Researching Integration of ART Test Cases into JCAT

Thesis Work for Bachelor of Computer Science

| Students:   | Petter Isberg & Thomas Åkerlund |
| Company:    | Ericsson AB                      |
| University: | School of Innovation, Design and Engineering
              Mälardalen University, Västerås |
| Company Supervisor: | Ian Kavanagh |
| University Supervisor: | Björn Lisper |
| University Examiner: | Björn Lisper |
| Date:       | 2013-06-11                       |
Abstract

Ericsson AB is a world-leading provider of telecommunications equipment and services to mobile and fixed network operators. Ericsson uses continuous integration testing to ensure that their equipment is top tier market products. The Ericsson WCDMA division has one older Erlang based software testing framework called ART and a newer Java based framework called JCAT. This thesis work aims to find a solution for JCAT to execute ART tests and evaluate and log the test results in JCAT.

This report describes the research results in executing ART from JCAT, gathering ART test case result data and sending it back to JCAT as well as a describing a proof of concept implementation using the research results. The suggested solution runs the ART tests by executing a Bash script. The communication was solved by using an event handler which gathers data from the ART tests and sends it back to JCAT using Erlang node communication.

The result is a solution which satisfies the requirements specified. If the implementation were to be used in a production environment it would still need some future work.

Sammanfattning

Ericsson AB är en världslig tillverkare av utrustning för telekommunikation samt tjänster för mobila och fasta operatörer. För att säkerställa kvaliteten på utrustningen använder sig Ericsson av testning med kontinuerlig integration. Ericssons WCDMA-avdelning har ett äldre ramverk för testning baserat på Erlang som heter ART samt JCAT som är ett nyare ramverk baserat på Java. Detta examensarbete siktar mot att hitta en lösning för att exekvera ART från JCAT och sedan evaluera och logga resultaten i JCAT.


Resultatet är en lösning som uppfyller kravspecifikationen. För att få implementationen redo för en skarp produktionsmiljö skulle det krävas en del framtida arbete.
Acknowledgements

We would like to thank Ericsson for giving us such a great thesis task.

We would like to thank our supervisor at Ericsson, Ian Kavanagh, for the help with the Erlang/ART implementation and for getting us in touch with the right people whenever there was a question he could not answer himself.

The section manager Sten Mogren have showed a lot of interest in our work, attended our weekly scrum meetings and gave us support when we visited the Ericsson offices in Stockholm.

We would also like to thank the Ericsson employees Martin Aldrin and Johan Lundberg for all the support and help during the work.

We want to thank supervisor Björn Lisper at Mälardalen University for his support.

Finally we would like to thank Daniel Flemström for all the support at the Industrial Research and Innovation Lab at Mälardalen University. Without him we would not have got in contact with Ericsson in the first place.
# Table of Contents

Abstract .......................................................................................................................... ii
Sammanfattning ............................................................................................................... ii
Acknowledgements ........................................................................................................ iii
Table of Contents .......................................................................................................... iv
Terms and Abbreviations ............................................................................................... vi

1 Background .................................................................................................................. 8
   1.1 Ericsson .................................................................................................................. 8
   1.2 The Thesis Task .................................................................................................... 8
   1.3 Software Testing .................................................................................................... 8
   1.4 The Erlang Programming Language ...................................................................... 9
      1.4.1 The Erlang Virtual Machine and Distributed Erlang ...................................... 10
      1.4.2 Open Telecom Platform (OTP) ...................................................................... 10
      1.4.3 Erlang Common Test .................................................................................... 11
      1.4.4 ART ............................................................................................................. 11
   1.5 Java ....................................................................................................................... 11
      1.5.1 The Programming Language and Virtual Machine ....................................... 11
      1.5.2 The TestNG Framework .............................................................................. 12
      1.5.3 JCAT .......................................................................................................... 14
2 Related Work .............................................................................................................. 14
   2.1 JInterface .............................................................................................................. 14
      2.1.1 Type Conversion ......................................................................................... 14
      2.1.2 Mailboxes .................................................................................................... 15
      2.1.3 Connections ............................................................................................... 15
      2.1.4 Remote Procedure Calls ............................................................................. 16
   2.2 WEA .................................................................................................................... 16
   2.3 RRS ....................................................................................................................... 16
3 Problem Formulation ................................................................................................. 16
   3.1 Tasks .................................................................................................................... 16
   3.2 Implementation Requirements ............................................................................ 17
4 Problem Analysis ....................................................................................................... 17
   4.1 Research ............................................................................................................... 17
   4.2 Implementation ..................................................................................................... 18
   4.3 Evaluation ............................................................................................................ 18
5 Work Method ............................................................................................................. 18
6 Solution ....................................................................................................................... 19
   6.1 Overview .............................................................................................................. 19
The following list contains terms and abbreviations used in this report. This is to simplify for the reader if any word is unknown. Many of the listed abbreviations are described in more detail in the report so reading the list is not required to understand the report.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ART</td>
<td>An older RNC and TGS test framework developed by Ericsson. Based on Erlang Common Test.</td>
</tr>
<tr>
<td>Bash</td>
<td>A Unix Shell. Scripts have the power that they do not need to be compiled before being executed which makes them very portable between different system architectures.</td>
</tr>
<tr>
<td>CI</td>
<td>Continuous Integration is a practice where developers are supposed to commit changes frequently. The CI software will automatically run the appropriate tests to verify that the code meets all requirements.</td>
</tr>
<tr>
<td>IRIL</td>
<td>The Industrial Research and Innovation Lab is collaboration between Ericsson, Mälardalen University and the Swedish Institute of Computer Science located at Mälardalen University in Västerås.</td>
</tr>
<tr>
<td>JInterface</td>
<td>A Java library that enables communication between Java and Erlang.</td>
</tr>
<tr>
<td>Jcart</td>
<td>The project name of this thesis work. Jcart is a combination of the abbreviations JCAT and ART.</td>
</tr>
<tr>
<td>JCAT</td>
<td>A Java test framework developed by Ericsson. Based on TestNG.</td>
</tr>
<tr>
<td>JUnit</td>
<td>Testing framework in Java oriented towards unit testing.</td>
</tr>
<tr>
<td>Nite</td>
<td>An Ericsson test framework written in Python.</td>
</tr>
<tr>
<td>OTP</td>
<td>The Open Telecom Platform is an Erlang platform for creating fault tolerant enterprise systems.</td>
</tr>
<tr>
<td>RNC</td>
<td>Radio Network Controller. An RNC is a hardware device used to, as the name suggests, control radio network traffic. [RNC3810]</td>
</tr>
<tr>
<td>Suite</td>
<td>A collection of tests that performs a larger logical test.</td>
</tr>
<tr>
<td>SUT</td>
<td>The System Under Test refers to the system (hardware or software) being tested.</td>
</tr>
<tr>
<td>Test</td>
<td>Can either refer to a single test case or a collection of test cases and test groups performing one logical test broken down into smaller subtests.</td>
</tr>
<tr>
<td>Test Case</td>
<td>A single testing unit that performs a test.</td>
</tr>
<tr>
<td>Test Group</td>
<td>A grouping of test cases.</td>
</tr>
<tr>
<td>TestNG</td>
<td>Testing framework in Java with functionality for unit, integration and functional testing.</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TGS</td>
<td>TGS or 3GSim generates simulated 3G traffic for RNC node testing.</td>
</tr>
<tr>
<td>VM</td>
<td>Short for Virtual Machine.</td>
</tr>
<tr>
<td>WEA</td>
<td>A test framework communication implementation that is used between Nite and JCAT.</td>
</tr>
<tr>
<td>XML</td>
<td>The Extensible Markup Language is an open standard structured markup language. XML is very good at structuring data by using key-value pairs. [xml.com]</td>
</tr>
</tbody>
</table>
1 Background

1.1 Ericsson
Ericsson is the product owner of this thesis work. Ericsson is a world-leading provider of telecommunications equipment and services to mobile and fixed network operators. Over 1,000 networks in more than 180 countries use Ericsson network equipment, and more than 40 percent of the world’s mobile traffic passes through Ericsson networks [Ericsson1]. Software Top 100 ranked Ericsson as the 5th largest software company globally when it comes to revenues in 2011 [Top100].

