Representation of asynchronous communication protocols in Scala and Akka

by

Joakim Eriksson

LIU-IDA/LITH-EX-A--13/038--SE

2013-06-28
Final Thesis

Representation of asynchronous communication protocols in Scala and Akka

by

Joakim Eriksson

LIU-IDA/LITH-EX-A--13/038--SE

2013-06-28

Supervisors: Mattias Evenson (Xdin AB, Linköping)
Lena Buffoni (IDA, LiU)

Examiner: Kristian Sandahl (IDA, LiU)
Abstract

This thesis work investigates how to represent protocols for asynchronous communication in the Scala programming language and the Akka actor framework, to be run on Java Virtual Machine (JVM). Further restrictions from the problem domain - the coexistence of multiple protocol instances sharing the same Java thread - imply that neither an asynchronous call waiting for response nor anything else can block the underlying Java threads.

A common way to represent asynchronous communication protocols is to use state machines. This thesis seeks a way to shrink the size of and to reduce the complexity of the protocol implementations by representing sequences of asynchronous communication calls (i.e. sequences of sent and received messages) as a type of procedure. The idea is find a way to make the procedures that contain asynchronous calls look like synchronous communication procedures by hiding the asynchronous details. In other words, the resulting procedure code should show what to do and not so much focus on how to overcome the impediment of the asynchronous calls.

With the help of an asynchronous communication protocol toy example, this report shows how such an protocol can be implemented with a combination of a state machine and a procedure representation in Scala and Akka. The procedure representation hides away the asynchronous details by using the Scala capability to use CPS-transformed delimited continuations. As a sub-problem, this thesis also shows how to safely schedule asynchronous communication timeouts with help of Scala and Akka within the restrictions of the thesis problem domain.
Acknowledgements

This thesis have been written with help from others. First, I want to thank my examiner Kristian Sandahl and institution supervisor Lena Buffoni. Their experience, insight and feedback has had a great impact on the direction of the work and the improvement of this report.

I want to thank my company supervisor Mattias Evensson, who has been a great source of information. He has had to put up with many difficult questions from me.

My opponent Hampus Malmström did not have an easy task giving constructive feedback on such a sizable report and narrow subject. I really appreciate the effort made.

Finally, a special thanks to my dear Linn who has spent a lot of time supporting the work and motivating me. Without her this thesis would not have been finished.

Vikingstad, June 2013

Joakim Eriksson
## Contents

1 Thesis introduction 1  
1.1 Background .............................................. 1  
1.2 Purpose ................................................. 2  
  1.2.1 A shift from state machine representation .......... 2  
  1.2.2 A representation suited for high concurrency ..... 2  
1.3 Scope and limitations ................................. 3  
1.4 Goals .................................................. 3  
1.5 Methodology and thesis overview ........................ 3  
1.6 Audience ............................................... 4  
1.7 Terminology ............................................ 5  
1.8 Typographical conventions .............................. 5

2 Technology introductions 7  
2.1 Introduction to Scala .................................. 7  
  2.1.1 Technical details of Scala ......................... 7  
  2.1.2 Comparison between Scala and Java ............... 8  
  2.1.3 Reasons for the focus on Scala .................... 10  
  2.1.4 Scala literature .................................. 11  
2.2 Introduction to Actors ................................ 12  
  2.2.1 The problem that actors solve .................... 12  
  2.2.2 Using actors ...................................... 13  
  2.2.3 The Scala actor library ............................ 14  
  2.2.4 The Akka framework ............................... 14  
2.3 Introductions to state machines ....................... 16  
  2.3.1 What are state machines used for? ............... 16

3 Investigation of current solution 18  
3.1 Common traits of the protocols ....................... 18  
3.2 Graphic visualization of program logic as a state machine .. 19  
3.3 Problems with the current solution ................... 19  
3.4 Technical solutions of interest and limitations ........ 19  
3.5 Focus on the developer ................................ 20
4 Investigation of possible techniques
  4.1 Generic SM implementations .................................. 21
    4.1.1 Example state machine .................................. 21
    4.1.2 Nested match/case implementation ......................... 23
    4.1.3 Transition table ........................................ 24
    4.1.4 The state pattern ....................................... 25
    4.1.5 The state machine compiler .............................. 29
  4.2 Akka specific SM implementations ............................ 30
    4.2.1 Actor methods become and unbecome ....................... 30
    4.2.2 Akka FSM implementation ................................ 32
    4.2.3 Composition of state machine and actor .................. 33
  4.3 Scala specific SM implementations ............................ 34
    4.3.1 Domain specific languages ............................... 34
    4.3.2 Internal DSL ........................................... 35
    4.3.3 External DSL ............................................ 38
  4.4 How to delay, take time and schedule ......................... 40
    4.4.1 Delay test bench evaluation ............................. 40
    4.4.2 Immediate response with no delay (ND) .................... 41
    4.4.3 Thread sleep (TS) ...................................... 41
    4.4.4 Java utility timer (GJT and LJT) ........................ 42
    4.4.5 Akka scheduler (AS) .................................... 43
    4.4.6 Akka FSM state timeout (FSMSTO) ......................... 43
    4.4.7 Akka FSM timers (FSMT) ................................ 44
    4.4.8 Akka receive timeout (RTO) ............................. 45
    4.4.9 Future with ask timeout (FTO) ........................... 45
    4.4.10 Future with Await.result (FAR) ......................... 46
    4.4.11 Comparison ............................................ 46
  4.5 Asynchronous Procedures ..................................... 50
    4.5.1 Continuations .......................................... 51
    4.5.2 Dataflow and Futures ................................... 54
    4.5.3 Thunk procedure ....................................... 54
  4.6 Investigation summary ....................................... 60

5 Specification of toy example .................................. 62
  5.1 Overview .................................................. 62
  5.2 Relation to thesis context ................................... 63
  5.3 NET states ................................................ 64
  5.4 Interaction with UE NET ..................................... 64
    5.4.1 Interaction between APPL and NET ....................... 65
    5.4.2 Interaction between RADIO and NET ...................... 65
    5.4.3 Interaction between UE and CORE side in NET layer ... 67
  5.5 Procedures in UE NET ....................................... 70
    5.5.1 A procedure notation ................................... 70
    5.5.2 Primitive procedures ................................. 71
    5.5.3 Composite procedures ............................... 72
List of Tables

4.1 Transition table for ballpoint pen state machine . . . . . . . . 22
4.2 Delay for actors using the techniques ND, GJT and AS . . . . 48
4.3 Delay for actors using Java thread sleep (TS) . . . . . . . . . 48
4.4 Delay for actors using the techniques LJТ and FAR . . . . . . 49
4.5 Delay for quantity of 1000 concurrent actors . . . . . . . . . . 49
4.6 Delay for quantity of 20000 concurrent actors . . . . . . . . . . 49
4.7 SM representation technique summary . . . . . . . . . . . . . 60
4.8 Delay technique summary . . . . . . . . . . . . . . . . . . . . . . 61
4.9 AP representation technique summary . . . . . . . . . . . . . . 61
List of Figures

4.1 State transition diagram for a ballpoint pen . . . . . . . . . . 22
4.2 The state pattern . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.3 Class diagram for ballpoint pen SM as state pattern . . . . . 28

5.1 NET overview . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
5.2 NAS overview . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
5.3 Toy example state machine . . . . . . . . . . . . . . . . . . . . 64
5.4 The powerOn interaction . . . . . . . . . . . . . . . . . . . . . 65
5.5 The powerOff interaction . . . . . . . . . . . . . . . . . . . . . 65
5.6 The is detached interaction . . . . . . . . . . . . . . . . . . . . 65
5.7 The connect interaction . . . . . . . . . . . . . . . . . . . . . . 66
5.8 The disconnect interaction . . . . . . . . . . . . . . . . . . . . 66
5.9 The send interaction . . . . . . . . . . . . . . . . . . . . . . . . 66
5.10 The receive interaction . . . . . . . . . . . . . . . . . . . . . . 67
5.11 The forcedDetach interaction . . . . . . . . . . . . . . . . . . . 67
5.12 The attach interaction . . . . . . . . . . . . . . . . . . . . . . . 68
5.13 The detach interaction . . . . . . . . . . . . . . . . . . . . . . . 68
5.14 The authentication interaction . . . . . . . . . . . . . . . . . . . 69
5.15 Large message interaction example . . . . . . . . . . . . . . . . 74
<table>
<thead>
<tr>
<th>Code Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A code listing sample</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>A Java example program</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>A long Scala example program</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>A short Scala example program</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>Classes common to all Ballpoint pen SM implementations</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Ballpoint pen SM as nested match/case</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Ballpoint pen SM as state-event tuple match/case</td>
<td>25</td>
</tr>
<tr>
<td>4.4</td>
<td>Ballpoint pen SM as transition table</td>
<td>26</td>
</tr>
<tr>
<td>4.5</td>
<td>Ballpoint pen SM as state design pattern</td>
<td>27</td>
</tr>
<tr>
<td>4.6</td>
<td>Ballpoint pen SM in some definition format</td>
<td>29</td>
</tr>
<tr>
<td>4.7</td>
<td>Ballpoint pen SM as actor with become method</td>
<td>31</td>
</tr>
<tr>
<td>4.8</td>
<td>Adapter between a ballpoint pen SM and an actor</td>
<td>31</td>
</tr>
<tr>
<td>4.9</td>
<td>Ballpoint pen SM as actor with FSM trait</td>
<td>33</td>
</tr>
<tr>
<td>4.10</td>
<td>Ballpoint pen SM as actor-SM composition</td>
<td>34</td>
</tr>
<tr>
<td>4.11</td>
<td>Internal DSL for SM example</td>
<td>36</td>
</tr>
<tr>
<td>4.12</td>
<td>Ballpoint pen SM with internal DSL</td>
<td>37</td>
</tr>
<tr>
<td>4.13</td>
<td>Pin code lock definitions</td>
<td>37</td>
</tr>
<tr>
<td>4.14</td>
<td>Pin code lock SM with internal DSL</td>
<td>39</td>
</tr>
<tr>
<td>4.15</td>
<td>Delay test bench message token</td>
<td>41</td>
</tr>
<tr>
<td>4.16</td>
<td>Delay test bench class AbstractDelayActor</td>
<td>41</td>
</tr>
<tr>
<td>4.17</td>
<td>Delay test bench class AbstractAkkaFsmActor</td>
<td>42</td>
</tr>
<tr>
<td>4.18</td>
<td>Delay actor with no delay (ND)</td>
<td>42</td>
</tr>
<tr>
<td>4.19</td>
<td>Delay actors using Thread.sleep</td>
<td>43</td>
</tr>
<tr>
<td>4.20</td>
<td>Delay actors using java.util.Timer</td>
<td>43</td>
</tr>
<tr>
<td>4.21</td>
<td>Delay actors using Akka scheduler</td>
<td>44</td>
</tr>
<tr>
<td>4.22</td>
<td>Delay actors using Akka FMS state timeout</td>
<td>44</td>
</tr>
<tr>
<td>4.23</td>
<td>Delay actors using Akka FMS timers</td>
<td>44</td>
</tr>
<tr>
<td>4.24</td>
<td>Delay actors using Akka receive timeout</td>
<td>45</td>
</tr>
<tr>
<td>4.25</td>
<td>Delay actors using Future with ask timeout</td>
<td>46</td>
</tr>
<tr>
<td>4.26</td>
<td>Delay actors using Future with Future with Await.result</td>
<td>47</td>
</tr>
<tr>
<td>4.27</td>
<td>Delimited Continuation example</td>
<td>53</td>
</tr>
<tr>
<td>4.28</td>
<td>Simple AP example - thunk procedure implementation</td>
<td>55</td>
</tr>
<tr>
<td>4.29</td>
<td>Simple AP example - usage in test program</td>
<td>56</td>
</tr>
<tr>
<td>4.30</td>
<td>Simple AP example - output from test program</td>
<td>56</td>
</tr>
</tbody>
</table>
## Nomenclature

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>asynchronous procedure</td>
</tr>
<tr>
<td>AS</td>
<td>Akka scheduler - a delay technique</td>
</tr>
<tr>
<td>CPS</td>
<td>continuation passing style</td>
</tr>
<tr>
<td>DSL</td>
<td>domain specific language</td>
</tr>
<tr>
<td>FAR</td>
<td>future with <code>Await.result</code> - a delay technique</td>
</tr>
<tr>
<td>FSM</td>
<td>finite state machine</td>
</tr>
<tr>
<td>FSMSTO</td>
<td>Akka FSM state timeout - a delay technique</td>
</tr>
<tr>
<td>FSMT</td>
<td>Akka FSM timers - a delay technique</td>
</tr>
<tr>
<td>FTO</td>
<td>future using ask timeout - a delay technique</td>
</tr>
<tr>
<td>GJT</td>
<td>global Java utility timer - a delay technique</td>
</tr>
<tr>
<td>JVM</td>
<td>Java virtual machine</td>
</tr>
<tr>
<td>LJT</td>
<td>local Java utility timer - a delay technique</td>
</tr>
<tr>
<td>ND</td>
<td>no delay - a delay technique</td>
</tr>
<tr>
<td>RTO</td>
<td>Akka receive timeout - a delay technique</td>
</tr>
<tr>
<td>RW</td>
<td>real-world implementation techniques</td>
</tr>
<tr>
<td>SM</td>
<td>state machine</td>
</tr>
<tr>
<td>TE</td>
<td>toy example implementation techniques</td>
</tr>
<tr>
<td>TS</td>
<td>thread sleep - a delay technique</td>
</tr>
</tbody>
</table>
Chapter 1

Thesis introduction

This chapter starts with a background, definition of the thesis work and ends with a small overview of this report.

1.1 Background

This thesis work is done at the Linköping office of the consulting firm named Xdin. In Linköping, Xdin has about 50 employees and works with embedded systems, test, quality assurance and system development among other things. This thesis work is related to Xdin’s operations in the development of test tools for mobile telecommunication systems.

Xdin develops performance test tools for network entities in these mobile telecommunication systems. The performance test tools are programs run on a Java virtual machine (JVM) and simulate many simultaneous entities to be connected to the device under test. These simulated connections are built up via multiple protocol layer (a protocol stack), that must be implemented in the test tools.

A common practice when implementing communication protocols is to represent the different protocols as state machines. What is currently done at Xdin is to use a graphical tool to draw state machines as state transition diagrams, in a plugin called UniMod\(^1\) in the Eclipse development environment (Eclipse IDE\(^2\)). The graphical tool then generates Java code that realizes the state machine that in its turn realizes a protocol.

The problem with the current solution at Xdin is that the state machines are hard to overview and hard to maintain. There are too many states and transitions in the state machines, for it to be easy to use even in the graphical tool.

A distinctive trait of these existing protocol state machines are that they only have a few long-term stable states. A simile would be the states on and

off. In addition to these stable states, there are large quantities of transition states that are there to bridge the stable states. The transition states often form sequences of multiple states, conducting asynchronous communication with other entities via messages sent back and forth between entities. In addition to this, there are also sequences of transition states to facilitate the communication error handling. All in all this amounts to a lot of transition states in between the long-term stable states.

1.2 Purpose

The idea behind the thesis work is to look at other ways than the currently used state machines, to represent protocols in the protocol stack. In essence, the thesis show techniques that aims to simplify the development and maintenance effort of the protocol implementations in the simulation test tool context.

1.2.1 A shift from state machine representation

As mentioned in the background, the protocol state machines have the format of sequences of transition states between the few stable states. These characteristics have led to the idea to represent the protocols as a small state machine in combination with a procedure representation. The procedure representation replaces the current sequences of transition states between the stable states.

The purpose of this thesis is to show that it is possible to in part break away from the current state machine representation in program code for asynchronous communication protocols, thereby reducing the intermediate translation to state machines from a protocol specification.

Further, the purpose of this thesis is to show that some asynchronous protocol procedures can be represented directly as a function or a procedure in the selected host programming language. The asynchronous nature of the protocol can be abstracted away, so the representation becomes compact and concise. In other words, the thesis aims to show how to reduce the translation effort between protocol specification and the representation in program code, compared to a exclusive use of state machines as protocol representation.

1.2.2 A representation suited for high concurrency

As mentioned in the background, the protocol implementations are used in a simulation context in a JVM environment. In the simulations, the number of simulated protocol instances far exceeds the number of possible JVM threads. This means that an abstraction must be used to separate the protocol instances (the concurrently executing entities) from the underlying JVM threads. As an effect, many protocol instances will share the same
JVM thread and this must be of consideration when representing a protocol in program code. Any operation in a protocol instance that blocks a JVM thread will also affect other protocol instances.

1.3 Scope and limitations

Some technology choices are made in order to limit the thesis scope. Since the use of the JVM is a must (a hard requirement), Scala is selected as the programming language to focus on. Scala can be run on the JVM and has some features not available in Java. One such feature is called delimited continuations and is in this thesis report shown to be useful in the thesis context.

This thesis focus on a specific Scala framework to facilitate the need for high concurrency environment. This specific framework is Akka, and is normally referred to as an actor framework.

The thesis does not so much focus the suitability of Scala and Akka in the application context and their suitability compared to other possible technology choices. The lion share of the focus in this report is on how to solve the thesis goals given the technical capabilities in Scala and Akka.

1.4 Goals

The overall goal of the thesis work is to show whether it is possible and practical to fulfill the thesis purpose with the use of Scala and Akka.

In detail, the first goal is to make an investigation of the technical capabilities in Scala and Akka, in relation to the thesis context.

The second goal is to make an implementation of a asynchronous protocol using Scala and Akka. This implementation will practically show how an implementation can fulfill the intentions stated in the purpose section. The implementation is based on a toy example that in some sense is representative for the type of protocols dealt with at Xdin.

The third goal is to some extent evaluate the implementation and discuss the technology choices available. An assessment is made on the practical usefulness and generalizability of the made implementation for other protocols, similar to the toy example protocol.

1.5 Methodology and thesis overview

The thesis work is divided into some sub parts. These are explained here to give an overview of the thesis work and report disposition:

1. An overview of some of the technologies used in this thesis is given at first. The introduced technologies are the Scala programming lan-
guage, actor systems, the Akka actor framework and finally state machines.

2. Investigation of the domain of the problem, to get a deeper understanding of the current solution and to get a grip of what properties are sought after in a solution.

3. Investigation of the chosen technologies and common or possible solutions to technical challenges. An example of such a technical challenge to investigate is how state machines usually are represented and if there are some specific types of representations facilitated by Scala or Akka. Another technical challenge to investigate is how to represent the procedures for asynchronous communication.

4. Investigation and identification of possible problems and areas of interest, when making an implementation in Scala and Akka. One such possible problem is how to enable the asynchronous communication timeouts without blocking or creating numerous threads.

5. Definition of a toy example protocol. A key focus of the toy example is to let it contain properties common in real world examples.

6. Implementation of the toy example, with chosen solutions based on earlier investigations. The rationale for making a concrete implementation is threefold. Firstly, it shows that it can be done. Secondly, a concrete implementation is a great communication tool and a good basis for conclusions. Thirdly, making a concrete implementation highlights unforeseen problem areas and stress tests the technology choices made.

7. Evaluation of the implemented solution, to in some sense make an assessment of whether the thesis work has fulfilled the goals set up in the thesis.

8. Discussion and conclusion based on the the investigations, implementation and evaluations made in the thesis.

The three tasks initial problem domain investigation, definition of a toy example and evaluation of solution (tasks 2, 5 and 7) are made with the help of the thesis supervisor at Xdin - Mattias Evensson. He is in this context regarded as the domain expert, and are therefore used as a source of information for the initial investigation, the toy example specification and the evaluation of the final toy example implementation.

1.6 Audience

The report aims to target an audience at approximately the same level as students at the near end of a computer science or computer engineering
degree. The report makes some assumptions on areas that the reader is expected to know.

The thesis focuses on specific problems, so the content might be most relevant to readers dealing with similar problems. To a wider audience, the thesis report gives discussions and concrete implementation examples of Scala’s delimited continuations, some features in Akka and gives a fairly comprehensive view of the alternative ways to implement state machines in Scala.

The report does often present code listings in Scala, to illustrate a technique or a problem solution, so at least a basic understanding of Scala will to a great extent help the reader.

1.7 Terminology

Two terms are extensively used in the report and need some explanation.

State Machine A state machine (SM) is normally regarded as more general compared to a finite state machine (FSM), because as a theoretical concept a state machine can have an infinite number of states.

In the communication protocol implementation context of this report the number of states in state machines is finite and therefore SM and FSM are used interchangeably. In this report, normally the terms state machine or just SM are used and the term FSM is only used for entities named FSM such as the Akka trait akka.actor.FSM. See section 2.3 for a general introduction and section 4.1 some concrete usage examples.

Asynchronous procedure An asynchronous procedure (AP) is in this report used as a name for a type of program code representation for asynchronous communication. As described in the thesis purpose in 1.2.1, an AP is an asynchronous protocol procedure represented directly as a function or a procedure in the selected host programming language with the asynchronous details of protocol messaging abstracted away. This thesis shows a solution for this AP representation in the Scala programming language.

This report do sometimes refer to AP to be synchronous looking and all that means is that the asynchronous details are abstracted away in the protocol procedure implementation.

1.8 Typographical conventions

These typographical conventions will be used throughout the report:

- Larger code listings for Java and Scala code will look like the example code in listing 1.1. The numbers on the left of the listing indicates the
line numbers in original source code, and may therefore not always begin with the number 1 (one).

- Names that refer to code, such as names of objects, values or functions are in a typewriter font.
- Names and things emphasized are *italic*, as long as they are not related to code.
- URLs are displayed as http://www.some.domain.

*Code listing 1.1: A code listing sample*

```java
val q = "The answer to the ultimate question of life is "
var a = 42
// comment: q is of type String and a is of type Int
println(q + a)
```
Chapter 2

Technology introductions

This chapter gives an overview and a short introduction to some of the technologies relevant to the thesis.

2.1 Introduction to Scala

“Scala is a big language, but you can use it effectively without knowing all of its details intimately.” - Horstman [6]

The fact that Scala is a big language is reflected in the thesis work. Scala solutions and examples presented in code listings are only examples on how to solve things - there are most often also other ways to do it. In this section only some features of Scala will be mentioned and the syntax will not be explained. This report will focus on a few Scala features and their application area in the thesis context. This section will only give an brief overview and try to give some motivation to the Scala focus in this thesis.

2.1.1 Technical details of Scala

Scala unifies object oriented programming with functional programming. Scala is fully object oriented which means that everything is an object. Almost everything in Scala is functional in the sense that it is an expression and returns a value. Even control structures like if and for return a value and the Java keyword void is replaced with the Scala class Unit. As mentioned by Kullberg [7], there is also a class named Option that replaces null and thereby considerably reduces the risk for generating NullPointerException.

Scala has advanced type inference, which means that types are figured out by the Scala compiler, so there are in many cases no need to explicitly write out the type for values, variables, parameters and function. It might however be a good idea to explicitly write out the type in certain places, just for clarity.
Scala is extensible. Odersky et al. [9] states that a language cannot always be perfect, but rather needs to be adaptable, and that Scala has the capability to grow new control structures. An observation on this subject is that from Scala version 2.8.0 onwards, the Scala compiler has a plugin that enables a feature that is called delimited continuation in Scala.

Kullberg [7] mentions both that Scala is open source and not own by a company. It is developed on a university and thereby paid for with taxes.

### 2.1.2 Comparison between Scala and Java

As explained in the background, Java is the language used in the current solution at Xdin. To give some perspective on the use of Scala in relation to the current use of Java, a comparison is made between the two languages. Further, all Scala books in the bibliography have made some type of comparison between Scala and Java and they have their obvious reasons.

According to Kullberg [7], Scala is closely integrated with Java and the Scala performance is comparable with Java. Also, the Scala compiler compiles Scala source code directly to JVM byte code\(^1\). For the JVM, there is no major difference between Scala and Java source code.

Code listings 2.1, 2.2 and 2.3 shows some differences between Java and Scala. All three examples do the same thing - sorts a copy of the string array named data and print it out to standard out stream.

#### Code listing 2.1: A Java example program

```java
import java.util.Arrays;
import java.util.Comparator;
import static data.Data.data;

public final class Java {
    public static void main(String[] args) {
        String[] dataByLength = data.clone();
        Arrays.sort(dataByLength, new Comparator<String>() {
            @Override
            public int compare(String a, String b) {
                return Integer.compare(a.length(), b.length());
            }
        });
        for (String element : dataByLength)
            System.out.println(element);
    }
}
```

\(^1\)Scala code can also be compiled to the .NET platform according to the official Scala site[1].
2.1. INTRODUCTION TO SCALA

Code listing 2.2: A long Scala example program, made to resemble the corresponding Java program in listing 2.1.

```scala
import data.Data.data

object Scala1 {
  def main(args: Array[String]): Unit = {
    val dataByLength: Array[String] =
      data.sortWith { (a, b) => a.length() < b.length() }

    for (element <- dataByLength)
      println(element)
  }
}
```

Code listing 2.3: A short Scala example program, intentionally written to be compact.

```scala
import data.Data.data

object Scala2 extends App {
  data sortWith { _.length < _.length } foreach println
}
```

Listing 2.1 shows an almost minimal\(^2\) Java program. The Scala code in listing 2.2 is written to resemble the Java code in listing 2.1. The code in listing 2.3 is intentionally written to be a small and compact Scala implementation, that do the same thing as the Java code in listing 2.1.

After working with Scala under approximately half a year (the work with this thesis), I have formed some opinions on the differences between Scala and Java. I usually end up explaining to other people that Scala is as Java on steroids - You can access things in Java Runtime Environment (JRE) system library without restrictions, but you program with syntax much more expressive and concise than ordinary Java syntax. On top of that, Scala has libraries that take full advantage of the extra functionality that Scala gives on top of Java.

Further, Scala’s functional programming and type inference are two features that are really handy when you have gotten used to them. Listing 2.3 shows the strength of functional programming, when passing functions as arguments - it is done twice in the single line of code (line 4) in the Scala2 class body.

**Less code** Scala classes sometimes becomes so short it is hard to motivate why they should have a separate file. Besides, there is no need to have separate files for separate Scala classes as it is needed for Java classes. There

\(^2\)disregarding white space characters, imports and variable names
are several examples of that in code listings in this report. According to Kullberg [7] Scala code size is approximately 25% - 50% of the corresponding Java code size.

**Makes some design patterns moot** Scala can to a large or at least some extent eliminate the need for some design patterns commonly used in Java:

- Visitor pattern is replaced with simple pattern matching.
- Decorator pattern can be replaced by the use of trait *mixin*.
- Strategy pattern and SAM\(^3\)-interfaces intended for dependency injection can to a large extent be replaced with, named or anonymous, functions as argument. In other words - they are replaced with functional programming. The *Comparator* in listing 2.1 compared to the anonymous comparison functions in listings 2.2 and 2.3 are excellent examples.

