Reliable Distributed Programming via Type-parameterized Actors and Supervision Tree

Jiansen HE

Doctor of Philosophy
University of Edinburgh
2014

To ...

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Jiansen HE*)

Table of Contents

Abstract

abstract abstract

Chapter 1

Introduction

blah blah blah

1.1 Contributions

The overall goal of the thesis is to develop a framework that makes it possible to construct reliable distributed applications written using and verified by our libraries which merges the advantages of type checking and the supervision principle. The key contributions of this thesis are:

- A formal model that captures the essence of the supervision principle. The model (Chapter 2) exhibits key features of the supervision principle used in existing libraries including Erlang, Akka, and TAkka. It provides a tool to compare the design alternatives of the supervision principle.
- The design and implementation of the TAkka library, where actors are parameterized by the type of messages it expects. The library (Chapter 3) mixes statical and dynamical type checking so that type errors are detected at the earliest opportunity. The library separates message types and message handlers for the purpose of supervision from those for the purpose of general computation. The decision is made so that type-parameterized actors can form a supervision tree. Interestingly, this decision coincides with our recommendation in the model analysis for improving availability. The TAkka library is carefully designed so that Akka programs can gradually migrate to their TAkka equivalents (evolution) rather than requiring providing type parameters everywhere (revolution). In addition, the type pollution problem can be straightforwardly avoided in TAkka.
- A framework for evaluating libraries that support the supervision principle. The evaluation (Chapter 4) compares the TAkka library and the Akka

library in terms of expressiveness, efficiency and scalability. Results show that TAkka applications add minimal runtime overhead to the underlying Akka system and have a similar code size and scalability compared to their Akka equivalents. Finally, we port the Chaos Monkey library for testing the supervision relationship and design a Supervision View library for dynamically capturing the structure of supervision trees. We believe that similar techniques can be applied to Erlang and new libraries that support the supervision principle.

- A model for analyzing the reliability and availability of fault-tolerant systems that use the *reactive* mechanism (supervision) and the *proactive* mechanism (software rejuvenation). The novel model (Chapter 5) overcomes the limitation of the classic software rejuvenation model where the failure rate is treated as a constant and failure recovery is ironically treated as a stochastic process. **♠add new contributions once achieved ♠ ♠efficient approximate estimation ♠ ♠? the classic model is the least accurate approximation. ? ♠**

Chapter 2

Actors and Their Supervision

2.1 Actor Programming

An actor responses to messages it receives ...

The Actor model defined by Hewitt et al. [1973] treats actors as primitive computational components. Actors collaborate by sending asynchronous messages to each other. An actor independently determines its reaction to messages it receives.

blah blah

2.2 The Supervision Tree

A supervision tree, first proposed in the Erlang OTP library [Ericsson AB., 2012], consists of two types of actors: workers and supervisors. A worker implements part of the business logic and reacts to request messages. A supervisor is responsible for initializing and monitoring its children, which are workers or supervisors for other actors, and restarting its children when necessary. The behaviour of a supervisor is defined by its *supervisor strategies*.

To study the properties of the supervision tree and compare design alternatives, an extensible formal model is required to capture the essence of the supervision tree. In this thesis, worker and supervisor are modelled as different Deterministic Finite Automata (DFAs) (Section 2.2.1). A supervision tree (Section 2.2.2) has supervisors as its internal nodes and workers as its leaf nodes. Further analyses (Section 2.2.3) show that attempts of unifying a supervisor node and a worker node result in problems or complexity. Our model analysis concludes with implementation suggestions given in Section 2.2.4.

2.2.1 Worker and Supervisor as Deterministic Finite Automata (DFAs)

Figure 2.1 gives the state graphs of DFAs that describe worker and supervisor in a supervision tree. Nodes in the graphs represent states of that DFA and arrows are transitions from one state to another. A transition is triggered either by the node itself or due to changes of the environment it resides.

