries], we will assume that the formats do not change.

One final problem with the input arises when JATT is forced to rerun some part of the test job that has not passed. The files in the result directory are updated in different ways and the challenge is to retrieve only the data from the last run, the new information. This will be discussed in-depth in section [6.1 Keeping Track of New File Contents between Reruns].

4.2 Holding Information in Nodes

A node in ADLA is simply a Java object which holds information that the rule engine needs when reasoning, i.e. facts. Although Drools can use objects of all classes as facts, we had to restrict these to a limited set. This is necessary because, as shown later, we need certain common attributes in all objects used while reasoning to be able to track the execution. The real gain of this will be revealed in section [4.3 Tracking Rule Execution]. Objects of the types in the restricted set are referred to as nodes, which in practice are objects from the inheritance hierarchy of the base class `Node`. If the object can not simply extend the `Node` class, it can be encapsulated in a new class in the `Node` hierarchy. This means objects of all classes can still be used as facts, as supported by Drools, with a little bit of extra work.

The information stored in nodes can both be raw input from the files or refined through reasoning. Every type of node holds a more or less unique set of information, so many of the `Node` classes need to be specially designed for its purpose. Some examples of nodes are:

- `Request` - represents the file Request.xml that defines the test job with list of tasks.
- `Task` - holds information about each task like its arguments and time out.
- `ErrorNode` - used as a wrapper for `ErrorNodeExceptions`, which the rule developers create when reporting errors in ADLA.
- `TestSummary` - summarizes the results from benchmarks and test suites.
- `JavaException` - holds the extracted information about a Java exception from a particular file.

In many cases, only something similar to a common variable needs to be stored, i.e. a name and value. For this purpose, the class `GeneralNode` has been created, which holds a `String` name and a value of the type `Object`. The advantage is that no specially designed class for storing simple information has to be created. This speeds up the development process. A disadvantage is that `GeneralNodes` only can be separated by their name attribute and not by class, which can lead to conflicts if one forgets that the particular name already is in use. Also, like every object, it is faster to check the class than an attribute, which can make the system a bit slower than if different classes would have been used.

4.3 Tracking Rule Execution

The internal structure of an expert system can be thought of as a pile of rules and a cloud of facts, which will be combined and executed automatically by the rule engine. One problem with this is that there is no direct way to determine how a certain conclusion was reached, which becomes very frustrating if one suspects the result to be erroneous. In addition, we totally agree with [Norvig, 1992, p. 531] in his statement:

> A system that can explain its solutions to the user in understandable terms will be trusted more.

Because there can be many ways to reach a certain conclusion in the system, we have to keep track of which rules are run and what objects are inserted or updated. As illustrated in figure 4.1, the problem with tracking the flow is that the only way of knowing which of all the possible ways through the rules and objects the system has taken is to track it while running. It is not possible to unambiguously determine this afterwards.

For the solution we needed common attributes for every node. One of the attributes is a list called `cameFrom` that stores a reference to the rule that inserted it into Drool's working memory, with the keyword `insert`. Now we can keep track of in which rules all the objects were inserted, but we still do not know which objects triggered each rule. For this purpose we also represent all rule executions as nodes, called rule nodes. This way the rule nodes will also possess `cameFrom` lists that will hold references to each object that was required to trigger the rule. One example of how the `cameFrom` list is used is shown in figure 4.1 and details on how the list is filled in can be found in section [6.2 Filling in the cameFrom Lists]. It is important to realize that in the same way that nodes represent objects, not classes, rule nodes represent rule executions and not the rule itself. So, if a rule is executed several times each execution will be represented by a separate rule node.

When an object is updated, e.g. when its attributes are changed, one lets Drools know by using the keyword `update`. Since an update can be just as important as an insert, we need to keep track of them too. A reference to every rule that updated an object node is stored in that node's `updatedBy` list. The references are stored in a similar way to that of the `cameFrom` list, but the list is empty for all rule nodes (rules cannot be updated). If the `cameFrom` and `updatedBy`

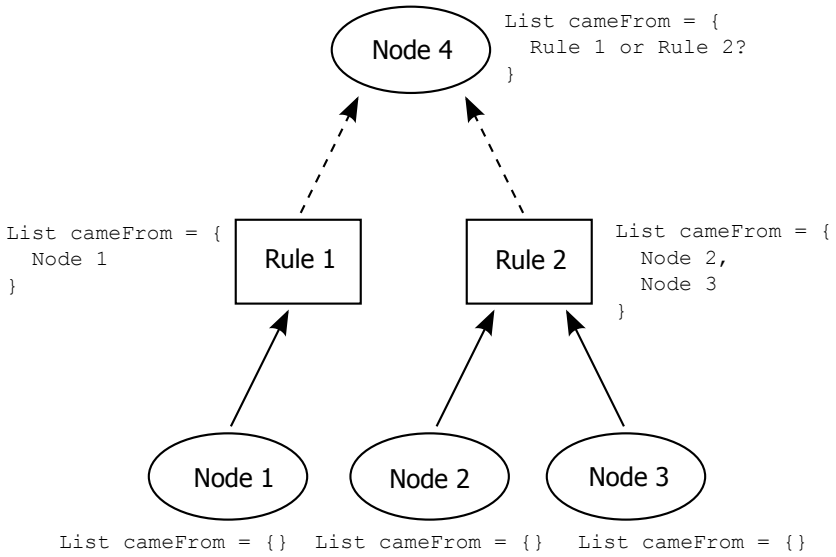


Figure 4.1: Shows the principle of how the nodes' `cameFrom` lists are used and the problem of tracking the flow in the system. One cannot unambiguously determine if Node 4 came from Rule 1 or Rule 2, if this was not logged when the node was created. The list `cameFrom` is a common attribute to nodes and rule nodes that stores references to where they came from. Arrows pointing at a rule node show that the rule required the object at the tail, and arrows pointing from a rule node shows objects that the rule created.

lists are followed back from a conclusion to its sources, it will form a chain of rules and nodes. All the nodes in the chain have contributed to the conclusion, directly or indirectly. A chain from a certain point in the system, back to its sources, is henceforth referred to as a why-chain. In practice, the why-chains are usually intertwined with each other and will form a graph. More about this in chapter [5 Visualizing ADLA].

4.4 Conclusions and Decisions

At the beginning of the project it was required that the results from ADLA, its conclusions and decisions, should be printed in prioritized order as XML. Further into the project this was changed, mainly because of the complexity of the conclusions from ADLA, as we will see in this section.

4.4.1 Identifying the Results

The first and biggest problem during the development has been to identify the results of the system, or in other words, how to separate facts from conclusions. The rule engine does not differentiate between facts and conclusions. This means that it is impossible to know if an object produced by a rule is supposed to be a conclusion or just be used for further reasoning by other rules. It could sometimes be a conclusion and sometimes not, depending on the combination of objects in the working memory and the kind of test job JATT has run.

Also, ADLA is supposed to find errors in multiple areas, so the problem gets even more complex because one never knows which objects hold the different results. Some of the results ADLA is supposed to produce are for example: decision about rerun for the test job, result from the test suite, information about failed tasks, core dumps and Java exceptions. If the system instead had one area of responsibility, e.g. to decide if the test job needed a rerun, finding the result would have been easier since it is stored in one specific object.

4.4.2 Prioritizing the Result

Even if all the results could have been identified, the prioritization would still be impossible to do automatically in the system. For example, the priorities for each conclusion could be kept in a list, but such a list would quickly become unmaintainable. This is because the prioritization can change depending on what other objects are in the system, but also because they continuously change when rules are rewritten or added. If the prioritization instead is performed by the users, when the result is finished, they will only need to prioritize the actual result and not all possible combinations.

4.4.3 Graphical Solution

An acceptable solution, where the system identifies the result, was never found. Instead, we were forced to involve the user to identify the result, the important information and to prioritize it. This was done with a graphical representation which shows the full picture of the system by finding all the leafs of facts and then follow them like why-chains. Such a graph had already been made for the function for asking the system how it reached a conclusion. The graph was extended to show all the why-chains in the system and new tools were added to be able to identify the results and decisions. Some of the tools described in section [5.3 User Interface] include colored nodes with explanations, node information window and the possibility to move the nodes in the graph.

In this way, the users can fairly easily disregard parts of the system that looks uninteresting. Another benefit is that it is easier to see the bindings between facts and rules compared to the non-graphical solution first used. The disadvantages are that the person has to have good knowledge about the structure of rules and objects in the system. Important information is embedded in the way the nodes are related to each other, through the connections between them. Information is also hidden on first sight, as it is stored in the nodes. Figure 4.2 shows the problem of knowing in which node the result is stored; if the users do not know where to start looking, they have to guess by trial and error.