**Increases productivity** Your mileage may vary depending on programming style and adoption rate, but I think Scala lets developers become more productive, compared to Java. If you consider the compact syntax and, let us call it, *dynamically typed language feel* this opinion is somewhat supported by the comparison between seven different languages made by Lutz Prechelt [11]. Worth noting is that Kullberg [7, p. 9] gives the productivity increase together with a decreased risk taking, compared to traditional languages such as Java, as the very reason to learn and adopt Scala.

### 2.1.3 Reasons for the focus on Scala

Here follow some factors for the focus on Scala in the thesis:

- Scala runs on the the JVM.
- Scala has support for concurrency programming via a concept called actors.
- Scala has support for a program construct called delimited continuations, via a technique called continuation passing style. This report shows how the support for delimited continuations can be use as a tool to realize the concept of asynchronous procedure concept.
- Scala is statically typed. According to Odersky et al. [9] static typing gives some advantages over dynamic typing. Source code propertied becomes verifiable, safe refactoring of code can be made and the static typing helps as documentation and communication in the source code.

\(^3\)Single Abstract Method
• Scala is not Java. Java is a part of the current status quo. Current solutions used at Xdin are written in Java. Java does have support for actors via the Akka library but lacks the official support for continuations, essential to achieve the goals of this thesis. See section 4.5 - Asynchronous Procedures for more details.

However, it should be noted that there also exists some third party continuation solutions for Java. Searching for “Java continuation” on Google, a frequently mentioned solution is Apache Commons Javaflow

• Scala integrates seamlessly to Java. Scala is not a big step from Java and this gives a shallow learning curve for Java developers who switch to Scala. Existing legacy code written in Java can be used with ease in Scala.

• Specifying one single language limit the scope of the thesis.

2.1.4 Scala literature

This thesis makes heavy use of Scala, but further explanation of the Scala language in general is out of scope for this thesis. There are many resources on Internet on the Scala language.

The Scala language is currently under development and is subject to change. From 2007 to 2013 there have been from one to five new releases each year, but Scala resources in the bibliography list will probably give a good understanding of the Scala language:

• Programming Scala by Dean Wampler and Alex Payne [15]. Old edition is accessible on Internet. Chapter on actors is not longer valid. Only covers the basics of the language.

• Programming in Scala by Martin Odersky, Lex Spoon and Bill Venners [9]. Old edition is accessible on Internet. This is a long and thorough read. Chapter on actors is no longer valid.

• Scala for the Impatient by Cay S. Horstman [6]. Currently not freely accessible on Internet. This book not as long as some of the other books (less than 400 pages), but does a great job covering advanced topics such as delimited continuations and Scala’s built in parsing capabilities. The downside is that it dives right in and skips much of the basics in Scala. The chapter about actors is out of date for the same reason as in the other books - since Scala version 2.10 the original Scala actors are deprecated and Akka Actors has become a part of the distribution.

---


5According to the distributions archive at the official Scala language site [1]

6Scala version 2.10 was released 4 January 2013
2.2 Introduction to Actors

“You can view the actor model as a special case of object-oriented programming where all communication between objects takes place via message passing, and when an object’s internal state changes only in response to messages.” - Haller and Sommers [5, ch. 2.2]

This section gives an introduction to actors in general and to the Akka framework implementation of actors in specific.

As given by the Akka documentation [14], an actor model is a strategy for implementing concurrent execution. A key concept of a actor model is to avoid shared mutable data (state) between concurrent executing entities in a program by not sharing any mutable data between the entities. These concurrent executing entities are called actors.

The communication between actors is done via asynchronous messages. The messages are asynchronous in the sense “send and forget” - no direct answers are given when a message is sent. Each message received by an actor ends up in what can be called a mailbox. There are no guarantees on the order of received messages, besides that consecutive messages sent from one specific actor to another specific actor are received in the same order they are sent. There is no guarantee that a sent message is delivered. There is also no specific guarantee regarding the time to deliver a message to an actor, besides that the time to deliver is a finite duration.

An actor processes messages that are received in the mailbox, one message at a time. The fact that only one message is processed at a time ensures that there is no race condition between received messages in an actor. Messages sent to an actor are stored in the mailbox until they are processed by the actor.

2.2.1 The problem that actors solve

According to Haller et al. [5] the actor model can be seen as a high-level concurrency abstraction that shields the programmer from the intricacies that can easily lead to errors. Odersky et al. [9] states that actor style programming lets you take advantage of multiple threads and processors in a computer system. They further state that the use of actors with mes-
2.2. INTRODUCTION TO ACTORS

Sage passing hide away the parts of multi-thread programming that are a “minefield of deadlocks and race conditions”.

No locks or monitors for shared-state synchronization

There is no need to synchronize the access to mutable data that is shared between multiple threads, because with actors the data do not need to be accessed directly. With actors, this can be solved by hiding the mutable data within a specific actor. The data can then be read or written to via immutable messages sent to the actor safekeeping the data.

The synchronization of messages sent and received are hidden within the actor system and do not need to be dealt with by the developer using the actors. According to Kullberg [7] actors are easier to reason about than using the synchronization mechanisms in e.g. Java. Kullberg [7] also states that deadlocks in Java are especially tricky, because they are not relieved until the entire JVM is restarted.

Using actors does not eliminate deadlocks and race conditions but Odersky et al. [9] states that using actor style programming reduces the risks involved.

2.2.2 Using actors

Odersky et al. [9, ch. 30], Horstman [6, p. 301] and Kullberg [7, p. 58-60] state some different key points to think about when using actors. Some of these key points are mentioned in this section.

Actors should not block

Horstman [6] states that when an actor blocks, it does not process additional messages received in the mailbox and may well lead to deadlocks. Making synchronous calls between actors (asking a question and awaiting a reply) is one way to block an actor.

An investigation on different ways to trigger a delay in actors is conducted in section 4.4. The investigation among other things shows the performance impact when an actor actually blocks either the underlying thread or the individual actor in the Akka actor framework.

Communicate with actors only via messages

Communication to an actor should be done via the actor message passing mechanism. Other means of communication may bypass the synchronization mechanisms in the actor and undo the benefits of using actors.

In the Akka framework for instance, a great effort is put into anonymizing the actor instances behind a indirect reference to the actor. The indirect access to actors in Akka can be broken out of, by gaining direct access to the actor instance. This direct access does however undo features such as
the actor life cycle management and the possibility to run the actor on a remote system, among other things.

**Prefer immutable messages**

Messages sent to an actor should be immutable. If the message is not immutable, the sender and the receiver can end up sharing access to a mutable message - i.e. a shared mutable state.

**Make messages self-contained**

Horstman [6] states that it is preferable to include contextual data in messages - that an actor should be able to understand a message in isolation, without having to keep track of related messages.

As stated earlier, the order of receiving messages is not always guaranteed and keeping data from earlier messages also means greater complexity in the receiving actor.

**An actor is a single threaded safe haven**

Actors put the focus on the communication between entities (threads or actors). By using a single entry point for the messages sent to an actor, all code inside an actor can be regarded as single threaded. Ordinary threads should not be mixed with actors.

### 2.2.3 The Scala actor library

Before the Scala release version 2.10, there were two separate and well-known actor libraries for Scala. The Scala library was included in the Scala language library while Akka was a completely separate entity. The Scala language library has however initialized the migration to Akka actors in Scala version 2.10, according to *The Scala Actors Migration Guide*\(^7\). The Scala actors will in the future be regarded as deprecated.

### 2.2.4 The Akka framework

Akka implements actors. Akka is a software framework that can be used both as a library and as a standalone program. It is designed for concurrency, remoting and fault tolerance. Some key points from the official Akka documentation [14]:

- Akka has both a Scala API and a Java API, so the Akka framework is not limited to the use of the Scala programming language.

2.2. INTRODUCTION TO ACTORS

- Akka actors can be seen as very lightweight event-driven processes (approximately 2.7 million actors per GB RAM).

- Actors can be run on remote systems and everything in Akka is designed to work in a distributed environment.

- Akka focuses on fault tolerance. Akka supports actor supervisor hierarchies, that can span over multiple JVMs to provide truly fault-tolerant systems. It can be used for writing highly fault-tolerant systems that self-heal and never stop.

Akka has many configuration options and features. Some of those features are mentioned in this report. The FSM-functionality is addressed as a way to implement state machines in section 4.2.2. The Akka dataflow-functionality is mentioned as a possible way to realize asynchronous procedures in section 4.5. An Akka feature called test kit is used as an actor test tool in the implementation of a toy example shown in this report.

Akka performance and scalability

The Akka documentation [14] does claim that Akka actors can be seen as very light-weight processes and this is also backed up by Horstman [6]. He makes a comparison between actor implementations with lightweight actors such as Akka and Scala actors, against Java threads. A computer that supports only a few thousand concurrent threads before the JVM runs out of memory, can support millions or even tens of millions of concurrent actors.

A practical evaluation between threads, thread based actors and event based actors in Scala is made by Haller and Odersky [4]. They note that the JVM has some deficiencies when it comes to concurrency, namely low maximum number of threads and high context-switch overhead. They do conclude that using event-based model for actors dramatically increases the efficiency and scalability compared both to thread based actors and implementation made by using bare threads directly.

One thing to note here is that Akka makes use of an event-based model for its actors. The actors used in the comparison by Haller and Odersky, is the basis for the Scala actor library - the library that as earlier mentioned is replaced by Akka.

The use of Akka in this thesis

This thesis builds on the precondition that Akka actors in highly scalable and enables the development of highly concurrent programs. The performance properties of Akka are assumed to be correct and are not investigated further, besides referring to the statements made by others in this section (2.2.4).

The major focus point on Akka in this thesis is on how to use and incorporate Akka and if some or any of the features of Akka is relevant in the thesis context.
2.3 Introductions to state machines

As mentioned in Erlang/OTP design principles manual[2], many applications can be modeled as FSMs. Protocol stacks are such an example.

If we define $S$ as the current state of a system, $E$ as a trigger event, $A$ as an action to take and $S'$ as some next state, a state machine can be described as a set of relations of the form:

$$S \times E \rightarrow A, S'$$

The meaning of this set of relations are that specific state event combinations ($S \times E$) results in an action $A$ and the system makes a transition to the state $S'$. In other words - If we are in state $S$ and the event $E$ occurs, we should perform the action $A$ and $S'$ becomes the new state of the system.

There are more formal definitions of state machines, but such definitions are not needed in the context of this thesis. UML-notation is used to graphically represent state machines in this thesis report, as in figure 4.1.

2.3.1 What are state machines used for?

“I use them in all levels of a system, from controlling the high-level GUI to the lowest-level communication protocols. They are almost universally applicable.” - Robert C. Martin [8, p. 419]

As mentioned earlier, state machines can be used to implement protocol stacks. Another, frequently mentioned in literature, use of state machines is in graphical user interfaces (GUI).

Inversion of control

A common denominator for these uses is the inversion of control (IOC). Both a state machine used in a communication protocol and a state machine used in a GUI are triggered by events produced by some other entity. In short - the control of the current state has been handed over to some other external entity that triggers the events in the state machine - hence the inversion of control.

Haller et al.[4] states that most programming models support event-driven programming only through inversion of control and says the following:

“Virtually all approaches based on inversion of control suffer from the following two problems: First, the interactive logic of a program is fragmented across multiple event handlers (or classes, as in the state design pattern). Second, control flow among handlers is expressed implicitly through manipulation of shared state.”
2.3. INTRODUCTIONS TO STATE MACHINES

One interpretation of what Haller et al. says is that because the separation in time between triggered events in an event-based system, the control flow needs to be divided up in separate pieces of code, where the separate pieces of code manipulates some shared state (control data).

As an extension of this - the use of state machines can be seen as an abstract concept to overcome the complexities in event-driven mechanisms and can also be seen as a development tool to specify program logic in an event-driven system.

An example of IOC and shared state

One Java program related example is requiring a user to enter a value by typing it the Java console / terminal and reading the value via System.in. Requiring a value via System.in is a blocking operation in Java, but the control flow of the program can be written as one long sequential and synchronous piece of code.

An alternative and more event-driven way is to use multiple Java Swing button and register an action listener in each button. The program logic to be triggered when a button is pressed is hidden in the listener registered in the button. The use of the buttons and the listeners are not blocking, but the control is handed over to the event mechanisms hidden within the buttons.

In the System.in version of the problem, only one value can be entered in the terminal if the program is written as such. In the Swing button version, multiple buttons can be pressed in sequence. One way to hinder the user from triggering multiple program control snippets is to use a state shared by all button listeners. The shared state can then signal whether button is already pressed or not.
Chapter 3

Investigation of current solution

The investigation of the current solution is based on an interview with the thesis supervisor Mattias Evensson, who in this case is regarded as the domain expert.

The purpose of the interview is to, at an early stage, establish a shared view of the thesis problem and the objectives. The original questions and answers of the interview can be seen in appendix A. This section presents a summary of the interview and the conclusions made based on the answers given.

3.1 Common traits of the protocols

A specific real world example of a protocol is given as a representative example. It is a protocol that is both hard to work with and do contain some properties that are representative for the type of protocols that they generally work with at Xdin:

- The protocols contain some few states that can be regarded as the stable states in the protocol. These states are the general modes of operation. In contrast to these stable states, there are states that are intermediate states in an ongoing transition procedure between the stable states. The intermediate transition states are in this report called transition states.

- The protocols contain procedures of several states in sequence, used as transition between the basic states.

- The protocols contain just three different events OK, TIMEOUT and FAIL.
• The procedures to execute in transition between the basic states can in turn make use of sub procedures.

3.2 Graphic visualization of program logic as a state machine

The protocol specification documentation does contain graphical representations of the protocols. There is no need for state machine visualization, especially if an implementation to a large extent resembles the specification. The takeaway here is that it can be regarded as a good thing if the implementation implemented does resemble the specification of the protocol.

3.3 Problems with the current solution

As mentioned earlier in the report, the current solution involves the task of graphically represent the protocol implementation as a state machine in a development tool called UniMod. There might be some shortcomings related to the specific tool, but there is also an intrinsic problem with using state machines, and thereby also representing them graphically.

The problem is that there is, what in the investigation is called, a state explosion in the transition procedures between the basic states. The reason for the state explosion is not explained further in the appended investigation, but some likely explanations are:

• Optional transition steps increasing the number of sub procedures.

• Error handling requiring tear down of partially applied transition procedures, leading to a lot of states just to handle the error handling.

3.4 Technical solutions of interest and limitations

Some limitations (or requirements) on these protocol implementations comes from the context they are used in. The protocol implementations are used in simulation tools that simulate thousands of concurrent entities, all simulated entities using some protocols.

This implies that performance and scalability are of great importance. Earlier experience shows that it is not possible to use one thread per entity\(^1\). The solution must run multiple simulated entities per JVM thread and thereby an entity cannot do anything that blocks the JVM tread (e.g. blocking asynchronous communication), because the blocking operation will affect other entities using the same thread.

\(^1\)The abbreviation UE (user equipment) is used in the investigation answers.
There are some technical concepts that might lead to interesting solutions to these problems:

- The Akka actor framework and possibly use of the Akka FSM functionality. As mentioned in the technical introduction on the Akka framework in section 2.2.4, using Akka actors is assumed to solve the performance and scalability problems for protocols implementations in the simulation context.

- The use of continuations to enable the description of protocol procedures containing asynchronous calls as sequential procedures program code. This is what in this thesis report is defined as asynchronous procedure (AP).

3.5 Focus on the developer

By judging from the answers in the interview, the problem is not so much the functionality of the current representation, but the problem seems rather to be that the current representation is hard to handle from a developer perspective. Some interpretations can be made from the information in the investigation.

One interpretation is that the development process is of importance when working with the protocol implementations. One example from the investigation answers is the need to be able to incrementally develop and improve the protocol implementation. A more specific example is the emphasis of testing in the development process and that the current type of representation leads problems when testing the implementation. The investigation indicates that the current type of protocol representation makes unit testing unnecessary complicates and slow.

Another interpretation is that making procedures containing asynchronous calls look like sequential procedures is thought to make the implementation of the protocol easier to handle. In other words - it is thought to be easier for a developer to write and reason about an AP than breaking up a procedure specification into multiple states in a state machine or some other representation using inversion of control.
Chapter 4

Investigation of possible techniques

This chapter focuses on the investigations needed to get a good foundation for the other parts of the thesis work.

The investigations focuses on roughly three things - ways to represent state machines, how to use timers and schedulers in the performance tool domain and how to actually represent asynchronous procedures.

The different ways to represent state machines are further divided up in common ways, special Scala way and special Akka ways to represent state machines.

4.1 Generic SM implementations

There are several common ways to represent state machines in program code, i.e. implementations techniques. Four common types of implementations techniques for state machines are listed in the book *Agile Software Development* by Robert Martin [8]. These are briefly explained in these sections.

4.1.1 Example state machine

To help illustrate the different techniques a simple example is constructed and used in large parts of this chapter. The example is a state machine representing a ballpoint pen. The ballpoint pen has a button to click, retracting or protracting the ballpoint tip, depending on the current state. With a ballpoint pen you can try to write things, but it will only disperse ink in one of its two states. Table 4.1 shows the transition table for a ballpoint pen state machine and figure 4.1 shows the state transition diagram for the same.
Table 4.1: Transition table for ballpoint pen state machine

<table>
<thead>
<tr>
<th>State</th>
<th>Event</th>
<th>Next state</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protracted</td>
<td>click</td>
<td>Retracted</td>
<td>retract</td>
</tr>
<tr>
<td></td>
<td>write</td>
<td>Protracted</td>
<td>disperse ink</td>
</tr>
<tr>
<td>Retracted</td>
<td>click</td>
<td>Protracted</td>
<td>protract</td>
</tr>
<tr>
<td></td>
<td>write</td>
<td>Retracted</td>
<td>do nothing</td>
</tr>
</tbody>
</table>

Figure 4.1: State transition diagram for a ballpoint pen

Commonalities, interfaces and separation of concerns

The Scala code that is reused in all implementations for the different ballpoint pen state machine representation examples can be seen in listing 4.1. The states in the `State` object and events in the `Event` object are the same in all types of state machine representations, besides in the state pattern representation. In the state pattern representation the state machine states are represented by separate classes.

The abstract class `BallpointPenController` is used to completely separate the ballpoint pen state machine logic from the actions to make in each transition. The point here is that the state machine logic becomes completely separated from the application area it is used in. As long as a state machine has access to concrete instance of a `BallpointPenController` interface, the state machine does not need to know whether the actions just are printed in a text file log or used to control a real pen.

The same idea works in the other direction for the usage of the abstract class `BallpointPen`, which represents the interface to the Ballpoint pen state machine. Here, the code that makes use of a state machine does not need to care which type of representation the state machine uses, because all state machines uses the same interface.

A bonus of separating the state machine logic from the application area it is used in, is that it makes the state machine logic easy to test because no application code need to be involved.
4.1. GENERIC SM IMPLEMENTATIONS

Code listing 4.1: Enumerations and abstract classes common to all Ballpoint pen state machine implementations.

```scala
package common

object State extends Enumeration {
  val protracted, retracted = Value
}

object Event extends Enumeration {
  val click, write = Value
}

abstract class BallpointPenController {
  def protract(): Unit
  def retract(): Unit
  def doNothing(): Unit
  def disperseInk(): Unit
}

abstract class BallpointPen(
  val action: BallpointPenController)
{
  def recieveEvent(event: Event.Value): Unit
}
```

4.1.2 Nested match/case implementation

This type of implementation is originally called a nested switch/case implementation, but in the context of Scala nested match/case implementation is a more suitable name.

The idea behind the nested match/case implementation is that there are two levels of nested match/case blocks, one level for the current state and one level for the current event. Listing 4.2 shows what this can look like for the ballpoint pen state machine implemented in Scala. In this code, each second level case corresponds to a separate transition in the state transition diagram in figure 4.1.

The support for pattern matching in match/case expressions in Scala enables a one-level match/case block for tuples of state and event. This can be seen in listing 4.3.

Martin [8, p. 424] states that for simple state machines, the nested match/case implementation is both elegant and efficient. Martin however, that for larger state machines the code devolves into page after page of case statements, which can be very difficult and error-prone to maintain.
Code listing 4.2: Ballpoint pen state machine as a nested match/case implementation in Scala.

```
package nestedmatchcase
import common._

trait NestedMatchCasePen extends BallpointPen {
  var state = State.retracted

  def recieveEvent (event: Event.Value) {
    state = state match {
      case State.retracted =>
        event match {
          case Event.click =>
            action.protract ()
            State.protracted
          case Event.write =>
            action.doNothing ()
            state // no state change
        }
      case State.protracted =>
        event match {
          case Event.click =>
            action.retract ()
            State.retracted
          case Event.write =>
            action.disperseInk ()
            state // no state change
        }
    }
  }
}
```

4.1.3 Transition table

Listing 4.4 shows a transition table implementation for the ballpoint pen state machine in figure 4.1.

This implementation technique is based on a data table that describes the transitions. This table is interpreted by an engine that handles the events. The engine looks up the transition that matched the event, invokes the appropriate action, and changes the state.

One advantage of the transition table technique is that it is easier to maintain compared to the nested match/case implementation. Another advantage is that, depending on the implementation, the table can be pro-
4.1. GENERIC SM IMPLEMENTATIONS

Code listing 4.3: Ballpoint pen state machine, as a state and event tuple match/case implementation in Scala.

```scala
package tuplematchcase
import common._

trait TupleMatchCasePen extends BallpointPen {
  var state = State.retracted

  def receiveEvent(event: Event.Value) {
    import State., Event..
    (state, event) match {
      case ('retracted', 'click') =>
        action.retract()
        state = State.retracted
      case ('protracted', 'click') =>
        action.protract()
        state = State.protracted
      case ('retracted', '_') => action.doNothing()
      case _ => action.disperseInk()
    }
  }
}
```

grammatically altered at runtime, allowing dynamic alteration of the logic of the state machine.

There can be speed costs associated with this approach. If the table lookup time is not constant, it takes time to search through the transition table. According to Martin [8, p. 425] this time may become significant for a large state machine.

4.1.4 The state pattern

Martin [8, p. 426] states that the state pattern combines the efficiency of the nested match/case statement with the flexibility of interpreting a transition table. It is a object oriented design pattern, that uses a state-interface and different concrete realizations to represent different states. Each realization of the state interface can then implement their own behaviors for each event, resulting in different behaviors in different states.

Figure 4.2 illustrates the state pattern in its generic form. Listing 4.5 and figure 4.3 shows how the state pattern can be implemented in Scala, for the ballpoint pen state machine.
CHAPTER 4. INVESTIGATION OF POSSIBLE TECHNIQUES

Code listing 4.4: Ballpoint pen state machine as a transition table implementation in Scala.

```scala
// ----- Code specific to the ballpoint pen state machine -----
package transitionTable
import common._

trait TransitionTablePen extends BallpointPen {
  import Event., State., action._

  def transitionTable = List(
    Transition(protracted, click, retracted, retract),
    Transition(retracted, click, protracted, project),
    Transition(protracted, write, protracted, disperseInk),
    Transition(retracted, write, retracted, doNothing))

  val engine = TransitionEngine(transitionTable)
  var state = retracted

  def receiveEvent(event: Event.Value) {
    state = engine.process(state, event)
  }
}

// ----- Generic and reusable code for other state machines -----

case class Transition[S, E](
  currentState: S, event: E,
  nextState: S, action: () => Unit)

case class TransitionEngine[S, E](
  table: List[Transition[S, E]]) {

  def process(state: S, event: E): S = {
    val transitionOption = table find
      { t => t.event == event && t.currentState == state }

    transitionOption match {
      case Some(transition) =>
        transition.action()
        transition.nextState

      case None =>
        throw newRuntimeException(“State–event combination ” +
          “%s–%s not found in table!”).format(state, event))
    }
  }
}

Some words about design patterns

Gamma et al. [3, p. 2] addresses the state pattern and other patterns in particular and design patterns in OOP in general. They describe design
4.1. GENERIC SM IMPLEMENTATIONS

Code listing 4.5: Ballpoint pen state machine, as a state pattern implementation in Scala.

```
package statepattern
import common._

abstract class State {
  def write(action: BallpointPenController): State
  def click(action: BallpointPenController): State
}

object RetractedState extends State {
  def click(action: BallpointPenController) = {
    action.protract()
    ProtractedState
  }
  def write(action: BallpointPenController) = {
    action.doNothing()
    this
  }
}

object ProtractedState extends State {
  def click(action: BallpointPenController) = {
    action.retract()
    RetractedState
  }
  def write(action: BallpointPenController) = {
    action.disperseInk()
    this
  }
}

trait StatePatternPen extends BallpointPen {
  var state: State = RetractedState
  def recieveEvent(event: Event.Value) =
    state = event match {
      case Event.click => state.click(action)
      case Event.write => state.write(action)
    }
}
```

patterns as

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this
solution a million times over, without ever doing it the same way twice"
Generally, a design pattern is a conceptual idea on how to organize some code to solve a particular and reoccurring problem. In that sense, also nested match/case and transition table can be regarded as design patterns.

### 4.1.5 The state machine compiler

**Code listing 4.6: Ballpoint pen state machine in a made up example definition format DSL used by some state machine compiler.**

```java
context BallpointPen
controller BallpointPenController
initialState Retracted

state Retracted {
  click Protracted protract
  write Retracted doNothing
}

state Protracted {
  click Retracted retract
  write Protracted disperseInk
}
```

Martin [8] mentions a fourth technique for implementing state machines. He calls this technique *state machine compiler* and it is in short a way to generate program code for the state machine implementation from a definition file, instead of writing the needed program code manually.

According to Martin [8, p. 431], the benefit of using state machine compiler is that the logic of the state machine is strongly isolated from the implementation of the actions and the compact definition requires a minimum of coding. Martin also states that the cost is in the use of state machine compiler, where you have to learn and use another development tool and a definition language.

In the example of the ballpoint state machine this could be realized by (1) defining the state machine in a definition file, (2) generate the Scala code with the state machine compiler tool and as a last step (3) create a implementation of the `BallpointPenController` interface in some class more bound to the application context.

Listing 4.6 is an example of how such a state machine definition can look like for the ballpoint pen state machine. The output Scala code of such a definition run through a state machine compiler can then look just as the state pattern in listing 4.5 or any other of the techniques mentioned in the report.

Using a state machine compiler can also incorporate other state machine related tools that are out of scope for an ordinary Scala development en-
environment or any other programming language development environment. Examples of such state machine related tools are validation of that the state machine is minimal, that all states are reachable from all other states (i.e. no acyclic parts in the state machine) and other constraints that the developer want to apply to the state machine.

4.2 Akka specific SM implementations

This section covers some different ways to combine actors and state machines. The connection between actors and state machines might not be as obvious at first, but is as a matter of fact so closely connected that the Akka actor framework documentation mentions two different solutions to integrate the two.

Section 4.2.1 shows one of the solutions mentioned in the Akka documentation. It is to use two methods available in all Akka actors - become and unbecome.

Section 4.2.2 shows the other way mentioned in the Akka documentation. The solution mentioned is to use the explicitly defined FSM trait as a mixin in the targeted actor.

A possible rationale for the emphasis on state machines in the Akka library documentation can be that the asynchronous nature of actors and the message passing between actors marries very well with the state machine concept, as a tool for logic representation. The state machine enables the actor to change its behavior in the context of multiple messages. The states machine event and the actor message can to some extent be seen as the same thing in two separate technology domains.

To get a complete image that covers all aspects of combining actors and state machines, a third way is investigated. Section 4.2.3 is added to show how to combine an actor with a completely separate state machine. This also shows that the actor and state machine can be loosely coupled and do not need to be tightly integrated with each other, in order to use state machines in the actor domain.

4.2.1 Actor methods become and unbecome

Code listing 4.7 shows an example implementation where the ballpoint pen state machine is implemented as an actor using the become method in given ActorContext class.

Code listing 4.8 shows one way to adapt the state machine interface to a concrete actor implementation. In this case the class AkkaBecomePen, that concrete class that implements the BallpointPen state machine interface, acts as an adapter to the actor BecomeActor.
4.2. AKKA SPECIFIC SM IMPLEMENTATIONS

Code listing 4.7: Ballpoint pen state machine, implemented as an actor changing behavior with become method.