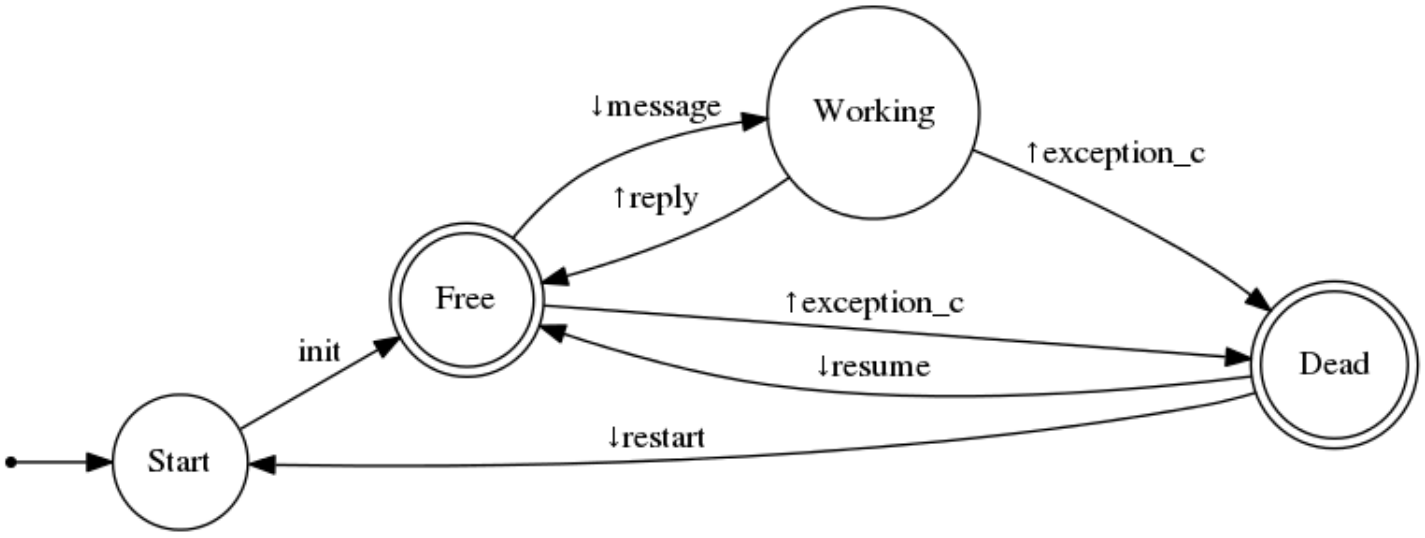
At any time, a worker may be in one of its four states: **Start**, **Free**, **Working**, or **Dead**. After automatically being initialized to the **Free** state, the worker can receive a messages from its outside environment and enters into the **Working** state where no other messages will be processed. If no error occurred when processing the message, the worker returns a result and go back to the **Free** state. An error may occur during the middle of message processing (e.g. software bugs) or when the environment changes (e.g. hardware failures), in which cases the worker reports its failure to its supervisor and goes to the **Dead** state. A dead worker can be resumed (with the restored internal state) or restarted (with an initial internal state) by its supervisor so that it can process new messages. The **Free** and **Dead** states are marked as accept states from which no further actions may occur. On the contrary, a **Working** worker will eventually emit a reply message or raise an exception; and all workers will be successfully initialized.

Similarly, Figure 2.2(b) gives the DFA of a supervisor whose only duty is supervising its children. A supervisor that in its **Free** state reacts to failure messages from its children. Meanwhile, a supervisor may fail at any time and reports its failure to its supervisor.