All nodes have something to tell, even the very existence of an object is often important. This makes it hard to decide which to show in the result, so instead all nodes are shown and the users can decide which are interesting. Together with ST-Ericsson we decided to skip the XML requirement and go for the new graphical solution. The real gain with this is that instead of using a separate function to find out how the system has reasoned, it is presented together with the result. This way every node gets more authority because it is possible to follow the path it came from. Sometimes the path even describes a node

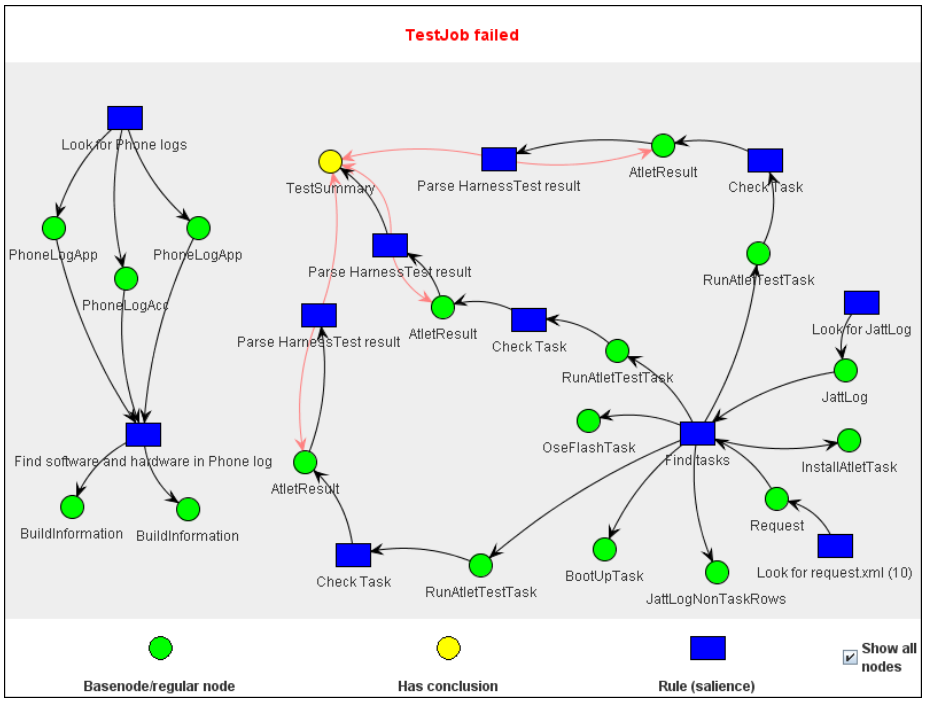


Figure 4.2: Shows the graphical result for the users to investigate. Due to the many nodes, finding the result one is looking for can be hard. Inexperienced users may have to resort to trial and error. The prioritization is done indirectly, as all humans do, when the users look at the graph and decide which nodes to investigate and in what order.

better than the node itself. With a non-graphical solution, this type of indirect information would have been harder to relay.

4.4.4 Passed or Failed Test Job

There is still the issue with users who are only interested in if the test job was successful or not. The main reason is that JATT needs to be called from their test program and the result must only be passed or failed. This is not possible with the above solution because the users have to be involved to identify the result. On the other hand, the reasons for a failed test and its justification need not be provided. To find a solution for this scenario we made a supplementary function. It checks that a result for the test job has been produced, that it is valid and that all tests in the suite were successful; otherwise the test job is marked as failed. In addition, if the tests in the test suite were passed but other errors were found the test job will be marked as failed. Simply put, we use the fact that every test job is supposed to return some kind of result (what else is the point of running it).

A problem is that the test job result sometimes contains minor faults, which should usually be ignored. Initially it is important that no such test job is allowed to pass because the errors could go through without notice. This means

it is better to let the test fail if there is some kind of uncertainty. The idea is that a detected problem, which is not an issue, needs to be handled in a rule to be circumvented.

4.5 Handling Errors

When we talk about errors in this section, we mean all kinds of Java exceptions, but also errors the rule developer has created for different reasons. The error handling in ADLA requires a special design due to the automatic nature of the system, as the intended way to run ADLA is as a step in JATT's execution. This presents a problem because no user is able to interact in case of an error. To make the problem even harder we want to avoid the use of log files for presenting errors to the user. This is because the very reason to build ADLA is to analyze log files so the user does not have to. The simplest solution would be to make ADLA totally quiet so no errors would be reported. The drawback would be that the user could be fooled into thinking everything went well, and the system would not be improved to avoid the errors. A more sophisticated way is required.

4.5.1 Presenting Errors with the Result

We wanted to present the errors together with the result because it can be affected by some kinds of errors. For example, if an error occurs so a fact is not added, a worst-case scenario could be that the system will come to the wrong conclusion. To pinpoint which parts of the system that are affected by a specific error it is important to be able to bind the errors to their sources. Because the same type of error may occur multiple times at different places, one needs to be able to distinguish between them to know which is most important. If one, for example, tries to read the same non-existing file in numerous locations the same error will be generated every time. If the errors are not bound they have to be more specific and contain enough information for the user to locate their sources. This puts unnecessary work on the rule developers to describe each error, so they are uniquely identifiable. If they instead are automatically bound to the rule that caused the error, one gets valuable information about its origin that sometimes says more than the actual description.

To accomplish this, the class `ErrorNode` is introduced to represent errors and behaves like any other node. The `ErrorNodes` are drawn in a different color to distinguish them from other nodes, more about this in subsection [5.3.1 Presenting the Graph]. In practice, an `ErrorNode` is merely a container for an `ErrorNodeException`, which is the actual error. This class extends `Exception`, which makes it possible to use it with Java's error handling facilities: `throw`, `try`, `catch`, etc. It holds a description of the error, a label that is used when the node is drawn and optionally a reference to another Java exception. This reference is set when the `ErrorNodeException` is created in response to another exception. Because it is optional the rule developer can create an error even though Java does not interpret it as an exception, e.g. if a report contains unreasonable, but correctly formatted, data.

`ErrorNodes` also have an attribute called `problem`. This can be set to `false` if ADLA on further reasoning finds the error irrelevant, for example when a specific error in some situation should not make the test job fail. The reason for not simply removing the `ErrorNode` is that it could be relevant to other rules.

4.5.2 Categorizing the Errors

The error handling is divided into three groups depending on where in the system the error originated, see figure 4.3. The first one consists of errors that occur outside the rule engine, e.g. output from ADLA could not be saved, no rule files were found, or there were syntax errors in the rules. Because the rule engine has not yet started, no result will be produced. This and the nature of the errors that occur at this point makes it impossible, or at least inappropriate, to present them in the graph. Instead we have chosen to print the errors, with the prefix ADLA, to the JATT log to avoid creating additional log files.

The second group includes all errors triggered from the rules that are explicitly handled, i.e. when an `ErrorNodeExceptions` is created. The idea is that all Java exceptions should be caught and if they still need to be reported to the user an `ErrorNodeException` should be thrown instead. The errors can of course be handled like in standard Java if they do not need to be reported. It is also possible to throw an `ErrorNodeException` even if it is not in response to a Java exception.

The last group includes all errors triggered from rules that are not handled, i.e. exceptions one forgot to catch. The only difference between this group and the second is that no label or custom description is assigned by the rule developers, which means they only contain the standard label "Unhandled error" and the regular exception text. This makes them less descriptive, which is why they should be avoided. Both exceptions from the second and third group are caught and transformed into `ErrorNodes` outside the rule engine as shown in figure 4.3. Once the exception is handled, the rule engine continues with the next rule, which means the rule that threw the exception will not be executed again, unless it is rescheduled.