1.2 The Thesis Task
Ericsson develops a large variety of hardware that is used to control and deliver mobile and fixed networks. There are several testing frameworks at Ericsson that are used to test the mobile network hardware and its software. Among them are JCAT and ART.

JCAT is written in Java and based on TestNG while ART is written in Erlang and is based on Erlang Common Test.

ART has been used for several years while JCAT is still being developed. The goal at Ericsson is to use JCAT only, but the transition takes time since there are several hundred test cases currently in ART that would have to be translated to JCAT. There is also a risk of introducing errors in previously correct test cases when translating them.

The problem with having these two testing frameworks running in parallel is that they are reporting to different databases making it difficult to inspect the results. It is also inconvenient to run the tests from different frameworks, especially when the tests are automated. To solve this problem, Ericsson wants to run all tests from JCAT as soon as possible.

This thesis work aims to find a way to run ART tests through JCAT as a temporary solution until the development of JCAT is complete. It is important that while ART is running the tests, JCAT will log all the results so that both ART and native JCAT test results end up in the same database.

1.3 Software Testing
Software can never be regarded as completely correct. The larger and the more complex the software, the more vulnerable it becomes to errors. Errors can be introduced into the project at all different stages of the development, from large scale planning to low level programming and module integration. The most important way of making sure the software behaves as planned is testing which can require up to 50 % of the software development cost and even more for safety-critical applications. It is also important to realize that tests can only show the presence of errors, they can never guarantee the absence of them. [Ammann08, p.10]

Software testing can be divided into different categories. Unit testing is about testing single units of code, for example testing each function in a class or module. Integration testing is above unit testing, focusing on testing that different units of the software works when they are integrated with each other. Functional testing, including acceptance testing, makes sure that the application functions correctly and according to the specification. [Eldh11, p.34]

Much of the testing is automated because of the costs and to make sure all relevant tests are actually run. Tests can be automated by using continuous integration (CI), where the developers are supposed to commit changes frequently and the CI software will automatically run the appropriate
tests whenever a code change has been committed. This gives the developers quick response to if their code works and whether or not it breaks some earlier functionality. Getting this response for every commit gives the possibility to always having working code by simply rolling back the version control system to the previous commit. [Duvallo07, p.12]

Tests are usually structured regarding what they test. A test case is a single method with an expected result. A test suite is a collection of related or unrelated test cases that can be run together. The system being tested by the test suites and cases is referred to as system under test (SUT). [Eldh11, p28] [IEEE 829]

### 1.4 The Erlang Programming Language

Erlang is a programming language created by Ericsson in 1986. The term concurrency oriented programming was coined by the creator Joe Armstrong to describe the language. [Armstrong07b, p.6-5]

Erlang was developed to be used in telephone switch applications where massive concurrency and high reliability was demanded. Erlang is often regarded to be a functional programming language with single assignment variables and functions with no side effects.

Erlang depend mostly on recursions instead of iterations since a variable that has been assigned cannot be assigned a new value. The single assignment property of Erlang means that processes usually don’t communicate with shared variables; inter process communications is instead implemented with message passing. Message passing is a core feature of the Erlang language since concurrency is such an important part of Erlang. Every Erlang process has a mailbox where incoming messages are stored and the process can access these messages asynchronously. Since the lack of shared variables reduces the risk of race conditions, it makes a large contribution to the stability and fault tolerance of Erlang.

Erlang has strong, dynamic typing of variables. This means that a variable is not declared with a type but it rather gets a type when it is assigned a value. A variable or a value is called a term in Erlang. A limited amount of variable types are used [Logan11, p 29]:

- Numbers (integer and floats)
- Binaries / Bit strings
- Atoms
- Tuples
- Lists (and strings)
- Unique identifiers (pids, ports, references)
- Funs (anonymous functions)

The type numbers include integers and floating point values and the type binaries are simply binary representations of values. The rest of the types are rather specific to Erlang.

Atoms are non-numerical constant symbols. Atoms like ‘ok’, ‘true’, ‘undefined’ and ‘Monday’ are stored in a dedicated table. Using an atom means using a reference to it in the table, making it fast to compare atoms for equality and making them more memory efficient than text strings.

Tuples are groups of terms enclosed in curly brackets. This makes it easy to pass around multiple values, for example when returning from a function. Tuples can contain any number and any types of terms, even other tuples, making it possible to create nested data structures.

Any type of terms can be mixed in a list. A string is actually an ordinary list of integers, only the output representation differs from any other list.
Erlang has some terms representing identifiers. The most interesting might be the pid, representing a process id. This is used when sending and receiving messages between processes and to check whether a process is still alive.

The last type is the fun, a variable type representing an anonymous function. Fun variables can be passed as arguments or used as return values when using higher order functions.

A heavily used and very powerful feature of Erlang is the pattern matching, making if statements almost obsolete. The programmer states different options, like in a switch statement. The system checks the options from top to bottom, using the first match. If no pattern matches, a bad match error is thrown.

```
area(Figure) ->
    case Figure of
    {square, Side} -> ok, Side * Side;
    {triangle, Base, Height} -> ok, (Base * Height) / 2
    end.
```

Return value often have an atom as the first term indicating whether the function executed correctly with the atom ‘ok’ or if it encountered an error. An error message from a function can for example look like

```
{error, "Error message"}.
```

By using pattern matching when receiving this value, the error can propagate upwards to a function dealing with the error.

```
{ok, Result} = area({circle, 5}).
```

Calling the area function with a circle will return a ‘badmatch’ error tuple since the function is only implemented for squares and triangles.

### 1.4.1 The Erlang Virtual Machine and Distributed Erlang

Erlang code is compiled into a machine independent intermediate language called Beam files, much in the same manner as Java uses its Java byte code. The Erlang VM handles scheduling of Erlang processes, I/O and memory management, including garbage collection.

Erlang has native support for distributed computing. If the Erlang VM is started with a node name as argument, the VM is considered to be a distributed Erlang node. Together with a distributed node the Erlang Port Mapper Daemon (EPMD) is started, if not already running. EPMD keeps track of the distributed nodes running on the current machine and what ports these nodes are listening to. This means that when connecting two nodes on different machines, they do not need to know which specific port number to connect to, only the name of the node and the machine it is running on.

When an Erlang node connects to another, for example by sending a ping or a message, a link is created and both nodes keep this connection until it is explicitly disconnected or breaks. Sending a message to a process on a connected node is made in the exact same way as sending a message to a local process since the process id (PID) term of a remote process contains information about what node the process is running on.

### 1.4.2 Open Telecom Platform (OTP)

The Open Telecom Platform (OTP) is a set of modules, interfaces and design principles that helps developers create stable, fail safe code. OTP has implementations for generic servers, events and supervisor processes. Creating a web server with OTP can be done by using a generic server in the background. Making it a web server is done by implementing the callback methods to serve web pages. In the same manner generic events have implementation for event managers that the
developer can attach specific event handlers with callback functions to properly handle the events according to the application.

### 1.4.3 Erlang Common Test

The Erlang standard distribution has two test frameworks, EUnit and Common Test. EUnit is used for unit testing while Common Test is used for large-scale testing and can run distributed tests while logging on one server. [Logan11, p.117-118]

Ericsson is using Common Test in their testing environment and there are a few terminologies that are important to understand in Common Test.

Suites are a collection of test cases and groups. Suites are used to simplify management and execution of tests and groups that belong together. The naming convention for suites is `<name>_SUITE.erl` where name is the name of the suite. [Common Test, p.3]

An implemented suite must include the interface function ‘all’. The function must return a list of all groups and test cases that are to be executed in the suite. There is also a set of optional interface functions for suites. Two interesting groups of functions are `init_per_TYPE` and `end_per_TYPE` which handles initialization and teardown of the types suite, groups and test cases.