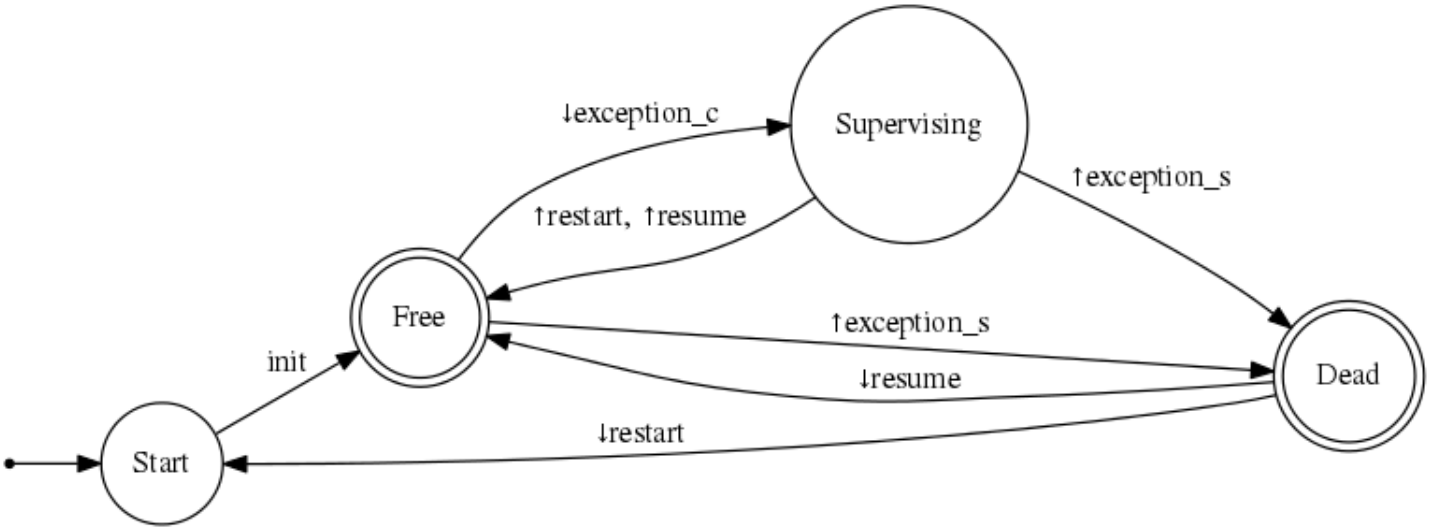
2.2.2 The Supervision Relationship and Supervisor Strategies

The structure of a supervision tree is straightforwardly defined in Figure 2.2. Each internal node is labelled by a supervisor strategy which contains a behaviour function whose type is `Exception` $\Rightarrow$ `Directive`, where `Exception` is the type of failures raised by children and `Directive` represents the final `system action` for that failure. Subtrees of a node is organized as a List.

There are 3 general supervisor strategies found in existing libraries: `OneForOne`, `AllForOne`, and `RestForOne`. If a supervisor adopts the `OneForOne` strategy, the behaviour function is applied to the failed child only. If a supervisor adopts the `AllForOne` supervisor strategy, the behaviour function is applied to all children when any of them fails. If a supervisor adopts the `RestForOne` supervisor



(a) Worker



(b) Supervisor

Figure 2.1: Worker and Supervisor as Deterministic Finite Automata

$$
\begin{array}{rcl}
\text{SupTree} & = & \text{Worker} \\
& | & \text{OneForOne}_D \quad \text{Sup} \quad [\text{SupTree}] \\
& | & \text{OneForAll}_D \quad \text{Sup} \quad [\text{SupTree}] \\
& | & \text{RestForOne}_D \quad \text{Sup} \quad [\text{SupTree}] \\
& | & \cdots \\
\\
\text{D} & = & \text{Escalate} \mid \text{Restart} \mid \text{Resume} \mid \text{Stop} \mid \cdots
\end{array}
$$

Figure 2.2: Supervision Tree

strategy, the behaviour function is applied to the failed child and children in the rest of the list. New supervisor strategies can be added to the model when required.