```scala
package akkaecome
import common._
import akka.actor._

/** Actor with the two behaviors protracted and retracted. */
class BecomeActor(action: BallpointPenController) extends Actor {
  import context._

  def protracted: Receive = {
    case Event.click =>
      action.retract()
      become(retracted)
    case Event.write =>
      action.disperseInk()
  }

  def retracted: Receive = {
    case Event.click =>
      action.protract()
      become(protracted)
    case Event.write =>
      action.doNothing()
  }

  def receive = retracted // ← initial behavior
}
```

Code listing 4.8: Adapter between a ballpoint pen state machine (instance of class BallpointPen) and the BecomeActor actor.

```scala
/** Adapter between BallpointPen and the BecomeActor. */
trait AkkaBecomePen extends BallpointPen {
  val pen = {
    val system = ActorSystem.create()
    val props = Props(new BecomeActor(action))
    system.actorOf(props)
  }

  def receiveEvent(event: Event.Value) = pen ! event
}
```
Stack multiple behaviors

Actor using the `become` functionality also has the functionality to stack multiple behaviors given by the `become` method and remove the topmost behavior with method `unbecome`.

In the example implementation in listing 4.7, this could be realized by replacing `become(retracted)` at line 12 with `unbecome` and overriding the default argument to discard old behaviors\(^1\) in `become(protracted)` at line 21 with the method call `become(protracted,false)`.

A possible pitfall

A possible pitfall with overriding the default behavior to discard old behaviors and instead combining the methods `become` with the method `unbecome` is that the actor can store an excessive amount of behaviors and eventually the JVM gets into memory trouble.

A little test has been conducted to verify that this pitfall is real if not careful to also remove stacked behaviors with the `unbecome` method. An actor gets the behavior to send a message to itself. Each time the message is received, the current behavior is stored on the behavior stack with a call to `become` and a new message is sent to itself. This creates a loop that forces the actor behavior stack to grow indefinitely.

The Scala program is executed on a computer with the OpenJDK 64-Bit Server VM JVM, the Scala code runner version 2.9.2, Akka version 2.0.4 and an option is given to the JVM that specifies that the maximum size of the memory allocation pool is 7000 MB.

After around 2.5 minutes and 120,000,000 calls to the `become` method, the computer starts to really walk on its knees. All approximately 7 gigabyte of memory given to the JVM is used up and further iterations are done in a much slower pace than it started with.

### 4.2.2 Akka FSM implementation

The functionality given by the Akka trait `akka.actor.FSM` is specifically created to facilitate state machines directly in actors. The code listing 4.9 gives an example how this could be done for the ballpoint pen state machine.

Compared to using the `become/unbecome` solution, the `FMS` trait also has some extra features available to ease the development of state machines, such as specifying state timeouts and to carry data between states. The Akka documentation \[14\] describes all features in details.

When looking in the source code of file `akka.actor.FSM.scala` in the Akka library version 2.0.4, I draw the conclusion that the FMS functionality is implemented as an ordinary actor, but does not explicitly make use of the `become/unbecome` functionality. The FMS implementation keeps all partial functions added with the `when`-method in a map with the given state as a

---

\(^1\)Given by the default boolean argument value `true`
4.2. AKKA SPECIFIC SM IMPLEMENTATIONS

Code listing 4.9: Ballpoint pen state machine, implemented as an actor specifically using the akka.actor.FSM trait as an actor mixin.

```
package akka_fsm
import common._
import akka.actor._

class FsmActor(action: BallpointPenController)
  extends Actor with FSM[State.Value, Unit] {
    import common.State., common.Event._
    val noData = Unit

    startWith(retracted, noData)
    when(protracted) {
      case Event('click', _) =>
        action.retract()
        goto(retracted)

      case Event('write', _) =>
        action.disperseInk()
        stay
    }

    when(retracted) {
      case Event('click', _) =>
        action.protract()
        goto(protracted)

      case Event('write', _) =>
        action.doNothing()
        stay
    }
  }
```

map key, and is therefore not vulnerable to the same problem as become/un-become can be.

4.2.3 Composition of state machine and actor

This section does not show any Akka specific state machine functionality. It only shows that the composition of a state machine and an actor can be done in a loosely coupled manner, where the actual SM implementation is injected into the actor as a constructor argument. Listing 4.10 shows an example on how this can be done.

This separation of actor and behavior logic can facilitate more specialized or specific types of SM representations. The separation enables the behavior logic representation to be more directed to the specific usage area than generic Akka SM representations using become method or the FSM trait.
CHAPTER 4. INVESTIGATION OF POSSIBLE TECHNIQUES

Code listing 4.10: Ballpoint pen state machine, implemented as an actor composed with an dependency injected state machine implementation.

```scala
/** Actor, where state machine is injected via constructor. */
class ComposedActor(fsm: BallpointPen) extends Actor {
  def receive = { case event: Value => fsm.receiveEvent(event) }
}
```

mentioned in sections 4.2.1 and 4.2.3.

Examples of when separation can be convenient is when one of the generic types of representations mentioned in section 4.1 is found to be easier to work with, than the ones given by the Akka framework. Perhaps the separation of actor and state machine helps the developer in some way such as e.g. testing.

Examples of more specialized types of behavior logic are given in section 4.3 - state machine implementations enabled by the Scala programming language - and in section 4.5 - asynchronous procedures. These sections show representations that can be more directed to the usage area of the behavior logic.

4.3 Scala specific SM implementations

This section focuses on functionality in the Scala programming language that can be of an advantage when implementing state machines. As it turns out, there are Scala features that can help when representing state machines, but the features are not restricted to only be used when implementing state machines.

This section will mostly deal with the concept of domain specific languages (DSL). This concept is not limited to Scala, but Scala happens to have some capabilities that make two different kinds of DSL easy to use.

In the scope of this section, the focus is on how Scala features can help the implementation of state machines. This focus area serves as a good example on how to use domain specific languages. However, one thing to note is that the Scala DSL features are general and can also help to leverage other parts of the protocol representations in this thesis.

4.3.1 Domain specific languages

In this context, a domain specific language is a programming language tailored to suite terms, expression and idioms of a specific domain or subject.

The book Programming Scala [15, ch. 11] states the following four benefits of using a well-designed DSL compared to using a plain programming approach without using a DSL:
4.3. SCALA SPECIFIC SM IMPLEMENTATIONS

- Implementation detail encapsulation
- Developer effort efficiency
- Improved communication between stakeholders, such as developers and domain experts.
- Increasing implementation quality by reducing the need for extensive translation between feature requirements and implementation source code.

*Programming Scala* [15, ch. 11] also mentions some drawbacks with DSLs. These drawbacks are mostly the difficulty to capture the details of the domain in the DSL and to cope with maintenance when the domain modeled in the DSL changes.

4.3.2 Internal DSL

Scala enables something called internal DSLs via some features in the Scala language. One interpretation of the prefix *internal*, is that Scala has features that let the developer create structures (classes, methods, implicit conversions, operator overloading\(^2\) and more) that lets Scala code look like another language. A good example of this is Baysick\(^3\), an internal DSL in Scala that is made to look like the Basic programming language.

**Ballpoint pen state machine as an internal DSL**

As a context related example, an internal DSL created for the state machine domain. All types of transitions can be seen in code listing 4.11. The intention is that the DSL should look like text interpretations of hand drawn transitions. It is built up with, among other things a *State* class that has four operator-like methods that takes either a event or another state as argument. The methods are \(-\rightarrow\), \(-\leftrightarrow\), \(-?\rightarrow\) and \(-?\leftrightarrow\). These methods, in their turn, return some type with other operator-like methods. Using some imagination, the combination of these methods ends up looking like some kind of transition arrows between states.

Listing 4.12 uses the state machine DSL to implement a state machine for the ballpoint pen example mentioned in section 4.1.

**A more advanced example - PIN code lock**

The ballpoint pen example is not a very large and advanced example. It has two states, two events and therefore a maximum of four state-event combinations.

---

\(^2\)Scala does not actually have operator overloading, because operators are methods in Scala.

Code listing 4.11: An example of how an internal DSL for SMs can look like.

```scala
// Ordinary transition with action.
state --- event /-- action --> nextState

// Ordinary transition without action.
state --- event --> nextState

// Recursive transition with action.
state *-- event /-- action

// Recursive transition without action.
state *-- event

// Default behaviour when the transition
// (i.e. state-event combination) is not defined.
state -?--> nextState

// Default behaviour with a defined action.
state -?-/-- action --> nextState
```

A more advanced example is a PIN code lock, with a ten digit numeric key pad and pin codes of length four. Another feature of the PIN code lock is that the user must use the code both when locking and unlocking the lock. The logic behind PIN code lock requires eight states and ten different events - one event for each digit pressed. This leads to a total of 80 different state-event combinations to take care of in an implementation.

Code listing 4.13 shows a definition of all events and actions for the PIN code lock. Code listing 4.14 shows the implementation of the state machine.

Here follow some indications of advantages to the developer, given by this specific DSL and the implementation of the PIN code lock in listing 4.14:

- Using some kind of object instance to uniquely identify different states, enables the states to be used as arguments to other Scala methods. As it is done here, the basic states `locked` and `unlocked` can easily be sent as arguments to the `correctPinCode` method. This method helps to build up a more advanced structure of intermediate states as a part of the transition between the basic states. In this specific example, eight states are created because the method `correctPinCode` is called twice.

This ability to programmatically structure the creation of a state machine can be a big advantage in large state machines. As it is represented in this DSL, the state instances in the resulting state machine are represented in a completely flat hierarchy, but the Scala code that
4.3. SCALA SPECIFIC SM IMPLEMENTATIONS

**Code listing 4.12: Ballpoint pen state machine implemented with an internal domain specific language (DSL) for state machines.**

```scala
package internaldsl
import common._

trait InternalDslPen extends BallpointPen {
  import dslstatemachine.{ State => DslState }
  import dslstatemachine.StateMachine
  import common.Event._

  val fsm = {
    def state = DslState[Value]_
    val retracted = state("ballpoint pen is retracted")
    val protracted = state("ballpoint pen is protracted")

    retracted ---- click |-- action.protract ---> protracted
    protracted ---- click |-- action.retract ---> retracted

    retracted --- write |-- action.doNothing
    protracted --- write |-- action.disperseInk

    StateMachine(retracted)
  }

  def receiveEvent(event : Value) = fsm.trigger(event)
}
```

**Code listing 4.13: Pin code lock events and actions definition.**

```scala
object PinKeyEvent extends Enumeration {
  val key0, key1, key2, key3, key4, key5, key6, key7, key8, key9 = Value
}

abstract class PinLockController {
  def lock()
  def unlock()
}
```

creates the state machine can be structured with the help of all code structure and organization concepts built into the Scala language, such as methods, objects and packages.

- Normally, as for the actor using the `FSM` trait in listing 4.9 among others, the state machine definitions tend to focus on the current state
and group together all events that can happen in that state. This state machine DSL can be regarded as centered on each individual transition to make between states. The developer can still cluster all transitions from a specific state together, but it enables the developer to also group the transitions in any way the developer finds advantageous. Examples are the cluster of transitions on lines 40-43, 47-48 and 51-54 in code listing 4.14. These transition clusters are grouped together based on other premises than that they originate from a common source state.

- Another advantage is that the state machine representations using the example DSL (i.e. listings 4.12 and 4.14) seem to be fairly compact compared to some other types of state machine representations such as using the state pattern or Akka FSM trait in listings 4.5 and 4.9 respectively.

### 4.3.3 External DSL

An external domain specific language encapsulates the same concepts as using an internal domain specific language. The purpose is to tailor some kind of representation to reflect the targeted domain.

The large differences between internal and external domain specific languages are that an internal DSL in fact is Scala code that is made to look and work in a certain way. On the other hand, an external DSL is not Scala code at all, but instead an arbitrary language format that might be intended to be parsed by some Scala program. One example of an external DSL is in fact the made up state definition format in listing 4.6.

The Scala standard library provides some tools called parser combinators. The parser combinators are found in standard library package `scala.util.parsing.combinator`. As the name indicates, these combinators can be combined to create a complete parser for some input format. The created parser can for instance be designed to generate an abstract syntax tree from a given source.

In the context of state machines, the external DSL tools given by Scala standard library are in fact just a convenient way to realize a state machine compiler, mentioned in section 4.1.5.
Code listing 4.14: Pin code lock state machine implemented with an internal domain specific language (DSL) for state machines.

```scala
object PinLock {
  import PinKeyEvent._
  import dslstatemachine.StateMachine
  import dslstatemachine.State
  type Action = () => Unit
  type S = State[PinKeyEvent.Value]
  type SM = StateMachine[PinKeyEvent.Value]

  def apply(action: PinLockController): SM = {
    // the basic states
    val locked = new S("locked")
    val unlocked = new S("unlocked")

    // transcend between basic states when pin code is correct
    correctPinCode(locked, unlocked, action.unlock)
    correctPinCode(unlocked, locked, action.lock)
    StateMachine(locked)
  }

  def correctPinCode(start: S, finish: S, action: Action) = {
    // intermediate states between start and finish
    val ok1 = new S("1st digit correct")
    val ok2 = new S("2nd digit correct")
    val ok3 = new S("3rd digit correct")

    // the correct code is 8879
    start --- key8 -->
    ok1 --- key8 -->
    ok2 --- key7 -->
    ok3 --- key9 -- action --> finish

    // if key is wrong at (current state) but at the
    // same time the first digits(s) in the correct sequence
    ok2 --- key8 --> ok2
    ok3 --- key8 --> ok1

    // start over if transition is not explicitly defined
    start ?--> start
    ok1 ?--> start
    ok2 ?--> start
    ok3 ?--> start
  }
}
```
4.4 How to delay, take time and schedule

This section explores the different ways to delay, take time and schedule tasks in Scala and Akka and in this section, some concrete types of time keeping are compared to each other.

In the scope of this thesis - implementation of asynchronous protocol - one potential usage area for this knowledge is to be able to trigger timeouts when there is no interaction between asynchronously communicating entities. A use case is that a message response is expected to be received within a certain time limit and some fault handling should be done if the time limit is not kept.

One of the preconditions related to this thesis work is that there can exist more protocol instances than the number of threads that the JVM can sustain. This generates some criteria on the time keeping mechanisms used. The criteria for the time keeping mechanisms are that they cannot block a Java thread and they cannot spawn a new Java thread for each time task.

4.4.1 Delay test bench evaluation

A Scala-based test bench is constructed and used in order to be able to compare the different ways to keep time. Some of the ways of keeping time are features in the Akka library, and the complete test bench is therefore centered on keeping time withing actors.

The conceptual idea of the test bench is that a dedicated test execution actor creates a number of delay actors. The delay actors will more or less at the same time receive a message token that should be delayed a certain number of milliseconds and then resent back to the test execution actor. All delay actors implement one of the techniques under investigation.

The idea is to use so many actors at the same time in a test, that if the tested technique happens to block the running Java threads, it will affect the other delay actors running in the same threads. Thereby, tests conducted with techniques that block Java threads will clearly stand out, because it will result in some delay actors taking notably longer time than expected.

The token to pass around

Most part of the test bench Scala code is omitted in this thesis report. However, some parts are included to both show how the delay actors work and how the different techniques can be used.

The token to be passed from and to the delay actors can be seen in listing 4.15. The token both contains the start time when the token is sent away to the delay actor. The start time enables the execution actor to calculate the actual delay of the token message, when token is received from delay actor. The delay duration value in the token tells the delay actor how long the token should be delayed.
4.4. HOW TO DELAY, TAKE TIME AND SCHEDULE

Code listing 4.15: Delay test bench message token

```scala
sealed trait TestMessage

case class TestToken(start: Long, delay: Long)
  extends TestMessage
```

Abstract delay actors

Some of the delay actors have much in common and does therefore extend a
common abstract delay actor just for convenience. The `AbstractDelayActor`
class in listing 4.16 is one of the abstract delay actors used in the tests. It
is a generic delay actor with a abstract `delay` method. The other abstract
delay actor is the class `AbstractAkkaFsmActor` in listing 4.17. It focuses on
features in the Akka `FSM` trait and is therefore more specialized.

Code listing 4.16: Delay test bench class `AbstractDelayActor`

```scala
import akka.actor._

abstract class AbstractDelayActor extends Actor {
  def delay(ms: Long)(delayedExpr: => Unit): Unit
  def receive = {
    case token: TestToken =>
      val replyToActor = sender
      delay(token.delay) { replyToActor ! token }
  }
}
```

4.4.2 Immediate response with no delay (ND)

To be able to determine how much the test bench in itself affects the result,
a delay actor that only makes an immediate response instead of doing an
actual delay is included. It can be seen in code listing 4.18.

4.4.3 Thread sleep (TS)

To explicitly do a thread sleep is the obvious anti-solution to this problem.
Listing 4.19 shows a usage example.
CHAPTER 4. INVESTIGATION OF POSSIBLE TECHNIQUES

4.4.4 Java utility timer (GJT and LJT)

There exists a Timer utility in the Java system library that can be used. According to Java 6 documentation [10], there is a one-to-one relation between timer and Java threads. Therefore two different tests are conducted with class java.util.Timer. One of the delay actors (GlobalJavaUtilTimerTaskActor) makes use of a single global timer instance. The other delay actor (LocalJavaUtilTimerTaskActor) creates a local timer instance for each cre-
4.4. HOW TO DELAY, TAKE TIME AND SCHEDULE

Code listing 4.19: Delay actors using Thread.sleep

```scala
final class JavaThreadSleepActor extends AbstractDelayActor {
  def delay(ms: Long)(delayedExpr: => Unit) = {
    Thread.sleep(ms)
    delayedExpr
  }
}
```

ated actor. Both actors can be seen in listing 4.20.

Code listing 4.20: Delay actors using java.util.Timer

```scala
import java.util.TimerTask
import java.util.Timer
import JavaUtilTimer.

object JavaUtilTimer {
  val timer = new Timer()
  implicit def exprToTimerTask(expr: => Unit) =
    new TimerTask() { def run() = expr }
}

class GlobalJavaUtilTimerTaskActor extends AbstractDelayActor {
  def delay(ms: Long)(delayedExpr: => Unit) =
    timer.schedule(delayedExpr, ms)
}

class LocalJavaUtilTimerTaskActor extends AbstractDelayActor {
  def delay(ms: Long)(delayedExpr: => Unit) =
    new Timer().schedule(delayedExpr, ms)
}
```

4.4.5 Akka scheduler (AS)

The Akka library has a scheduler that can be utilized. Listing 4.21 shows how this can be done in actor AkkaSchedulerActor.

4.4.6 Akka FSM state timeout (FSMSTO)

The Akka FSM trait has the ability to generate a message of type StateTimeout when there is no message received within some time limit. How this can be used is shown in actor AkkaFsmStateTimeoutActor in listing 4.22.
CHAPTER 4. INVESTIGATION OF POSSIBLE TECHNIQUES

**Code listing 4.21: Delay actors using Akka scheduler**

```scala
import scala.concurrent.duration.DurationLong

final class AkkaSchedulerActor extends AbstractDelayActor {
  import context._

  def delay(ms: Long)(delayedExpr: => Unit) =
  context.system.scheduler.scheduleOnce(ms milliseconds)(delayedExpr)
}
```

**Code listing 4.22: Delay actors using Akka FSM state timeout**

```scala
import scala.concurrent.duration.FiniteDuration

final class AkkaFsmStateTimeoutActor extends AbstractAkkaFsmActor {
  val timeoutEvent = this.StateTimeout
  def delay(duration: FiniteDuration)(state: State) =
  state forMax (duration)
}
```

### 4.4.7 Akka FSM timers (FSMT)

**Code listing 4.23: Delay actors using Akka FSM timers**

```scala
import scala.concurrent.duration.FiniteDuration

final class AkkaFsmTimerActor extends AbstractAkkaFsmActor {
  val timeoutEvent = "myTimeoutEvent"
  val myTimer = "myDelayTimer"
  val noRepeat = false

  def delay(duration: FiniteDuration)(state: State): State = {
    setTimer(myTimer, timeoutEvent, duration, noRepeat)
    state
  }
}
```

The Akka FSM trait has the ability to set multiple generic timers iden-
4.4. HOW TO DELAY, TAKE TIME AND SCHEDULE

Code listing 4.24: Delay actors using Akka receive timeout

```scala
import scala.concurrent.duration.Duration
import scala.concurrent.duration.DurationLong
import akka.actor._

class AkkaReceiveTimeoutActor extends Actor {
  import context._

  def idle: Receive = {
    case token: TestToken =>
      setReceiveTimeout(token.delay milliseconds)
      become(active(sender, token))
  }

  def active(target: ActorRef, token: TestToken): Receive = {
    case ReceiveTimeout =>
      target ! token
      setReceiveTimeout(Duration.Undefined) // no more timeout
      become(idle)
  }

  def receive = idle
}
```

tified with string names. According to the Akka documentation [14], these named timers complement state timeouts because they are not affected by intervening reception of other messages. Also, these timers can be canceled using the method call `cancelTimer(name)` which is guaranteed to work immediately. How this can be used is shown in actor `AkkaFsmTimerActor` in listing 4.23.

4.4.8 Akka receive timeout (RTO)

This generic feature of all Akka actors resembles the Akka FSM state timeout, to a large extends. Actor `AkkaReceiveTimeoutActor` in listing 4.24 shows how this can be used.

4.4.9 Future with ask timeout (FTO)

Both Scala and Akka contain the concept of a `Future`. It can be regarded as a placeholder for a not yet given result - i.e. a placeholder for a `future` result.

The delay actor example in listing 4.25 may look a little bit convoluted, because the delay actor `AkkaFutureTimeoutActor` asks itself a question that it do not answer. This should trigger a timeout exception that can be used
Code listing 4.25: Delay actors using Future with ask timeout

```scala
import scala.concurrent.duration.DurationLong
import akka.pattern.AskTimeoutException
import akka.pattern.ask
import akka.util.Timeout.durationToTimeout

class AkkaFutureTimeoutActor extends AbstractDelayActor {
  import context._

  val timeoutQuestion = "Hello?"
  val doNothing = ()

  override def receive = super.receive orElse {
    case timeoutQuestion => doNothing
  }

  def delay(ms: Long)(delayed: => Unit) =
    ask(self, timeoutQuestion)(ms milliseconds).onFailure {
      case _ : AskTimeoutException => delayed
    }
}
```

as a timer.

### 4.4.10 Future with Await.result (FAR)

An instance of a future is not in itself blocking Java threads, but waiting for the result using `Await.result` is blocking according to the Akka documentation [14].

The actor in listing 4.26 should block the Java thread while waiting on a `AskTimeoutException`. This technique, explicitly waiting for a future result, is taken into account in this investigation just to ensure that it actually is a bad technique.

### 4.4.11 Comparison

All different techniques are run in the delay test bench with an expected delay of 1000 milliseconds (1 second). The tests are run on a desktop computer that at the time of writing (and test execution) can be regarded to have a reasonable performance.

Each test conducted makes use of a specific quantity of concurrently executed delay actors. All actors make use of the same delay technique. Each delay actor in a test gets a token message to delay. The actual delays for all message tokens are measured and collected. Both a mean delay value
4.4. HOW TO DELAY, TAKE TIME AND SCHEDULE

Code listing 4.26: Delay actors using Future with `Await.result`