Groups are a collection of test cases and/or groups. A group is defined within a suite and is used to wrap common initialization of tests into one initialization of the entire group.

Test cases are the actual tests, implementing the test functionality that is run by the Common Test framework.

Test Specification files, also called spec files, are files that specifies which tests should be run during a test execution and how. Using specification files instead of execution parameters simplifies repeated or automatic execution. [Common Test, p.40]

### 1.4.4 ART

The ART framework is an Ericsson test framework that is an extension of Erlang Common Test. ART is a collection of suites, groups, test cases, event handlers, logging managers, execution scripts and more. The framework is used for functional testing of RNC and 3GSim nodes developed by Ericsson.

### 1.5 Java

Java is a high-level Object-Oriented C-style language developed by Sun and later Oracle [Savitch10, p.40] [Oracle1]. Java focuses on simplicity together with architecture neutrality and multi-platform availability by using the Java Virtual Machine (Java VM) [Oracle2] [Oracle3]. By using a virtual machine a java program can be run on any platform and architecture with almost no modification, as long as there is a Java VM implementation for it. This is one of the key strengths of Java. Cross platform support made Java very popular when the World Wide Web was introduced, letting developers create programs to be run on web pages on different systems. Official numbers states that Java is now running on several billion devices over the world, from embedded systems to servers [Java.com].

#### 1.5.1 The Programming Language and Virtual Machine

There are two different program types that can be written in Java, the first is applications and the second is applets. An application is, as the name suggests, a standard computer program. Applets on the other hand are small Java programs that are used to run as web applications in web browsers.
Applications may either run as a console program or a window program, but applets always run as window programs.

A Java program consists of classes, methods and the main method. In Java, the name of a class must be the same as the name of a file [Savitch10, p.42]. It is common that each class in Java is located in its own file but there are a few exceptions such as inner classes.

To run a Java program, it must first be compiled into Java byte-code [Savitch10, p.41]. The byte-code is not natively readable by a computer and this is where the Java Virtual Machine (Java VM) comes into use. The Java VM is an interpreter that translates the Java byte-code into machine-language instructions [Savitch10, p.41]. These instructions can then be executed by the machine instructed to run the Java program. The reason for this is portability. A Java program can be run on any computer that has the Java VM available. There are of course special cases, where architecture specific instructions cannot be interpreted by other architectures.

There is a set of eight primitive types in Java. All other types are objects inheriting from the Object superclass. The following table is a list of the primitive types [Savitch10, p.48].

<table>
<thead>
<tr>
<th>Type</th>
<th>Kind of Value</th>
<th>Memory Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>true or false</td>
<td>1 byte</td>
</tr>
<tr>
<td>char</td>
<td>single character (Unicode)</td>
<td>2 bytes</td>
</tr>
<tr>
<td>byte</td>
<td>integer</td>
<td>1 byte</td>
</tr>
<tr>
<td>short</td>
<td>integer</td>
<td>2 bytes</td>
</tr>
<tr>
<td>int</td>
<td>integer</td>
<td>4 bytes</td>
</tr>
<tr>
<td>long</td>
<td>integer</td>
<td>8 bytes</td>
</tr>
<tr>
<td>float</td>
<td>floating-point number</td>
<td>4 bytes</td>
</tr>
<tr>
<td>double</td>
<td>floating-point number</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>

Two facts about primitives in Java that are worth noting is that there are no unsigned values in Java and a String is not a primitive type even though it can be used like one [Savitch10, p.48].

Java has support for a feature called annotations, used to add metadata to classes and methods. Annotations are connected to the elements using the at-sign (@). This is used by Java to for example mark methods overridden from the superclass or to mark methods as deprecated, using @Override or @Deprecated before the method declaration, respectively. These annotations are checked by the Java compiler to issue warnings or errors when using deprecated methods or trying to override final methods. There is also support in Java to make custom annotations that can be controlled during runtime. This enables annotations to be used to mark classes or methods as tests as described in the next section. To programmatically check the presence of such annotations during runtime, the method isAnnotationPresent() is available in the Class and Method super classes [Spell05, p.637].

1.5.2 The TestNG Framework

TestNG is an open source framework designed both for unit testing, functional testing and for integration tests. The framework is inspired from the Java unit testing framework JUnit. To support more than unit testing TestNG has functionalities that JUnit were missing such as annotations, dependencies between test methods, testing multithreading and support for test parameters.
There are two essential parts to use TestNG; the test classes implementing the test functionality and an XML file telling the framework which test cases to run. Figure 1 shows the structure of a simple test class using some the TestNG annotations and Figure 2 shows an example of a JCAT XML file.

```java
package examplePkg;

import org.testng.annotations.*;

public class MyTestClass {
    @BeforeClass
    public void beforeClass() {
        // Do some configuration that all the tests in the class needs.
    }

    @BeforeMethod
    public void beforeMethod() {
        // Do some configuration before each test method.
    }

    @Test
    public void myTest() {
        // Do the test!
    }

    @AfterMethod
    public void afterMethod() {
        // Do some cleanup of the @BeforeMethod changes.
    }

    @AfterClass
    public void afterClass() {
        // Do some cleanup of the @BeforeClass changes.
    }
}
```

**Code 1. Structure of TestNG test class.**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="MySuite">
    <test name="MyTest">
        <classes>
            <class name="MyTestClass">
                <methods>
                    <exclude name="myTestCase" />
                </methods>
            </class>
        </classes>
    </test>
</suite>
```

**Code 2. Structure of JCAT XML file.**

The suite is the top element of the testing structure in JCAT XML file and consists of one or more independent tests. The test has an internal state and provides a context for the test classes allowing them to share data.

A class is a collection of methods that belong together. Methods are the actual test cases. The test methods in TestNG can belong to one or more groups and the execution or exclusion of groups is controlled in the XML file. Groups are intended to give the tester more flexibility in regard of what test cases he or she wants to run without having to recompile the test classes. [Beust07, p.119]
The XML file defines the suites and tests independently while the classes, groups and methods must exist in java code before they can be used. [Beust07, p119]

1.5.3 JCAT

Java Common Automation Tester (JCAT), is an Ericsson test framework based on TestNG. Earlier versions of JCAT were based on JUnit3 but Ericsson decided to change framework to TestNG instead of upgrading it to JUnit4. JCAT is an extension to TestNG to give better support for different Ericsson SUT’s, automated tests and more logging options. [JCAT, p.11]

2 Related Work

2.1 JInterface

JInterface is a library for Java that is written by Ericsson. It handles communication between Erlang processes and its main purpose is to let Java communicate with Erlang processes as if Java was an Erlang node itself. It is also possible for two Java processes to communicate using JInterface since they would both appear as Erlang nodes to each other. [JInterface UserGuide]

JInterface classes can be split into two categories, those that handle the communication with Erlang nodes and those handling type conversions between Erlang and Java types.

JInterface is used in Java by creating an OtpNode instance acting like an Erlang node. The node can then communicate asynchronously with other Erlang processes by creating an OtpMbox instance behaving like an Erlang process mailbox. JInterface communicates with other Erlang processes using either its pid or registered name just like a native Erlang node.

2.1.1 Type Conversion

JInterface can do conversions between Erlang and Java types. The following table describes the type mappings.

<table>
<thead>
<tr>
<th>Erlang type</th>
<th>Java type</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom</td>
<td>OtpErlangAtom</td>
</tr>
<tr>
<td>binary</td>
<td>OtpErlangBinary</td>
</tr>
<tr>
<td>floating point type</td>
<td>OtpErlangFloat or OtpErlangDouble (depending on the value)</td>
</tr>
<tr>
<td>integral type</td>
<td>OtpErlangByte, OtpErlangChar, OtpErlangShort, OtpErlangUShort, OtpErlangInt, OtpErlangUInt or OtpErlangLong (depending on the integral value and sign)</td>
</tr>
<tr>
<td>list</td>
<td>OtpErlangList</td>
</tr>
<tr>
<td>pid</td>
<td>OtpErlangPid</td>
</tr>
<tr>
<td>port</td>
<td>OtpErlangPort</td>
</tr>
<tr>
<td>ref</td>
<td>OtpErlangRef</td>
</tr>
<tr>
<td>tuple</td>
<td>OtpErlangTuple</td>
</tr>
</tbody>
</table>
### Mailboxes

Mailbox communication in Erlang is used for asynchronous message passing between Erlang processes. Each process has an implicit mailbox that is used to receive messages sent to it. Mailboxes have no size limit on the message queue they hold. [Logan11, p.106]

The mailbox is communicated with through the registered name or pid of the owning process. It is also possible to explicitly create a mailbox with a specified name that other processes can send messages to.

JInterface uses `OtpMbox` to mimic this functionality. The `OtpMbox` created within JInterface gets a unique pid but can also be contacted through a registered mailbox name.

A JInterface mailbox is created using the following code.