Main.java

```
public void execute() {
 StatefulKnowledgeSession ksession;
 try {
  ksession = initExpertSystem();
 } catch(Exception e) {
  logger.info("[ADLA] Unable to inititalize expert system");  (A)
  return;
 }

 while(true) {
  try {
   ksession.fireAllRules(); // The rule engine executes rules
   break;
  } catch(ConsequenceException e) {
   ErrorNodeException err;
   if(e.getCause() instanceof ErrorNodeException)
    err = ((ErrorNodeException)e.getCause());
   else
    err = new ErrorNodeException("Unhandled error", "An unexpected error was caught", e.getCause());
   ksession.insert(new ErrorNode(err));
  }
 }
}
```

Rule.drl

```
rule "Parse TestSuit report"
 when
  $ts : TestSuite()
 then
  $ts.parse();                                                                    (B)
  if($ts.totalTests != $ts.passedTest + $ts.failedTests)
   throw new ErrorNodeException("Sanity check failed", "The total sum is not equal to the parts");
  update($ts);
end
```

TestSuite.java

```
public class TestSuite {
 public int totalTests;
 public int passedTest;
 public int failedTests;

 public void parse() throws Exception {
  FileReader fr;
  try {
   fr = new FileReader("reportFile.dat");
  } catch(FileNotFoundException e) {
   throw new ErrorNodeException("TestSuite error", "The report file was not found", e);  (C)
  }
  totalTests  = fr.read();
  passedTest  = fr.read(); // Whoops, throws IOException                            (D)
  failedTests = fr.read();
 }
}
```

Figure 4.3: Shows the principle of how the error handling works. In the while loop the rule engine is started with the method `fireAllRules`, which does not return until the rule engine is finished. If an exception is thrown in a rule or a method it calls, it is caught in the while loop and handled. When `fireAllRules` is called again it continues with the next rule. Label A shows an example from group 1, how an error message is written to the JATT log. Labels B and C are examples of group 2 errors and show how the user can report errors from rules, with or without an attached Java exception. Label D is a group 3 example, where an exception is not handled and results in an unhandled error.

Chapter 5

Visualizing ADLA

This chapter describes the graphical part of ADLA, which is designed as an applet. It is separated from the reasoning part, both in this report and in the code. The reasoning part produces an XML file, containing graph data, which will be stored in JATT's result directory. This file should not be confused with the withdrawn requirement of XML output, previously discussed. Whenever the users want to look at the result they visit the JATT web page where our applet retrieves the file and displays it as a graph.

5.1 Reasons for the Visualization

The main reason was to present how ADLA reasoned, to clarify why it reached a certain conclusion. We refer to this as a why-chain and it tells what rules were executed and what facts were used and created by the rules to reach the conclusion. Facts that do not contribute to a conclusion are not included in that conclusion's why-chain. One of the reasons for using an expert system is that the reasoning behind a result is generally not trivial. This is why the ability to explain how it was reached is especially important to us. The explanation had to be graphical as described in section [6.3 Presenting Why-Chains].

As mentioned in section [4.4 Conclusions and Decisions] we also needed a clearer way to present the results from ADLA than the originally intended XML output. We concluded that the only reasonable solution for the problem of identifying and prioritizing the result is to let the user decide after each run. This was done by extending the graphical solution for the why-chains to include the result as well.

Finally, it would be helpful to find out if some specific fact was present in ADLA, even though it did not contribute to a conclusion (not included in a why-chain). This is useful when expanding the system with new facts and rules.

5.2 JUNG - Java Universal Network/Graph

As stated on its web page (http://jung.sourceforge.net (verified 2010-05-20)), JUNG is a framework written in Java used for "modeling, analysis, and visualization of data that can be represented as graphs". It is delivered in JAR files that contain classes, which are used directly in the application. JUNG is freely available under the BSD license. We use it to visualize the graphs that ADLA output.

To use JUNG, one first chooses a suitable graph type, in our case a `DirectedSparseMultigraph`. After that, simply insert the vertices and edges into the graph. In JUNG it is possible to define custom `Vertex` and `Edge` classes, which can hold data that becomes available at certain events. Next, select a layout that controls how the nodes are placed when rendered. For our purpose the `ISOMLayout` looked the best. Finally everything is rendered on a `VisualizationViewer`, which is a `JPanel`.

JUNG lets one interact with the graph in various ways, for example to zoom and move around. To allow for a better view of a certain part of the graph, the vertices can also be moved around, either one at a time or in groups.

In JUNG various listeners can be set up to catch different events in the graph, e.g. when the mouse is clicked. A reference to the clicked item is provided, for example one of the `Vertex` objects. Different transformations are also supported, which lets one change the appearances of the items in the graph. We use them to change the color and shape for different node types, and the color of certain edges. On JUNG's web page both example graphs and a manual can be found.

5.3 User Interface

Figure 5.1 shows the main view of the user interface for a failed Sun TCK test job. This test job is specially chosen because it includes many of the typical elements we want to describe here. Unless otherwise noted, it will be used for all the examples in this section.

5.3.1 Presenting the Graph

The largest part of the user interface is the graph panel where the why-chains and results are presented. Each node represents either an object or a rule execution.

Arrows

The nodes in the graph are connected to each other with arrows. The flow begins in the nodes with no incoming arrows (in most cases a rule node) and continues in the direction of the arrows, alternating between object nodes and rule nodes. Because of how the rule engine works two rule nodes, or two object nodes, cannot be connected to each other. The simple reason is that objects cannot create other objects and a rule cannot force another rule to run.

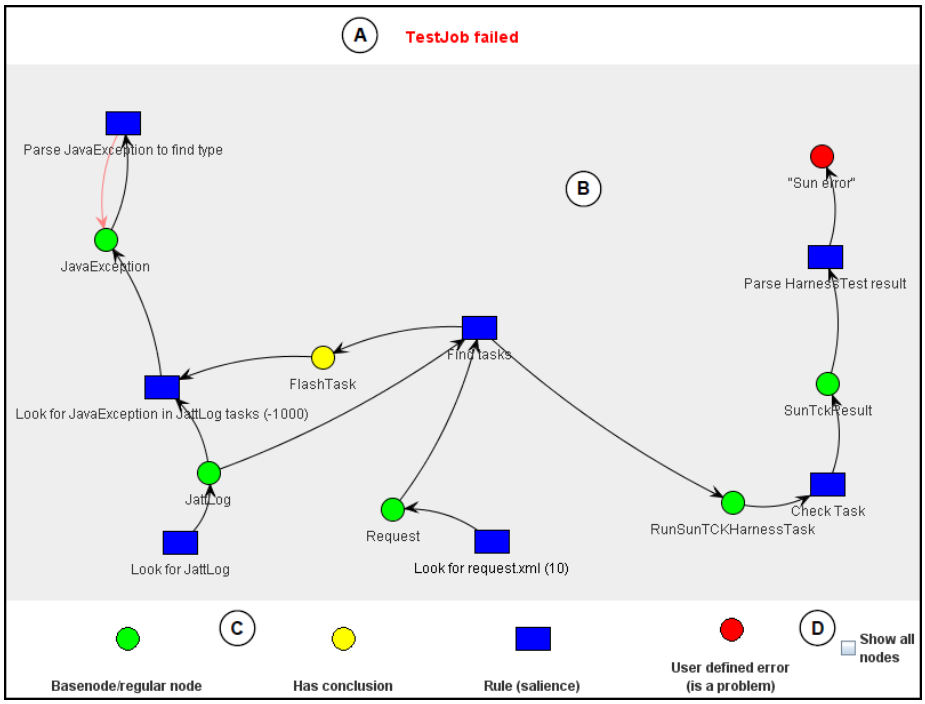


Figure 5.1: The main view of the user interface. Label A shows the status indicator, which displays the status of the test job. Label B marks the graph panel used for drawing the graph that contains the result and the why-chains. Label C is the info panel, which displays the explanation of the nodes currently found in the graph panel. Label D shows a checkbox used to toggle the graph panel between displaying only important nodes and all nodes.

Figure 5.2 shows an example of the different arrows used in the graph. An arrow pointing at a rule node means that the object node at the tail was required to trigger the rule, which is described by the condition part of the rule (when). In the opposite case, where an arrow is pointing away from a rule node, it means the rule (in the then-part) has created the object at the arrow's head. If the arrow instead is light red, the node at its head was updated during the rule execution.

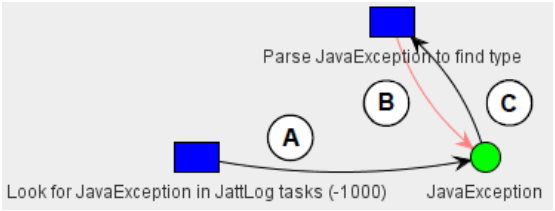


Figure 5.2: Shows the different types of arrows used in the graph. Label A is an example of an arrow where the rule creates an object. Label B marks an arrow where the object gets updated by the rule. Label C shows an arrow where the object `JavaException` is required to trigger the rule.