```scala
import java.util.concurrent.TimeoutException
import scala.concurrent.Await
import scala.concurrent.duration.DurationLong
import akka.pattern.ask
import akka.util.Timeout.durationToTimeout

class AkkaFutureAwait extends AbstractDelayActor {
  import context._

  val timeoutQuestion = "Hello?"
  val doNothing = ()

  override def receive = super.receive orElse {
    case timeoutQuestion => doNothing
  }

  def delay(ms: Long)(delayed: => Unit) = {
    val future = ask(self, timeoutQuestion)(10 * ms milliseconds)
    try {
      Await.result(future, ms milliseconds)
    } catch {
      case _: TimeoutException => delayed
    }
  }
}
```

and a max delay value in milliseconds are calculated, based on all token message delays in a test. The mean value is rounded to whole milliseconds.

Table 4.2 shows three different delay techniques. The first technique (ND) shows the intrinsic delays in the delay test bench when used on the test computer. The table shows that the test system in itself introduces significant delays when using 10000 and 20000 concurrent delay actors. For this reason no test makes use of more than 20000 actors.

The second technique in table 4.2 shows the delays when using a global instance of the `java.util.Timer` class (GJT). This technique shows the best delay performance of all techniques up to around 10000 concurrent delay actors and then seems to fall behind when using 20000 concurrent delay actors.

The second technique in table 4.2 shows the delays when using the Akka scheduler (AS). The delay values for the Akka scheduler are also representative for some of the other techniques - Akka FSM timer (FSMT), Akka FSM state timeout (FSMSTO), Akka receive timeout (RTO) and the use of future timeout (FT).
Table 4.2: Delay in milliseconds, for different quantities of concurrent delay actors using three different techniques - no delay (ND), global timer (GJT) and Akka scheduler (AS).

<table>
<thead>
<tr>
<th>quantity</th>
<th>no delay (ND)</th>
<th>global timer (GJT)</th>
<th>Akka scheduler (AS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>max</td>
<td>mean</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>1001</td>
</tr>
<tr>
<td>200</td>
<td>0</td>
<td>10</td>
<td>1001</td>
</tr>
<tr>
<td>500</td>
<td>10</td>
<td>20</td>
<td>1003</td>
</tr>
<tr>
<td>1000</td>
<td>16</td>
<td>40</td>
<td>1008</td>
</tr>
<tr>
<td>2000</td>
<td>25</td>
<td>40</td>
<td>1033</td>
</tr>
<tr>
<td>5000</td>
<td>67</td>
<td>112</td>
<td>1097</td>
</tr>
<tr>
<td>10000</td>
<td>198</td>
<td>382</td>
<td>1583</td>
</tr>
<tr>
<td>20000</td>
<td>600</td>
<td>1370</td>
<td>1583</td>
</tr>
</tbody>
</table>

Table 4.3: Delay in milliseconds, for different quantities of concurrent delay actors using Java thread sleep (TS).

<table>
<thead>
<tr>
<th>quantity</th>
<th>mean</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1002</td>
<td>1011</td>
</tr>
<tr>
<td>20</td>
<td>1401</td>
<td>2001</td>
</tr>
<tr>
<td>50</td>
<td>4042</td>
<td>5002</td>
</tr>
<tr>
<td>100</td>
<td>8041</td>
<td>9001</td>
</tr>
<tr>
<td>200</td>
<td>16041</td>
<td>17001</td>
</tr>
</tbody>
</table>

As expected, the use of `Thread.sleep` generates delays that deviate considerably from the expected delay of 1000 milliseconds. This can be seen in table 4.3. The actual resulting delays follow the number of delay actors almost linearly. The mean and max values follow each other quite closely, and this can be a sign of that the delay actors not only interfere with each other, but also blocks the test execution actor thread, since the execution actor thread is one of the threads the Akka actor system uses for all actors.

The techniques using local instances of `java.util.Thread` (LJT) and the use of `Await.result` (FAR) are both expected to perform badly in the test bench. They do not perform as badly as the use of `Thread.sleep` (TS), but do still perform notably worse than most of the other techniques.

As seen in table 4.4, these two techniques seems to perform worse for the quantity of 5000 concurrent delay actors than for a quantity of 10000 concurrent delay actors. I find no apparent explanation for this behavior. This behavior is reproduced when the test bench is run multiple times - even when the relative order between tests of different techniques is rearranged in the test bench. However, one conclusion than can be made is that these two techniques do worse than many of the other techniques for large quantities of concurrent delay actors.

As a comparison between different delay-techniques, table 4.5 and 4.6...
### Table 4.4: Delay in milliseconds, for different quantities of concurrent delay actors using two different techniques - local timer (LJT) and future await (FAR).

<table>
<thead>
<tr>
<th>quantity</th>
<th>local timer (LJT)</th>
<th>future await (FAR)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>max</td>
</tr>
<tr>
<td>1 000</td>
<td>1 029</td>
<td>1 061</td>
</tr>
<tr>
<td>2 000</td>
<td>1 070</td>
<td>1 150</td>
</tr>
<tr>
<td>5 000</td>
<td>7 073</td>
<td>7 966</td>
</tr>
<tr>
<td>10 000</td>
<td>1 317</td>
<td>1 644</td>
</tr>
<tr>
<td>20 000</td>
<td>2 320</td>
<td>3 102</td>
</tr>
</tbody>
</table>

### Table 4.5: Delay in milliseconds, for different techniques for the specific quantity of 1 000 concurrent delay actors.

<table>
<thead>
<tr>
<th>delay technique</th>
<th>mean</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>no delay (ND)</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>global timer (GJT)</td>
<td>1 003</td>
<td>1 011</td>
</tr>
<tr>
<td>local timer (LJT)</td>
<td>1 029</td>
<td>1 061</td>
</tr>
<tr>
<td>future await result (FAR)</td>
<td>1 041</td>
<td>1 111</td>
</tr>
<tr>
<td>Akka scheduler (AS)</td>
<td>1 070</td>
<td>1 101</td>
</tr>
<tr>
<td>future timeout (FTO)</td>
<td>1 092</td>
<td>1 101</td>
</tr>
<tr>
<td>Akka FSM timer (FSMT)</td>
<td>1 100</td>
<td>1 107</td>
</tr>
<tr>
<td>Akka FSM state timeout (FSMSTO)</td>
<td>1 101</td>
<td>1 101</td>
</tr>
<tr>
<td>Akka receive timeout (RTO)</td>
<td>1 200</td>
<td>1 202</td>
</tr>
</tbody>
</table>

### Table 4.6: Delay in milliseconds, for different techniques for the specific quantity of 20 000 concurrent delay actors.

<table>
<thead>
<tr>
<th>delay technique</th>
<th>mean</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>no delay (ND)</td>
<td>600</td>
<td>1 370</td>
</tr>
<tr>
<td>Akka FSM timer (FSMT)</td>
<td>1 328</td>
<td>1 864</td>
</tr>
<tr>
<td>Akka FSM state timeout (FSMSTO)</td>
<td>1 331</td>
<td>1 862</td>
</tr>
<tr>
<td>Akka scheduler (AS)</td>
<td>1 380</td>
<td>1 990</td>
</tr>
<tr>
<td>Akka recieve timeout (RTO)</td>
<td>1 404</td>
<td>2 042</td>
</tr>
<tr>
<td>future timeout (FTO)</td>
<td>1 412</td>
<td>2 100</td>
</tr>
<tr>
<td>global timer (GJT)</td>
<td>1 583</td>
<td>2 301</td>
</tr>
<tr>
<td>local timer (LJT)</td>
<td>2 320</td>
<td>3 102</td>
</tr>
<tr>
<td>future await result (FAR)</td>
<td>182 048</td>
<td>183 032</td>
</tr>
</tbody>
</table>
show compilations of all techniques for specific quantities of concurrent delay actors. Table 4.5 shows the comparison for 1 000 concurrent delay actors, while table 4.6 shows the comparison for 20 000 concurrent delay actors. The techniques are ordered by their mean delay. The use of Java thread sleep (TS) is not included in these tables.

Given the data collected in the test, it seems like there are several viable alternatives to choose from. It seems that the Akka related techniques FSMT, FSMSTO, AS and RTO have more or less equal performance, but may differ in suitable usage area. Using a global Java timer (GJT) seems to have the best accuracy of all up to a specific limit of concurrent timer tasks. Techniques not to use are LJIT, FAR and TS.

### 4.5 Asynchronous Procedures

As stated in the terminology definition for AP in section 1.7 the intended meaning and usage is to represent consecutive asynchronous calls to other entities in some representation that in a way resembles synchronous communication by hiding the asynchronous implementation details.

Question 3 in the initial investigation interview (see appendix A) shows how such an AP could look like in pseudo code:

```java
def myProcedure() {
    try {
        send(new SetupLowerLayerReq());
        resp = receive();
        if (resp.result == FAIL)
            throw new FailureException();
        send(new MessageReq);
        resp = receive();
        if (resp.result == FAIL)
            throw new FailureException();
    } catch (TimeoutException | FailureException) {
        send(new TeardownLowerLayerReq());
        resp = receive();
    }
}
```

This procedure pseudo code makes use of the two functions `send` and `receive` as functions to send and receive asynchronous messages without needing to deal with the time delay between a call to `send` and the result returned from a call to `receive`. The intention of this pseudo code is also that it does not block the underlying Java thread while waiting to receive the response message.
The hallmark of AP  One thing can be identified in the pseudo code, in an effort to formalize what is sought after in an AP. The AP needs to halt the execution when requiring a response by calling the `receive` function.

Expressed in more general terms, the sought functionality of an AP is that the AP should halt the execution and in some way store the current execution state when some kind of `input` is required or expected. The execution of the AP should be continued when input is given at some later point in time. The following pseudo code tries to visualize what is sought after:

```scala
def asynchronousProcedure() {
    expression1
    input
    expression2
    input
    expression3
}
```

The pseudo code is synchronous looking, but the execution is halted in two places, before `expression2` and `expression3`, when input is expected.

Concrete technologies  I have found two different candidate technologies to use when implementing AP. One technology is an API in the Akka framework that is called `dataflow`. The dataflow API, do in its turn make use of the Scala support for `delimited continuations`. The other technology for realizing AP is to directly use the Scala support for delimited continuations, in a concept that in this thesis report will be called `thunk procedure`.

Realizing AP via thunk procedures require heavy use of the Scala support for delimited continuations. Therefore will section 4.5.1 give an introduction to continuations in Scala, while section 4.5.2 and 4.5.3 deals with dataflow and thunk procedures respectively.

The section about continuation is just a brief introduction to the delimited continuations used in Scala, in the context of implementing AP. I recommend the book *Scala for the Impatient* [6] for a pedagogical and thorough introduction on the subject of delimited continuations in Scala.

4.5.1 Continuations

“Continuations are a powerful construct that allows you to implement control flow mechanisms other than the familiar branches, loops, function calls, and exceptions. For example, you can jump back into a prior computation or reorder the execution of parts of a program. These capabilities can be mind-bending; they are not intended for application programmers, but library implementors can harness the power of continuations and make them
Continuation can be regarded as a language construct that can capture the runtime environment at a given point in execution to be continued at some later point in execution or time. In other words, continuations give the developer the ability to suspend and resume sequential code paths in a controlled way, and conceptually this is specifically what is searched for in order to realize AP.

In the case of Scala, a specific type of continuation is implemented, namely delimited continuations. As the name implies the delimited continuations are limited in range. In the Scala case, the delimited continuation is limited to the code block given to a function called reset.

Technical details of delimited continuations in Scala

The JVM does not have the ability to provide continuations. In the Scala case, this is circumvented by making a code transformation of the delimited continuation (everything inside the reset block) to something that is called continuation passing style ( ), according to Horstman [6, p. 325, 332].

Ryan Stansifer [13, p. 257] describes CPS as each call to a function implicitly takes the remainder of the program as an argument. Stansifer means that this enables the function to always to continue forward in its execution and never “return” to some previous state.

To be able to do the CPS transformation, the Scala compiler makes use of a compiler plugin that takes care of the code transformation. According to the Scala documentation [1], using the compiler option -P:continuations:enable enables this compiler plugin.

As mentioned in the Scala introduction in section 2.1, Scala is a statically type language but the type inference saves much of the hassle with writing out the types explicitly. However, types do some times need to be written out explicitly anyway. To facilitate the CPS transformation of the delimited continuations in Scala, more type information must be given than just the direct parameter type or return type. Therefore do many types in a delimited continuation need to be annotated with annotations like @cpsParam[B,C] or @cps[A], to specify all type information that the CPS transformation needs.

Delimited continuation examples in Scala

To in some way understand how the delimited continuations are used in thunk procedures in section 4.5.3, a small example on delimited continuations are given in listing 4.27.

The delimited continuations in Scala make use of two keywords (actually functions) - reset and shift. The code block given as argument to reset is the outer bounds of the delimited continuation. shift is always used
inside a reset block. The shift function takes a function that requires a specific function as argument. In the example listing 4.27, shift is given an anonymous function with parameter $k: \text{Int} \Rightarrow \text{Int}$. The parameter named $k$ is in its turn a function from Int to Int.

A key thing to identify here is what the argument function named $k$ actually is. The function $k$ is the rest of the code, from the end of the shift block to the end of the reset block. In this example, $k$ is a function that contains the expression on line 9, and could be defined as:

```
def k(i: Int) = i + 1
```

The resulting expression of the delimited continuation looks like the following expression:

```
val result = ((3 + 1) + 1) * 2
```

**Performance of delimited continuations in Scala**

Performance is of importance in the context of the thesis problem, with many instances of code running concurrently in multiple threads. Using delimited continuations in Scala do not need to affect the performance of Scala code negatively.

Research done by Rompf et al. [12], shows that CPS-transformed delimited continuations can be implemented on stock VMs in ways that are competitive in terms of performance. They conclude based on benchmarks that the high-level approach of, what they call, selective code transformation of delimited continuations to CPS “performs competitively” to ordinary Scala code without delimited continuations.
4.5.2 Dataflow and Futures

The use of the dataflow API in the Akka framework seems as a viable solution when realizing AP. The Akka documentation [14] states that Akka implements Oz-style dataflow concurrency by using a special API for Scala Futures that enables a complimentary way of writing synchronous-looking code that in reality is asynchronous. The dataflow API in Akka makes use of delimited continuations to be able to work.

Future is a trait representing a value that may not have been computed yet. A Promise is essentially the promise (or write-side) of a Future (read-side).

However, use of the dataflow API adds dependency to the Akka framework besides adding dependencies to the Scala continuation compiler plugin. Further, the close integration between dataflow and the use of the Future and Promise will probably in most use cases add an extra level of complexity to the implementation of AP.

4.5.3 Thunk procedure

The principle behind the thunk procedure is that the execution of a procedure can be stored away as a value, to be continued at a later time. The not yet executed part of the procedure is in this case returned to the code that called the procedure in the first place.

The returned value is in this report called thunk. The thunk is (or has - depending on implementation) a function that can be called at any later time to continue the execution form the earlier halted procedure execution state.

Consecutive calls to the AP via calls to the returned thunk may generate further thunks if the AP for some reason is halted further times.

A simple thunk procedure example

To show how thunk procedures can be implemented in Scala with the help of delimited continuations, a little example is put together with code listings 4.28, 4.29 and 4.30.

The concept of this example is to show how to implement AP requiring arbitrary number of inputs of type Int. The example is somewhat limited, because there is no (general) way to tell when the AP has reached its end.

Line 5 in listing 4.28 contains the recursive definition of the Thunk class. It contains a value named input and the value input is of type function with argument type Int and return type Thunk.

The class ProcedureRunner in listing 4.28 contains two methods - run and input. These two methods are the locations where the use of the delimited continuation blocks reset and shift are used. The run method is a wrapper for reset. The method run is the method to call when executing an AP. The
4.5. ASYNCHRONOUS PROCEDURES

Code listing 4.28: Simple AP example - thunk procedure implementation.

```scala
import util.continuations.reset
import util.continuations.shift
import util.continuations.cps

case class Thunk(input: Int => Thunk)

object ProcedureRunner {

  val ignoreInput: Thunk = Thunk { input =>
    println(" ignored: " + input)
    ignoreInput
  }

  def input = shift { k: (Int => Thunk) => Thunk(k) }

  def run(func: => Unit @cps[Thunk]) = reset {
    func
    ignoreInput
  }
}
```

last return value in the reset block is a value called ignoreInput, which is a recursive thunk handling any superfluous input.

The input method is a wrapper for shift and is called from within an AP when more input is required or expected.

Listing 4.29 shows a usage example of the AP implementation in listing 4.28. Lines 26 to 31 shows the actual AP block. Input is requires on lines 28-30. Input is given to the AP via the returned thunks on lines 35, 36 and 39.

Listing 4.30 shows the generated output in a test run of this simple example.

One thing to note is that the thunks can be rerun multiple times. As seen in the listing, input is given to thunkOne on both lines 35 and 39 in listing 4.29.

Another thing to note is that the output “procedure start” and “procedure second line” is only displayed once, despite that the first thunk (thunkOne) is run twice. The reason for this is that the first thunk is generated and returned from the input request made on line 28 in listing 4.29 and that the output text “procedure start” and “procedure second line” is printed on the lines before the thunk is generated - i.e. the AP lines 26 and 27 is not a part of the first thunk returned.

Further, the input values 23, 29, 53 and 59 are ignored, because in both cases the AP has finished before these inputs are given. Thereby are these inputs taken care of by the ignoreInput thunk used on line 18 in listing 4.28.
CHAPTER 4. INVESTIGATION OF POSSIBLE TECHNIQUES

Code listing 4.29: Simple AP example - usage in test program.

```scala
object ThunkProcedureSmallExample extends App {

  val thunkOne: Thunk = run {
    println("procedure start")
    println("procedure second line")
    println("value 1: " + input)
    println("value 2: " + input)
    println("value 3: " + input)
    println("procedure end")
  }

  println("\nFeeding with some input")
  val thunkTwo = thunkOne.input(13)
  thunkTwo input 17 input 19 input 23 input 29

  println("\nRestart procedure from first thunk")
  thunkOne input 41 input 43 input 47 input 53 input 59
}
```

Code listing 4.30: Simple AP example - output from test program.

```
procedure start
procedure second line

Feeding with some input
value 1: 13
value 2: 17
value 3: 19
procedure end
ignored: 23
ignored: 29

Restart procedure from first thunk
value 1: 41
value 2: 43
value 3: 47
procedure end
ignored: 53
ignored: 59
```

A advanced thunk procedure example

To illustrate a solution of the shortcomings of the simple example, a more advanced example is shown here. This, more advanced example, have a remedy to two problems in the simple example. First, the advanced example
4.5. ASYNCHRONOUS PROCEDURES

enables the possibility to be able to determine when the AP has reached its end. Second feature of the more advance example is the possibility to return a final return value from the AP.

The example makes use of a procedure called `isFirstFivePrimes` that tests if the first five input values (integers) given are the first five prime numbers in ascending order (i.e. 2, 3, 5, 7 and 11). If the first numbers are the prime numbers, `isFirstFivePrimes` should return the Boolean value `true` and in other cases it should return `false`.

**Code listing 4.31: Advanced AP example - thunk procedure implementation.**

```scala
import scala.util.continuations.cps
import scala.util.continuations.reset
import scala.util.continuations.shift

object ThunkProcedure {
  def runProcedure(proc: => Thunk @cps[Thunk]): Thunk =
  reset { proc }

  def input: Int @cps[Thunk] =
    shift { rest: (Int => Thunk) => InputThunk(rest) }

  def terminate(answer: Boolean): Nothing @cps[Thunk] =
    shift { rest: (Nothing => Thunk) => AnswerThunk(answer) }
}

sealed trait Thunk { def <<(input: Int): Thunk }

case class InputThunk(nextThunk: (Int => Thunk)) extends Thunk {
  def <<(input: Int) = {
    println("Input value " + input + " given.")
    nextThunk(input)
  }
}

case class AnswerThunk(answer: Boolean) extends Thunk {
  def <<(input: Int) = {
    println("Ignore input value " + input + ". ")
    println("Answer already given.")
    this
  }
}
```

The code listings that belong to the advances example are listings 4.31 (the thunk procedure implementation) and 4.32 (example AP application and execution of it). The output of the test program is shown in listing 4.33.

Listing 4.31 show the central parts of thunk procedures. It contains the procedure runner with methods to request `input` and to `terminate` an AP with a Boolean answer as the result of the procedure. It also contains the
CHAPTER 4. INVESTIGATION OF POSSIBLE TECHNIQUES

Code listing 4.32: Advanced AP example - usage of AP in a test program.

```scala
object PrimeProcedure {
    import ThunkProcedure

    def assertInput(expected: Int): Unit @cps[Thunk] = {
        println("Expecting value " + expected + ".")
        if (input != expected) terminate(false)
    }

    def isFirstFivePrimes: Thunk = runProcedure {
        assertInput(2)
        assertInput(3)
        assertInput(5)
        assertInput(7)
        assertInput(11)

        println("Procedure Success.")
        terminate(true)
    }
}

object PrintUtil {
    def printAnswer(thunk: Thunk) = thunk match {
        case AnswerThunk(answer) =>
            println("Answer is " + answer + ".")
        case InputThunk(_) =>
            println("No answer given yet − to few inputs.")
    }

    def printTestTile(title: String) =
        println("\n### Test " + title + " ###")
}

object PrimeTest extends App {
    import PrintUtil._
    import PrimeProcedure._

    val is3rdTo5thPrimes: Thunk = isFirstFivePrimes << 2 << 3

    printTestTile("1: First five primes are correct")
    printAnswer(is3rdTo5thPrimes << 5 << 7 << 11 << 98 << 99)

    printTestTile("2: Missing fifth prime")
    printAnswer(is3rdTo5thPrimes << 5 << 7)

    printTestTile("3: Fourth prime is wrong")
    printAnswer(is3rdTo5thPrimes << 5 << 77 << 11)
}
```

thunk definition. In this case a thunk can either be a request for more input or it can be an answer (i.e. the final return value) from an AP. In either
4.5. ASYNCHRONOUS PROCEDURES

Code listing 4.33: Advanced AP example - output from test program.