```java
OtpMbox mbox = node.createMbox();
```

The variable `node` in this case is an instance of `OtpNode`. This mailbox is created without a registered name and can therefore only be communicated with using its pid.

To create a mailbox that can be communicated with, not only by its pid, but also by its registered name, the following code is used.

```java
OtpMbox mbox = node.createMbox("name");
```

In this case `name` is the desired name to be registered.

It is also possible to register a name for a mailbox after it has been created.

```java
mbox.registerName("name");
```

The variable `mbox` is an instance of `OtpMbox`.

The reason for using registered names instead of pids is that a pid cannot be known in advance while a name can be.

### Connections

Erlang connections are loosely connected and are implicitly set up when nodes are communicating for the first time. While for example pinging a node the JInterface `OtpNode` will set up a connection if it does not already exist to the node being pinged. This happens with any kind of process communication in Erlang and is not specific for ping. [Erlang Doc]

To see if another node is accessible the same technique as other types of communication can be used. By simply sending a ping to the remote node the response will tell whether it is available or not.

```java
if (node.ping("name", 2000)) { The node is available. }
else { The node is not available. }
```

The variable `node` is an instance of the JInterface object `OtpNode`. The ping method will return a boolean value telling if the ping was successful or not. The second parameter in ping specifies the timeout in milliseconds since the ping was sent.

In the above example, if ping gets a response it has implicitly also created a connection to the responding node that is registered with the name “name”.

---

<table>
<thead>
<tr>
<th>Erlang type</th>
<th>Java type</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
<td>OtpErlangObject</td>
</tr>
</tbody>
</table>
A connection can only be established between two nodes using the same security cookie. A security cookie is a string of characters that is used for authentication on incoming connections to verify that the incoming connection is allowed.

### 2.1.4 Remote Procedure Calls

Remote procedure calls, in short RPC, is node to node communication where one node asks the other to execute a specific function with a set of parameters. Remote procedure calls are very useful in Jinterface when trying run Erlang code while using Java.

The following code asks the Erlang node to execute the function date in the module `erlang` with an empty list as parameter:

```java
connection.sendRPC("erlang", "date", new OtpErlangList());
OtpErlangObject received = connection.receiveRPC();
```

The variable `connection` in this example is an instance of `OtpConnection`.

### 2.2 WEA

WCDMA Extension Adapter (WEA) is an internal Ericsson software framework developed to be used in a similar way to what this thesis work is intended. WEA is a framework that lets JCAT execute functions from the test framework Nite that is based on the programming language Python. This implementation uses sockets for communication between JCAT and Nite. The socket communication is written in Python and is designed to let JCAT connect to it and run desired functions.

There is a framework called Jython (previously JPython) that at first glance resembles Jinterface. The reason why it was not used in the development of WEA is that instead of being a communications framework it is an inter language compilation framework, meaning that Python code is compiled to Java Bytecode using Jython. This caused problems for the WEA developers because some libraries used by Nite could not be compiled by Jython.

### 2.3 RRS

RRS is an internal Ericsson database that is used to store ART test case information and results. In Erlang Common Test and by extension ART it is possible to assign an event handler that receives events about the state and results of the testing. RRS takes advantage of this feature by adding an event handler to the ART execution that sends the event information to the RRS database during the testing. The information is then stored and is easily accessible by Ericsson staff.

### 3 Problem Formulation

#### 3.1 Tasks

The problem has been divided into several tasks to give a clear view of what the thesis work consists of.

1. Investigate communication between ART and JCAT. A method is needed for JCAT to get information from ART about what tests have been carried out and their results. Research is required to find out what possible solutions there are and to compare their respective strengths and weaknesses.
2. Investigate execution of ART test cases from JCAT. JCAT shall be able to seamlessly execute ART test cases as if they were JCAT test cases.
3. Evaluation of the possible solutions according to what Ericsson needs. The evaluation shall compare areas such as usability, reliable communication, logging and data translation.

4. A proof of concept implementation using the results of (1), (2) and (3).
   a. JCAT shall be able to seamlessly run one ART test case and evaluate the results.
   b. JCAT shall be able to seamlessly run one ART test suite and evaluate the results.

5. A technical report describing the research, evaluations, implementations and results. The report should also describe the problems and possible solutions found during research and implementation. This report together with the proof of concept can then be used by Ericsson to implement the solution into JCAT.

It is important to note that the goal of this thesis work is not to create a complete solution to be directly integrated in Ericsson’s existing environment but rather to create an implementation that can show a possible solution to the problem.

3.2 Implementation Requirements

The following requirements should be fulfilled by the proof of concept implementation.

1. Usability. The proof of concept shall be usable in a way that Ericsson can apply the fundamental concepts of the implementation into their environment.
2. Communication. JCAT shall be able to communicate with ART and ART shall be able to respond.
3. Data translation. JCAT shall be able to translate the ART data into usable Java objects.
4. Logging. Tests that JCAT run through ART should be logged in the same way as if they were executed directly by JCAT.

4 Problem Analysis

The thesis work was divided into several sub problems. The sub problems had three distinct areas; research, implementation and evaluation.

4.1 Research

A large portion of initial research was needed. First, how would JCAT and ART communicate? This question did not have one single answer since there were three steps that the communication had to go through and each of those steps could communicate in different ways. The communication steps were setup, execution and teardown. Possible solutions were to use JInterface, sockets, process execution, files or, since Ericsson runs their tests on Linux, to use a Linux terminal connection.

The next problem to solve was how to run ART from JCAT. This was a part of the setup communication step. There were several different options when solving this, such as using the ART execution script created by Ericsson or possibly run it manually either though a Linux terminal connection or by node to node communication using JInterface.

Another problem that was closely related was how to set up the environment so that the test cases can run properly. The environment setup could be done automatically by using runner scripts while executing through a Linux terminal connection or through JInterface the environment needs to be setup manually. The need for the environment setup came from the fact that ART test cases expect the software and hardware to be in a certain state for the tests to work as intended.

The next problem was how to transfer test data from ART to JCAT. This could happen either during the testing using events or after the tests using ART log files, sockets or Erlang node to node communication. A part of this problem was to make sure that all the gathered data from the ART test
cases were evaluated correctly in JCAT and also that the test data was satisfactory equivalent to the information that JCAT logs from native tests.

At the end of the ART tests the JCAT framework had to know that the ART execution was finished to be able to do any teardown work. This could be solved either through events or through signaling using Jinterface or sockets.

### 4.2 Implementation

It was hard to anticipate what implementations were needed before doing the research and evaluations. The reason for this was the multitude of selections available to solve the sub problems.

In general the proof of concept would require a Java proxy that was capable of communicating with Erlang. Furthermore some kind of ART test case wrappers in Java would be needed to simplify the evaluation of ART test cases in JCAT on a larger scale.

On the ART side some kind of communication implementation was needed to relay information from the test cases.

### 4.3 Evaluation

Evaluation of different technologies was required for every part of the research to ensure that the proof of concept implementation was both suitable for Ericsson and that it was the best solution that could be found within the timeframe of this thesis work.

The most important part to evaluate was how the communication would work through the three steps setup, execution and teardown. Also evaluating what the best approach was to start the ART test cases from JCAT was important since it would determine how easy it would be to do changes to proxied test cases in JCAT.

### 5 Work Method

The thesis work was planned for ten weeks. This gave a specific timeframe to achieve the set goals. Since the work included a lot of research and implementation of a solution to an unusual problem it was hard to estimate the time needed to get a good proof of concept ready. Because of this the waterfall method was not a good candidate as the work method. The work method that was selected was instead agile development inspired by scrum. There are two reasons that the choice was based on. The first is that agile development does not leave out any part of the development in the case that it is not completed by the end of the time period. This means that there would be a proof of concept, report and presentation ready by the end of the planned ten weeks even if some parts have not reached the expected state. The second reason is that Ericsson use agile development within their organization. Having this project using the same work method should make it easier to synchronize with Ericsson employees.

To make the project as efficient as possible the distribution of research, implementation, evaluation and report writing was in different proportions during the project. At the beginning, a larger part of the time was dedicated to research, since a good information base was needed to be able to do the other parts of the project. Later on as the research information base grew, more time was put into implementation and at the end of the project most of the time was put into evaluation and report writing.

Using the nonlinear division of time across the work areas made sure that all areas was progressing during the whole project and eliminated the risk of a lower quality of work because of dependency problems.
6 Solution

The solution to this thesis work is based on previous work, existing applications and innovative design. A lot of focus was put into using as much of existing technologies and implementations used by Ericsson as possible to make the solution easy to implement into their future production environment. Class, method and variable names in this report are written in italic to make references to the implemented code easier to follow.

6.1 Overview

This overview section describes how the solution works without details or reasons. This should make it easier to follow the detailed description from section 6.2 and forward.

In Figure 3 there are three distinct areas, the JCAT Preprocessing and Execution area, the ART Execution area and the JCAT Post-processing area. The JCAT Preprocessing and Execution area contains the classes and logic which handles ART specification generation, execution and event management. The ART Execution area is where the actual execution of the ART tests is done. The JCAT Post-processing area holds the classes and logic for the data distribution, test result evaluation and logging.

![Diagram](image)

**Figure 3.** The complete overview of the proof of concept.

In Figure 3 through 6 the circles and dotted lines are logic handled by TestNG. The rectangles are the classes created for the proof of concept. The diamonds and dotted lines in the ART Execution area represent the ART execution logic. In Figure 4 through 6 the text on the arrows describes which type of data that is passed in the direction of the arrow. The barrel represents the SUT. The hexagon represents all the JCAT ART suite classes.

The best place to start in order to understand the proof of concept is at the *Initializer* class which appears on the top left side in Figure 3 directly after TestNG(1). This class is the entry point for the ART specification file generation, execution and event management and is also where the
preprocessing phase begins. This class implements the TestNG`IMethodInterceptor` which contains the method `intercept`. The `intercept` method is only run if at least one test class that is an ART suite is selected.

The `Initializer` class uses the `intercept` method to first generate the specification file based on all the selected test cases and groups in JCAT. The specification file is generated by the `ARTSpecGenerator` class which is located above the `Initializer` class in Figure 3. The `ARTSpecGenerator` class will return the file path to the specification file generated.

The `ARTExecutioner` class, located below the `Initializer` class in Figure 3, handles the execution of ART. It gets information from the `Initializer` class about the test setup, such as ip-addresses to the nodes in the SUT and where the specification file resides. Using this information, ART is executed by starting a new process which runs a Bash script. This is where the preprocessing phase ends and the proof of concept enters the execution phase.

The Bash script that is used to execute ART is the same script that is used in the production environment at Ericsson to execute ART. This script is represented by the diamond called ART Executioner Script in Figure 3. The script has been changed from the original used by Ericsson to receive a parameter telling the script to use a new event handler as well as the name of the node that is created using JInterface on the JCAT side. The new event handler will receive events about when ART test cases start, where they are logged and when they have finished. The event information is sent by the new event handler to the JInterface mailbox using distributed Erlang node-to-node communication.