Node Types

To make the graph easier to understand we assign different shapes and colors to the nodes, depending on their types. If the shape is a circle the node represents an object, if the node is a rectangle it represents a rule execution. The nodes are referred to as object nodes and rule nodes, respectively. In figure 5.3, the six different types currently defined in the system are presented; new types can be added as needed, see [A.3 Add New Node Types]. Beginning from the left, the light green node represents an ordinary object and is the standard appearance if an object does not meet any other criteria. The dark green node is for objects from the specific class `GeneralNode`. A yellow node indicates that the object's `conclusion` attribute is set, which means some important information is likely to be available. The blue rectangle represents a rule execution, which means that the specific rule with the same name has been run. Some rule nodes have a number after its name; this is the rule's salience, i.e. its priority. Finally, the two red nodes signify `ErrorNodes`. The difference between them is that an error represented by a light red node was flagged as not being a problem and could be ignored.

Figure 5.3: All the node types currently defined in the system. The circles represent object nodes as opposed to the rectangle, which identifies the rule node.

Show All Nodes

In figure 5.4 the checkbox "Show all nodes" has been checked. If one compares it to figure 5.1, the difference is that the graph now also shows the nodes that are considered not important. For a node to be important at least one of the following criteria must be fulfilled:

- The node's `conclusion` attribute has been set and the node is also a leaf in the graph. For a node to be considered a leaf it must not have been used to trigger a rule that created an object.
- The node is an `ErrorNode` that has been marked as being a problem.
- The node's `alwaysDisplay` attribute has been set to true.
- The node is included in another important node's why-chain, i.e. the node has contributed to an important node.

The reason for using the function that removes non-important nodes is to avoid having them clutter the graph, making it harder to overview. The ability to toggle them back on is useful to find out if some specific fact was present in ADLA, even though it was not marked as important. This is especially helpful when expanding the system with new facts and rules and one wants to make sure a certain object has been inserted.

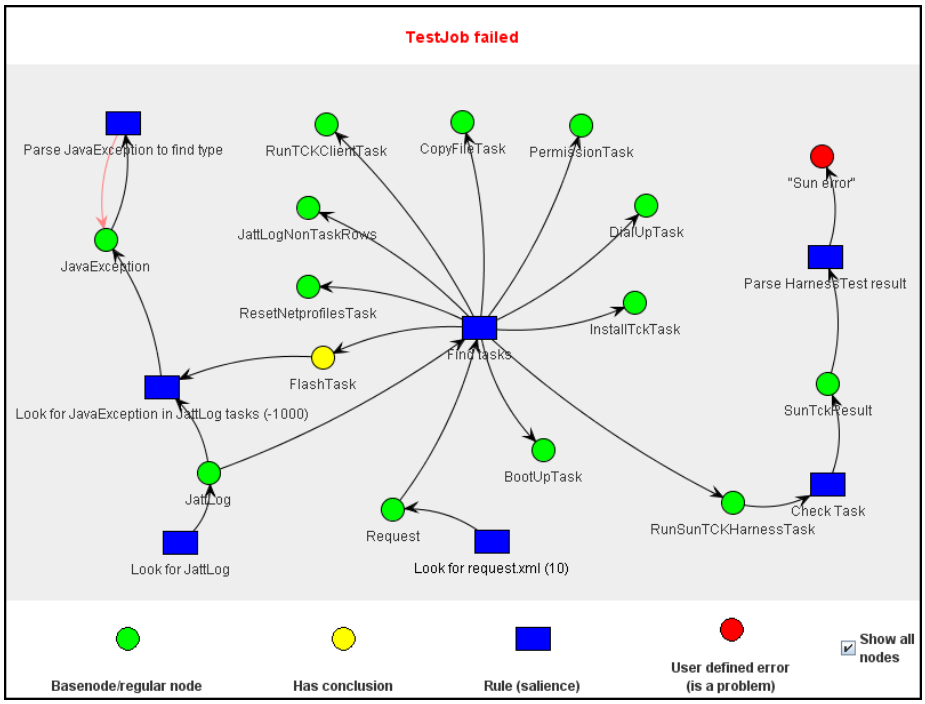


Figure 5.4: The main view of the user interface with all nodes from Drools' working memory visible.

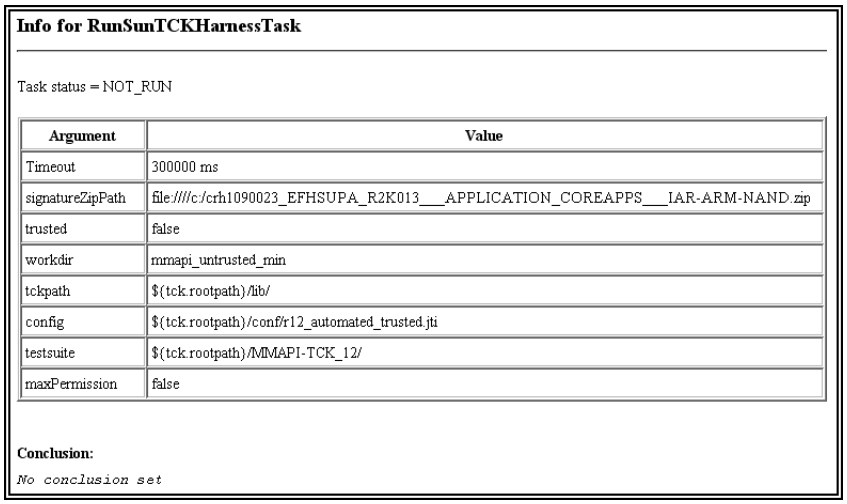
Rule executions that do not insert or update any objects will never be shown in the graph, not even when "Show all nodes" is checked. This could for example happen when a rule inserts objects inside an if-statement, only the executions that actually performs the insertion are shown. One reason for this is that, we figured, such rules have not produced anything that can be used by other rules or led to a conclusion. Another reason is to minimize the number of nodes shown in the graph as much as possible, or the graph would get too crowded. To see a summary of all the rules that have been executed one can look in the file ruleTimes.txt, which lists all rules and their execution times.

5.3.2 Information Window

To be able to easily present information stored in the nodes, the ability to right-click on them has been added. This brings up a window, showing node-specific information. The window size adjusts itself to the displayed contents.

Node Information

For object nodes, the rule developer has the ability to decide the contents of the window by overriding the method `getNodeInfo` in the `Node` classes. Because it should return an HTML formatted `String`, tables and other formatting can also be used besides regular text. If the method is not overridden the standard attribute `conclusion`, which is common to all object nodes, is shown. An example of the information window for an object node is shown in figure 5.5.



Info for RunSunTCKHarnessTask

Task status = NOT_RUN

Argument	Value
Timeout	300000 ms
signatureZipPath	file:////c:/crh1090023_EFHSUPA_R2K013___APPLICATION_COREAPPS___IAR-ARM-NAND.zip
trusted	false
workdir	mmapi_untrusted_min
tckpath	${tck.rootpath}/lib/
config	${tck.rootpath}/conf/r12_automated_trusted.jti
testsuite	${tck.rootpath}/MMAPI-TCK_12/
maxPermission	false

Conclusion:

No conclusion set

Figure 5.5: The node information window for the object node `RunTCKHarnessTask`. This particular node is a `Task` and contains that task's different arguments from the request.xml file, e.g. the time out. One can also see that the task status is set to "not run", which means that the task for some reason was not performed.

Rule Information

For rule nodes the contents of the window is standardized to include common attributes and the code for the rule, see figure 5.6. `AgendaGroup` indicates which layer the rule belongs to and can currently be either "layer1" or "layer2". A `salience` of zero means no special priority was assigned to this rule. It is often hard to locate in which of the rule files a specific rule is defined, which is why the path is shown as the attribute `ruleFileName`. The `executionTime` is presented for each rule node to make it possible to identify potential performance issues. A full summary of all rules run and their execution times are printed to the file ruleTimes.txt.

To make the rule easier to read, each line of the rule code is numbered. The initial reason for the numbering was to enable users to more easily find a certain line pointed out by an exception's stack print, in an `ErrorNode`. Because Drools translates every rule into a separate Java file, which is then compiled, we thought it could be solved by numbering each rule individually as shown in figure 5.6. Unfortunately, we discovered that the line numbers specified do not correspond to neither the absolute line numbers of the rule file, nor to the individual numbering shown in the figure. After some testing we found out that it was hard to predict how many lines the when-part of the rule will occupy in the Java file, after Drools' translation. If a way to calculate the row number where the then-part starts could be found, the problem would have been solved.