```plaintext
1 Expecting value 2.
2 Input value 2 given.
3 Expecting value 3.
4 Input value 3 given.
5 Expecting value 5.
6
7 # # Test 1: First five primes are correct # #
8 Input value 5 given.
9 Expecting value 7.
10 Input value 7 given.
11 Expecting value 11.
12 Input value 11 given.
13 Procedure Success.
14 Ignore input value 98. Answer already given.
15 Ignore input value 99. Answer already given.
16 Answer is true.
17
18 # # Test 2: Missing fifth prime # #
19 Input value 5 given.
20 Expecting value 7.
21 Input value 7 given.
22 Expecting value 11.
23 No answer given yet – to few inputs.
24
25 # # Test 3: Fourth prime is wrong # #
26 Input value 5 given.
27 Expecting value 7.
28 Input value 77 given.
29 Ignore input value 11. Answer already given.
30 Answer is false.
```

case - the thunk is generated from the procedure runner or returned as a result from an earlier thunk requiring more input.

Both types of thunks implement a method <<, that consumes a input. This can be compared with the left shift operator for `std::ostream` in the C++ programming language. The `InputThunk` uses the given input value to continue the execution of the AP, while the `AnswerThunk` ignores the given input.

Listing 4.32 shows how an AP can look like an how to run it. Just as in the simple example, a key thing to identify here is that the execution of `isFirstFivePrimes` is halted each time is reaches the call to method `input`. In the advanced example `input` is called at row 38 in listing 4.32. As seen at line 10 in listing 4.31, each time the method `input` is executed, the `shift` block stores the rest of the `isFirstFivePrimes` procedure not yet executed in the variable called `rest`. The variable `rest` is a variable of type function taking an `Int` and returning a `Thunk`. The variable `rest` is then stored in a new `InputThunk` instance that is returned to the piece of code that initially
executed the AP. The execution of `isFirstFivePrimes` is halted, because the variable `rest` (of type function) is not at this stage called with an integer as argument.

At line 69 in listing 4.32, the two first thunk of the call to AP `isFirstFivePrimes` is fed with input values 2 and 3. The third thunk is stored as value `is3rdTo5thPrimes`. As seen in output in listing 4.33 at line 5, the `is3rdTo5thPrimes` thunk is awaiting the input value 5 when starting the three different tests.

### 4.6 Investigation summary

This section contains some tables that tries to summarize the the technologies investigated in this chapter.

Table 4.7 shows a summary of the SM representation techniques mentioned. Table 4.8 shows a summary of the timeout techniques mentioned. The last table - table 4.9 shows a summary of the AP representation techniques mentioned in this chapter.

As the investigations show, there are multiple viable techniques to choose from when representing state machines and using delays in Scala.

---

**Table 4.7: SM representation technique summary, where I/D means ‘implementation dependent’**

<table>
<thead>
<tr>
<th>SM repr. technique</th>
<th>extra tool required</th>
<th>SM tailored syntax</th>
<th>potentially inefficient</th>
<th>Akka integrated</th>
<th>data driven logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested match/case</td>
<td></td>
<td>I/D</td>
<td>I/D</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Transition table</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State pattern</td>
<td>yes</td>
<td></td>
<td></td>
<td>I/D</td>
<td></td>
</tr>
<tr>
<td>SM compiler</td>
<td>yes</td>
<td>yes</td>
<td>I/D</td>
<td>I/D</td>
<td>I/D</td>
</tr>
<tr>
<td>Akka become</td>
<td>yes</td>
<td></td>
<td></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Akka FSM</td>
<td>yes</td>
<td></td>
<td></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Scala internal DSL</td>
<td>yes</td>
<td></td>
<td></td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

---

60
### 4.6. INVESTIGATION SUMMARY

#### Table 4.8: Delay technique summary.

<table>
<thead>
<tr>
<th>Delay technique</th>
<th>Acceptable performance</th>
<th>Akka integrated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Akka FSM timer (FSMT)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Akka FSM state timeout (FSMSTO)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Akka scheduler (AS)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Akka receive timeout (RTO)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>future timeout (FTO)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>global timer (GJT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>local timer (LJT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>future await result (FAR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread sleep (TS)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Table 4.9: AP representation technique summary.

<table>
<thead>
<tr>
<th>AP repr. technique</th>
<th>Based on delimited continuations</th>
<th>Akka integrated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thunk procedure</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Akka dataflow</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
Chapter 5

Specification of toy example

The purpose of this chapter is to define a protocol toy example specification that can act as a mannequin and guide for some of the implementation techniques investigated in this thesis work.

Sections 5.1 and 5.2 give an overview of the toy example and explains the real world context of the toy example. The detailed specification of the toy example is given in sections 5.3, 5.4 and 5.5.

5.1 Overview

This section gives an overview of the toy example protocol and the domain it operates in.

The toy example is defined as a protocol layer between two other protocol layers. Figure 5.1 shows a schematic overview of the layers and the entities involved in the NET protocol layer. The toy example centers on the user equipment (UE) side of the NET layer. A UE can for instance be a mobile phone or any other communication device that uses the mobile communication network.

The protocol layer above is called APPL and can be thought of as the application layer. Just as an example, one application of the application layer could be IP traffic.

The layer that the toy example focuses on is called NET and manages the connection between UE and the communication network core, by communication to a network entity that in the toy example is called CORE.

The protocol layer below is called RADIO and is a protocol layer that manages the radio communication link between the UE and some radio base station (RBS).
5.2 Relation to thesis context

As mentioned in the answer to question 2 in the initial investigation (see appendix A), there is a state machine for a protocol layer that is, as stated in the investigation, difficult to work with. That protocol is called NAS (Non Access Stratum), and is a specification given by the 3rd Generation Partnership Project (3GPP) on a part of the LTE standard. LTE is abbreviation for Long Term Evolution and is sometimes referred to as 4G (fourth generation) mobile communication standard.

The purpose of the toy example is to illustrate the specific features and idiosyncrasies of some of the real protocols that is the source for this thesis work. The toy example is therefore loosely based on some parts of the NAS specification and the operating environment of NAS.

However, the toy example is a simplification inspired by the NAS specification, but is not an actual real world example. This is why the content of the toy example and the names used in the toy example is somewhat separated from NAS or other real world examples.

---

Figure 5.1: NET overview, in context of different abstraction layers in the communication stack between entities.

Figure 5.2: NAS overview, in context of different abstraction layers in the communication stack between entities.

Figure 5.2 shows a schematic overview of the layers and the entities involved in the NAS protocol layer.

Compared to the toy example the RRC (Radio Resource Control) is
roughly the same as the RADIO layer and the NAS layer is roughly the same as the NET layer. Also, ENB (Evolved NodeB) and MME (Mobility Management Entity) is almost the same as the things called RBS and CORE. The only names and abbreviations use in the rest of the toy example definition belong to the toy example and are the layers APPL, NET and RADIO and the network entities UE and CORE.

5.3 NET states

Figure 5.3 shows the states of the UE side of the NET layer. Idle and Connected are the basic states while AttachOngoing and DetachOngoing denotes the current transition between the basic states and can therefore be regarded as short term transition states.

5.4 Interaction with UE NET

This section tries to describe all interactions and expected messaging between UE NET and related entities, with the help of sequence diagrams. Section 5.5 focuses on the internal structure, i.e. the behavior, of UE NET.

All interactions with the UE side NET layer are directly to the layers above and below the net layer in the UE, i.e. the APPL and RADIO layers respectively. Indirectly the UE side of the NET layer can communicate to CORE side of the NET layer, via messages to and from the RADIO layer within the UE. The radio layer delegates the messages further.
5.4.1 Interaction between APPL and NET

The sequence diagrams in figures 5.4 to 5.6 describe the interactions between the APPL layer and the NET layer.

![Diagram](image)

*Figure 5.4: The powerOn interaction between application layer and NET when requesting to power on.*

![Diagram](image)

*Figure 5.5: The powerOff interaction between application layer and NET when requesting to power off.*

![Diagram](image)

*Figure 5.6: The is detached interaction between application layer and NET when signaling that UE is detached from CORE.*

5.4.2 Interaction between RADIO and NET

The sequence diagrams in figures 5.7 to 5.10 describe the interactions between the RADIO layer and the NET layer.
CHAPTER 5. SPECIFICATION OF TOY EXAMPLE

Figure 5.7: The `connect` interaction between NET and RADIO when requesting to create a RADIO layer connection.

Figure 5.8: The `disconnect` interaction between NET and RADIO when requesting to disconnect a RADIO layer connection.

Figure 5.9: The `send` interaction between UE NET and RADIO when UE NET wants to send a payload $x$ to CORE NET. If RADIO successfully sends message further, the RADIO will respond with an acknowledgment to NET.
5.4. INTERACTION WITH UE NET

5.4.3 Interaction between UE and CORE side in NET layer.

The sequence diagrams in figures 5.11 to 5.14 describe the interactions between the UE and CORE side in NET layer.

Figure 5.10: The receive interaction between UE NET and RADIO when CORE NET sends a payload y to UE NET.

Figure 5.11: The forcedDetach interaction between application layer, UE NET and CORE NET when CORE NET forces a detach. The DETACH ACCEPT response message is sent before disconnection.
Figure 5.12: The attach interaction between UE NET and CORE NET when UE tries to attach to CORE. The CORE NET might want to authenticate the UE and therefore the authentication interaction might be applied after the initial attach request, depending on whether CORE NET responds with an attach accept or an authentication request.

Figure 5.13: The detach interaction between UE NET and CORE NET when UE tries to detach from CORE.
5.4. INTERACTION WITH UE NET

Figure 5.14: The authentication interaction between UE NET and CORE NET when UE NET tries to authenticate itself towards CORE NET. Depending on some variable the UE NET might not send an authentication response to CORE NET, but instead send an authentication failure.
5.5 Procedures in UE NET

The purpose of the procedures in this section is to tie together the fragmented sequence diagrams in section 5.4. The sequence diagrams in section 5.4 needs to be put into a larger context that describes the complete behavior of UE NET. Figure 5.3 together with the procedures in this section does just that.

Some criteria put on the procedures:

- The procedures in this section directly correspond to the entities with the same names in the interaction in the sequence diagrams and actions in the state transition diagram 5.3.

- The action/interaction to make when a procedure returns a failure or success is given by the corresponding interaction sequence diagram. For some procedures, interaction is expected to occur based on the return value, but for some it is not expected. This should be obvious when comparing sequence diagrams with procedures.

  The difference comes from the fact that some methods are called as sub-methods while other are called as topmost procedures in figure 5.3. This can be regarded as an artifact from the split in representation between a state transition diagram and a procedure representation.

- When a procedure fails, everything should revert to the previous state. However, this does not apply to the failure of detach or disconnect procedures, that should end up in an idle (disconnected) state when failing.

5.5.1 A procedure notation

To be able to describe the procedures in the UE NET, a simple procedure notation (a kind of pseudo language) is used. The point of this procedure notation is to be abstract and minimalistic, so the intent of the procedures can be described, but at the same time do not rely on any particular language features of Scala or any other programming language. This section focuses on defining the procedure notation. The actual representation in Scala is neither given nor of importance at this stage.

Conventions

Here follows some conventions for the procedure representation:

- A block of statements are denoted by being indented at the same indentation level.

- S and F represents the success resp. failure of a procedure.
• If a procedure calls a sub-procedure and the sub-procedure returns $F$, the calling procedure immediately also returns $F$ (this can be compared to exception in some languages).

• If no of the statements in a procedure returns $F$ the procedure will return $S$. In other words, the return of $S$ is implicit.

**Keywords**

Six procedure notation keywords are defined for the procedure notation:

- **if** An *if*-statement. Gives the ability to execute conditional blocks of statements, given a boolean expression.

- **else** The else part of an *if*-statement.

- **return** Makes a procedure return a specific value.

- **whenfail** Sets a statement (or blocks of statements) to execute if the current procedure fails from this point forward. Compared to Java or Scala, this can be regarded as a tear-down mechanism in the *catch* part of a *try-catch* block around something that might throw an exception.

- **nondet** To simplify the abstract procedure notation, some parts are wrapped in a *nondet* block signaling that this block is non-deterministic and only executed if the first statement can be executed successfully.

  If the first statement cannot be executed successfully, not even the first statement is executed. The *nondet* block is in this case regarded as an an success.

  If the first statement in the *nondet* block is executed successfully, a failure in a subsequent statement in the *nondet* block is to be regarded as a failure for the entire *nondet* block. This is analogous to the convention that a failure of a statement in a procedure block at the same time is regarded as a failure of the entire procedure.

**5.5.2 Primitive procedures**

These procedures are regarded as being primitive, and in some sense their implementation is not of importance, because they can be regarded as the basic building blocks of procedures. In a more practical perspective, they can be regarded as abstract functions or interface methods, where the implementation details are not of interest to the procedures that use them.

- **detachComplete** Signal to the application layer that the UE is detached from CORE.

- **connect** Create a RADIO-layer connection.
CHAPTER 5. SPECIFICATION OF TOY EXAMPLE

disconnect  Disconnect the currently existing RADIO-layer connection.

send    Send a message to CORE NET over existing RADIO-layer connection.

receive  Wait for a specific message from CORE NET over existing RADIO-layer connection. This procedure will fail if given timeout is exceeded.

5.5.3 Composite procedures

These composite procedures are built up of other statements given the primitive procedures, other composite procedures and the procedure notation.

powerOn  When application layer wants to power on, this procedure makes sure that the underlying RADIO- and NET-layers are properly set up.

  connect
  whenfail disconnect
  attach

powerOff Tear-down of RADIO- and NET-layers when application layer wants to power off.

  detach
  disconnect

attach Attaches the UE NET to the CORE NET. The CORE NET might also request the UE to authenticate itself, as a part of the attach procedure.

  send(ATTACH_REQUEST)
  nondet receive(ATTACH_FAIL, timeout)
      return F
  authentication
  receive(ATTACH_ACCEPT,timeout)
  send(ATTACH_COMPLETE)

detach Detached the UE NET from the CORE NET.

  send(DETAIL_REQUEST)
  receive(DETAIL_COMPLETE,timeout)

authentication This method also have access to a value shouldAuthenticate, that denotes whether the UE NET will try to authenticate itself towards the CORE NET or not.
5.6 A larger example

Figure 5.15 shows a large example to show that the specifications in sections 5.3, 5.4 and 5.5 results in.

The sequence diagram shows the all interaction that might take place when the application layer wants to power on and all interactions are completed successful. In the Idle protocol state, a POWER ON message is sent to the NET layer from the APPL layer. The NET layer makes all necessary interaction with the RADIO layer stated in the powerOn procedure before returning a POWER ON DONE message to the APPL layer and finally ending up in the Connected state.
Figure 5.15: Large message interaction example, with successful `powerOn` procedure complete with all sub-procedures including authentication.
Chapter 6

Implementation of toy example

This chapter starts with an explanation of the design choices made in the implementation. After that, some features of the implementation will be highlighted in the implementation presentation. The chapter ends with an evaluation of the implementation, based on code review with and questions to the domain expert.

6.1 Technical choices for implementation

There are some choices to make in the implementation of the toy example. The technical investigations in chapter 4 give multiple option for both the representation of the protocol state machine and how to trigger protocol timeouts. Only one way to implement AP is thoroughly investigated.

6.1.1 Mapping between specification and implementation

To minimize the differences between the toy example specification and the implementation, the implementation is divided up in the same way the specification is in chapter 5.

The implementation is built up with a 4-state state machine using some procedures in the transition between the states in the state machine. The state machine is based on figure 5.3 on page 64.

The toy example procedures specified in section 5.5 are implemented as AP with help of the thunk procedure solution illustrated in section 4.5.3 on page 54.
6.1.2 State machine

The state machine implementation is realized with the help of the functionality Akka FSM, shown in section 4.2.2 on page 32.

6.1.3 Asynchronous Procedure

As already mentioned, the thunk procedure solution using Scala’s capability to use delimited continuations is used when implementing the procedures in the protocol specification as AP.

As given in the specification, the protocol procedures are specified to return a value denoting either success or failure of the procedure. In the implementation, this is denoted with the Scala values true and false respectively.

The implementation of the AP functionality, using thunk procedures, is in a way completely independent to the implementation of the toy example protocol. Therefore the implementation of the AP functionality is made as a separate and generic implementation that has no ties to the toy example. This is reflected in the implementation code listings appendix in this report, where the AP implementation in section D.1 is separated from the protocol definition and test code in sections D.2 and D.3.

The implementation of the toy example however, makes extensive use of the AP functionality.

6.1.4 Timeout

The protocol timeouts in the specification is realized with the use of FSM-STO (Akka FSM state timeout), explained in section 4.4.6 on page 43. This suits well with the use of Akka FSM in the state machine implementation.

An advantage with the use of FSMSTO is that there is no need to manually cancel the timeout if a message is received within the timeout duration.

Another advantage with the use of FSMSTO is that the timeout is signaled to the FSM-using actor via a message named StateTimeout. This enables the ability to trigger the timeout explicitly by sending the StateTimeout message directly to the targeted actor. This comes in extra handy when doing tests on the FSM-actor state machine. The alternative would be that the test code waits the entire timeout duration in order to trigger the timeout. The waiting alternative can make test suits with many or long timeouts run really slow.

6.1.5 Test framework

A test framework is used in order to make unit tests for a individual component in the implementation. The unit tests helps to verify the functionality of the individual components in the implementation.
In this implementation the test framework *ScalaTest*\(^1\) is used. ScalaTest is one of the two commonly referred test frameworks specifically written for Scala and not just Java. The alternative test framework would in this case be *Specs2*\(^2\).

### 6.1.6 Mock framework

To be able to make unit tests for a individual component in the implementation, the component needs to be isolated form the rest of the code, by using mock objects or stub objects as replacements for the other parts of the implementation needed in the tested component.

Another advantage of the mock objects is that they usually can be instrumented to verify that e.g. a method is called three times with specific arguments. This instrumentation is an test aid, handy when making e.g. unit test verifications of some component.

There exist a lot of mock frameworks that helps with the construction of these mock objects. Most often, the differences between different mock frameworks comes down to personal preferences. For this implementation a mock framework called *Mockito*\(^3\) is used.

### 6.2 Implementation result

The entire implementation is appended in appendix D starting on page 101. This section highlights some implementation details.

#### 6.2.1 Code structure

As seen in appendix D, the implementation code is divided up into three major parts. These parts are:

- The generic AP implementation using the thunk procedure technique. It resides in appendix D.1.
- The code implementing the toy example specification. It resides in appendix D.2.
- The test code that tries to verify that the toy example implementation behaves correctly. It resides in appendix D.3.

The code in the tree different parts is further divided up in Scala packages based on functionality. The Scala packages correspond to folders in the code listings in the appendix.

---


6.2.2 Asynchronous procedure implementation

*Code listing 6.1: Toy example thunk definition in implementation file thunkprocedure/Procedure.scala.*

```scala
/** Recursive definition of continuation wrapper. */
case class Input[In, Out](proceed: In => Thunk[In, Out])

/**
 * A thunk is either a procedure part waiting for input or
 * a result value.
 */
type Thunk[In, Out] = Either[Input[In, Out], Out]
```

The entire AP functionality consists of one file for the basic functionality and one extra file to enable some syntax sugar for thunks. The central parts are the thunk definition shown in listing 6.1. The thunk definition shows that a thunk literally is either a request for input or a output result.

*Code listing 6.2: Toy example procedure runner method definitions in implementation file thunkprocedure/Procedure.scala.*

```scala
type T = Thunk[In, Out]

/** Require input. I.e. pause, return thunk and wait until
 * input is given to thunk.
 */
def input: In @cps[T]

/** Runs the procedure given as parameter. */
def apply(procedure: => Out @cps[T]): T

/** Called when terminating a procedure from within. */
def terminate(result: Out): Nothing @cps[T]

/** Runs a sub-procedure from within another procedure. */
def runSubProcedure(procedure: => Out @cps[T]): Out @cps[T]
```

The thunk procedure method definition shown in listing 6.2, where `apply` is called when starting an AP and the three methods `input`, `terminate` and `runSubProcedure` are all called from within a running AP. These are the only methods found to be needed when using AP to implement all features the toy example protocol procedures.
6.3. EVALUATION OF IMPLEMENTATION

6.2.3 Toy example state machine part

The toy example state machine part is located in the protocol.fsm package. The protocol state machine is located in class NetFsm in listing D.5.

6.2.4 Toy example asynchronous procedures part

The asynchronous procedures needed in the toy example are divided up in several Scala classes in an attempt to make them easier to test. The AP implementations are divided in a way where all sub-procedures used in a AP resides in another class.

This division in combination with mocking the behavior of sub-procedure classes with the Mockito mock framework enables easy testing of all procedures individually, without interference from sub-procedure implementations.

6.2.5 Tests of toy example

The testing of the implementation focuses on two major types of testing. The first type of testing is, as already mentioned, unit testing of each procedure class individually. These test specifications reside in the protocol.procedure package.

The other type of testing is protocol message driven testing of the protocol state machine. These tests make use of the Akka TestKit testing utility and these tests reside in the protocol.fsm package.

Mock framework deficiencies

In the development and testing of the protocol implementation, some test tool deficiencies were discovered. The Mockito mock framework is unable to handle the CPS type annotations used for the delimited continuations in Scala. Delimited continuations are in this case used for the realization of AP.

The code in the protocol.procedure.mockable package is specifically developed to tackle the problem to mock methods with parameter type or return type with CPS type annotations.

6.3 Evaluation of implementation

The implementation shows that the protocol specification can be implemented with Akka FSM and AP realized with Scala’s delimited continuations.

Being the first implementation attempt of the toy example, the implementation can be regarded as a first try or first implementation iteration. There is most often room for improvements in any implementation and the current implementation can be seen as a base for further improvements. An
evaluation of the implementation helps to highlight problems and solutions in the implementation.

The implementation evaluation in this section is heavily based on notes from a code review evaluation session and evaluation questions and answers in written correspondence. Both evaluations are directed to the thesis domain expert. The data from these two evaluations are appended in appendix B and C.

6.3.1 Technical details

The code review in the evaluation session focuses on technical details in the implementation. A summary of the notes from the evaluation session is given in this section.

Naming conventions

In the code review session, some of the comments involve the naming of things in the code. Some names in the implementation do not follow naming conventions or have the capability to mislead developer reading the code. This is obviously something that can be improved. The naming of things do not have so much to do with the technical aspects of the implementation, but can have an impact on how easy the code is to develop and maintain. Some mentioned naming problems:

- The use of the term cut (class under test) instead of e.g. fsm in some tests.
- The use of the prefix nondet in some places. It could be replaced with another prefix like e.g. conditional or if.

Interface between SM and AP

There are some improvements to make in the interface boundary between the two types of representation in the implementation - the state machine and asynchronous procedure representations. Two objections are:

- As seen in the adapter class protocol.fsm.Procedures, there are a little bit too much logic in between the SM and AP parts of the implementation. The logic works, but it might be regarded as good code hygiene to move the logic found on line 46 to 51 in listing D.7 to the class FsmProceduresImpl.
- The use of the thunk procedures realization of AP requires the state machine in the class NetFsm in listing D.5 to use the method attachBehavior already in the IDLE state. The same objection works for using detachBehavior in the CONNECTED state.
The reason for this is an artifact from the fact that a thunk procedure can immediately generate a thunk containing a final answer on the initial call to the thunk procedure.

Testing

One opinion in the evaluation session is that some parts of the testing looks really nice while other parts are hard to read. Generally, the code in the integration tests in class `protocol.fsm.IntegrationTest` in listing D.18 looks nice while procedure test specifications such as class `protocol.procedure.CoreProcedureSpec` in listing D.21 are hard to read.

The difference between these two types of tests is the difference in abstraction level of what they test. The integration tests execute tests on the message level between different protocol layers, while the procedure test specifications focus on the interaction between thunk procedures. There might be more message centric ways to test the asynchronous procedures that can improve readability of the procedure tests.

Parts of the implementation that stand out as good examples

One of the reasons for the implementation is to just show that an implementation can be done and how it can be done. In addition to this some parts of the implementation are mentioned as good implementation examples in the evaluation session:

- The state machine in the `NetFsm` class in listing D.5 is judged to be compact and easy understood.

- Another example being easy to read and understand is the integration tests in the test specification class `IntegrationTest` in listing D.18.

- The use of the AP representation is a big step forward compared to both the result from the UniMod tool and alternatively making use of callback methods in Java to accommodate asynchronous calls.

6.3.2 Implementation validity and applicability

This section gives a summary of a set of questions and answers that to some extent examines the validity and applicability of a implementation similar to the concrete implementation of the toy example.

The questions used try to put an implementation similar to the toy example implementation in relation to the type of implementations used today at the Xdin. Despite the concrete toy example implementation, the questions are somewhat hypothetical in nature because it makes the comparison

---

4This comment is not based on the implementation evaluations in appendix B and C, but instead received 8 April 2013 from domain expert after a thesis report review.
between the representation techniques used in the single instance of toy example implementation and some distant real-world representation practices used today at Xdin.

In the rest of this section, the techniques used in the toy example implementation will be abbreviated $TE$ and while the real-world techniques used at Xdin involving Java and Unimod will be abbreviated $RW$.

**Differences**

The largest difference between TE and RW is that TE allows that non-blocking I/O can use a representation that looks like sequential blocking I/O in Java.

**Gains**

The possible gains with using TE is that the code and tests gets more easy to read and in the extension cheaper to maintain and to develop.

**Problems**

The largest obstacle with using TE is that the migration from RW might need considerable time. Also, it is not obvious how to in small steps partially migrate from RW to TE.

**Missing aspects**

The toy example specification is designed to cover most special cases existing in RW implementations. There is a high probability that some special cases are missed, but they might not show them self until a real world implementation of TE are tried.

**Conclusions**

The starting point for this thesis is the hypothesis that using the AP concept together with the alleged advantages of Scala over Java should give an development edge over current RW techniques. The ability to realize AP in Scala is something that speaks strongly for the use of TE.

However, without a closer and more extensive comparison between TE and RW, no definite conclusion can be made.
Chapter 7

Discussion

This chapter discusses the results and findings in this thesis report and also tries to mention some of the alternative solutions not used in the implementation. Further, it mentions some of the things out of scope of the thesis, but might be interesting to look more into. Finally, the chapter tries to capture some thoughts on the goals and methodology in the thesis.

7.1 Generalization and applicability

The implementation evaluation has not given any indications that the techniques used in the toy example implementation are not generalizable and applicable to other similar protocol implementations.

The following sections make some more detailed elaborations on the generalization and applicability on other protocols similar to the toy example.

7.1.1 The asynchronous procedure implementation

As mentioned earlier, the AP part of the toy example implementation resides in appendix D.1. This part of the toy example implementation can be seen as very generalized solution. Both the AP input type and output type are generic and therefore not tied to any particular usage area.

In general the AP implementation should be able to use in, if not in most, but at least some places where inversion of control is needed, as the term is described in section 2.3.1.

The changes proposed to deal with the missed timeout functionality in section 7.1.3 would make the AP implementation even more general. Then the AP can double as both a generator and a consumer of data via input\textsuperscript{1} thunks.

\textsuperscript{1}In that case not only used for input
7.1.2 Applicability on protocols similar to toy example

The principles and design choices used in the toy example protocol implementation should be able to handle similar types of protocols.

Something to be aware of is that the procedures for the Net protocol layer is only specified to have two different return values - true and false, meaning success and failure. This can be a limiting factor that can create problems for some types of protocol procedure representations.

The first thing to note is that the Net protocol layer procedures are an abstraction (specialization) on top of the generic AP implementation. The second thing to note is that to simplify the implementation, multiple types of failure are grouped into the Net procedure return value false. These types of failures are

- Received messages that is interpreted as failure messages.
- Timeouts for not received, but expected messages.
- Explicitly triggered failures in Net procedures.

If the two procedure return values - success and failure - do not suite a specific protocol representation, the principles and design choices used in the toy example protocol implementation may not be directly applicable.

7.1.3 Improved timeouts

One thing that to some extent is overlooked and missed in the toy example is the usage and consequence of message timeouts. As it is now, the specification does not give an explicit and detailed specification on the usage and behavior of timeouts when awaiting response (input) in a procedure. The toy example implementation do somewhat miss to deal with timeouts in accordance to the bits of information given in the specification. As it is now, message timeouts are dealt with in the state machine and not delegated to the asynchronous procedure thunks. This results in that the procedures lose the ability to make a proper error handling when a timeout occurs.

Pieces of the current implementation can be seen in listings 6.1 and 6.2 on page 78 and the complete file can be seen in listing D.1 on page 101.

This does not invalidate the toy example implementation and the techniques used in it. The addition of proper handling of input timeouts in the thunk procedure implementation can be seen as an extension or generalization of the current implementation. A redefinition of a thunk input request to enable passing a timeout duration argument, could then look like

```scala
case class Input[InArg, In, Out](
  arg: InArg, proceed: In => Thunk[InArg, In, Out])
```

compared to the definition given in listing 6.1