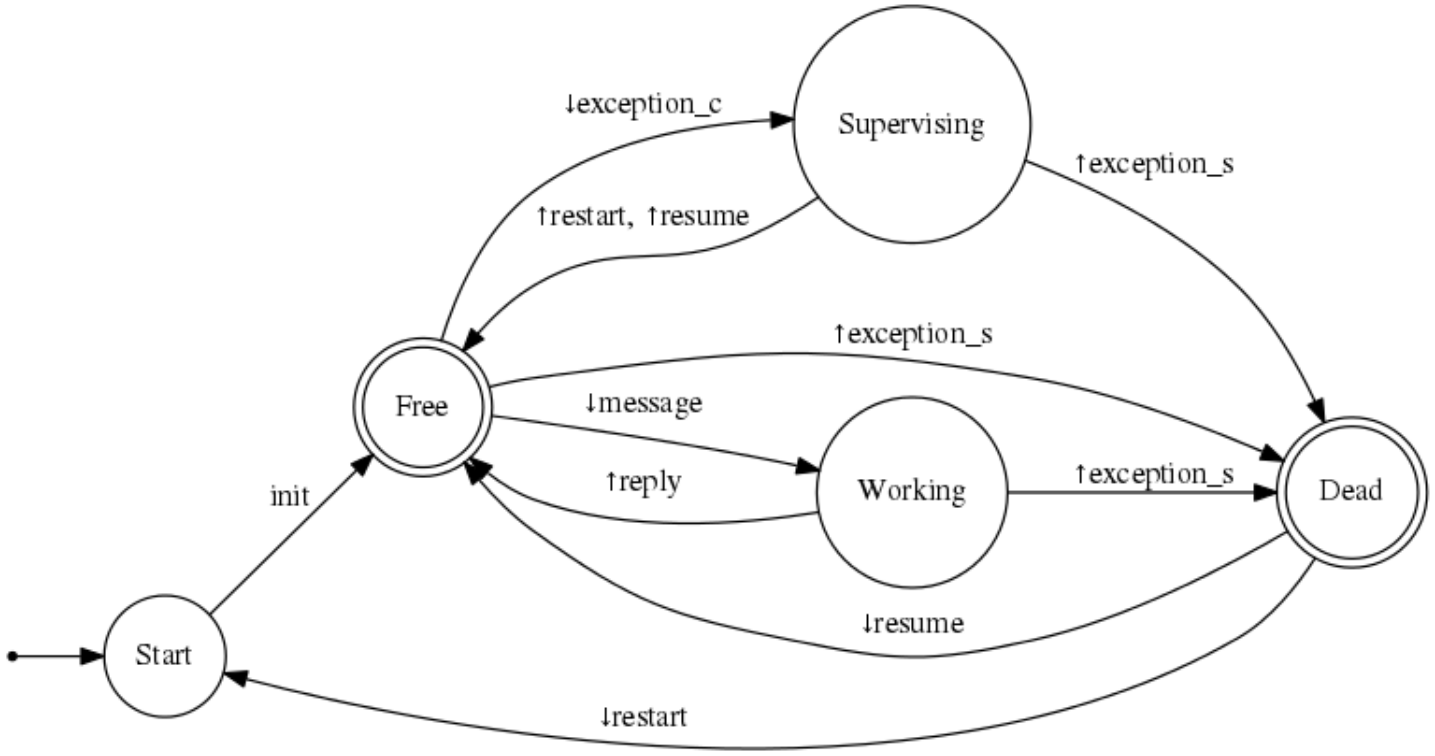
The Erlang OTP library [Ericsson AB., 2012] implements all three supervisor strategies whereas the Akka library [Typesafe Inc. (a), 2012; Typesafe Inc. (b), 2012] does not implement the `RestForOne` strategy because children in Akka is not ordered. Simulating the `RestForOne` supervisor strategy in Akka requires ad-hoc implementation that groups related children and defines special messages to trigger actor termination. No evidence shows that the lack of the `RestForOne` strategy will result in difficulties when rewriting Erlang applications in Akka.

There are 4 directives found in existing libraries: `Escalate`, `Restart`, `Resume`, and `Stop`. The `Escalate` directive throws the exception to the supervisor of the supervisor; the `Restart` directive restart the failed child with its initial internal state; the `Resume` directive restart the failed child with the restored internal state and ask the child to process the message again; finally, the `Stop` directive terminates the failed actor permanently. New directives can be added to the model when required.