While the JInterface mailbox is receiving messages from the ART event handler the `ARTExecutioner` class gathers the information event by event and puts the information into a container structure where the bottom level container class represents one ART test case. This class will receive events that belong to the same test case and extract information from them to form the data from a complete test case. Each of these containers that hold the ART test cases are put into ART suite containers which each represents an ART suite.

When all the events have been received the ART suite containers will be saved in a context attribute that can later be accessed by other classes in the same JCAT test. Finally the `Initializer` will remove all the ART test cases from executing except for the test cases named `ARTTestCase`. After this the `Initializer` returns and the post-processing phase starts.

TestNG reaches the `ARTDataProvider` class, shown in the area called JCAT Post-processing in Figure 3. This class will get all the ART suite containers that were saved in the context attribute by the `Initializer` class. This data will then be distributed to each JCAT class that represents an ART suite. The class will receive the test data for the ART suite with the same name as the class.

The `@BeforeMethod` annotated method will run before the test method `ARTTestCase` is called to assert the test case data. This method will change the reported name of the `ARTTestCase` method, so that when it has executed and have been logged, it will look like the `ARTTestCase` method was named the same thing as the ART test case that was asserted.

Finally the data is asserted by the `ARTTestCase` test method. The results of the assertion will be logged with the name that was set by the `@BeforeMethod` annotated method. This cycle of distributing the data from the `ARTDataProvider` class and changing the name and finally asserting the results of the ART test case will be done repeatedly until all the executed ART test cases has been logged in JCAT.

### 6.2 Research Results and Design Decisions

To start the execution from JCAT the selected test classes representing ART test suites had to be identified. The best solution found was to use a TestNG method that received an `ITestContext`
parameter containing lists of included and excluded test cases. The selected solution was to create a class that implements the TestNG interface `IMethodInterceptor` with the method `intercept`. The reason is that it has the `ITestContext` and also has a list of all the methods that will run in the current test, as well as the possibility to change the test methods that will run by returning a new list with a different set of test cases [TestNG JavaDoc]. A different option would have been to use the `IMethodSelector` interface which can also change the list of methods that will run. One difference is that the `IMethodSelector` has no direct way of adding excluded test methods to the list to be run, there is only the option of excluding already included methods. The `IMethodSelector` must also be included in the JCAT XML file and would run even if there are no ART suite test classes selected. The `IMethodInterceptor` on the other hand is selected from each ART suite test class. This means that the `intercept` method of the `IMethodInterceptor` class will only run if there are ART suite test classes selected in the JCAT XML file.

Using the `ITestContext` parameter solves the problem with knowing which test cases have been selected, but to find selected classes and groups requires a more elaborate solution. After extensive research into TestNG the result was that there are no TestNG objects that show which groups and complete classes that have actually been selected. A decision was made to create test cases representing ART groups and that each ART suite class includes a test case called `all` which, if selected, represents that the whole ART suite should be run.

Sockets or Erlang message passing via JInterface were regarded for the communication. Using the built in support for message passing in Erlang results in a simple, straightforward solution that is easy to understand and maintain. Socket communication on the other hand would have needed explicit setup of sockets and serialization of the transmitted data. Implementing such a solution would have required more time, a risk of introducing errors and would not have contributed to anything more than what JInterface already does.

During the research regarding execution of the ART framework the main idea was to set up an Erlang node functioning as a server. This Erlang server would take requests via remote procedure calls from JCAT to execute ART tests. The decision was however made to execute ART via an existing Bash script, used by Ericsson to execute ART tests from their automation tool.

These methods of execution and data gathering have some great benefits. On the Java side the usage of JInterface, an existing library that comes with the official Erlang distribution, results in less code that needs to be maintained for Ericsson. On the ART side, using the executioner script guarantees that the execution of ART using JCAT will always be done in a correct way. Because of the executioner script is being updated by the ART developers, the JCAT developers at Ericsson will not have to spend time updating their own way of executing ART.

To log the ART results in JCAT the obvious solution was to let every test case in ART correspond to a test method in JCAT. During the research it became apparent that this posed some problems both with dependencies and with test groups. To handle dependencies between ART test cases, these would have to be added into the JCAT test classes, but this would not be enough. TestNG, and therefore JCAT, does not handle automatic dependency solving when single test cases are included in the XML file. Based on these research findings the decision was to discard all ART corresponding test methods in JCAT during the initialization phase. The ART results are instead logged by repeatedly executing a single test method, `ARTTestCase`, letting it mimic the ART test cases’ and configuration functions’ (e.g. `init_per_suite`) names.

### 6.3 JCAT, ART and Communication

The Erlang Common Test framework, and thereby ART, has the option to include custom event handlers. This is already used by Ericsson to store the test results into their database called RRS.
In ART there is a Bash script called *execute* which is used by Ericsson in their production environment to execute ART. After some setup, the execute script calls the Bash script *execute_inside* which handles the rest of the execution. The proof of concept uses the *execute_inside* script directly because of differences between the IRIL environment, where most of the thesis work was done, and the Ericsson production environment. By researching how the *execute_inside* script handles the existing RRS event handler, changes were made to include support for the new event handler in a similar way. The events generated by Common Test are listed in Table 2. These events include information on what test that have been executed and their results but they don’t have any information about the actual commands used during execution, debug and error information or the terminal output from the SUT. This information is instead accessible in the original ART log files that the events provide the paths to.

<table>
<thead>
<tr>
<th>ART Event Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_logging</td>
<td>Informs that logging has started. Includes the path where all log files will be generated.</td>
</tr>
<tr>
<td>test_start</td>
<td>Informs that the test is starting. Includes the date, time and log path.</td>
</tr>
<tr>
<td>start_info</td>
<td>Informs how many tests, suites and test cases that will be run.</td>
</tr>
<tr>
<td>tc_logfile</td>
<td>Tells the current suite and case and the path to the log file.</td>
</tr>
<tr>
<td>tc_start</td>
<td>Tells that the test case has started. Contains the suite and the case that is started.</td>
</tr>
<tr>
<td>tc_done</td>
<td>Tells that the test case has completed. Contains suite, test case and result. Result is either the atom ‘ok’ if the test case passed or result is a tuple that starts with error and then contains error information.</td>
</tr>
<tr>
<td>tc_user_skip</td>
<td>Test case skipped by user. Includes suite, test case and reason.</td>
</tr>
<tr>
<td>tc_auto_skip</td>
<td>Test case skipped automatically. Usually caused by a problem in init_per_suite, init_per_group or init_per_test.</td>
</tr>
<tr>
<td>test_stats</td>
<td>Tells the number of test cases passed, failed and a tuple with counts of user skipped and auto skipped test cases.</td>
</tr>
<tr>
<td>test_done</td>
<td>Indicates that the test has completed. Includes date and time.</td>
</tr>
<tr>
<td>stop_logging</td>
<td>Indicates that the logging has stopped.</td>
</tr>
</tbody>
</table>

*Table 2. The different ART events listed by their names.*

To support the JCAT framework the *execute_inside* script was adapted to receive parameters to activate the JCAT event handler and to receive the name of the JCAT node that are supposed to receive the forwarded events. Setting the node name during execution enables the possibility to run several tests simultaneously by different users, which is a necessity at Ericsson.

The Erlang implementation of the event handler sets up the connection to the JCAT node during its initialization. All received events on the ART side are then forwarded to the JCAT node, using the distributed Erlang message passing functions described in the Erlang section in the background. By using Erlang message passing, asynchronous communication is used. The test execution or the event handler does not have to be halted to wait for the Java side to receive the messages. Figure 4 shows
how the communications flow looks around the ART side of the proof of concept. In the figure, *JcartEvh* is the event handler that forwards the events to the JCAT JInterface node.

**Figure 4.** The ART side of the proof of concept. The modified ART Executioner script starts the ART tests and the Erlang nodes with the JCART event handler passes the events with the test results on to JCAT.

### 6.4 Preprocessing and Execution

#### 6.4.1 Initialization

The *Initializer* class is the class implementing the *IMethodInterceptor* interface and its *intercept* method. This class is used for the generation of the ART specification file by using the *ARTSpecGenerator* class. It initiates the execution of ART and receives the ART test results from the *ARTExecutioner* class. The *Initializer* class finally changes the list methods that will be run before returning the control to the JCAT framework and the post-processing phase. Figure 5 shows an overview of how the *Initializer* class interacts with the other classes.
Figure 5. Interaction between the Initializer class and other classes. This is where the ART specification file is generated with the help of the ARTSpecGenerator class, and where ART is executed from. The control of execution and the list of test methods to be run by JCAT are then returned by the Initializer class to the JCAT framework.

6.4.2 The ART Specification File

Every JCAT test class that represents an ART suite has an all method to indicate if the entire suite has been selected in the XML file. If specific test cases are chosen from a class, the all method would be automatically excluded. The entries in the ART specification file for a specific ART suite class depends on if the all method is among the included test cases. If it is, the whole suite will be added and any exclusions with it. If the all method is not present among the methods that will run for a specific class then all the selected groups and test cases will be added to the test specification instead. Code 1 shows how the entries for a class called example_SUITE would look if the all method was included. It shows that the entire suite is included and then any excludes that were chosen in the JCAT XML file is also added to the ART specification file. Code 2 shows how the lines for the same class would look if the all method was not included. This listing shows that only the specific test cases and groups that were selected in the JCAT XML file are also included in the ART specification file.

Code 1. ART specification file when the ‘all’ method was included.