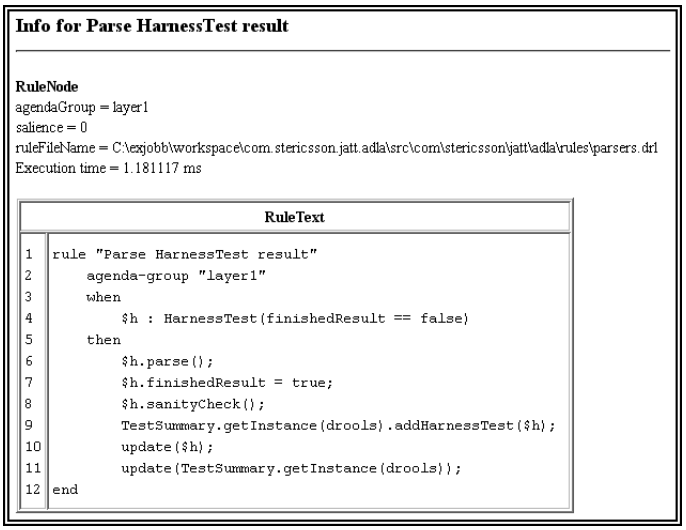


Figure 5.6: The node information window for the rule node `Parse HarnessTest result`.

5.3.3 Test Job Status Indicator

The status indicator reflects the result of the test job for those users who only need to know if the test job passed or failed, see subsection [4.4.4 Passed or Failed Test Job]. In the current solution the status can be passed, failed, error or not set, where any result besides passed would be considered a failure. The results passed or failed is collected from the test suite report, but even if the tests in the suite passed, it is still possible to get an error from ADLA if some other problem was discovered.

5.4 Map for System Overview

When the amount of rules and objects grew, and some rules needed a priority to work as intended, we realized that if there would be more than one rule developer, this could become a problem. To write a new rule one has to know what other rules exist, and what objects they are using, to be able to use them as originally intended. If the rule developer does not understand how an object is used or if a new rule gets the wrong priority, the intended system execution could be jeopardized.

To solve this we constructed a map that dynamically shows all rules and objects currently used in the system, and how they are connected to each other. This is similar to how the why-chain graph was done, except all possible connections are displayed, even to objects inside for example if-statements. This way the rule developer could get an overview of the system and get an understanding of how it is intended to be used.

Initially we thought the map was very practical and used it a lot, but as ADLA got more complex several limitations made it problematic to show the real

picture of the system. For example, the inheritance structure used, e.g. by `HarnessTest`, made the map display the wrong objects. In addition, if methods are called from the rules, the function has no possibility to analyze them.

To sum up, the map program is a good idea in theory, but limitations in the implementation made it misleading. Mainly because lack of time we have decided to exclude the function for the time being. Figure 5.7 shows an example of what the part already implemented looks like.

The map is run as a stand-alone program through the class `RunMap` in com.stericsson.jatt.adla.map. Before running, specify the path to the rule directory by setting the variable `RULE_DIR`. Also, make sure all packages that contain nodes are included in the call to `initNodeClasses`, in the `MapGraph.generateMapGraph` method.

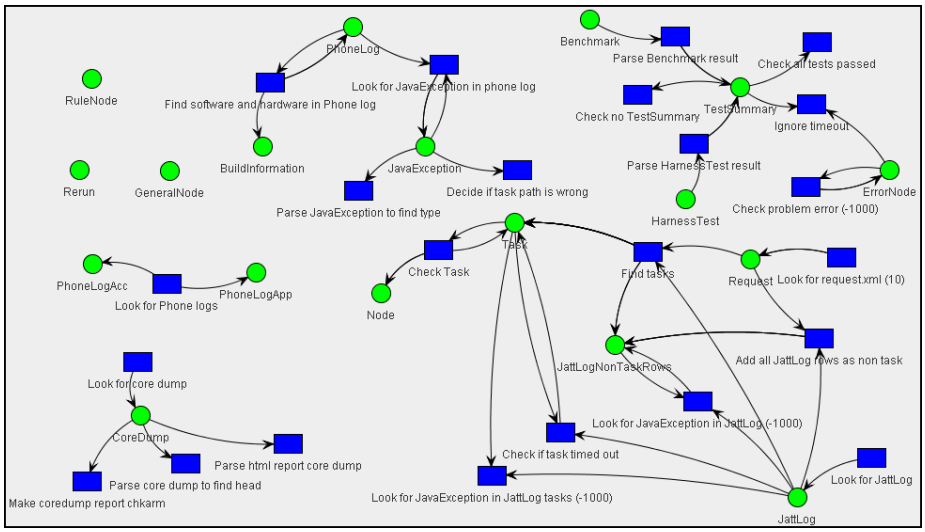


Figure 5.7: The map that intended to give the rule developers an overview of the rules and object nodes in the system.

Chapter 6

Problems and Solutions

6.1 Keeping Track of New File Contents between Reruns

When some of the tests in the test suite have failed, JATT will sometimes try to rerun the failed parts. This means the files in the result directory are updated with the data from the last run. The challenge is to only retrieve the new information so the new result is not contaminated by earlier produced data.

For some files, e.g. the JATT log, the new data is appended to the end of the file without any delimiter. To extract the new data from this kind of files we keep track of their sizes and assume the bytes appended after the last recorded file size is the new data. If the sizes are the same, the file is treated as only containing old information and is excluded. Other files, like the phone logs, are just created with new names and will therefore only contain new information. Because their names are not previously recorded, they are all treated as new files. A suggestion to handle these two types of files has been implemented in the class `FileSizeOverview`, but unfortunately the code is untested because the simulator does not support execution of test suites, which is required for testing. The class writes down the file sizes it finds in FileSizesOverview.txt and stores it in JATT's result directory. The file is then read and updated with new file sizes on each rerun. To use this functionality a method called `getDataSinceLastRerun` is available in the class.

An initially unnoticed problem was a third type of file where the old data is just overwritten with new. With the previous solution, strange errors would have been likely to arise because the files would have been treated as one of the other types. This drawback combined with the fact that it could not be tested, made us decide to skip the feature. Nevertheless, a theoretical solution could be to include the file attribute "last changed" to decide if the file is new or not. It would then be possible to detect if the file had been replaced.

One thing to note is that some files, e.g. request.xml, are not supposed to be covered by this functionality, because they are required for every rerun and are

never modified. The part of the functionality that is implemented therefore needs to be explicitly told which files in the result directory that should be monitored. This means that if the developer does not use the methods provided, everything will behave as normal. For the functionality to work ADLA must be called by JATT between each rerun, or else the old files would not be recorded properly.

6.2 Filling in the cameFrom Lists

During the project different techniques have been used for filling in the `cameFrom` lists. The lists are used for binding each node to the rule that created it, and for binding each rule to the nodes it depends on for its execution, see section [4.3 Tracking Rule Execution].

The main reason for changing how the lists are filled in was to make it as easy as possible for the rule developers to write rules. Since the first attempt our goal has been to make it less intrusive, i.e. get rid of all unnecessary steps when writing rules. Ideally, the user should never have to think about this function, everything should be handled automatically behind the scenes.

6.2.1 First Attempt - Java Varargs

In the first solution, the user had to manually pass a reference to each object used in the rule, whenever a node was created and inserted. These objects were inserted directly into the node's `cameFrom` list. Not only that, but the name of the rule was also required as an argument to the constructor. An example is shown below in the rule named "First attempt":

```
1  rule "First attempt"
2    when
3      $t : Task()
4      $r : Request()
5      JattLog() // Will not be connected
6    then
7      // some use of $t and $r
8      insert(new Node(arguments, "First attempt", $t, $r));
9  end
```

The first thing to notice is that every object used in the condition part (when) of the rule needs to be assigned a variable in order to be passed to the `Node` constructor, e.g. `$t` and `$r`. The constructor receives the objects as varargs, which means any number of objects can be passed. `Node` in the example refers to any class in the `Node` hierarchy, and `arguments` refers to the specific arguments of those classes' constructors. The object `JattLog` is not used in the consequence part of the rule; it is only a condition for the rule to execute. Normally, in standard Drools, it would have been correct not to assign it to a variable. However, with this solution it is required to pass the `JattLog` object to the node's constructor, in the same way as for the `Task` and `Request` objects, and therefore

it must be assigned to a variable as well. In the example, someone has made an error and forgotten to pass it, which means it will not be shown as having triggered this rule. This could be very misleading when viewing the graph.

The worse thing with this solution is the lack of feedback when passing the objects, there is no way to know if some objects are forgotten. The reason for this is that one must be able to pass any number of objects, so the program can not just count them to make sure they are all there. At least an error is given if the rule name is missing, but there is no check if the name is correct or not.