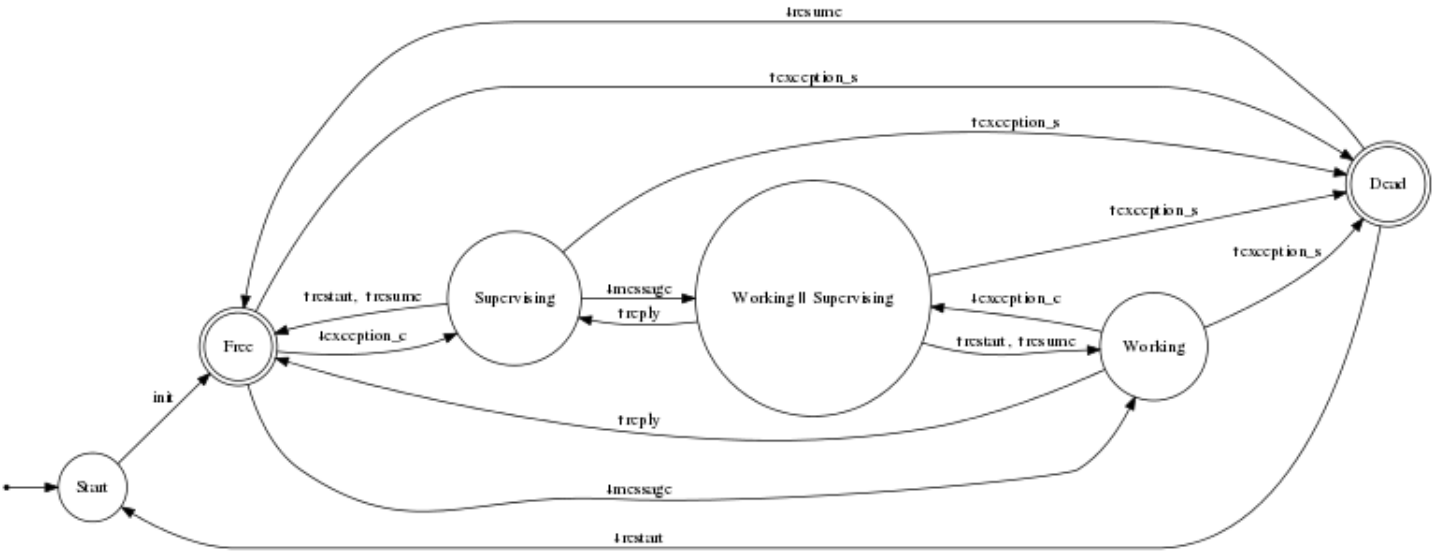
The Akka library [Typesafe Inc. (a), 2012; Typesafe Inc. (b), 2012] implements all four directives whereas the Erlang OTP library [Ericsson AB., 2012] only provides the `Restart` directive and the `Stop` directive. The `Escalate` directive can be simulated in Erlang by manually defining a function that terminates the supervisor when one of its children fails. The `Resume` directive is not needed in Erlang OTP because an Erlang actor does not have a mutable internal state.

2.2.3 Unifying Supervisor and Worker

Notice that the model for supervisor and worker are similar. Can we define a *unified* model that matches both supervisor and worker? Of course we can, and



(a) Supervisor AND Worker



(b) Supervisor PAR Worker

Figure 2.3: A Node that Unifies Supervisor and Worker

indeed there are three ways to do so. This section will present three unified models and discuss the trade-off of using those models.

A General Actor Model The model for worker in Figure 2.2(a) is no more than an actor that can be resumed or restarted when it fails. If an exception raised from a child is viewed as a request message to its supervisor and a restart/resume command is viewed as a reply message to the child, the supervisor model (Figure 2.2(b)) coincides with the worker model (Figure 2.2(a)). For this reason, in Erlang and Akka, both supervisors and workers are implemented as actors. Defining supervisors and workers in terms of the general actor model is a reasonable implementation strategy; however, a specialized model for supervisor is desired in model analyses so that properties and behaviours of supervisors can be better discussed.

Combining Supervisor and Worker The supervision tree defined in Section 2.2.2 requires supervisors as its internal nodes and workers as its leaves. In contrast, the Akka library adopts the alternative design where an internal node can be both a supervisor for other nodes and a worker that implements part of the business logic. The supervisor process and the worker process in Akka share the message box and are compounded to the message handler of the actor. Figure 2.4(a) gives a DFA that models the internal node of an Akka supervision tree. It is clear that the internal node cannot restart or resume failed child when it is processing a message for other purposes, and it cannot perform a computational task when it carries out its duty as a supervisor. In both cases, the performance of the system suffers.