```java
{suites, "/bin/src/ART/suites/", example_SUITE}.
{skip_cases, "/bin/src/ART/suites/", example_SUITE, [tc_example_case]}.  
{skip_groups, "/bin/src/ART/suites/", example_SUITE, [grp_example_group]}.  
```

Code 2. ART specification file when the ‘all’ method was not included.

```java
{cases, "/bin/src/ART/suites/", example_SUITE, [tc_example_case]}. 
{groups, "/bin/src/ART/suites/", example_SUITE, [grp_example_group]}. 
```

The solution to represent groups and the suite selection as test cases is not optimal since it does not follow the standards of TestNG and thereby JCAT, but it does work. To make sure that users know if a test case in JCAT represents an ART test case, group or suite, the names have been given specific
identifiers at the start. A test case starts with `tc`, a group starts with `grp` and the entire suite is represented by the `all` test case method.

The `ARTSpecBuilder` class was created to take advantage of the groups and suite representation solution to generate the ART specification file. Figure 5 on the previous page shows how it interacts with the `Initializer` class. The method `buildSpec` in `ARTSpecBuilder` takes two parameters, a list of all the methods that will run and an instance of the `ITestContext` interface. Using these two parameters the ART specification file is generated. The `buildSpec` method is called by the `Initializer` class and returns the path to the specification file.

The second step in the `Initializer` class' `intercept` method is to use the returned specification file path, together with node information about which RNC and TGS nodes that the ART test cases will test, to call the method `runART` in the class `ARTExecutioner`.

### 6.4.3 The ARTExecutioner Class

The `ARTExecutioner` class is designed for two purposes. The first purpose is to execute ART, the second is to receive all the events from the event handler and produce a logical structure of ART test cases in suites. Figure 5 on the previous page shows how the `ARTExecutioner` interacts with the `Initializer` class.

Code 3 shows how the execution is done from the `ARTExecutioner` class. Its constructor receives the path to the generated specification file, an RNC and a TGS node. These parameters are used to build the execution command. The name of the JInterface node is sent during the execution so that the `JcartEvh` event handler knows where to report back to. The path to the generated ART specification file is supplied at the end of the command string. The server name in this example is used statically during the prototyping but should later be changed to an option.

