Vakkaria
Vaadin-Akka Integration Experiment

Alexandre Zua Caldeira
zuacaldeira@gmail.com

September 23, 2016

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Misanthrop

Chapter 2

Javices

A **javice** is anything related to Java. *Um tique ou um vício de programação. Um padrão ou um anti-padrão; uma classe ou um comentário bem construídos. Uma boa ou uma má prática. Uma obsessão, uma experiência ou um improviso.*

I have a recurring obsession: improvisation. I assume my hypothesis that all the software needed in the future is still to be done. Since the future is the future, nothing on it already exists. The future's software is not yet here nor there. This means we can create it. Then let's do it! And that is improvisation. So two challenges: develop future's software units, improvising.

The problem about developing the future's software is that we have to build it today, and today we only have today's data-structures and algorithms. To become the pieces of future's software systems they must be incredibly simple. Only that way they can be reusable and easy to adapt and compose.

The problem about improvisations is that you have to master the ability to, out of known elementary pieces, develop unknown, intriguing, challenging and pleasant compositions. And that is only possible mastering the individual pieces and the interaction between them. Improvisers are players and composers at the same time, and this time is real-time. Live. Here and now, and that is a *timeline*.

2.1 Timelines

We are extremely dependent on our conception of time. We organize most of our activities on the assumption of a linear time flow. Based on a concept of an experienced *instant* called now, we created a notion of past, present and future, and tools to include those concepts in our semantic reasoning system. Timelines are data structure that store the history of a particle or system of particles, organized by the default time flow, spanning from $-\infty$ up to $+\infty$. Each point in the timeline is a pair (e, t_e) saying that event e occurred on instant t_e. Not so simple. Not so far.

Let's start by specifying *particle*, *events* and *history*.

2.1.1 Particles

A particle is a minute fragment or quantity of matter, with physical properties such as volume and mass. [1]

[1] https://en.wikipedia.org/wiki/Particle, 18 Sep. 2016

2.1.2 Events

Not all events are instantaneous single events.

Chapter 3

*Akka*ros

3.1 *Bakkaros*

3.2 *Vakkaros*

The main entrance is controlled by a doorman, the `WelcomeMVCActor`. No one enters in the system without his acknowledgement. The behavior of this mvc actor is to redirect users to the registration hall or to the main access control zone that leads into Akkaria, where other actors will navigate and interact with the user through the system. These actors are the `RegisterMVCActor` and `LoginMVCActor`. The session type w is governed by the `WelcomeMVCActor`, and is a typical session type for a delegator, where upon message reception, possibly after some internal tasks, the delegator transfers user interaction to other actors, in this case, to the registration or login mvc actors, with session types R and S, respectively.

$$W = \&\{register : R, login : L\} \tag{3.1}$$

The finite automata representing this session type is depicted in Figure 3.1.

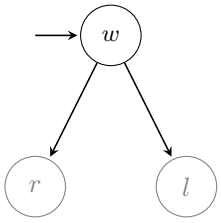


Figure 3.1: WelcomeMVCActor Finite automata.

The `RegisterMVCActor` governs things from state r. He is responsible to present the user a register form, collect it and try so persist it internally in the database. Many things can go wrong during the registration process. The input data provided by the client, *username, password, fullname* is bound to business rules they must observe. Also, there can be system failures or any other abnormal situations that may prevent the registration process to be completed. This means that we need a fault tolerance mechanism to react adequately to the circumstances. A session type r that includes fault tolerance is given below in equation 3.2. Parameters are data that the user must provide. When receiving such a message, the register actor enters in a state v_r where the operation is validated.

$$\begin{cases} r & = \&\{register(email, password, fullname) : v_r\} \\ v_r & = +\langle SUCCESSFUL : l, INVALID : r, FAILED : w\rangle \end{cases} \tag{3.2}$$

Validation can result in three possible results: `SUCCESSFUL, INVALID, FAILED`. If the data provided from client violates any business rule, the registration will be denied. The actor sends an invalid message to the caller and resumes to it's initial state r. If the data is valid, but a system error or exception occurs during data persistence in the database, failure will be reported to the caller and state resumed to w. If successful, user interaction will progress to the login state, under supervision of the `LoginMVCActor`. The finite automata representing this interaction is shown in Figure 3.2.

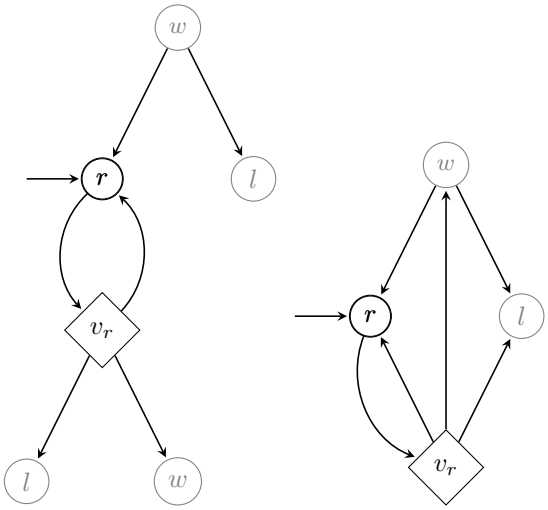


Figure 3.2: `RegisterMVCActor` normalized finite automata

Figure 3.3: `RegisterMVCActor` session type.