Supervisor in Parallel with Worker One way to get around the limitation of the above design is executing the supervision process and the worker process in parallel, as shown in Figure 2.4(b). From the perspective of model analysis, the above parallel model is equivalent to the one which separates the supervisor process and the worker process into two nodes, and treat them either as siblings or as a supervisor and a child. Of course additional dependencies of the life cycle of those two logically separated nodes shall be addressed in the sense that the failure of one node will cause the failure of the other.

To summarize, a node that naively combining the role of supervisor and worker (Figure 2.4(a)) has less availability than a node that place the supervision process and the worker process in parallel (Figure 2.4(b)). Interestingly, during

the process of designing and implementing the TAkka library (Chapter 3), we realized that the type of supervision messages should be separated from the type of other messages. However, partly because we would like to reuse the Akka implementation and partly because we did not have the above model at that time, the process of handling supervision messages is not separated from the process of handling other messages.

2.2.4 Implementation Considerations

The two DFAs given in Figure 2.1 are abstracted for model analyses. In an implementation of the supervision tree model, following issues might be considered.

Distributed Deployment Supervision Tree represents the logic relationship between nodes. In practice, nodes of a supervision tree may be deployed in distributed machines. A child node may be restarted at or shipped to another physical or virtual machine but stays in the same local position of the supervision tree.

Heart-Beat Message At run-time, failures may occur at any time for different reasons. In some circumstances, failure messages of a child may not be delivered to its supervisor, or even worse, not be sent by the failed child at all. To build a system tolerant to the above failures, supervisors need to be aware of the liveness of their children. One commonly used approach is asking the child to periodically send heart-beat messages to its supervisor. If no heart-beat message is received from a child within a time-out, that child is considered dead by the supervisor and an appropriate recovery process is activated. In our model, a logical exception is sent from a child to its supervisor when no heart-beat message is delivered within a time-out, and a restart or resume message is sent to a suitable (virtual) machine where the recovered node will reside.

Message Queuing In the model given in Figure 2.2(a), a node can processed one message a time when it is **Free**. The model does not exclude the case where messages are queuing either in memory or a distributed database. Similarly, when a node is resumed from a failure, the message it was processing before that failure may either be restored or be discarded.

2.3 Akka Basics

2.3.1 Akka Actor

An Akka Actor has four essential components as shown in Figure 2.4: (i) a receive function that defines its reaction to incoming messages, (ii) an actor reference pointing to itself, (iii) the actor context representing the outside world of the actor, and (iv) the supervisor strategy for its children.

```
1 package akka.actor
2 trait Actor{
3   def receive:Any=>Unit
4   val self:ActorRef
5   val context:ActorContext
6   var supervisorStrategy: SupervisorStrategy
7 }
```

Figure 2.4: Akka Actor API

Figure 2.5 shows an example actor in Akka. The `receive` function of the Akka actor has type `Any⇒Unit` but the defined actor, `ServerActor`, is only intended to process strings. At Line 16, a `Props`, an abstraction of actor creation, is initialized and passed to an actor system, which creates an actor with name `server` and returns a reference pointing to that actor. Another way to obtain an actor is using the `actorFor` method as shown in line 24. We then use actor references to send the actor string messages and integer messages. String messages are processed in the way defined by the receive function.

Undefined messages are treated differently in different actor libraries. In Erlang, an actor keeps undefined messages in its mailbox, attempts to process the message again when a new message handler is in use. In versions prior to 2.0, an Akka actor raises an exception when it processes an undefined message. In recent Akka versions, an undefined message is discarded by the actor and an `UnhandledMessage` event is pushed to the event stream of the actor system. The event stream may be subscribed by other actors who are interested in particular event messages. To handle the unexpected integer message in the above short example, an event handler is defined and created with 6 lines of code.