Pros

- No nodes are affected if any references are forgotten, just the edges between them. In other words, the reasoning part is not affected, just the graphical representation.

Cons

- The user must remember to pass the objects and the rule name to every new `Node` inserted.
- No feedback is given if an object is forgotten.

It was soon obvious that a new solution was needed; even though we ourselves were the developers we kept making mistakes and ended up with strange errors in the graph. It would be foolish to think that someone else would have a better chance of succeeding.

6.2.2 Second Attempt - The Drools Object

We tried hard to find a better solution, but found no alternative by continuing to enhance the class structure. The key to the new solution came unexpectedly from the minor annoyance of being forced to write the rule name in multiple places within the same rule, as required in the first solution. We figured it would have been strange if one could not, at least, get the name of the currently executing rule, to at least improve that part. Therefore, instead of trying to enhance our own structure we changed focus and examined Drools' manual and API. The method `drools.getRule().getName()` was found in the Drools manual [JBoss Community, 2009, 4.8.3. The Right Hand Side] and the built-in object `drools` was examined in the debugger. This way we also found that references to the objects in the rules were accessible through the same object with the method `drools.getTuple().getFactHandles()`. The rule writing was hugely improved; now the only mandatory argument to the `Node` constructor was the `drools` object.

```
1  rule "Second attempt"
2    when
3      $t : Task()
4      $r : Request()
5      JattLog() // Is connected
6    then
7      // some use of $t and $r
8      insert(new Node(arguments, drools));
9  end
```

It was no longer necessary to assign `JattLog` to a variable; it would be connected anyway. This was a significant step towards our goal of eliminating all unnecessary steps in the process of writing rules. The only thing one had to remember was to pass the `drools` object to the constructor, and if it is forgotten a compiler error message would inform of it.

Pros

- Feedback in the form of a compiler error if one forgets to pass the required `drools` object.
- Always the same *one* object that needs to be passed.
- No need to assign objects to variables any more if they are not actually used in the rule.

Cons

- One still needs to pass the `drools` object, which is an extra step in the rule writing.

At this point we thought we had found the best possible solution. We were no longer actively searching for an alternative, but progress in another part of the code revealed an even simpler solution.

6.2.3 Third Attempt - Event Listeners

During the development of the error handling functionality we needed a way to keep track of which rule was executing when an exception was thrown. If the technique with the `drools` object from the second attempt had been used it would have required `drools` to be passed to any method that needed to throw an exception. This would have been very impractical. To find a better solution we investigated the source code for the built-in Drools logger, because we knew from the manual that it logged all the rule executions to a file. For more information about the Drools Logger, see subsection [3.2.4 Drools Logger and Audit View]. It was discovered that it used several internal event listeners. For example, `AgendaEventListener` is called on activation-related events, including before and after a rule is executed. `WorkingMemoryEventListener` is called when objects are inserted, updated and retracted. We combined these two listeners to a logger for both error handling and rule tracking called `ExecutingRuleLogger`.

One should keep in mind that there is a `cameFrom` list in both rule nodes and object nodes. The lists in the rule nodes are filled in when the method `beforeActivationFired` is triggered, whenever a rule is about to run. All of the objects required for triggering the rule are inserted into the `cameFrom` list of the rule node. A reference to the rule is also saved for later and used when filling in the `cameFrom` list in the object nodes. This is done in the method `objectInserted` and the saved reference is added to the `cameFrom` list of the inserted node.

This change improved the rule example even more:

```
1  rule "Third attempt"
2    when
3      $t : Task()
4      $r : Request()
5      JattLog() // Is connected
6    then
7      // some use of $t and $r
8      insert(new Node(arguments));
9  end
```

Pros

- Nothing extra is required, only the regular arguments.

Cons

- *None*

In this third attempt, we have actually reached our goal to not introduce new steps to the user when writing rules. Drools' standard syntax is used without any additional requirements.

6.3 Presenting Why-Chains

A why-chain shows which rules and objects have contributed to reach a certain node. Our main goal was to make the overview of the why-chain as good as possible, and at the same time easy to generate. At this point we saw the why-chains as trees and tried to print them as such. Figure 6.2 shows a small sample of this, but also the main problem with this non-graphical solution. The object `JattLog` is printed twice and it is impossible to see if it is supposed to be the same object or another object with the same name. In this case it is the same object, which means it got two parents and therefore why-chains can not be thought of as tree-structures. This also demonstrates the problem that the entire structure underneath `JattLog`, in this case the rule `Look for JattLog`, will be printed twice. One can easily see that the non-graphical solution will be impossible to overview and extremely hard to interpret as the number of nodes grows.

We needed a better way to present the why-chains, which both allowed non-tree structures and offered a good overview of the why-chain. The solution was to print it in a graphical form as shown in figure 6.1. It is now obvious that `JattLog` is one single object, which is required by two rules, as illustrated by the arrows. If two nodes with the same name are present it means two different objects have been created. In the same way, if more than one rule node with the same name is present the rule has been triggered multiple times.

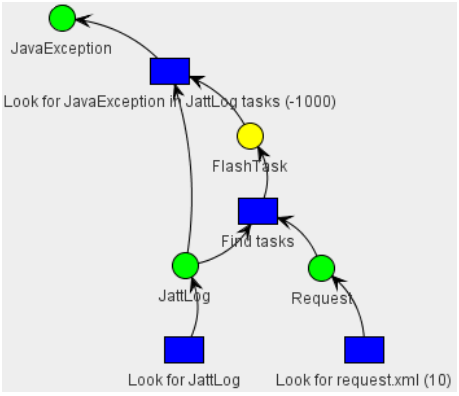


Figure 6.1: Graphical representation of the why-chain.

```
JavaException
  Look for JavaException in JattLog tasks
    FlashTask
      Find tasks
        JattLog
          Look for JattLog
        Request
          Look for request.xml
    JattLog
      Look for JattLog
```

Figure 6.2: The arrow shows the problem with a non-graphical representation of the why-chain.

6.4 Representing Non-Existent Objects

One problem when developing ADLA was how to represent objects that do not exist. For example, we have a rule that needs to be triggered if the object `TestSummary` does not, and will not, exist. This could happen if the test report is missing. A naive solution could be to just put `not TestSummary()` as the rule's condition. The problem then is that the rule could be executed before the `TestSummary` is produced. One must somehow make sure that everything that could lead to a `TestSummary` being produced is executed first. This can be done in numerous ways, choose the most suitable for the particular problem.

- With the keyword `salience` it is possible to prioritize the rules in Drools. Of all the rules considered for execution, the one with the highest priority will be executed. The standard salience for the rules is 0. To make a rule execute after another simply assign it a lower salience (negative numbers are allowed). The advantage with this method is the simplicity. The disadvantage is that if it is used too much it is hard to overview all the rules with modified priorities and it may be risky to add or adjust them. This was among other things the problem that should have been alleviated by the Map functionality, see section [5.4 Map for System Overview].
- Critical objects can be inserted into the working memory before the rule engine starts. This way, if an object does not exist, one knows for sure

that it will not appear later. This of course assumes that there are no rules that create it.

- The rule that creates the critical object can insert another object instead, if the critical object should not be created. This object symbolizes the non-existence of the critical object. In the example above one would for example introduce the object `NotTestSummary` to symbolize that the object `TestSummary` had not been created. A variant of this is to let the rule always create a special object to signal that it has been run.
- In Drools rules can be grouped into so-called agenda-groups. The rules in a group can only be executed if that group has focus. In this way layers of rules can be created, which are executed one at a time. If one knows that a critical object only could have been created on a previous layer, the rule can presume that if it does not exist, it will not be created.
- A similar problem occurs if one wants to execute several rules on the same object and want to make sure they are executed in the correct order. This can be done with some of the techniques above, but a better solution is to introduce an attribute in the object that keeps track of what state the object is in. This is used as a condition in the rules, and the attribute is also updated by the rules. This method requires slightly more work than prioritization, but is more robust.

We have prepared ADLA to use two agenda-groups, as described above. The rule attribute `agenda-group` can be either "layer1" or "layer2", where layer1 is run first. If the attribute is not set to either of them ADLA will not run, but leaves a message with what rule, in which rule file has the invalid agenda-group. We call them layers to signal that they will be executed in order, the second starts when the first one is finished. This is like running two separate rule bases, but on the same working memory. This is useful for example when one wants to ignore an error under special circumstances. The rule that suppresses the error is placed in the second layer to make sure it does not affect the other reasoning, and that the information it bases the decision on is complete. Adding new layers is very easy; simply add it to the `agenda_groups` list, found in our class `Globals`. The layers will receive focus and be executed in order, starting with the first one in the list.