```java
// Create the command.
String[] cmd = new String[] {"/bin/bash", "-c",
   execRoot + "/execute_inside " + // The ART executioner script.
   " -rnc " + rnc.getNode().getNodeIp() + " "
   + rnc.getNode().getResource().getNodeUser() + " "
   + rnc.getNode().getResource().getNodePassword() +
   " -tgs " + tgs.getNode().getNodeIp() + " "
   + tgs.getNode().getResource().getNodeUser() + " "
   + tgs.getNode().getResource().getNodePassword() +
   " -ci jcart " + node.node() + // Use the Jcart event handler.
   " -server RTU19023B " + // The server that the slave should run on.
   " -spec " + specFile}; // The ART .spec file.
if (!verbose) cmd[2] += " >\&1"; // Direct printouts to /dev/null

// Execute the command.
Runtime runtime = Runtime.getRuntime();
Process proc = runtime.exec(cmd);
```

**Code 3.** Execution of ART using a Linux Bash shell together with an existing Ericsson shell script.

### 6.4.4 Event Handling

In JCAT the event handling is done in several steps. The first step is to receive the data from the mailbox. There are two modes for the event management. One is with verbose enabled where the entire Bash output as well as the events is printed. The other is with verbose disabled where no information is printed during event management.
While using verbose mode, reading from the mailbox is using a timeout because the process that was started to run the ART executioner script prints to an output that could become full. If that output is full the execution would be stalled until it no longer is full. The timeout is used to avoid this. When a timeout occurs or after a mailbox message have been processed the program flushes all the characters from the process output by reading them and printing them to the console output. If verbose mode is disabled, the output is piped to /dev/null automatically by Bash. Code 3 shows that the verbose mode sets whether the /dev/null pipe should be included or not.

When a message is received in the JInterface mailbox it is checked if it is an event. If it is, it is sent to another method for further handling. That method will then extract the information and put it into the ARTTestCaseData class which in turn is stored in the class ARTSuiteTestCaseCollection that stores all test results for a suite. This way of dividing the events gives a simple interface when the data is to be distributed to a specific JCAT ART suite class.

When all the ART tests have been run and all the events have been divided into data containers the data is put into an attribute in the ITestContext. This attribute can then be accessed from any method that has access to ITestContext in the current test.

### 6.4.5 Changing the List of Test Cases

The last step that the intercept method in the Initializer class does is to create a new list of test cases that will be run by JCAT. The list is changed so that all the test methods that belong to a class representing an ART suite is removed except for the test methods named ARTTestCase. ARTTestCase is included in every JCAT ART suite class since all of them extend the class ARTSuite which contains the method.

The reason that all the test methods are removed is that, to identify which groups that are to be included in the ART specification file, the ART groups cannot be represented by the TestNG standard annotation but are instead represented by test methods which are named grp in the beginning. Since the groups are represented by test methods instead of group annotations the annotation @BeforeGroup will not be triggered by TestNG. The only way to assert all the test cases that have been run by ART is to use the same method over and over again that just changes the name.

To identify that the class is an ART suite a new annotation was created. The annotation is called ARTTest. By using this annotation the ART suite classes are easily distinguishable from other test classes.

If the user chooses to run only a subset of the available test cases, the rest will automatically be excluded by JCAT. This includes the ARTTestCase method needed to log the test results. The solution was to find this specific test method in the list of excluded methods and add it into the list of test cases that should be run by JCAT. By doing this manually it is ensured that the evaluation of the ART test case data will always be done.

### 6.5 Post-processing

When all the ART test methods are removed the only test methods that remain is the ARTTestCase methods that resides in the ARTSuite class which all JCAT ART suite classes extends. This method uses a data provider that supplies the method with an array of data where each element will be used for one test in that method. Figure 6 shows the ARTDataProvider class’ interaction with the other classes and TestNG.
The data provider gets all the ART test data from the context attribute and for each ART suite class that requests data to the ARTTestCase method, the data provider will find the correct data for that ART suite class and return it. The @BeforeMethod annotated method in each JCAT ART suite class will run before every ARTTestCase run. This method will change the name of the upcoming test method by using the Java reflection library to change a private variable called m_methodName. The result is that every run of the ARTTestCase method will be logged with the name of the test case that was run in ART.

The ARTTestCase method will finally evaluate and log the data supplied. Code 4 shows the simplicity of evaluating and logging the data. The testCase variable is an instance of ARTTestCaseData. This variable holds the data for one ART test case which belongs to the ART suite that the current JCAT ART suite class represents. The ARTTestCase method puts a link to the complete ART test case log file in the JCAT log in case more debug information is needed. The method then proceeds to assert the ART test case result. If the ART test case was skipped, a skip exception is thrown to mark the JCAT test method as skipped as well. If it was not skipped the result string will be checked if it is ‘ok’. If not, the test case failed in ART and therefore will also fail in JCAT which means that the reason for failure will be included automatically by JCAT.
public void ARTTestCase(Field methodName, BaseTestMethod testMethod, ARTTestCaseData testCase) {
    setTestCase("ART Test Case", testCase.toString());

    // Add some log info.
    if (!testCase.getLogfile().isEmpty()) {
        setTestInfo(" <a href=" + testCase.getLogfile() + ">">External ART "
            + "log file</a>");
    }
    if (!testCase.getGroupName().isEmpty()) {
        setTestInfo("ART Group: " + testCase.getGroupName());
        setTestInfo("ART Test Case: " + testCase.getFunctionName());
    }

    // Evaluate the results.
    if (testCase.getResult().contains("skip"))
        throw new SkipException(testCase.getReason());
    else {
        if (!testCase.isPassed()) {
            String failedMsg = "Testcase " + testCase.getFunctionName() + "
                has failed (" + testCase.getReason() + ")";
            setAdditionalResultInfo(ellipsizeString(failedMsg, 50));
            setTestInfo(failedMsg);
        }
    }
    assertTrue(testCase.getReason(), testCase.isPassed());
}

7 Results

The result is a prototype which both fulfills all tasks and most of the requirements. The prototype is easy to use, especially for someone who has previous experience of using JCAT.

The prototype is designed to be intuitive to use, all the user have to do is to select which ART suite classes to run in the standard JCAT XML file. It is possible to select specific groups or test cases but the user will have to remember that the groups are not selected as groups in JCAT normally are. The groups are instead selected as if they were test cases.

To use the prototype the user will have to set up ART so that it is working without JCAT. The same goes for JCAT, it must be configured and working. The resulting ART test case logs looks like the test has been run directly from ART as expected and the JCAT logs look like ordinary JCAT log files. Figure 7 shows how the log looks when both JCAT and ART test cases have been run.
7.1 Result Analysis

To test that the proof of concept works as it was expected a set of tests were defined. The defined tasks state that the proof of concept shall be able to run one ART test case and one ART suite.

The tests were defined as the following.

- Run one test case from an ART suite in JCAT.
- Run one complete ART suite from JCAT.
- Run one test case from one ART suite and a complete ART suite at the same time from JCAT.
- Run one ART suite where one test case and one group have been excluded from JCAT.
- Run one ART test case and one group that belong to the same suite from JCAT.
- Run three ART suites from JCAT. One that runs the entire suite, one that excludes one test case and one that only runs one group.

All of the defined tests have been run and all of them passed successfully.

It was also deemed interesting to see the difference in execution times between running ART natively and running ART from JCAT and logging the data. The execution time results were that one complete suite in ART (**_SUITE) took 1 minute and 27 seconds. When running the same suite from JCAT and logging the results in JCAT it took 1 minute and 41 seconds. This small time difference comes from starting the JCAT framework and shows that using JCAT to execute ART is viable from a timing perspective.

In terms of how the execution is done from JCAT there are some less intuitive aspects. One is representing test groups as if they were test cases when including or excluding them in the XML file. While this is not standard behavior in JCAT it is still simple to use the solution to run ART tests. One other drawback with the current proof of concept is that it is representing the `init_per_testcase`, `init_per_suite`, `end_per_testcase` and similar configuration functions in ART as test cases in the JCAT log files. This gives the impression that more test cases have been executed than what is actually the case.
**Figure 7.** The final JCAT log with ART and JCAT test results. All test cases in the table are ART test cases except for the first two which are native JCAT test methods.
8 Future Work

There are several improvements and changes that need to be made in order for the proof of concept in this thesis work to be production ready. Some of these changes need more research while for others there was not enough time to implement the proper solution.

8.1 Class Generation

A functionality that would greatly benefit the production implementation of this prototype is to have a program that would automatically generate and compile all the ART suite class files for JCAT. This program could use the ART suite files and the function all to find which test cases and groups that should be generated. It would also be possible to find the test cases and groups through the build in function module_info which can list all the exported functions in a module. This script or program should preferably be run before JCAT is run to make sure that the ART suite class files are up to date.

8.1.1 Comments on Generated Classes

Another feature that could be included with the class generation is having the generator also include comments about what the test groups, functions and the entire suit does. This would make it a lot simpler for the users of JCAT since they would not have to view the ART suite files to see what the tests do.

A work around to extracting the comments would be to include a Javadoc link for each test case and group to their original ART documentation.

8.2 ART Groups in JCAT

The prototype representation of ART groups in JCAT is using test cases that start with grp. This behavior does not conform with the JCAT and TestNG standards where groups are represented using the groups list in the test annotation [TestNG Doc]. During the research no other satisfactory solution was found. A solution to this would require more research.

8.3 Use of Before and After Methods in JCAT

The init and end functions in ART have equivalents in the JCAT @Before- and @After methods. An example of these configuration methods is that the ART init_per_testcase function corresponds to the JCAT @BeforeMethod method. These are in the current implementation logged in JCAT as ordinary test cases, only the names shows that they are in fact configuration functions. If the log files are to be analyzed by a program it could be of some importance that these functions are represented correctly.

8.4 Error Handling

Both during and after execution extensive error handling is needed to make the solution stable. The prototype has very little error handling and would not detect if the ART execution fails or if the process would suddenly crash.

Some error handling is also needed after the execution is done. Code 5 shows an example where the name of a Java method is changed to match the ART test cases. In this example there is need for error handling in the cases that the parameters are not initialized.
@BeforeMethod
public void BeforeARTTestCase(Object[] params) {
    Field methodName = (Field) params[0];
    BaseTestMethod testMethod = (BaseTestMethod) params[1];
    ARTTestCaseData testCase = (ARTTestCaseData) params[2];
    try {
        methodName.set(testMethod, testCase.getFunctionName());
    } catch (Exception e) {
        e.printStackTrace();
        logger.error(e.toString());
    }
}

Code 5. Code example for method name changing where proper error handling is missing.

8.5 Verbose Mode
During execution of ART the current solution prints out all the bash output to the Java console. While this is desirable during testing it might not be during usage in the production environment.

The suggested solution is to change the implementation so that it is optional whether the output should be printed to the Java console or if it should be redirected to /dev/null. This option could be controlled from the JCAT XML file by using the property tag or by checking the TestNG verbosity level, also available in the XML file.

8.6 Users Guide
A user guide needs to be written in order to fully understand the differences between executing ART test cases and executing native JCAT test cases. The guide would need to include the differences between execution, options that change the behavior of ART and some requirements for the execution to work.

8.7 Integration into Production Environment
To actually be able to use the JCAT ART integration proposed in this report, the suggested solution would have to be included in the files and frameworks in the version management system on Ericsson’s servers.

8.8 Support for Parallel Execution in ART
Ericsson sometimes executes ART test cases in parallel to speed up the testing. The proof of concept does not support parallel execution of test cases in ART. To implement support for this feature the event management would have to be changed in the ARTExecutioner class so that it can store events that are received out of order.

9 Summary
In regard to the tasks the first task was to research the communication between ART and JCAT. The prototype has completed this task with great success since the communication is very stable and works without any interaction or setup from the user.

The second task was to research the execution of ART from JCAT. This task is closely related to the first task and just as the first task this task is also solved with excellence. The execution does not
require any already running processes to initiate the ART test suites. The prototype uses the same execution methods as the production environment at Ericsson is using. This makes sure that the execution scripts used is always up to date and any fixes are applied automatically to the solution as well.

The third task was to make sure that the solution fits Ericsson’s needs. This has been the deciding factor in every decision that has been made and the final prototype does very few alterations to the existing Ericsson environment while still fulfilling its task.

The fourth task was to create a prototype which would seamlessly run an ART test case and an ART suite from JCAT. This task has been completed. To choose what to run a user simply have to select the test cases, groups or entire classes (suites) in the JCAT XML file just as a standard JCAT test would be with the one difference that groups are chosen like test cases are.

The final task was to write a report that describes the details of the work, the solution and known problems. This report is the completion of that task.

Almost all of the requirements that were defined have been fulfilled. The first requirement was usability. That requirement has been partially fulfilled. The prototype works great in the lab environment using the same tools as the production environment does. But it has not been possible to test the proof of concept in the production environment due to time and access restrictions.

The second requirement was regarding communication. This requirement is also only partially fulfilled since it does work in the lab environment but, just as the previous requirement, have not been tested in the production environment due to time and access restrictions.

The third requirement was data translation. JCAT is able to fully translate all ART event data into usable Java objects that can easily be distributed to the methods that needs it.

The fourth and final requirement was logging. This requirement states that “Tests that JCAT run through ART should be logged in the same way as if they were executed directly by JCAT”. This requirement has been fully completed. JCAT logs all the ART test case data and also links to the debugging and error details to make it easier to find the source of problems.

All in all the results are satisfying. The solution should be useful to Ericsson and relatively simple to put into production.

10 Conclusion

We are proud of the results of this thesis work. We have created an implementation that work well and shows clearly that it is possible to run ART tests from JCAT.

This thesis work has been very educational for both of us. We had some previous knowledge of ART, JCAT, TestNG and Common Test before the thesis work started which helped us get the result described. Now, after this thesis work, we feel that we have a deeper insight and better understanding of these test frameworks.

We feel that we made a lot of good decisions during the thesis work which has made the proof of concept turn out great. One issue that we had, which we did not prepare for, was the design decisions. Those were decisions where there were no correct answers but rather just choices about how information would be displayed or how something should be selected. One good example is where we had to choose between displaying test cases ART style, where the init and end test cases are shown, or display them in JCAT style, where they are not. In a proof of concept those choices do not matter much but in a production implementation they may be essential to produce a high quality product. In the future we will try to prepare better for decisions like this.
It has been great to do the thesis work with Ericsson with all the support and attention we have gotten. We would highly recommend future students to do their thesis work at Ericsson as well.
11 References

11.1 Printed References

ISBN: 978-0-52188-038-1


ISBN: 978-0-32150-310-7


ISBN: 978-9-17485-037-6


ISBN: 978-1-93398-878-8

ISBN: 978-0-13136-589-6


11.2 Internet References


http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4578383

[Java.com] Learn About Java Technology.  
http://www.java.com/en/about/


http://www.oracle.com/technetwork/java/simple-136065.html

http://www.oracle.com/technetwork/java/simple-142339.html

http://www.oracle.com/technetwork/java/intro-141325.html

[RNC3810] RNC3810—Ericsson’s first WCDMA radio network controller.  

http://testng.org/javadoc/org/testng/IMethodInterceptor.html

[TestNG Doc] TestNG Documentation: 5.2 - Test Groups.  
http://testng.org/doc/documentation-main.html#test-groups

[Top100] Software Top 100.  

[xml.com] O’Reilly: XML From the Inside Out. What is XML?  
http://www.xml.com/pub/a/98/10/guide0.html?page=2