```scala
case class Input[In, Out](proceed: In => Thunk[In, Out])
```
Put into the toy example context, the type argument \texttt{InArg} could be specified to class \texttt{scala.concurrent.duration.DurationLong} or something like that, while \texttt{In} would be bound to a concrete type that represents either a valid input argument or a timeout. A type that fit that interpretation could be the class \texttt{Option[Message]}, instead as the currently used \texttt{Message} class. Obviously modifications also need to be made at other places too, to enable to handle timeouts in the toy example procedures.

7.2 Other solutions

This section tries to give some perspective on the toy example implementation technique used and alternative techniques.

7.2.1 Other implementation choices in Scala and Akka

The investigations in chapter 4 shows that there are multiple techniques that can be used when implementing the toy example and similar protocols. Multiple state machine implementations and multiple ways to trigger timeouts are shown to be valid choices, depending on context.

The toy example implementation makes some specific choices for techniques to use, but other choices could be made instead. The specific techniques used only show one way to implement the toy example. The main purpose of the toy example is to demonstrate that an implementation actually can be done in accordance to the thesis purpose and goals and how that might look like.

This report avoids to make any general conclusion on which of the investigated state machine implementation techniques that are most suitable when representing protocols similar to the toy example protocol, in Scala and Akka.

Structures in perspective of multiple protocol layers

In the toy example implementation, the protocol state machine is tightly connected directly with an actor via the Akka \texttt{FSM} functionality. This can be put in contrast to state machine representations not tied to an actor, such as using a transition table.

In a larger perspective, one could choose to group multiple protocol layer implementations together inside a single actor. The advantage or disadvantage with using an actor per protocol layer versus using one actor per UE is neither investigated nor indicated to make a difference given the information gathered in this thesis.
7.2.2 Java and Akka

As earlier mentioned in the comparison between Scala and Java in section 2.1.3 on page 10, Akka has support for direct use in Java. This enables Java implementations to take advantage of at least the concurrency and scalability features of Akka. However, the referred section also mentions that Java do not have native support for continuations, but it can be supported in Java via third party byte code processing in Apache Commons javaflow.

Some of the other advantages of Scala over Java will however be lost such as generally more compact and productive syntax and specifically type inference, pattern matching, functional programming and the reduced risk for null pointer problems.

7.2.3 Other JVM languages

There exist a lot of languages for the JVM today. According to Kullberg [7], Scala where the only statically type JVM language besides Java up to year 2010, when Groovy++ emerged. At the time of writing this report, a quick search for “statically typed JVM language” on Google results in languages such as Xtend, Yeti, Kotlin, Fusio, Fantom, Ceylon and Talc. A Dr. Dobb’s article A Long Look at JVM Languages written by Eric Bruno, November 2012, gives an introduction to some of them.

There are also some dynamically typed JVM languages such as Jython, Clojure and Rhino, but there are some drawbacks with using dynamically typed languages, as earlier mentioned in section 2.1.3. However, Kullberg [7] states that Clojure is a functional language suitable for parallel programming.

In perspective of these other languages, Scala is probably at least good enough. Some of the reasons to use Scala in comparison with other JVM languages are delimited continuations, static typing with type inference, relative large adoption rate³, commercial support⁴, close integration with Java and performance comparable to Java.

7.3 Ideas for further investigation

Some questions and ideas have been raised during the investigation and implementation parts of this thesis. Some of these ideas are possibly of interest for further investigation and are summarized here.

7.3.1 Remove transitions states in state machine

Is it possible and does it give any advantages to remove the explicit definition of transition states in state machine part of the implementation?

³Not compared to Java.
7.3. IDEAS FOR FURTHER INVESTIGATION

One example is the toy example state machine implementation in class NetFsm, in listing D.5 on page 106. As it is right now in NetFsm, the transition states ATTACH and DETACH does not attribute much to the current state of the protocol when the state machine is in transition in between the stable states IDLE and CONNECTED. The behavior of the transition states ATTACH and DETACH are almost equal to stable states IDLE and CONNECTED respectively. Further, in transition between IDLE and CONNECTED, the current state of the protocol are kept in the returned AP thunk passed around as optional data in the state machine. There exists multiple ways to solve this.

One idea would to use the existing stable states or a single transition state for all transitions between stable states. The behavior to execute when an AP has ended (i.e. returned a final value) can be sent as data in a tuple with the already sent AP thunk. In the toy example, this could be realized by letting NetFsm extend from

\[
\text{FSM}[\text{NetState}, \text{Option}[\text{(NetThunk, Behavior)}]]
\]

instead of as currently implemented, inherit from

\[
\text{FSM}[\text{NetState}, \text{Option}[\text{NetThunk}]]
\]

Another idea is that the removal of the transition states is something that relates to the boundary between the SM and AP parts of the protocol representation. If it turns out to be convenient to use the transition state, they could perhaps be implicitly defined by some type of SM-AP boundary abstraction.

7.3.2 Complete protocol as an asynchronous procedure

Could the entire protocol be represented as an AP, instead of splitting the representation in a SM part and an AP part? This might have some developer advantage, reducing the number techniques used when representing a protocol and eliminating the border between SM and AP.

One technical detail relevant to this is how the CPS delimited continuations in Scala handles looping and/or tail recursion. Are there potential problems related to this?

7.3.3 Improve testing of asynchronous procedures

How can the testing of AP implementations be improved? In the toy example implementation evaluation in section 6.3, this stands out as an area with room for improvement.

One problem is that the test specifications are hard to read. Should the tests be centered more around message passing instead of procedure calling?

Another problem is that the Mockito mock framework have problem with the Scala type annotation related to the use of delimited continuations. Will the Scala type annotation problem be fixed in Mockito or are there
other mock frameworks more suitable for the task? Can the test code be constructed in another way that circumvents the problem in a better way that it is implemented in the toy example?

7.3.4 Remote actors for even greater numbers of simulated entities

The numbers of simulated entities is not something this thesis has put direct focus on. Not much information is given on the real world application context and the preconditions given in the initial investigation in section 3.

However, the use of the Akka framework gives great capabilities for both concurrency in a single JVM and distribution between multiple JVM on multiple computer nodes in a network. Besides concurrently simulate many UE in one JVM, this gives at least the possibility to also distribute the simulated entities on multiple computers.

7.4 Thesis relevance and changing technologies

This section tries to give some perspective on the relevance of the thesis work.

Scala, Akka and other software are moving targets. During the thesis work period, updates have been made many of the software tools used in the thesis. It seems that the development of Scala and the ecosystem around Scala is quite active at the time of writing. Some of the software that have had at least one new releases during the compilation of the thesis are Scala, Akka, ScalaTest and the Scala IDE plugin used in the Eclipse IDE.

New functionality is added and old functionality is removed in these updates. Some of the solutions found in this thesis might become obsolete in the future.
Chapter 8
Conclusions

The conclusions in this chapter will focus three specific areas besides conclusions related to the thesis goals.

As set out in the thesis goals, the thesis shows that it definitely is feasible to implement asynchronous protocols by using a state machine for a few stable states and using a procedure representation that hides away the asynchronous details of the protocol in between the stable states, without being blocking. In other words - Implementation of asynchronous protocol can be done in Scala and Akka using a combination of SM and AP.

One way to implement this is shown with the help of an implementation based on the toy example specification.

8.1 Representation of state machines
Multiple ways to represent state machines in Scala exists and have been shown. Some generic ways of representing state machines are shown in section 4.1 and some Akka and Scala specific representations are shown in sections 4.2 and 4.3.

8.2 Asynchronous procedure
This thesis shows that AP can be realized with use of Scala delimited continuations.

8.2.1 AP usefulness
Based on the evaluation of the implementation, no definite claims can be made on whether the AP representation is better and easier to work with in the perspective of the developer. The opinion of the domain expert is although that this is so.
In some sense, the usefulness of the AP representation is relative. It can be really handy in a new green field project, but can contain large investment costs for when transitioning from other technologies, as mentioned in the implementation evaluation questions in appendix C.

The internal DSL section 4.3.2 and the syntax sugar class ThunkWrapper in listing D.2 shows that there are features in Scala that help to shape the syntax to suite the purpose. With the help of this flexibility in Scala, the use of AP can be tailored to suite different application areas and the conceptual models used by the developer. The attach methods on line 37-43 in listing D.9 shows how the basic AP implementation can be adapted to something that looks very much similar to the actual attach method specification pseudo code in section 5.5.3.

8.2.2 AP performance

Based on the findings regarding the performance of the CPS-transformed Scala code in section 4.5.1, no significant computational performance penalty is added to using delimited continuations in Scala. Therefore, there should not be any significant performance difference associated with the use of AP in an protocol representation instead of using non-CPS-transformed code such as state machines or callback functions.

8.3 Akka

Based on claims made by others, Akka is clearly a much better alternative than thread for enabling the capability to run many (thousands) of concurrently asynchronous protocols with the purpose to simulate user equipment.

Further, based on the Akka capability to run on remote systems, it should be able to simulate systems that are out of performance scope of a single JVM computer node.

8.4 Results from delay test bench

Scheduling and delay techniques are of importance when dealing with asynchronous communication that need to contain mechanisms to trigger communication timeouts.

As seen in table 4.6 on page 49 there are four good ways to use scheduler or timeout delay, related to the Akka library. These are Akka FSM timer (FSMT), Akka FSM state timeout (FSMSTO), Akka scheduler (AS) and Akka receive timeout (RTO).

Using a global Java timer (GJT) is the best choice when using a delay technique not related to Akka,

The types of delay technique to avoid are the use of a local Java timer (LJT), future await result (FAR) and thread sleep (TS). As shown in table
4.3 TS is the absolute worst technique to use.

8.5 Conclusions from implementation evaluation

The evaluation of the toy example implementation shows that there are some implementation and testing details to improve to be easier to work with as a developer.

However, the implementation is a concrete implementation example that demonstrates the techniques searched for in this thesis. The implementation works as a solid foundation for further advancements and other implementations.
Appendix A

Questions and answers in initial investigation

This appendix contains some questions and answers to these questions. The purpose of the questions is to make an initial investigation of, and to get more knowledge of the domain for the thesis work.

The questions are directed to people with domain knowledge, at the software consulting company (Xdin). In practice the thesis supervisor at the company (Mattias Evensson) is one of the few persons with good knowledge of the thesis context, and is therefore the single person answering the questions.

Some questions did not get an answer and are therefore removed from the appendix. Because the questions are formulated with only superficial knowledge of the domain, it is probable that some of the questions without answer were irrelevant or too hard to answer.

As seen in the questions and answers some information where already given at an earlier introduction to the thesis problem, but are implicitly confirmed by being included in the questions. Such information is the notion of the events OK, TIMEOUT and FAIL in these protocols.

Both the questions and answer are presented as is, in the original language (Swedish). Questions are in bold text with the original question numbering.

A.1 Questions and answers in Swedish

Fråga 2

En del i att hitta en lösning är att veta vad lösningen ska vara en lösning till. Sen tidigare har jag information om att dessa tillståndsmaskiner har vissa särdrag. De består av ett få antal grundtillstånd. Utöver grundtillstånden, så finns en mängd
”övergångstillstånd” för att överbrygga dessa grundtillstånd. Många av övergångstillstånden hanterar asynkron kommunikation. Generellt så kan ett asynkront anrop resultera i händelserna OK, TIMEOUT och FAIL. Finns det mera särdrag? Delar många tillstånd samma ”event” eller samma ”action”? Finns det många förgreningar, symmetrier eller långa procedurer av tillstånd i tillståndsmaskinerna?

Den tillståndsmaskin vi har mest problem med är NAS. NAS specen går att ladda ner från: http://www.3gpp.org/ftp/Specs/archive/24_series/24.301/24301-9a0.zip

På sidan 43 finns figur 5.1.3.2.2.7.1 som är en grov skiss på en tillståndsmaskin för NAS. I figuren kan man se de fasta tillstånden EMM-DEREGISTERED och EMM-REGISTERED samt procedurer för att gå mellan dem, ATTACH, SR (Service Request), TAU (Tracking Area Update) och DETACH.

Så där har du huvudsärdragen: några få stabila tillstånd och procedurer med asynkrona meddelanden för att gå mellan dem. Om man går in på mer detaljer så finns det en del till särdrag:

- De stabila tillstånden kan delas in i undertillstånd, se tex kapitel 5.1.3.2.3 och 5.1.3.2.4.
- Procedurerna har en massa felfall som ska hanteras på olika sätt, se tex 5.5.1.2.5 och 5.5.1.2.6 som beskriver felfall för attach.
- Under en procedur kan det startas subprocedurer, exempelvis så kan en autentiseringsprocedur startas under attach, service request och tracking area update.

Fråga 3

Finns det några typlösningar eller koncept som är extra intressanta att titta på? (t.ex. tekniska möjligheter i Scala, mer allmänna utvecklingskoncept, mentala tankeabstraktioner med mera)

Akka och continuations tror jag mycket på. Min förhoppning är att det går att beskriva de fasta tillsånden plus eventuellt ett extra tillstånd för varje övergång/procedur med Akka’s FSM ramverk och att det med hjälp av tex continuations går att beskriva procedurerna sekventiellt, ex på sekventiell procedur i pseudokod:

```scala
def myProcedure() {
  try {
    send(new SetupLowerLayerReq());
    resp = receive();
    if (resp.result == FAIL)
      throw new FailureException();
    send(new MessageReq);
    resp = receive();
    if (resp.result == FAIL)
  }
```

93
Fråga 5
Finns det några typer av tekniska begränsningar att ta i beaktning? (t.ex. multi-/singelträdning, prestanda, kompatibilitet med andra)


Fråga 6
Finns det något behov av att grafiskt kunna visualisera tillståndsmaskinerna? (t.ex. till dokumentation)

Det är inte jätteviktigt. Som jag nämnt ovan så finns det figurer i specarna, förhoppningen är att implementationen ska bli så lik/bra att det går att använda defigurerna för visualisering.

Fråga 7
Om det finns problem med nuvarande (grafiska) lösning, vad är dessa problem och är dessa relaterade till specifikt program/verktyg eller till grafiska verktyg som generellt koncept?

Både och, Unimod har en hel del tillkortkommanden som det antagligen finns andra verktyg som inte har. Tillståndsexplosionen som följer av procedurerna har jag dock svårt att se att något grafiskt verktyg kan råda bot på. Därmed inte sagt att inte ett grafiskt verktyg skulle kunna fungera bra.

Fråga 8
Tidigare har nämnts att det bör vara lätt att ändra, läsa och testa tillståndsmaskinerna. Finns det även andra saker som är bra att fokusera på?

Prestanda och skalbarhet är viktigt, men jag tror inte att de ska vara något större problem. Tack vare att vi har så många UE:ar som är oberoende av varandra så löser sig skalbarheten rätt lätt, så länge som lösningen är icke-blockerande så att flera UE:ar kan köra i samma tråd.
Fråga 9

Olika ord kan vara laddade med olika betydelse och de betydelserna kan vara olika i olika sammanhang. Följande två frågor är ställda i kontext av att implementera tillståndsmaskiner i Scala på Xdin.

Vad menas (enl. dig) med att kunna ändra och läsa tillståndsmaskinerna och har du några åsikter om när det är lätt?

Med läsa tillståndsmaskinerna menar jag att de ska vara lätt att förstå för både nya och gamla utvecklare.

Att implementera en hel standard är ett jättejobb så det kan vara lämpligt att börja med en delmängd och sedan succesiut utöka stödet, det är en källa till förändringar. En annan är att standarden kontinuerligt utvecklas, så att stödja en ny version medför också att förändringar behöver göras. En tredje källa till ändringar är rättning av fel.

Det jag menar med lätt att ändra är alltså att det ska vara lätt att införa den typen av förändringar. Hur man mäter detta har jag inget svar på.

Fråga 10

Vad menas (enl. dig) med att testa tillståndsmaskinerna och har du några åsikter om när det är lätt eller hur det bör fungera?

Som jag ser det så är det så finns det två typer av tester som vi är intresserade av, scenariotester och unittester.

Scenariotesterna testar en viss sekvens av meddelanden/event tex en hel attach-procedur följd av en detach-procedur. Den typen av tester fungerar ok i vår nuvarande lösning.

Appendix B

Notes from implementation evaluation session

The following notes are from two evaluation sessions, done at 19 February 2013, with the thesis supervisor at the company - Mattias Evensson. The notes are taken by me, thesis author, and thereby reflect my view of the evaluation session. The notes are retroactively cleaned up and contextual information is added, so the notes can be understood by other readers. The notes are originally written in Swedish but translated to English when cleaned up.

The evaluation sessions focus on the code in the toy example implementation and source code details are discussed, so the following notes supposedly has the most value when compared to the actual implementation source code in appendix D.

B.1 The notes

The SM in class protocol.fsm.NetFSM

- The SM looks compact and nice. It is easy to understand.

- The states \texttt{ATTACH} and \texttt{DETACH} could possibly be renamed to \texttt{ATTACH\_ONGOING} and \texttt{DETACH\_ONGOING} to emphasize the transitional nature of the states.

- It looks strange when using the \texttt{attachBehavior} method in the \texttt{IDLE} state. Should just be something like \texttt{goto(ATTACH) using procedures.powerOn()}. The same applies for \texttt{detachBehavior} in the \texttt{CONNECTED} state.

- A note about the partial function given to the function \texttt{whenUnhandled} is that when an unexpected message is received, a log message is pro-
duced and the state machine makes a transition to the IDLE state. Normally the same is done in actual implementations, but with the difference that the state machine stays in the same state (i.e. no transition between states).

**The interface between procedures and SM in class protocol.fsm.Procedures**

- There is too much logic in the powerOn method. It would be better if the failure handling and the communication to the application layer is delegated to the fsmProcedures instance used in Procedures.

**The integration testing in class protocol.fsm.IntegrationTest**

- It is not recommended to use Thread.sleep in test code.

- General impression of the integration tests is that they look really nice. It is obvious what the tests are doing and they are easy to read. This is how tests should look like. Even the error messages are good and easy to understand.

**Notes on procedure test specifications**

- Generally the tests in the procedure specification test classes are hard to read. They are not as easy as the integration tests when it comes to how they are presented.

- In procedure specification test class protocol.procedure.CoreProcedureSpec it looks like there are a lot of repetitive code (code duplication) that can be improved.

**Notes on code structure**

- Some test specification uses the class protocol.fsm.FsmTestUtil.Fixture as a test fixture. In these cases, it is not obvious that the second argument in e.g. Fixture(idle,attach(input)) is the expected end state (in this case the attach state). In other words, it is not completely clear that this actually is a verification of the end state.

- Sometimes it might not be completely clear what the difference between the methods input and nondetInput in trait protocol.procedure.Environment, and in extension also the, in some procedures used, methods receiveExpected and nondetReceive.

**Notes on method and variable naming**

- Many tests make use of the name cut (class under test) and there are often more suitable names as fsm.
• The related methods \texttt{whenFail} and \texttt{recoverWith} can be confusing. Example suggestions on alternative names are \texttt{attempt} and \texttt{ifFail} respectively.

• The prefix \texttt{nondet} in \texttt{nondetInput} and \texttt{nondetReceive} could be changed to another more descriptive prefix, like \texttt{if} or \texttt{cond} so e.g. \texttt{nondetReceive} becomes \texttt{ifReceive} or \texttt{condReceive}. 
Appendix C

Questions and answers for implementation evaluation

The purpose of the questions in this appendix is to, together with the evaluation session notes in appendix B, make an evaluation of the resulting toy example implementation.

Just as in the initial investigation in appendix A, the questions are answered by the thesis supervisor at the company (Mattias Evenson) and both the questions and answer are presented as is, in the original language (Swedish). Questions are in bold text.

C.1 Questions and answers in Swedish

Fråga 1

Jag har förkortat “en Scala-implementation liknande exempelimplementationen” till S i alla frågor. Vad är de största skillnaderna mellan att använda S jämfört med Unimod eller andra Java-implementerar?

Största skillnaden mot vad som är möjligt att göra med Unimod och Java-implementationer är att procedurer med icke-blockerande I/O kan beskrivas sekventiellt. Det ser i princip ut på samma sätt som när man har blockerande I/O i Java.

Om man i stället jämför med den implementation vi har, så skulle jag säga att största skillnaden är uppdelningen i en FSM del plus ett antal procedurer i stället för allt i ett enda stort diagram. Fast det skulle ju ha gått att göra samma uppdelning med Unimod och andra Java-lösningar.

Fråga 2

Vinster med att använda S?
De vinster jag hoppas att S ska ge är mer lättläst kod och mer lättlästa tester. Med andra ord att det ska bli billigare att underhålla och vidareutveckla koden.

Fråga 3 och 4
Problem med att använda S? Hur är applicerbarheten för S i en verklig situation?
Det största problemet jag ser är att portningen antagligen skulle ta rätt lång tid och det är inte uppenbart att det går att göra den inkrementellt i små steg.

Fråga 5
Finns det några aspekter av ”verkliga implementationer” som inte alls behandlas av S?
När jag specade upp vad din exempel implementation skulle innehålla så försökte jag få med alla de specialfall vi har i den ”riktiga” implementatio-
nen. Jag är rätt säker på att jag missat ett antal fall, men vilka det är lär man väl få se först när man försöker sig på att göra en skarp implementation.
Appendix D

Toy example implementation source code

This appendix is divided up in three sections. Each section corresponds to a logical section in the toy example implementation code. The code in section D.1 shows the code that enables asynchronous procedures. The code in section D.2 shows the actual code that implements the toy example specification - in other words, the code that could be called the production code. The code in section D.3 shows the test code for the protocol implementation, and is the code that asserts that the protocol implementation works as expected.

D.1 Asynchronous procedures

This section shows the code that enables asynchronous procedures. This code is very generic and is not tied to the toy example protocol in any way. The file Procedure.scala in listing D.1 contains all essential code while the file ThunkWrapper.scala in listing D.2 contains helper code that makes the asynchronous procedure thunks easier to work with, via implicit conversion from the ProcedureThunk type to the ThunkWrapper trait.

Code listing D.1: Source file thunkprocedure/Procedure.scala

```scala
package thunkprocedure

import scala.util.continuations.reset
import scala.util.continuations.shift
import scala.util.continuations.cps

/**

```
* Helper object that makes using ProcedureRunner and Thunk easy.
*
/** Recursive definition of continuation wrapper. */
case class Input[In, Out](proceed: In ⇒ Thunk[In, Out])
/**
 * A thunk is either a procedure part waiting for input or
 * a result value.
 /**
type Thunk[In, Out] = Either[Input[In, Out], Out]
/**
 * Wraps a thunk in a ThunkWrapper, to enable some syntax sugar.
 /**
imPLICIT def wrapThunk[In, Out](
t: Thunk[In, Out]): ThunkWrapper[In, Out] =
ThunkWrapper.wrappedThunks(t)
/**
/* Quick way to create a procedure runner. */
def createRunner[In, Out]: ProcedureRunner[In, Out] =
  new RunnerImpl[In, Out]()
/**
 * Runs a code block and returns a Thunk when
 * input is required or when code block returns.
 */
trait ProcedureRunner[In, Out] {
  import ProcedureThunk
  type T = Thunk[In, Out]
  /**
   * Require input. I.e. pause, return thunk and wait until
   * input is given to thunk.
   */
  def input: In @cps[T]
  /**
    * Runs the procedure given as parameter. */
  def apply(procedure: ⇒ Out @cps[T]): T
  /**
   * Called when terminating a procedure from within. */
  def terminate(result: Out): Nothing @cps[T]
  /**
   * Runs a sub-procedure from within another procedure. */
  def runSubProcedure(procedure: ⇒ Out @cps[T]): Out @cps[T]
}
/**
 * Implementation of ProcedureRunner. */
private[thunkprocedure] class RunnerImpl[In, Out]
extends ProcedureRunner[In, Out] {
  import Procedure.Input
}
D.1. ASYNCHRONOUS PROCEDURES

```scala
def input = shift { k: (In ⇒ T) ⇒
  Left(Input[In, Out](k))
}

def apply(procedure: ⇒ Out @cps[T]) =
  reset { Right(procedure) }

def terminate(result: Out) =
  shift { k: (Nothing ⇒ T) ⇒ Right(result) }

def runSubProcedure(procedure: ⇒ Out @cps[T]) = {
  def loop(thunk: T): Out @cps[T] =
    if (thunk.hasResult) thunk.forceResult
    else loop(thunk << input)

  loop(apply { procedure })
}
```

---

**Code listing D.2: Source file thunkprocedure/ThunkWrapper.scala**

```scala
package thunkprocedure

import thunkprocedure.ProcedureThunk

/** Enables some syntax sugar for thunks. */
trait ThunkWrapper[In, Out] {
  def isWaitingForInput: Boolean
  def hasResult: Boolean
  def forceResult: Out
  def result: Option[Out]
  def thunk: Thunk[In, Out]

  /**
   * Add input to thunk and return result from result–thunk.
   * To few input values will throw exception. Excessive
   * inputs will be ignored.
   */
  def apply(ins: In*): Out

  /**
   * Add input to thunk and return next thunk. Adding input to
   * result–thunk will be ignored.
   */
  def <<(in: In): Thunk[In, Out]
}

object ThunkWrapper {
  implicit def wrapThunk[In, Out](
    t: Thunk[In, Out]): ThunkWrapper[In, Out] =
    new ThunkWrapperImpl[In, Out](t)
}
```

103
APPENDIX D. TOY EXAMPLE IMPLEMENTATION SOURCE CODE

D.2 Protocol definition

This section shows the actual code that implements the toy example definition - in other words, the code that could be called the production code. The toy example protocol implementation consists of 13 source code files divided up in three Scala packages (i.e. three source tree folders), depending on functionality.

Two files is used both the state machine part and the asynchronous procedure part, and are therefore placed in the root folder protocol. Three files belong to the state machine part of the implementation and are placed in the folder protocol/fsm. Eight files belong to the asynchronous procedure part of the implementation and are placed in the folder protocol/procedure.

D.2.1 Source code folder protocol

Code listing D.3: Source file protocol/Message.scala

```scala
package protocol

/**
 * Messages between different protocol layers. Some messages also have explicit unapply-methods to support pattern matching.
 */
object Message {

  val ATTACH_ACCEPT = Attach(Accept())
  val ATTACH_COMPLETE = Attach(Complete())
  val ATTACH_REQUEST = Attach(Request())
  val ATTACH_FAIL = Attach(Fail())
  val AUTH_FAIL = Authenticate(Fail())
```

104
D.2. PROTOCOL DEFINITION

val AUTH_RESPONSE = Authenticate(Response())
val AUTH_REJECT = Authenticate(Reject())
val CONNECT = Connect(Request())
val CONNECT_DONE = Connect(Done())
val CONNECT_FAIL = Connect(Fail())
val DETACH_ACCEPT = Detach(Accept())
val DETACH_COMPLETE = Detach(Complete())
val DETACH_REQUEST = Detach(Request())
val DISCONNECT = Disconnect(Request())
val DISCONNECT_DONE = Disconnect(Done())
val POWER_ON = PowerOn(Request())
val POWER_ON_DONE = PowerOn(Done())
val POWER_ON_FAIL = PowerOn(Fail())
val POWER_OFF = PowerOff(Request())
val POWER_OFF_DONE = PowerOff(Request())
val SEND_ACK = SendAck()
val SEND_FAIL = SendFail()

val AUTH_REQUEST = new {
  def apply(id: Int) = Authenticate(RequestWithId(id))
  def unapply(m: Message) = m match {
    case Authenticate(RequestWithId(id)) ⇒ Some(id)
    case _ ⇒ None
  }
}

val RECEIVE = new {
  def apply(payload: Message) = Receive(payload)
  def unapply(m: Message) = m match {
    case Receive(payload) ⇒ Some(payload)
    case _ ⇒ None
  }
}

val SEND = new {
  def apply(payload: Message) = Send(payload)
  def unapply(m: Message) = m match {
    case Send(payload) ⇒ Some(payload)
    case _ ⇒ None
  }
}

sealed trait Message {}

case class Disconnect(submessage: SubMessage) extends Message
case class Disconnect(submessage: SubMessage) extends Message
case class Attach(submessage: SubMessage) extends Message
case class Authenticate(submessage: SubMessage) extends Message
case class Detach(submessage: SubMessage) extends Message
case class PowerOn(submessage: SubMessage) extends Message
case class PowerOff(submessage: SubMessage) extends Message
case class Receive(payload: Message) extends Message
case class Send(payload: Message) extends Message
case class SendAck() extends Message
case class SendFail() extends Message
sealed trait SubMessage {}

case class Accept() extends SubMessage

case class Complete() extends SubMessage

case class Done() extends SubMessage

case class Fail() extends SubMessage

case class Reject() extends SubMessage

case class Response() extends SubMessage

case class RequestWithId(id: Int) extends SubMessage

Code listing D.4: Source file protocol/Stratum.scala

package protocol

/** Interface used for other protocol layers. */
trait Stratum {

/** Sends a message to other protocol layer. */
def output(message: Message): Unit

}

D.2.2 Source code folder protocol/fsm

Code listing D.5: Source file protocol/fsm/NetFsm.scala

package protocol.fsm

import scala.concurrent.duration.DurationInt

import NetState.ATTACH
import NetState.CONNECTED
import NetState.DETACH
import NetState.IDLE
import akka.actor.Actor
import akka.actor.FSM
import protocol.Message
import protocol.Message.DETACH_REQUEST
import protocol.Message.POWER_OFF
import protocol.Message.POWER_ON
import protocol.procedure.NetThunk
import protocol.procedureThunkSugar.ProcedureFailure
import protocol.procedureThunkSugar.ProcedureInput
import protocol.procedureThunkSugar.ProcedureSuccess
import thunkprocedure.Procedure.wrapThunk

/**
 * Implementation of the Net-layer FSM, as an actor and the
 * FSM DSL.
 */


class NetFsm(procedures: Procedures) extends Actor
with FSM[NetState, Option[NetThunk]] {

  type Behavior = PartialFunction[NetThunk, State]
  val timeout = 1 second

  val attachBehavior: Behavior = {
    case ProcedureInput(thunk) =>
      goto(ATTACH) using Some(thunk) forMax timeout
    case ProcedureSuccess() => goto(CONNECTED) using None
    case ProcedureFailure() => goto(IDLE) using None
  }

  val detachBehavior: Behavior = {
    case ProcedureInput(thunk) =>
      goto(DETACH) using Some(thunk) forMax timeout
    case ProcedureSuccess() => goto(IDLE) using None
    case ProcedureFailure() => goto(IDLE) using None
  }

  startWith(IDLE, None)

  when(IDLE) {
    case Event(POWER_ON, _) =>
      attachBehavior(procedures.powerOn())
    case Event(message, _) => goto(IDLE) using None
  }

  when(ATTACH) {
    case Event(msg: Message, Some(thunk)) =>
      attachBehavior(thunk << msg)
  }

  when(CONNECTED) {
    case Event(POWER_OFF, _) =>
      detachBehavior(procedures.powerOff())
    case Event(DETACH_REQUEST, _) =>
      detachBehavior(procedures.forcedDetach())
  }

  when(DETACH) {
    case Event(msg: Message, Some(thunk)) =>
      detachBehavior(thunk << msg)
  }

  whenUnhandled {
    case Event(StateTimeout, _) => goto(IDLE) using None
    case Event(message, _) =>
      log.warning("Unhandled message {} in {},
        message, stateName)
      goto(IDLE) using None
  }
}
package protocol.fsm

/** The state in the Net FSM. */
object NetState {
  val IDLE = Idle()
  val CONNECTED = Connected()
  val ATTACH = Attach()
  val DETACH = Detach()
}

sealed trait NetState
  case class Idle() extends NetState
  case class Connected() extends NetState
  case class Attach() extends NetState
  case class Detach() extends NetState

/** Procedures used by the Net FSM. It is related to
  * FsmProcedures, but does not contain any sign of CPS-
  * annotations and are therefore mockable in tests.
  * Also, in relation to FsmProcedures, Procedures are extended
  * to suite the context of the messages in the Net FSM that
  * triggers these procedures and sends back answers to the
  * application layer.
  */
trait Procedures {
  def powerOn(): NetThunk
  def powerOff(): NetThunk
  def forcedDetach(): NetThunk
}

class ProceduresImpl(radio: Stratum, appl: Stratum)
  extends Procedures {

  private val env =
      new EnvironmentImpl(createRunner[Message, Boolean])
D.2. PROTOCOL DEFINITION


private val fsmProcedures = {
  val prim = new PrimitiveProceduresImpl(env, radio, appl)
  val auth = new AuthProceduresImpl(env, prim)
  val core = new CoreProceduresImpl(env, prim, auth)
  new FsmProceduresImpl(env, prim, core)
}

def powerOn() = env {
  env.whenFail {
    fsmProcedures.powerOn()
  } recoverWith {
    appl.output(POWER_ON_FAIL)
  }
  appl.output(POWER_ON_DONE)
}

def powerOff() = env {
  fsmProcedures.powerOff()
  appl.output(POWER_OFF_DONE)
}

def forcedDetach() = env(fsmProcedures.forcedDetach())

D.2.3 Source code folder protocol/procedure

Code listing D.8: Source file protocol/procedure/AuthProcedures.scala

package protocol.procedure
import scala.util.continuationscps
import protocol.Message.AUTH_FAIL
import protocol.Message.AUTH_REJECT
import protocol.Message.AUTH_REQUEST
import protocol.Message.AUTH_RESPONSE
import protocol.Message.RECEIVE

trait AuthProcedures {
  /** Authenticates the UE if authentication is required. */
  def authentication(): Unit @cps[NetThunk]
}

/** Implementation where auth. request ID is regarded as valid
 * when it is even.
 */
class AuthProceduresImpl{
  environment: Environment,
  primitiveProcedures: PrimitiveProcedures)
extends AuthProcedures {
import primitiveProcedures .
import environment .
def nondetReceive = NondetReceiveSugar(environment) .

def authentication() =
  nondetReceive {
    case AUTH_REQUEST(id) if shouldAuthenticate(id) =>
      send(AUTH_RESPONSE)
      nondetReceive {
        case AUTH_REJECT => failure()}
    case AUTH_REQUEST(id) if !shouldAuthenticate(id) =>
      send(AUTH_FAIL)
      failure()}

private def shouldAuthenticate(id: Int) =
  (id % 2) === 0  // is even
}

---

Code listing D.9: Source file protocol/procedure/CoreProcedures.scala

package protocol.procedure

import scala.util.continuations cps

import protocol.Message.ATTACH_ACCEPT
import protocol.Message.ATTACH_COMPLETE
import protocol.Message.ATTACH_FAIL
import protocol.Message.ATTACH_REQUEST
import protocol.Message.DETACH_COMPLETE
import protocol.Message.DETACH_REQUEST

trait CoreProcedures {

  /**
   * Attaches the UE to Net core, via messages to Radio layer.
   */
  def attach(): Unit @cps[NetThunk]

  /**
   * Detaches the UE from the Net core, via messages to Radio layer.
   */
  def detach(): Unit @cps[NetThunk]
}

class CoreProceduresImpl(
  env: Environment,
  primitiveProcedures: PrimitiveProcedures,
  authProcedures: AuthProcedures)
  extends CoreProcedures {

  import primitiveProcedures .

}
import authProcedures._
import env._
def nondetReceive = NonDetReceiveSugar(env).

def attach() = {
  send(ATTACH_REQUEST)
  nondetReceive { case ATTACH_FAIL => failure() }
  authentication()
  receiveExpected(ATTACH_ACCEPT)
  send(ATTACH_COMPLETE)
}

def detach() = {
  send(DETACH_REQUEST)
  receiveExpected(DETACH_COMPLETE)
}
* (i.e. in both cases), procedure P regarded as used
* (consumed) and is thrown away after input is tested on
* it.
* If the input message is applicable on P, also the
* message will be consumed and another input message is
* required for further processing of partial procedures
* and the final receive call. Partial procedure will be
* matched in the order they are added via the nondetReceive
* method.
* All failures (terminations) triggered inside a partial
* procedure P will propagate.
* [1] i.e. when a deterministic input is expected.
*/

def nondetInput(partProc: PartProc): Unit

/**
 * Invokes a procedure failure. It terminates the procedure
 * execution and returns false. This can be compared to
 * throwing an exception and a catch block that returns
 * false.
 */
def failure(): Nothing @cps[NetThunk]

/**
 * Enables to run sub procedures and handle failure, i.e
 * when sub-procedure returns false. Does not run sub-
 * procedure, only defines it in a RecoverExecutor.
 */
def whenFail(a: => Unit @cps[NetThunk]): RecoverExecutor

/**
 * Implements basic features of environment. */
trait EnvironmentBasics extends Environment {
def failure() = runner.terminate(failedProcedure)
def input() = runner.input
def apply(procedure: => Unit @cps[NetThunk]) = runner {
    procedure
    successfulProcedure
}
}

/** Entity returned from whenFail-block. */
trait RecoverExecutor {
/** Defines recover code and executes sub-procedure. */
def recoverWith(
    b: => Unit @cps[NetThunk]): Unit @cps[NetThunk]
}

/**
 * Handles failure recovery. Implements the recovery of failed
 * sub-procedures, together with RecoverExecutor.
 */
trait FailRecover extends Environment {
D.2. PROTOCOL DEFINITION