Chapter 7

Result

The result of the project is ADLA; a system written in Java to analyze logs and dump files from automated tests on mobile phones. The analysis is performed with a third-party rule engine called Drools. ADLA can be run both as a stand-alone application, to be able to reevaluate previously generated results, and as an integrated part of JATT.

ADLA contains two separate parts, one for the automatic reasoning and another to graphically present the results. Graph data is transferred between them by means of an XML file, which is stored in JATT's result directory. The graphical part of ADLA is an applet for viewing and interacting with the result. The applet uses JUNG, a third-party framework to draw graphs.

The focus of the project has changed from creating a complete knowledge-filled expert system, tailored to ST-Ericsson's needs, to become more of a demonstration of an expert system's potential and a foundation for further development. The main reason for this has been the problem of finding relevant expert knowledge to insert into ADLA. Another contributory factor is how fast the different mobile platforms are changing. Some of the test suites that were used when we started the project are now more or less obsolete, while others have been added. This shows how important it is to keep the knowledge up to date.

7.1 ADLA Program Flow

Figure 7.1 shows the program flow between ADLA's major components. The standard way to run is through JATT with the method `execute`, illustrated by the arrow between JATT and ADLA. The alternative way is as a stand-alone program through `RunExpertsystem`.

When ADLA has been started it initializes the rule engine Drools, which reads the rules from the rule files and starts the analysis. As a first step in the analysis, files in JATT's result directory are parsed and inserted as facts in Drools' working memory. Drools continues to reason by combining the rules with the facts until no more rules can be executed. When the reasoning is

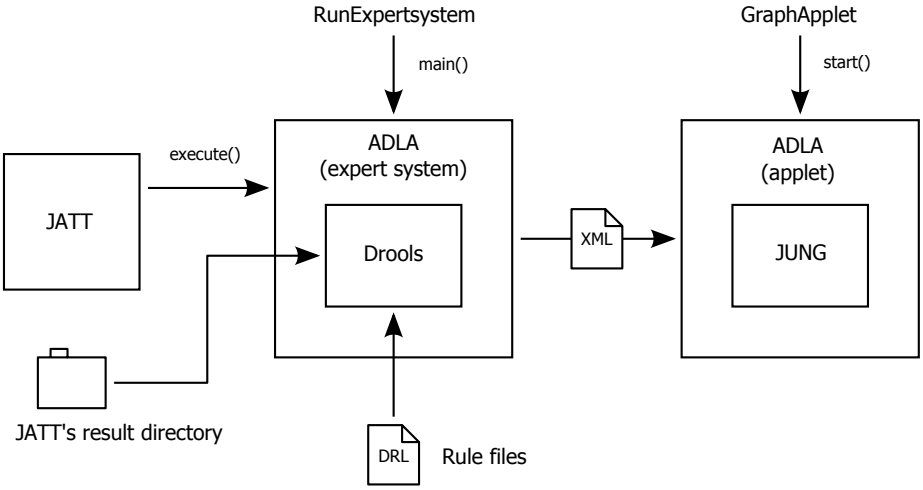


Figure 7.1: Program flow between ADLA's major components.

done ADLA produces an XML file containing graph data. All files produced by ADLA are stored in a sub-folder in the result directory. For viewing the result the file is opened by the graphical part of ADLA. This part uses the framework JUNG to visualize and make it possible for the user to interact with the result.

The map is stand-alone developer tool separated from the other parts of ADLA. It intends to give the rule developers an overview of all the rules and objects in the system and how they are connected. This becomes more important as the system grows. As seen in figure 7.2 the map functionality uses a small part of Drools to parse the rules. Together with ADLA's source code it is able to connect objects and rules and show them as a graph using the JUNG framework. The map is discussed in section [5.4 Map for System Overview].

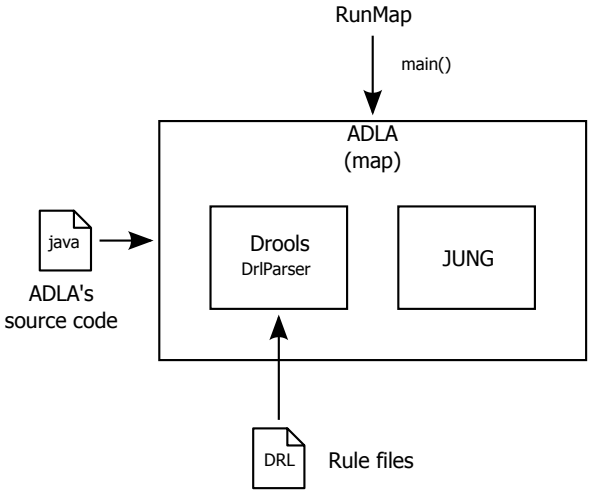


Figure 7.2: Program flow for the map program.

7.2 ADLA Features

Here is a selection of features in ADLA:

- The following test suites are supported: JDTS, Atlet, LabView, Sun TCK, IBM TCK, Nokia TCK, Gatling TCK and Benchmarks.
- Task objects are created from request.xml, so that errors and activities can be bound to them.
- Each exception found in the JATT log is bound to the task run when it occurred.
- The JATT log is used to find failed tasks and to determine an individual status for each task to one of the following: `NOT_SET`, `RUN`, `TIMED_OUT`, `NOT_RUN` or `CRASHED`.
- Core dumps are automatically translated into HTML reports using chkArm, ST-Ericsson's standard application.
- The header from the core dump file is displayed if no translations could be made by chkArm. If the report is available the "Decoded error" from chkArm, containing explanations of error codes, is used.
- Special exception type, `ErrorNodeException`, is provided to simplify error reporting in rules.
- Special status for the test suite result is available to be able to run JATT and ADLA as a step in other test software. The status can be `NOT_SET`, `PASSED`, `ERROR` or `FAILED`.
- A summary of all rules executed and how long they took to run is generated to the file ruleTimes.txt. This makes it possible to identify rules with performance issues.
- Partial support for handling reruns of the test suites with output to the same result directory [6.1 Keeping Track of New File Contents between Reruns].
- The result is drawn as a graph with the possibility to see how the result was reached.
- Dynamically updated explanation for each node type currently used in the graph.
- Possibility to switch between nodes marked as important and all nodes that were present in Drools' working memory.
- Extra information about nodes is presented by a simple right-click. For rule nodes both rule code and its attributes are shown. For object nodes the message can be customized by the rule developer.
- A map that can visualize all objects and rules used in the system and how they are connected to each other.

Chapter 8

Conclusion

The quality of an expert system largely depends on the knowledge in it. Therefore, it is important that the person with the best knowledge for each area of competence formulates the knowledge; one does not ask a lawyer questions about surgery, one asks the best surgeon available. Because we are no surgeons our focus was to find experts with suitable knowledge, but even with help from our supervisors we had no luck.

Some knowledge was still needed in order to design the features we wanted. So instead we started to look for more common problems by going through a large amount of result directories from old test jobs, to find suitable knowledge. Many of the errors found were too complex for us to understand, others turned out to be rare bugs that will not appear again, or bugs that already had been fixed. It turned out that finding knowledge was one of the hardest things in the entire project! Our focus was changed to designing a system with great extensibility and features rather than one filled with knowledge.

8.1 Work Strategy

It has been our ambition through the entire work process to be as agile as possible. We have never hesitated to make huge changes that affected the entire system, as long as it benefits the users or the code structure. The advantage with this strategy is that workarounds and other code patching are minimized in favor of renewed designs, which should improve the code quality. Many of the examples in chapter [6 Problems and Solutions] have changed successively through the work process.

Having a good design of the code is extra important in this project compared to others. This is because some of the users will change and add both code and rules in their everyday use of the system. When adding new areas of competence having a good design is especially important.