2.3.2 Supervision

The Supervision Tree principle is proposed but optional in Erlang. On the contrary, the Akka library makes supervision obligatory by restricting the way

```
class ServerActor extends Actor {
  def receive = {
    case m:String => println("received message: "+m)
  }
}

class MessageHandler(system: ActorSystem) extends Actor {
  def receive = {
    case akka.actor.UnhandledMessage(message, sender, recipient) =>
      println("unhandled message:"+message);
  }
}

object ServerTest extends App {
  val system = ActorSystem("ServerTest")
  val server = system.actorOf(Props[ServerActor], "server")

  val handler = system.actorOf(Props(new MessageHandler(system)))
  system.eventStream.subscribe(handler,
                  classOf[akka.actor.UnhandledMessage]);
  server ! "Hello World"
  server ! 3

  val serverRef = system.actorFor("akka://ServerTest/user/server")
  serverRef ! "Hello World"
  serverRef ! 3
}


/*
Terminal output:
received message: Hello World
unhandled message:3
received message: Hello World
unhandled message:3
*/
```

Figure 2.5: A String Processor in Akka

of creating actors. Actors can only be initialized by using the `actorOf` method provided by `ActorSystem` or `ActorContext`. Each actor system provides a guardian actor for all user-created actors. Calling the `actorOf` method of an actor system creates an actor supervised by the guardian actor. Calling the `actorOf` method of an actor context creates a child actor supervised by that actor. Therefore, all user-created actors in an actor system, together with the guardian actor of that actor system, form a tree structure. Obligatory supervision unifies the structure of actor deployment and simplifies the work of system maintenance.

Each actor in Akka is associated with an actor path. The string representation of the actor path of a guardian actor has format *akka://mysystem@IP:port/user*, where *mysystem* is the name of the actor system, *IP* and *port* are the IP address and the port number which the actor system listens to, and *user* is the name of the guardian actor. The actor path of a child actor is actor path of its supervisor appended by the name of the child actor, either a user specified name or a system generated name.

Figure 2.6 defines a simple calculator which supports multiplication and division. The simple calculator does not consider the problematic case of dividing a number by 0, where an `ArithmeticException` will be raised. We then define a safe calculator as the supervisor of the simple calculator. The safe calculator delegates calculation tasks to the simple calculator and restarts the simple calculator when an `ArithmeticException` is raised. The supervisor strategy of the safe calculator also specifies the maximum number of failures its child may have within a given time range. If the child fails more frequently than the allowed frequency, the safe calculator will be stopped, and its failure will be reported to its supervisor, the system guardian actor in this example. The terminal output shows that the simple calculator is restarted before the third message and the fifth message are delivered. The last message is not processed because both calculators are terminated since the simple calculator fails more frequently than allowed.

```
case class Multiplication(m:Int, n:Int)
case class Division(m:Int, n:Int)

class Calculator extends Actor {
  def receive = {
    case Multiplication(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Division(m:Int, n:Int) =>
      println(m +" / "+ n +" = "+ (m/n))
  }
}
class SafeCalculator extends Actor {
  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 2, withinTimeRange = 1 minute) {
      case _: ArithmeticException =>
        println("ArithmeticException Raised to: "+self)
        Restart
    }
  val child:ActorRef = context.actorOf(Props[Calculator], "child")
  def receive = { case m => child ! m }
}
  val system = ActorSystem("MySystem")
  val actorRef:ActorRef = system.actorOf(Props[SafeCalculator],
"safecalculator")

  calculator ! Multiplication(3, 1)
  calculator ! Division(10, 0)
  calculator ! Division(10, 5)
  calculator ! Division(10, 0)
  calculator ! Multiplication(3, 2)
  calculator ! Division(10, 0)
  calculator ! Multiplication(3, 3)
/* Terminal Output:
3 * 1 = 3
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
10 / 5 = 2
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
java.lang.ArithmeticException: / by zero
3 * 2 = 6
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
java.lang.ArithmeticException: / by zero
*/
```

Figure 2.6: Supervised Calculator

Chapter 3

TAkka: Design and Implementation

Chapter 4

TAkka: Evaluation

Chapter 5

A Precise Model for Software Rejuvenation

Chapter 6

Future Work

Chapter 7

Conclusion

Bibliography

Ericsson AB. (2012). OTP Design Principles User's Guide. `http://www.erlang.org/doc/pdf/otp-system-documentation.pdf`.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. pages 235–245.

Typesafe Inc. (a) (2012). Akka API: Release 2.0.2. http://doc.akka.io/api/akka/2.0.2/. Accessed on Oct 2012.

Typesafe Inc. (b) (2012). Akka Documentation: Release 2.0.2. http://doc.akka.io/docs/akka/2.0.2/Akka.pdf. Accessed on Oct 2012.