```scala
def whenFail(a: ⇒ Unit @cps[NetThunk]) =
  new RecoverExecutorImpl({ a }, runner)

class RecoverExecutorImpl(
  primaryProcedure: ⇒ Unit @cps[NetThunk],
  runner: NetProcedureRunner)
  extends RecoverExecutor {
    def recoverWith(
      secondaryProcedure: ⇒ Unit @cps[NetThunk]: Unit @cps[
        NetThunk] = {
        val successWithPrimary =
          runner.runSubProcedure {
            primaryProcedure
            successfulProcedure
          }
        if (!successWithPrimary) {
          secondaryProcedure
          failure
        }
      }
    )

  } //∗∗ Implements the nondeterministic receive functionality. */

trait NondetInput extends Environment {
  private val nondetQueue = Queue[PartProc]()

  private def findInNondet(
    message: Message): Message @cps[NetThunk] =
    if (nondetQueue.isEmpty) message
    else {
      val pFunc = nondetQueue.dequeue
      val nextMessage =
        if (pFunc.isDefinedAt(message)) {
          pFunc.apply(message)
          runner.input
        } else message
      findInNondet(nextMessage)
    }

  abstract override def input() = findInNondet(super.input())

  def nondetInput(partProc: PartProc): Unit = {
    nondetQueue += partProc
  }

  } //∗∗ Contains a complete implementation for Environment, using
```
APPENDIX D. TOY EXAMPLE IMPLEMENTATION SOURCE CODE

```scala
* multiple partially implemented traits.
*/
class EnvironmentImpl(val runner: NetProcedureRunner)
extends Environment
with EnvironmentBasics
with FailRecover
with NondetInput {}

---

Code listing D.11: Source file protocol/procedure/FsmProcedures.scala

```scala
package protocol.procedure

import scala.util.continuations.cps

import protocol.Message.DETACH_ACCEPT

trait FsmProcedures {
  /** Powers on everything from Net layer and down. */
  def powerOn(): Unit @cps[NetThunk]

  /** Powers off everything from Net layer and down. */
  def powerOff(): Unit @cps[NetThunk]

  /** Powers off after net core forced UE to detach. */
  def forcedDetach(): Unit @cps[NetThunk]
}

class FsmProceduresImpl(
  env: Environment,
  primitives: PrimitiveProcedures,
  core: CoreProcedures)
extends FsmProcedures {

  import primitives._

  import core._

  import env._

  def powerOn() = {
    connect()
    whenFail {
      attach()
      recoverWith {
        disconnect()
      }
    }
  }

  def powerOff() = {
    detach()
    disconnect()
  }

  def forcedDetach() = {
    send(DETACH_ACCEPT)
  }
```
D.2. PROTOCOL DEFINITION

```
disconnect()
detachComplete()
}

Code listing D.12: Source file protocol/procedure/NondetReceiveSugar.scala

```package protocol.procedure
import scala.util.continuations cps
import protocol.Message
import protocol.Message.RECEIVE

/**
 * Wraps partial procedures so Environment.nondetInput works
 * on messages wrapped in RECEIVE message. This enables
 * procedures that are more like the declarative protocol
 * definitions and do not need to have knowledge about the
 * messages used in the RADIO-layer, i.e. the RECEIVE-message.
 */
object NondetReceiveSugar {

def decorateWithRecieve(innerProc: PartProc) =
  new PartProc {
    def apply(m: Message): Unit @cps[NetThunk] =
      m match {
        case RECEIVE(innerMessage) =>
          innerProc(innerMessage)
      }

    def isDefinedAt(m: Message) =
      m match {
        case RECEIVE(innerMessage) =>
          innerProc.isDefinedAt(innerMessage)
        case _ =>
          false
      }

      env.nondetInput(decorateWithRecieve(innerProc))
  }
```

Code listing D.13: Source file protocol/procedure/package.scala

```package protocol
import thunkprocedure.ProcedureThunk
import thunkprocedure.ProcedureRunner
import util.continuations cps
```
APPENDIX D. TOY EXAMPLE IMPLEMENTATION SOURCE CODE

```scala
/** Some procedure definitions. */
package object procedure {

/** A partially applicable Net procedure. */
type PartProc = PartialFunction[Message, Unit @cps[NetThunk]]

/** All procedures in the Net layer needs a runner of these types. */
type NetProcedureRunner = ProcedureRunner[Message, Boolean]

/** All procedures in the Net layer returns these types of thunks. */
type NetThunk = Thunk[Message, Boolean]

/** The value of a successful procedure. */
val successfulProcedure = true

/** The value of a failed procedure. */
val failedProcedure = false
}

Code listing D.14: Source file protocol/procedure/PrimitiveProcedures.scala

package protocol.procedure
import scala.util.continuations.cps
import protocol.Message
import protocol.Message.CONNECT
import protocol.Message.CONNECT_DONE
import protocol.Message.CONNECT_FAIL
import protocol.Message.DETACH_COMPLETE
import protocol.Message.DISCONNECT
import protocol.Message.DISCONNECT_DONE
import protocol.Message.RECEIVE
import protocol.Message.SEND
import protocol.Message.SEND_ACK
import protocol.Message.SEND_FAIL
import protocol.Stratum

trait PrimitiveProcedures {

/** Creates radio layer connection. */
def connect(): Unit @cps[NetThunk]

/** Disconnect the currently existing radio layer. */
def disconnect(): Unit @cps[NetThunk]

/** Send a message net core over existing radio layer */

connection.

116
```
D.2. PROTOCOL DEFINITION

```scala
/*
def send(payload: Message): Unit @cps[NetThunk]
/**
 * Wait for a specific message from net core over existing
 * radio layer connection.
 */
def receiveExpected(expected: Message): Unit @cps[NetThunk]

/**
 * Wait for a any message from net core over existing radio
 * layer connection.
 */
def receive(): Message @cps[NetThunk]

/**
 * Signals to the application layer that UE is detached
 * from core.
 */
def detachComplete(): Unit @cps[NetThunk]

class PrimitiveProceduresImpl(
  env: Environment,
  radio: Stratum,
  appl: Stratum) extends PrimitiveProcedures {

  import env._

  private def throwException(
    actual: Message,
    expected: String = "") = {
    val messagePart =
      if (expected == "") ""
    else " when expecting " + expected
    throw new Exception("Received unexpected message " +
      messagePart)
  }

  def connect() = {
    radio.output(CONNECT)
    input() match {
      case CONNECT_DONE ⇒ ()
      case CONNECT_FAIL ⇒ failure()
      case m ⇒ throwException(m, 
        "connect confirmation (CONNECT_DONE or CONNECT_FAIL)")
    }
  }

  def disconnect() = {
    radio.output(DISCONNECT)
    input() match {
      case DISCONNECT_DONE ⇒ ()
      case m ⇒ throwException(m, "DISCONNECT_DONE")
    }
  }
```
```scala
def detachComplete() = {
  appl.output(DETACH_COMPLETE)
}

def send(payload : Message) = {
  radio.output(SEND(payload))
  input() match {
    case SEND_ACK => ()
    case SEND_FAIL => failure()
    case m => throwException(m, "send confirmation (SEND_ACK or SEND_FAIL)")
  }
}

def receiveExpected(expected : Message) = {
  input() match {
    case RECEIVE(expected) => ()
    case m => throwException(m, expected.toString)
  }
}

def receive() = {
  input() match {
    case RECEIVE(m) => m
    case m => throwException(m)
  }
}

object ThunkSugar {

  object ProcedureSuccess {
    def apply() = Right(true)
    def unapply(thunk: NetThunk) =
      if (thunk.hasResult) {
        val isSuccessful = thunk.forceResult
        if (isSuccessful) Some() else None
      } else None
  }

  object ProcedureFailure {
    def apply() = Right(false)
  }

  package protocol.procedure
  import protocol.Message
  import thunkprocedure.Procedure.Input
  import protocol.Message

  /**
   * Syntax sugar that enables pattern matching of NetThunks.
   */
  object ThunkSugar {

    object ProcedureSuccess {
      def apply() = Right(true)
      def unapply(thunk: NetThunk) =
        if (thunk.hasResult) {
          val isSuccessful = thunk.forceResult
          if (isSuccessful) Some() else None
        } else None
    }

    object ProcedureFailure {
      def apply() = Right(false)
  }

```

Code listing D.15: Source file `protocol/procedure/ThunkSugar.scala`

```
D.3 Protocol tests

This section shows the test code for the protocol implementation, and are the code that asserts that the protocol implementation works as expected.

Many files in the test source code tree corresponds to files in the source code tree. As an example, the test code file `protocol/procedure/CoreProceduresSpec.scala` corresponds to the source code file `protocol/procedure/CoreProcedures.scala`.

The test code folder `protocol/procedure/mockable` do not correspond to a folder in the source code tree. It is instead a folder containing code that helps the testing in the folder `protocol/procedure` by facilitating mockable traits lacking references to any continuations.

D.3.1 Test code folder protocol

Code listing D.16: Test source file `protocol/CompleteSpecificationSuite.scala`

```scala
package protocol

import org.scalatest.Specs
import protocol.fsm.IntegrationTest
import protocol.fsm.NetFsmSpec
import protocol.procedure.AuthPoceduresSpec
import protocol.procedure.CoreProceduresSpec
import protocol.procedure.EnvironmentSpec
import protocol.procedure.FsmProceduresSpec
import protocol.procedure.PrimitiveProceduresSpec

/** Runs all Net layer tests */
class CompleteSpecificationSuite extends Specs({
  new AuthPoceduresSpec(),
})
```
new CoreProceduresSpec() ,
new EnvironmentSpec() ,
new FsmProceduresSpec() ,
new PrimitiveProceduresSpec() ,
new NetFsmSpec() ,
new IntegrationTest() )

D.3.2 Test code folder protocol/fsm

Code listing D.17: Test source file protocol/fsm/FsmTestUtil.scala

package protocol.fsm
import org.scalatest.BeforeAndAfterAll
import org.scalatest.FlatSpec
import org.scalatest.mock.MockitoSugar
import NetState.ATTACH
import NetState.CONNECTED
import NetState.DETACH
import NetState.IDLE
import akka.actor.ActorSystem
import akka.testkit.TestFSMRef
import akka.testkit.TestKit
import protocol.Message.AUTH
import protocol.Message.RECEIVE
import protocol.procedure.NetThunk
import protocol.procedureThunkSugar.ProcedureFailure
import protocol.procedureThunkSugar.ProcedureInput
import protocol.procedureThunkSugar.ProcedureSuccess

/**
 * Test utility specific to the Net layer FSM domain,
 * extending the Akka test kit for actor testing.
 */
abstract class FsmTestUtil
  extends TestKit(ActorSystem("NetFsmSpec"))
  with FlatSpec with MockitoSugar with BeforeAndAfterAll {
  type FSMRef = TestFSMRef[NetState, Option[NetThunk], NetFsm]
  val arbitraryMessage = RECEIVE(AUTH_FAIL)

  /** Different kinds of thunks. */
  val input: NetThunk = ProcedureInput { message ⇒ input }
  val failure: NetThunk = ProcedureFailure()
  val success: NetThunk = ProcedureSuccess()

  /** State-data pairs */
  type StateData = (NetState, Option[NetThunk])
  val idle: StateData = (IDLE, None)
  val connected: StateData = (CONNECTED, None)
  def attach(thunk: NetThunk): StateData =
    (ATTACH, Some(thunk))
D.3. PROTOCOL TESTS

```scala
def detach(thunk: NetThunk): StateData =
  (DETACH, Some(thunk))

/** Abstract methods. */
def createProcedures(): Procedures
def verificationAfterTest(procedures: Procedures): Any

class Fixture(startState: StateData, endState: StateData) {
  def test(testCode: (FSMRef, Procedures) => Any) = {
    val procedures = createProcedures()
    val classUnderTest: FSMRef =
      TestFSMRef(new NetFsm(procedures))
    classUnderTest.setState(startState._1, startState._2)
    testCode(classUnderTest, procedures)
    assert(classUnderTest.stateName == endState._1)
    assert(classUnderTest.stateData.isDefined == endState._2.isDefined)
    verificationAfterTest(procedures)
  }

  override def afterAll { system.shutdown() }
}

Code listing D.18: Test source file protocol/fsm/IntegrationTest.scala
```

---

1 package protocol.fsm
2 import NetState.ATTACH
3 import NetState.IDLE
4 import akka.actor.ActorRef2Scala
5 import akka.testkit.ImplicitSender
6 import protocol.Message
7 import protocol.Message.ATTACH_ACCEPT
8 import protocol.Message.ATTACH_COMPLETE
9 import protocol.Message.ATTACH_REQUEST
10 import protocol.Message.AUTH_FAIL
11 import protocol.Message.AUTH_REQUEST
12 import protocol.Message.AUTH_RESPONSE
13 import protocol.Message.CONNECT
14 import protocol.Message.CONNECT_DONE
15 import protocol.Message.DETACH_ACCEPT
16 import protocol.Message.DETACH_COMPLETE
17 import protocol.Message.DETACH_REQUEST
18 import protocol.Message.DISCONNECT
19 import protocol.Message.DISCONNECT_DONE
20 import protocol.Message.POWER_OFF
21 import protocol.Message.POWER_OFF_DONE
22 import protocol.Message.POWER_ON
23 import protocol.Message.POWER_ON_DONE
24 import protocol.Message.POWER_ON_FAIL
25 import protocol.Message.RECEIVE
26 import protocol.Message.SEND
27 import protocol.Message.SEND_ACK
28```
import protocol.Stratum

trait Direction

case class ToRadio(m: Message) extends Direction
case class ToAppl(m: Message) extends Direction

/**
 * Some larger integration tests using both the Net layer
 * FSM and Net layer procedures.
 */

class IntegrationTest extends FsmTestUtil
    with ImplicitSender {
        import Message _

        def createProcedures() = {
            val radio = new Stratum {
                def output(m: Message) = {
                    println("#RADIO# " + m)
                    testActor ! ToRadio(m)
                }
            }

            val appl = new Stratum {
                def output(m: Message) = {
                    println("#APPL# " + m)
                    testActor ! ToAppl(m)
                }
            }

            new ProceduresImpl(radio, appl)
        }

        def verificationAfterTest(procedures: Procedures) = ()
        val validId = 56 // even == valid authentication ID
        val invalidId = 3  // odd == invalid authentication ID

        "IDLE -> POWER_ON" should "end up in CONNECTED, " +
        "when successful" in
        new Fixture(idle, connected).
        test { (cut, _) =>
        cut ! POWER_ON

        expectMsg(ToRadio(CONNECT))
        cut ! CONNECT_DONE
        expectMsg(ToRadio(SEND(ATTACH_REQUEST)))
        cut ! SEND_ACK
        cut ! RECEIVE(AUTH_REQUEST(validId))
        expectMsg(ToRadio(SEND(AUTH_RESPONSE)))
        cut ! SEND_ACK
        cut ! RECEIVE(ATTACH_ACCEPT)
        expectMsg(ToRadio(SEND(ATTACH_COMPLETE)))
        cut ! SEND_ACK

        expectMsg(ToAppl(POWER_ON_DONE))
        }

        it should "end up in IDLE, " +
        "when authentication ID is invalid" in
new Fixture(idle, idle).
test { (cut, _) \Rightarrow

cut ! POWER_ON

expectMsg(ToRadio(CONNECT))
cut ! CONNECTDONE
expectMsg(ToRadio(SEND(ATTACH_REQUEST)))
cut ! SEND_ACK

cut ! RECEIVE(AUTH_REQUEST(invalidId))
expectMsg(ToRadio(SEND(AUTH_FAIL)))
cut ! SEND_ACK
expectMsg(ToRadio(DISCONNECT))
cut ! DISCONNECT_DONE

expectMsg(ToAppl(POWER_ON_FAIL))
}

it should "end up in IDLE, when timeout " +
"and then ignoring messages that follow." in
new Fixture(idle, idle).
test { (cut, _) \Rightarrow

cut ! POWER_ON
expectMsg(ToRadio(CONNECT))
assert (cut.stateName === ATTACH)

// State timeout is specified at 1 second.
Thread.sleep(1500)
assert (cut.stateName === IDLE)

// should be ignored in IDLE, no exceptions thrown

cut ! CONNECTDONE
}

"CONNECTED $\rightarrow$ POWER_OFF" should "end up in IDLE, " +
"when successful" in
new Fixture(connected, idle).
test { (cut, _) \Rightarrow

cut ! POWER_OFF

expectMsg(ToRadio(SEND(DETACH_REQUEST)))
cut ! SEND_ACK

cut ! RECEIVE(DETACH_COMPLETE)
expectMsg(ToRadio(DISCONNECT))
cut ! DISCONNECT_DONE

expectMsg(ToAppl(POWER_OFF_DONE))
}

"CONNECTED $\rightarrow$ DETACH_REQUEST" should "end up in IDLE, " +
"when successful" in
new Fixture(connected, idle).
test { (cut, _) \Rightarrow
Code listing D.19: Test source file protocol/fsm/NetFsmSpec.scala

```scala
package protocol.fsm

import org.mockito.Mockito.verify
import org.mockito.Mockito.verifyNoMoreInteractions
import org.mockito.Mockito.when
import protocol.Message.DETACH_REQUEST
import protocol.Message.POWER_OFF
import protocol.Message.POWER_ON

/**
  * Unit tests for the Net−layer FSM.
  */
class NetFsmSpec extends FsmTestUtil {

  def createProcedures() = mock[Procedures]
  def verificationAfterTest(procedures: Procedures) =
    verifyNoMoreInteractions(procedures)

  "IDLE --> POWER_ON" should "end up in ATTACH " +
  "when waiting for input" in
  new Fixture(idle, attach(input)).
  test { (cut, procedures) =>
    when(procedures.powerOn()).thenReturn(input)
    cut! POWER_ON
    verify(procedures).powerOn()
  }

  it should "end up in IDLE when procedure fails" in
  new Fixture(idle, idle).
  test { (cut, procedures) =>
    when(procedures.powerOn()).thenReturn(failure)
    cut! POWER_ON
    verify(procedures).powerOn()
  }

  it should "end up in CONNECTED when procedure succeeds" in
  new Fixture(idle, connected).
  test { (cut, procedures) =>
    when(procedures.powerOn()).thenReturn(success)
  }

  cut! DETACH_REQUEST
  expectMsg(ToRadio(SEND(DETACH_ACCEPT)))
  cut! SEND_ACK
  expectMsg(ToRadio(DISCONNECT))
  cut! DISCONNECT_DONE
  expectMsg(ToAppl(DETACH_COMPLETE))
}
```

D.3. PROTOCOL TESTS

 cut ! POWER_ON
 verify ( procedures ). powerOn()

"ATTACH" should "end up in ATTACH " +
"when waiting for input" in
new Fixture ( attach ( input ) , attach ( input ) ).
test { ( cut , _ ) } ➞ cut ! arbitraryMessage }

it should "end up in IDLE when procedure fails" in
new Fixture ( attach ( failure ) , idle ).
test { ( cut , _ ) } ➞ cut ! arbitraryMessage }

it should "end up in CONNECTED when procedure succeeds" in
new Fixture ( attach ( success ) , connected ).
test { ( cut , _ ) } ➞ cut ! arbitraryMessage }

"CONNECTED -> DETACHREQUEST" should "end up in DETACH " +
"when waiting for input" in
new Fixture ( connected , detach ( input ) ).
test { ( cut , procedures ) } ➞
when ( procedures . forcedDetach () ) . thenReturn ( input )
cut ! DETACH_REQUEST
 verify ( procedures ). forcedDetach ()
}

it should "end up in IDLE when procedure fails" in
new Fixture ( connected , idle ).
test { ( cut , procedures ) } ➞
when ( procedures . forcedDetach () ) . thenReturn ( failure )
cut ! DETACH_REQUEST
 verify ( procedures ). forcedDetach ()
}

it should "end up in IDLE when procedure succeeds" in
new Fixture ( connected , idle ).
test { ( cut , procedures ) } ➞
when ( procedures . forcedDetach () ) . thenReturn ( success )
cut ! DETACH_REQUEST
 verify ( procedures ). forcedDetach ()
}

"CONNECTED -> POWER_OFF" should "end up in DETACH " +
"when waiting for input" in
new Fixture ( connected , detach ( input ) ).
test { ( cut , procedures ) } ➞
when ( procedures . powerOff () ) . thenReturn ( input )
cut ! POWER_OFF
 verify ( procedures ). powerOff ()
}

it should "end up in IDLE when procedure fails" in
new Fixture(connected, idle).

test { (cut, procedures) \Rightarrow
  when(procedures.powerOff()).thenReturn(failure)
cut ! POWER_OFF
  verify(procedures).powerOff()
}

it should "end up in IDLE when procedure succeeds" in
new Fixture(connected, idle).

test { (cut, procedures) \Rightarrow
  when(procedures.powerOff()).thenReturn(success)
cut ! POWER_OFF
  verify(procedures).powerOff()
}

"DETACH" should "end up in DETACH " +
"when waiting for input" in
new Fixture(detach(input), detach(input)).

test { (cut, _) \Rightarrow cut ! arbitraryMessage }

it should "end up in IDLE when procedure fails" in
new Fixture(detach(failure), idle).

test { (cut, _) \Rightarrow cut ! arbitraryMessage }

it should "end up in IDLE when procedure succeeds" in
new Fixture(detach(success), idle).