8.2 Aims and Goals Evaluation

The list of aims and goals in the introduction has changed many times during the project, in discussions with ST-Ericsson. For example, it was agreed to skip the requirement of XML formatted output in favor of the graphical solution. The following reflects the requirement list from section [1.2 Aims and Goals]:

- Because the rules are written directly in Eclipse there was no need for us to design an input interface.
- During a meeting with one of the experts developing chkArm we were informed that the team behind the program already had decided against some features (dump report analysis) we had intended to include in ADLA. Besides, they would actually have been better placed directly in chkArm. The memory leak detection is a part of the dump report analysis since it is in the dump reports potential memory leaks are found. The detection would have had to be a small expert system itself. Instead, it was agreed to include either the dump header from the core dump file, or the decoded header from the dump report.
- The output format from the reasoning part has changed to be graph data, which can be viewed in the graphical part of ADLA. This part is designed as an applet to also meet the requirement of displaying the result on a web page.
- To make sure it is easy to add new knowledge to ADLA the use case in [A.1.4 Adding a New Rule] was used and checked periodically with ST-Ericsson.
- Of course it is possible to add new areas of competence to ADLA; each step is documented in the use case [A.1.5 Adding a New Area of Competence].
- An applet is provided for displaying the result on a web page.
- All test suites, regardless of their priorities, listed in section [1.4 Boundaries] can be read by ADLA, except for Android CTS. This is because no example of a failed Android test could be provided in order for us to write a parser.
- As already pointed out, it was agreed that a specific part of the dump report was enough to cover ST-Ericsson's needs.
- ADLA automatically runs chkArm, which makes the translations from the error codes to their assigned names.

The cancelled requirements above have one thing in common, they are all about core dumps and dump report analysis. The dump report is generated from the core dump by the program chkArm, which has previously been started manually using a script. We translated the script to fit in ADLA, so the report is now generated automatically and included in the result when a core dump file is detected.

According to the expert we have talked to, dump analysis is the last resort to locate the cause of a dump. If the error is repeatable, a much better choice is to use a debugger like Trace32. Core dumps are an unusual outcome of tests run through JATT, so when the required time to create a working analysis was discovered, ST-Ericsson did not want us to waste time on it. Instead we agreed that the header part of the dump report was enough to cover their needs at the moment. If the features are still wanted in the future it is possible to extend ADLA with the needed knowledge, the expertise is available.

To compensate for the items cancelled we focused more on other types of errors and functionalities to improve the user experience. For example, we built a graphic explanation to the result and created an exception handler. For more examples see section [7.2 ADLA Features].

8.3 Future Work

Knowledge to put in the system has constantly been in short supply throughout the whole project. Much more work has to be put into finding expert knowledge and useful common knowledge suitable for ADLA.

The core dump functionality must be completed with a timer for timeout. This is because it uses the external program chkArm, which sometimes freezes mysteriously when run through Java.

Target limits for the benchmark suites have not been available, so at the moment no benchmark tests will fail as a consequence of a performance value not being good enough.

Bibliography

Joseph Giarratono and Gary Riley. *Expert Systems Principles and Programming*. PWS Publishing Company, third edition, 1998. ISBN 0-534-95053-1.

Jan Goyvaerts and Steven Levithan. *Regular Expressions Cookbook*. O'Reilly Media, Inc., 2009. ISBN 978-0-596-52068-7.

JBoss Community. *Drools Expert User Guide*, 5.0 final edition, May 2009. http://www.jboss.org/drools/documentation.html (manual verified 2010-06-02).

Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers, Inc., 1992. ISBN 1-55860-191-0.

Alex Toussaint. *Java Rule Engine API JSR-94*. BEA Systems Inc, 1.0 edition, September 2003. JCP final review (doc/jsr94_spec.pdf) included in the zip file from http://jcp.org/aboutJava/communityprocess/final/jsr094/index.html (verified 2010-05-25).

Appendix A

Manual and Guidelines

A.1 Use Cases

These use cases describe the steps needed to perform some typical tasks in ADLA. This only reflects how the system behaved at the time of delivery.

A.1.1 Set Up ADLA to Run Through JATT

ADLA is built as an Eclipse plugin to be able to start ADLA from JATT. The activator `AdlaActivator` is located in com.stericsson.jatt.adla.internal. This class uses `ExpertSystem` from com.stericsson.jatt.adla.expertsystem and the JATT entry point is the method `execute`.

1. Create a new instance of `ExpertSystem`, which takes the path to the rule directory as an argument.
2. Call `execute`, which takes an `ITestJobContext` as an argument.
3. ADLA should be called after JATT has finished running the test job.

A.1.2 Running ADLA as a Stand-Alone Application

The class `RunExpertsystem` in package com.stericsson.jatt.adla.expertsystem starts ADLA without JATT, which is useful when re-running old test jobs.

1. Set `Globals.TEST_JOB_NAME` to the test job name, e.g. "DbTestJob-61".
2. Set `Globals.TEST_JOB_DIR` to the path of the result directory (where the JATT log is located), e.g. `new File("C:\\testdata\\DbTestJob-61")`.
3. Execute the class `RunExpertsystem`.
4. Now the reasoning part of ADLA has created a file called output.xml in the ADLA subdirectory of the result directory.
5. To view the result, follow the steps in [A.1.3 Viewing the Result].

A.1.3 Viewing the Result

The class `GraphApplet` in package com.stericsson.jatt.adla.applet starts the graphical part of ADLA, which is used to view the result.

1. Set the `file` variable to the path of the output.xml file created by the reasoning part of ADLA. It can be found in the ADLA subdirectory of the result directory.
2. Run the applet.

A.1.4 Adding a New Rule

When you want to add knowledge to ADLA, it has to be formulated in a rule. To do this the right way you must overview the rules and objects in order to locate which are likely to be affected by the new rule. You may find the map helpful (but do not trust it too much), see section [5.4 Map for System Overview].

1. Find a suitable rule file for the rule to be placed in, or create a new.
2. Make sure the objects that the rule will interact with exist, or you will have to create them.
3. Write the rule. See [3.2.2 Rule Files and Rule Syntax] for some basic syntax.
4. Decide which layer is best suited for the rule and set the attribute `agenda-group`.
5. If the rule needs to be prioritized set the attribute `salience`.
6. Run ADLA in stand-alone mode to try the new rule on a previously run test job where the outcome can be predicted.

A.1.5 Adding a New Area of Competence

1. Create the classes needed for the new area of competence. Make sure they all extend `Node` or one of its sub-classes. Place them in the most suitable package, either `basenodes`, `nodes` or `parsers`. Choose the basenodes package if the class is used to store parsed information directly from the input files, parsers if the class is used to parse a test suite or else place it in nodes.
2. Override the method `toString` to control the label in the graph. The default is the name of the class.
3. Override the method `getNodeInfo` to control the extra information displayed when right-clicking the node in the graph. The `String` returned is formatted as HTML and the default is to print the `conclusion` attribute. The heading is added later and should not be included, also the attribute `alwaysDisplay` is printed if it is set.
4. Write the rules needed to describe the knowledge, see [A.1.4 Adding a New Rule].

A.1.6 Removing a Rule

When removing a rule you must make sure that all dependencies are checked so other rules are not left unreachable. Even a small change in a rule can trigger a chain reaction so other rules are not executed. You may find the map helpful (but do not trust it too much), see section [5.4 Map for System Overview].

1. Find the rule in one of the rule files.
2. Identify which objects are inserted, updated and removed (also in called methods) and make sure you do not break anything.
3. Delete the rule.

A.2 Remove Rule When ADLA is Running

In some situations it may be useful to remove a rule while the system is running. When you, in a rule, update the same object that is required for triggering the rule, you may end up in an infinite loop. This is because when an object is updated a new pattern match is performed, which will detect that the rule's condition part is true and the rule will be executed again.

To avoid this problem Drools has the rule attribute `no-loop`, which prevents a rule from rescheduling itself. However, if two rules keep activating each other this does not help. We have provided a method that removes the rule from the knowledge base, which means it cannot be reactivated until ADLA is restarted. The method is placed in the class `RuleUtils` and is called `removeThisRule`. It takes the object `drools` as an argument and is intended to be called directly in the rule. This way you are sure that the rule is not executed again.

A.3 Add New Node Types

The node types control the shape and color of the nodes when they are drawn. They are defined in the class `VertexType` in the package com.stericsson.jatt.adla.graph. To add a new node type, simply add a new line in the `types` vector. The first argument to the `VertexType` is the id, which can be set to anything because it is only used as an identifier internally in the graph file. The second argument is the label text that is printed on the nodeInfo-panel in the applet. `\n` is recognized so you can decide where the newline should be. The third argument is the color of the node and should be written as a regular hex coded RGB string, e.g. "FF8080" for light red. The fourth argument specifies the shape of the node and at the moment only "Rectangle" and "Ellipse2D" are recognized. More shapes can be added in the class `VertexType` in the package com.stericsson.jatt.adla.applet. The last two arguments specify the width and height of the shape, in pixels. Finally you need to decide when to use the new node type, which is done in the method `getNodeVertexType`. The method is called once for every node in the graph and should return the id of the type you want to use for that particular node.

Index