test { (cut, _) \Rightarrow cut ! arbitraryMessage }

D.3.3 Test code folder protocol/procedure

Code listing D.20: Test source file protocol/procedure/AuthProceduresSpec.scala

package protocol.procedure

import org.mockito.Mockito.verify
import org.mockito.Mockito.verifyNoMoreInteractions
import org.mockito.Mockito.when
import org.scalatest.FlatSpec
import org.scalatest.mock.MockitoSugar
import protocol.Message
import protocol.Message.ATTACH_ACCEPT
import protocol.Message.AUTH_FAIL
import protocol.Message.AUTH_REJECT
import protocol.Message.AUTH_REQUEST
import protocol.Message.AUTH_RESPONSE
import protocol.Message.RECEIVE
import protocol.procedure.mockable.PrimitiveMock
import protocol.procedure.mockable.continue
import protocol.procedure.mockable.terminate
import thunkprocedure.Procedure.wrapThunk

class AuthPoceduresSpec extends FlatSpec with MockitoSugar {
  import Message._

  trait CUT extends Fixture {
    val (primitiveProcedures, primitiveMock) = PrimitiveMock(environment)
    val cut: AuthProcedures = new AuthProceduresImpl(environment, primitiveProcedures)
    val invalidId = 1 // odd
    val validId = 2 // even

    def triggerNondetWithExpectedAttachAcceptReceive() =
      assert (environment.input =!= ATTACH_ACCEPT)

    def tearDown = {
      verifyNoMoreInteractions(primitiveMock)
    }

    "Authentication" should "succeed,", + 
    "when received message is not AUTH_REQUEST."
    in new CUT with Success {
      def setup = {} // no setup
      run {
        cut.authentication
        triggerNondetWithExpectedAttachAcceptReceive()
      } withInteraction { t =>
        t <<= Message.ATTACH_ACCEPT
      }
    }

    it should "succeed,", +
    "when received message is AUTH_REQUEST with valid ID"
    "and send AUTH_RESPONSE succeeds"
    "and no AUTH_REJECT is received."
    in new CUT with Success {
      def setup = continue(when(primitiveMock.send(AUTH_RESPONSE)))
      run {
        cut.authentication()
        triggerNondetWithExpectedAttachAcceptReceive()
      } withInteraction { t1 =>
        val t2 = t1 <<= RECEIVE(AUTH_REQUEST(validId))
        verify(primitiveMock).send(AUTH_RESPONSE)
        assert (t2.isWaitingForInput)
        t2 <<= ATTACH_ACCEPT
      }
    }
  }
}
it should "fail," +
"when received message is AUTH_REQUEST with valid ID" +
"and send AUTH_RESPONSE succeeds" +
"and finally an AUTH_REJECT is received." in
new CUT with Failure {

def setup =
    continue (when(primitiveMock.send(AUTH_RESPONSE)))

run {
    cut.authentication
    triggerNondetWithExpectedAttachAcceptReceive()
} withInteraction { t1 =>
    val t2 = t1 << RECEIVE(AUTH_REQUEST(validId))
    verify(primitiveMock).send(AUTH_RESPONSE)
    assert(t2.isWaitingForInput)
    t2 << RECEIVE(AUTH_REJECT)
}
}

it should "fail," +
"when received message is AUTH_REQUEST with invalid ID" +
"and send AUTH_FAIL succeeds." in
new CUT with Failure {

def setup =
    terminate (when(primitiveMock.send(AUTH_RESPONSE)))

run {
    cut.authentication
    triggerNondetWithExpectedAttachAcceptReceive()
} withInteraction { t1 =>
    val t2 = t1 << RECEIVE(AUTH_REQUEST(invalidId))
    verify(primitiveMock).send(AUTH_RESPONSE)
    t2
}
}

it should "fail," +
"when received message is AUTH_REQUEST with invalid ID" +
"and send AUTH_FAIL succeeds." in
new CUT with Failure {

def setup =
    continue (when(primitiveMock.send(AUTH_FAIL)))

run {
    cut.authentication
    triggerNondetWithExpectedAttachAcceptReceive()
} withInteraction { t1 =>
    val t2 = t1 << RECEIVE(AUTH_REQUEST(invalidId))

D.3. PROTOCOL TESTS

```scala
verify(primitiveMock).send(AUTH_FAIL)
t2
}

it should "fail," +
"when received message is AUTH_REQUEST with invalid ID " +
"and send AUTH_FAIL fails." in
new CUT with Failure {
  def setup =
    terminate(when(primitiveMock.send(AUTH_FAIL)))
  run {
    cut.authentication
    triggerNondetWithExpectedAttachAcceptReceive()
  } withInteraction { t1 =>
    val t2 = t1 << RECEIVE(AUTH_REQUEST(invalidId))
    verify(primitiveMock).send(AUTH_FAIL)
t2
  }
}
```

Code listing D.21: Test source file protocol/procedure/CoreProceduresSpec.scala

```scala
package protocol.procedure
import org.mockito.Mockito
import org.mockito.Mockito.verify
import org.mockito.Mockito.verifyNoMoreInteractions
import org.mockito.Mockito.when
import org.scalatest.FlatSpec
import org.scalatest.mock.MockitoSugar
import protocol.Message.ATTACH_ACCEPT
import protocol.Message.ATTACH_COMPLETE
import protocol.Message.ATTACH_REQUEST
import protocol.Message.DETACH_COMPLETE
import protocol.Message.DETACH_REQUEST
import protocol.procedure.mockable.AuthMock
import protocol.procedure.mockable.PrimitiveMock
import protocol.procedure.mockable.continue
import protocol.procedure.mockable.terminate

class CoreProceduresSpec extends FlatSpec with MockitoSugar {
  trait CUT extends Fixture {
    val (primProcedures, primMock) = PrimitiveMock(environment)
    val (authProcedures, authMock) = AuthMock(environment)
    val inOrder = Mockito.inOrder(primMock, authMock);
    val cut: CoreProcedures =
      new CoreProceduresImpl(environment,
```
def tearDown = {
    verifyNoMoreInteractions(primMock)
    verifyNoMoreInteractions(authMock)
}

"Attach" should "succeed, " +
"when all subprocedures are successful." in
new CUT with Success {

def setup = {
    continue (when(primMock.send(ATTACH_REQUEST)))
    continue (when(authMock.authentication()))
    continue (when(primMock.receiveExpected(ATTACH_ACCEPT)))
    continue (when(primMock.send(ATTACH_COMPLETE)))
}

run { cut.attach } and {
    inOrder.verify(primMock).send(ATTACH_REQUEST)
    inOrder.verify(authMock).authentication()
    inOrder.verify(primMock).receiveExpected(ATTACH_ACCEPT)
    inOrder.verify(primMock).send(ATTACH_COMPLETE)
}

it should "fail, " +
"when send ATTACH_COMPLETE fails." in
new CUT with Failure {

def setup = {
    continue (when(primMock.send(ATTACH_REQUEST)))
    continue (when(authMock.authentication()))
    continue (when(primMock.receiveExpected(ATTACH_ACCEPT)))
    terminate (when(primMock.send(ATTACH_COMPLETE)))
}

run { cut.attach } and {
    inOrder.verify(primMock).send(ATTACH_REQUEST)
    inOrder.verify(authMock).authentication()
    inOrder.verify(primMock).receiveExpected(ATTACH_ACCEPT)
    inOrder.verify(primMock).send(ATTACH_COMPLETE)
}

it should "fail, " +
"when it receives ATTACH_FAIL." in
new CUT with Failure {

def setup = {
    continue (when(primMock.send(ATTACH_REQUEST)))
    continue (when(authMock.authentication()))
    terminate (when(primMock.receiveExpected(ATTACH_ACCEPT)))
}

run { cut.attach } and {

showspaces showspaces
86 inOrder . verify (primMock) . send (ATTACH REQUEST)
87 inOrder . verify (authMock) . authentication ()
88 inOrder . verify (primMock) . receiveExpected (ATTACH_ACCEPT)
}

it should "fail, " +
"when authentication fails." in
new CUT with Failure {
  def setup = {
    continue (when (primMock . send (ATTACH REQUEST)))
    terminate (when (authMock . authentication ()))
  }

  run { cut . attach } and {
    inOrder . verify (primMock) . send (ATTACH REQUEST)
    inOrder . verify (authMock) . authentication ()
  }
}

it should "fail, " +
"when send ATTACH REQUEST fails." in
new CUT with Failure {
  def setup = {
    terminate (when (primMock . send (ATTACH REQUEST)))
    run { cut . attach } and {
      inOrder . verify (primMock) . send (ATTACH REQUEST)
    }
}

"Detach" should "succeed, " +
"when all subprocedures are successful." in
new CUT with Success {
  def setup = {
    continue (when (primMock . send (DETACH REQUEST)))
    continue (when (primMock . receiveExpected (DETACH_COMPLETE)))
  }

  run { cut . detach } and {
    inOrder . verify (primMock) . send (DETACH REQUEST)
    inOrder . verify (primMock) . receiveExpected (DETACH_COMPLETE)
  }
}

it should "fail, " +
"when receive DETACH COMPLETE fails." in
new CUT with Failure {
  def setup = {
    continue (when (primMock . send (DETACH REQUEST)))
    terminate (when (primMock . receiveExpected (DETACH_COMPLETE)))
  }

131
APPENDIX D. TOY EXAMPLE IMPLEMENTATION SOURCE CODE

```
package protocol.procedure

import org.scalatest.FlatSpec
import thunkprocedure.Procedure

import protocol.Message
import protocol.Message..

class EnvironmentSpec extends FlatSpec {

  trait EnvironmentFixture {

    val runner = createRunner[Message, Boolean]
    val env: Environment = new EnvironmentImpl(runner)
  }

  trait NondetFixture extends EnvironmentFixture {

    import env._

    var connectOneIsExec = false
    var disconnectIsExec = false
    var authFailIsExec = false
    var connectTwoIsExec = false

    def run(): NetThunk = env {

      nondetInput {
        case DISCONNECT => disconnectIsExec = true
        case CONNECT => connectOneIsExec = true
      }

      nondetInput {
        case AUTH.FAIL => authFailIsExec = true
      }
    }

  }

  it should "fail, " +
  "when send DETACH_REQUEST." in
  new CUT with Failure {

    def setup =
    terminate(when(primMock.send(DETAIL_REQUEST)))

    run { cut.detach } and
    verify(primMock).send(DETAIL_REQUEST)

  }

  inOrder verify (primMock).send (DETACH_REQUEST)
  inOrder verify (primMock).receiveExpected (DETACH_COMPLETE)
}
```

D.3. PROTOCOL TESTS

```python
nondetInput {
    case CONNECT ⇒ connectTwoIsExec = true
}

// Ordinary receive trigger messages to be
// matched against application of nondetReceive.
assert(input == SEND_ACK)

}
def assertResult(thunk: NetThunk, expectedVal: Boolean) = {
    assert(thunk.hasResult)
    assert(thunk.forceResult == expectedVal)
}

"Apply" should "return thunk with true, " +
"when procedure is completed" in
new EnvironmentFixture {
    val thunk = env()
    assertResult(thunk, true)
}

"Failure" should "return thunk with false" in
new EnvironmentFixture {
    val thunk = env { env.failure() }
    assertResult(thunk, false)
}

"Receive" should "return thunk with input request, " +
"when procedure want to receive input" in
new EnvironmentFixture {
    val thunk1 = env {
        assert(env.input == Message.DETACH_COMPLETE)
        assert(env.input == Message.AUTH_RESPONSE)
    }
    assert(thunk1.isWaitingForInput)
    val thunk2 = thunk1 << Message.DETACH_COMPLETE
    assert(thunk2.isWaitingForInput)
    val thunk3 = thunk2 << Message.AUTH_RESPONSE
    assertResult(thunk3, true)
}

"WhenFail" should "not recover but continue " +
"when successful" in
new EnvironmentFixture {
    import env.
    var continuesAfterWhenfailBlock = false
```
val thunk = env {
    whenFail {
        // successful procedure that does nothing
        recoversWith {
            fail("Recover block should not be executed.")
        }
        continuesAfterWhenFailBlock = true
    }
    assert(continuesAfterWhenFailBlock)
    assertResult(thunk, true)
}

it should "recover when failure occurs" +
"and then terminate" in
new EnvironmentFixture {
    import env.
    var recoverBlockIsExecuted = false

    val thunk = env {
        whenFail {
            env.failure() // trigger failure
            recoverWith {
                recoverBlockIsExecuted = true
            }
            fail("Procedure should terminate "+
            "after recover block.")
        }
    }
    assert(recoverBlockIsExecuted)
    assertResult(thunk, false)
}

"NonDetReceive" should "not be executed" +
"when message do not match" in
new NondetFixture {
    val thunk = run() << SEND_ACK
    assert(disconnectIsExec == false)
    assert(connectOneIsExec == false)
    assert(authFailIsExec == false)
    assert(connectTwoIsExec == false)
    assertResult(thunk, true)
}

it should "overshadow later applications of" +
"nonDetReceive that also match." in
new NondetFixture {
    val thunk = run() << CONNECT << SEND_ACK
    assert(disconnectIsExec == false)
    assert(connectOneIsExec)
    assert(authFailIsExec == false)
assert ( connectTwoIsExec == false )
assertResult ( thunk , true )
}

it should "forget applications of " +
"nondetReceive that has already matched." in
new NondetFixture {
    val thunk = run () << DISCONNECT << CONNECT << SEND_ACK
    assert ( disconnectIsExec )
    assert ( connectOneIsExec == false )
    assert ( authFailIsExec == false )
    assert ( connectTwoIsExec )
    assertResult ( thunk , true )
}

it should "forget earlier applications of " +
"nondetReceive that might match at a later stage." in
new NondetFixture {
    val thunk = run () << AUTH_FAIL << CONNECT << SEND_ACK
    assert ( disconnectIsExec == false )
    assert ( connectOneIsExec == false )
    assert ( authFailIsExec )
    assert ( connectTwoIsExec )
    assertResult ( thunk , true )
}

it should "not forget other applications of " +
"nondetReceive that also match current message, " +
"but are overshadowed at the moment." in
new NondetFixture {
    val thunk1 = run () << CONNECT
    assert ( connectOneIsExec )
    assert ( connectTwoIsExec == false )
    connectOneIsExec = false // reset value
    val thunk2 = thunk1 << CONNECT
    assert ( connectOneIsExec == false )
    assert ( connectTwoIsExec )
    connectTwoIsExec = false // reset value
    val thunk3 = thunk2 << SEND_ACK
    assert ( disconnectIsExec == false )
    assert ( connectOneIsExec == false )
    assert ( authFailIsExec == false )
    assert ( connectTwoIsExec == false )
    assertResult ( thunk3 , true )
}

it should "be completely forgotten " +
"when ordinary receive are reached" in

```scala
new EnvironmentFixture {
  import env._

  val thunk = env {
    nondetInput {
      case CONNECT => fail("Should mismatch")
    }
    assert(input == SEND_ACK)
    assert(input == CONNECT)
  } << SEND_ACK << CONNECT

  assertResult(thunk, true)
}
```

**Code listing D.23: Test fixture for procedure tests**

```scala
package protocol.procedure

import scala.util.continuations cps
import org.scalatest.FlatSpec
import org.scalatest.mock.MockitoSugar.mock
import protocol.Message
import thunkprocedure.Procedure.createRunner
import thunkprocedure.Procedure.wrapThunk

/** Test fixture for procedure tests */
trait Fixture extends FlatSpec {
  val environment: Environment =
    new EnvironmentImpl(createRunner[Message, Boolean])

  // abstract
  def expectedResult: Boolean
  def setup: Unit
  def tearDown: Unit
  def procedureWithValidationCode(
    procedure: => Unit @cps[NetThunk]) : Unit @cps[NetThunk] =
    procedure

  // concrete
  def thunkValidations(finalThunk: NetThunk) = {
    assert(finalThunk.hasResult == true,
      "Thunk should contain result after interactions.")

    assert(finalThunk.forceResult == expectedResult,
      "The thunk result is expected to " +
      "be " + expectedResult + ", but isn’t.")
  }

  def run(procedure: => Unit @cps[NetThunk]) = new {
```
D.3. PROTOCOL TESTS

```scala
def withNoInteraction =
  withInteraction { thunk ⇒ thunk }

def and(f: ⇒ Unit) =
  withInteraction { thunk ⇒ f; thunk }

def withInteraction(
  interaction: (NetThunk) ⇒ NetThunk): Unit = {
  def runProcedure() = environment {
    procedureWithValidationCode { procedure }
  }

  setup
  val initialThunk = runProcedure()
  val finalThunk = interaction(initialThunk)
  tearDown
  thunkValidations(finalThunk)
}

/**
 * Behavior implementation for some parts if Fixture.
 * Ensures that the procedure fails.
 */
trait Failure extends Fixture {
  def expectedResult = false

  override def procedureWithValidationCode(
    procedure: ⇒ Unit @cps[NetThunk]) = {
    procedure
    fail("This line should not be executed.")
  }
}

/**
 * Behavior implementation for some parts if Fixture.
 * Ensures that the procedure succeeds.
 */
trait Success extends Fixture {
  def expectedResult = true

  var secondLineIsExecuted = false

  override def procedureWithValidationCode(
    procedure: ⇒ Unit @cps[NetThunk]) = {
    val result = procedure
    secondLineIsExecuted = true
    result
  }

  abstract override def thunkValidations(finalThunk: NetThunk) = {
    assert(secondLineIsExecuted,
```
"Procedure should continue, not terminate")

super.thunkValidations(finalThunk)
}
}

---

Code listing D.24: Test source file protocol/procedure/FsmProceduresSpec.scala

```scala
package protocol.procedure

import org.mockito.Mockito
import org.mockito.Mockito.verify
import org.mockito.Mockito.verifyNoMoreInteractions
import org.mockito.Mockito.when
import org.scalatest.FlatSpec
import org.scalatest.mock.MonoMockSugar

import protocol.Message.DETACH_ACCEPT
import protocol.procedure.mockable.CoreMock
import protocol.procedure.mockable.PrimitiveMock
import protocol.procedure.mockable.continue
import protocol.procedure.mockable.terminate

class FsmProceduresSpec extends FlatSpec with MockitoSugar {
  trait CUT extends Fixture {
    val (primitiveProcedures, primitiveMock) = PrimitiveMock(environment)
    val (coreProcedures, coreMock) = CoreMock(environment)
    val inOrder = Mockito.inOrder(primitiveMock, coreMock);
    val cut: FsmProcedures = new FsmProceduresImpl(
      environment, primitiveProcedures, coreProcedures)

    def tearDown = {
      verifyNoMoreInteractions(primitiveMock)
      verifyNoMoreInteractions(coreMock)
    }

    "PowerOn" should "fail and disconnect, " + "when attach fails." in
    new CUT with Failure {
      def setup =
        terminate(when(coreMock.attach()))
      run(cut.powerOn) and {
        inOrder.verify(primitiveMock).connect()
        inOrder.verify(coreMock).attach()
        inOrder.verify(primitiveMock).disconnect()
      }
      it should "succeed and not disconnect, " +
    }
```

138
"when attach succeeds." in new CUT with Success 
{
  def setup =
  continue(when(coreMock.attach()))
  run(cut.powerOn) and {
    inOrder.verify(primitiveMock).connect()
    inOrder.verify(coreMock).attach()
  }
}

"PowerOff" should "fail," + "when detach fails." in new CUT with Failure 
{
  def setup =
  terminate(when(coreMock.detach()))
  run(cut.powerOff) and {
    verify(coreMock).detach()
  }
}

it should "succeed and disconnect," + "when detach succeeds." in new CUT with Success 
{
  def setup =
  continue(when(coreMock.detach()))
  run(cut.powerOff) and {
    inOrder.verify(coreMock).detach()
    inOrder.verify(primitiveMock).disconnect()
  }
}

"ForcedDetach" should
"first send DETACH_ACCEPT," + "then disconnect" + "and then detachComplete." in new CUT with Success 
{
  def setup = {} // no setup
  run { cut.forcedDetach } and {
    inOrder.verify(primitiveMock).send(DETACH_ACCEPT)
    inOrder.verify(primitiveMock).disconnect()
    inOrder.verify(primitiveMock).detachComplete()
  }
}
import org.mockito.Mockito.verify
import org.mockito.Mockito.verifyNoMoreInteractions
import org.scalatest.FlatSpec
import org.scalatest.mock.MockitoSugar
import protocol.Message.AUTH_FAIL
import protocol.Message.CONNECT
import protocol.Message.CONNECT_DONE
import protocol.Message.DETACH_COMPLETE
import protocol.Message.DISCONNECT
import protocol.Message.DISCONNECT_DONE
import protocol.Message.RECEIVE
import protocol.Message.SEND
import protocol.Message.SEND_ACK
import protocol.Message.SEND_FAIL
import protocol.Stratum
import thunkprocedure.Procedure.wrapThunk

class PrimitiveProceduresSpec extends FlatSpec with MockitoSugar {

  trait CUT extends Fixture {
    val radio = mock[Stratum]("radio")
    val appl = mock[Stratum]("appl")
    val cut: PrimitiveProcedures =
        new PrimitiveProceduresImpl(environment, radio, appl)

    def setup = {} // no setup
    def tearDown = {
      verifyNoMoreInteractions(radio)
      verifyNoMoreInteractions(appl)
    }

    "Connect" should "succeed after " +
    "output CONNECT to radio layer " +
    "when receives CONNECT_DONE." in
    new CUT with Success {
      run {
        cut.connect()
      } withInteraction { t =>
        verify(radio).output(CONNECT)
        t <<= CONNECT_DONE
      }
    }

    it should "fail after " +
    "output CONNECT to radio layer " +
    "when receives CONNECT_FAIL." in
    new CUT with Failure {
      run {

\begin{verbatim}
    cut.connect()

    } withInteraction { t \Rightarrow
      verify(radio).output(CONNECT)
      t <<= CONNECT_FAIL
    }

    "Disconnect" should "succeed after " +
    "output DISCONNECT to radio layer." in
    new CUT with Success {
      run(cut.disconnect) withInteraction { t \Rightarrow
        verify(radio).output(DISCONNECT)
        assert(t.isWaitingForInput)
        t <<= DISCONNECT_DONE
      }
    }

    "DetachComplete" should "succeed after " +
    "output DETACH_COMPLETE to appl layer." in
    new CUT with Success {
      run(cut.detachComplete) and
      verify(appl).output(DETACH_COMPLETE)
    }

    "Send" should "succeed after " +
    "output SEND payload to radio layer," +
    "when receives SEND_ACK." in
    new CUT with Success {
      run {
        cut.send(AUTH_FAIL)
      } withInteraction { t \Rightarrow
        verify(radio).output(SEND(AUTH_FAIL))
        t <<= SEND_ACK
      }
    }

    it should "fail after " +
    "output SEND payload to radio layer," +
    "when receives SEND_FAIL." in
    new CUT with Failure {
      run {
        cut.send(AUTH_FAIL)
      } withInteraction { t \Rightarrow
        verify(radio).output(SEND(AUTH_FAIL))
        t <<= SEND_FAIL
      }
    }

    "ReceiveExpected" should "succeed," +
    "when input is as expected." in
\end{verbatim}
new CUT with Success {
    run {
        cut.receiveExpected(DISCONNECT)
    } withInteraction { _ << RECEIVE(DISCONNECT) }
}

"Receive" should "return the content of a " +
"RECEIVE-message." in
new CUT with Success {
    run {
        assert (cut.receive() == DISCONNECT)
    } withInteraction { _ << RECEIVE(DISCONNECT) }
}

D.3.4 Test code folder protocol/procedure/mockable

Code listing D.26: Test source file protocol/procedure/mockable/AuthMock.scala

package protocol.procedure.mockable

import org.scalatest.mock.MockitoSugar.mock
import protocol.procedure.CoreProcedures
import protocol.procedure.Environment
import protocol.procedure.AuthProcedures

/** Mock trait that hides cps-annotations. */
trait AuthMock {
  /** Determines whether this procedure should fail or not. */
  def authentication(): Boolean
}

object AuthMock {
  def apply(env: Environment): (AuthProcedures, AuthMock) = {
    def mf = mightFail(env) _
    val am = mock[AuthMock]
    val ap = new AuthProcedures {
      def authentication() = mf { am.authentication() }
    }
    (ap, am)
  }
}

Code listing D.27: Test source file protocol/procedure/mockable/CoreMock.scala
package protocol.procedure.mockable

import org.scalatest.mock.MockitoSugar.mock

import protocol.procedure.CoreProcedures
import protocol.procedure.Environment

/** Mock trait that hides cps-annotations. */
trait CoreMock {
  /** Determines whether this procedure should fail or not. */
  def attach(): Boolean

  /** Determines whether this procedure should fail or not. */
  def detach(): Boolean
}

object CoreMock {
  def apply(environment: Environment): (CoreProcedures, CoreMock) = {
    def mf = mightFail(environment)
    val cm = mock[CoreMock]
    val cp = new CoreProcedures {
      def attach() = mf { cm.attach() }
      def detach() = mf { cm.detach() }
    }
    (cp, cm)
  }
}

/** Mock object helpers */
package object mockable {

  /**
   * Triggers an procedure failure (termination) if given
   * argument function returns true.
   *
   */
  def mightFail(env: Environment)(doFail: => Boolean) = if (doFail) env.failure()

  /**
   * Sugar. Procedure termination is associated with mock
   * methods returning true.
   *
   */
  def terminate(stubbing: => OngoingStubbing[Boolean]): Unit = stubbing.thenReturn(true)
}
```scala
// ** Sugar. Continues evaluation of procedure is associated
// * with mock methods returning false.
// */
def continue(stubbing: => OngoingStubbing[Boolean]): Unit =
  stubbing.thenReturn(false)
}

Code listing D.29: Test source file protocol/procedure/mockable/PrimitiveMock.scala
```
D.3. PROTOCOL TESTS

```python
def send(payload: Message) =
    mf { pm.send(payload) }
def receiveExpected(expected: Message) =
    mf { pm.receiveExpected(expected) }
```
Index

Akka FMS, 32
AS, 43
asynchronous procedure, 5
Await.result, 46
become, 30
composition of SM and actor, 33
continuation, 51
continuation passing style, 52
CPS, 52
dataflow, 51
delimited continuations, 51
design pattern, 26
domain specific language, 34
DSL, 34
Eclipse, 1
FAR, 46
finite state machine, 5
FSMSTO, 43
FSMT, 44
FTO, 45
Future, 45
GJT, 42
internal DSL, 35
Java virtual machine, 1
LJT, 42
match/case implementation, 23
ND, 41
RTO, 45
RW, 82
state machine, 5
state machine compiler, 29
state pattern, 25
switch/case implementation, 23
TE, 82
thread sleep, 41
thunk procedure, 51
toy example, 62
transition table, 24
TS, 41
unbecome, 30
UniMod, 1
Bibliography


[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida: http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Joakim Eriksson