This last automata introduces two new features: v_r, a select state marked by a diamond, and cycles introduced by the interaction of the verification state v_r and other system states. In circle states, actors waits for client messages, and on diamond states the actor communicates it's internal states to clients. Clients will then synchronize with the actor, providing data until the data is valid leading the actor to state l, or an error occurs bringing the actor into state w. States l and w are governed by other actors, so we change both state and host actor. State v_r is the state that implements fault tolerance: instead of terminating the interaction, client and actors behave safely even in the presence of failures, error or invalid data input.

The `LoginMVCActor` behaves in a similar way (Figure 3.2). In state l, the actor is expecting a *login* message with the input data. When the client sends such message, the actor performs a state transition to v_l where the result of the operation will be communicated back to the client for state synchronization. This happen in v_l, where fault tolerance mechanisms are implemented in cases of system failure or violation of business requirements. If no violation nor failure is detected, and user redirected to state u, the user area under the control of a new mvc actor: `UserMVCActor`, an *avatar* in Akkaria.

3.2.1 WelcomeMVCActor

3.2.2 RegisterMVCActor

Creating the RegisterMVCActor

MVC actor live in the ui and provides a bridge to the backend actors, forming together a sort of bidirectional communication channel. Users and views only have direct access to MVC actors, not to business actors. MVC actors acts as controllers that will internally coordinate processes and resources, including business actors, to accomplish the requested task. The source code of

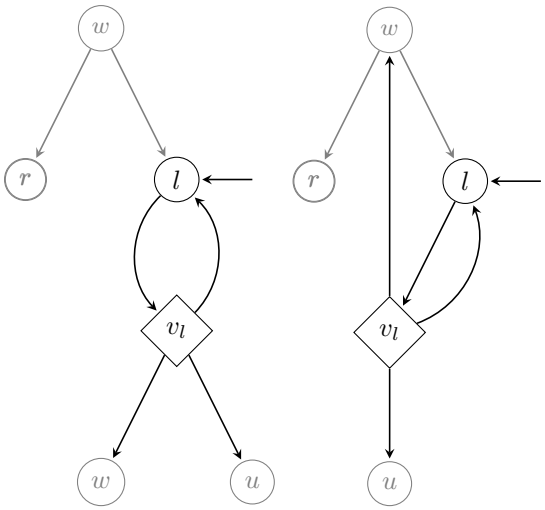


Figure 3.4: `RegisterMVCActor` normalized finite automata

Figure 3.5: `RegisterMVCActor` session type.

`RegisterMVCActor` is listed in Listing 3.2.2 To create a reference to this actor:

The business rules or requirements for the registration process exist.

- Usernames are email valid addresses; users can register only once. The existence of a registration is determined by the existence of one and one only `Registration` relationship for an account whose username matches the input *username*. A username is valid if respects:

$$\forall r = (u, a) \in R,\ u \in U,\ a \in A : a.username \neq username \tag{3.3}$$

- Passwords have to be strong and will be stored encrypted
- Full names are made of at least two names

After registration there will be new data units in the database. In particular:

$$\begin{cases} \exists^1 u \in U : & u.fullname = fullname \\ \exists^1 a \in A : & a.username = username \\ \exists^1 r = (u', a') \in R : & a' = a, u' = u \end{cases} \tag{3.4}$$

The behaviour of `WelcomeActor` is to coordinate the registration and login processes. This behaviour is represented by session type

$$\begin{cases} W &= \&\{register : R, login : L\} \\ R &= \&\{register(username, password, fullname) : V_r\} \\ V_r &= +\langle true : L, false : R, error : W\rangle \\ L &= \&\{login(username, password) : V_l\} \\ V_l &= +\langle true : U, false : L, error : W\rangle \end{cases} \tag{3.5}$$

In this example, in *Welcome* state the actor offers two services, *register* and *login*. To access the service clients must send one of these messages. Exchanging this message will bring the communication into a new state, either *Register* or *Login*. In state *Register* the actor expects a message wrapping user input. Input validation, business rule validation and error handling are reported back to the client for state synchronization (*ValidateRegistration*). If validation and operation were successful, we can go to the login state *Login*. Registered users can go directly to

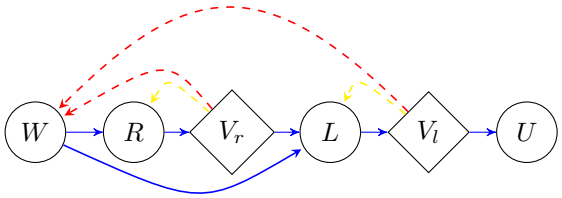


Figure 3.6: `WelcomeActor` behavior described by finite state machine with states W, R, V_r, L, V_l, U. Some states are represented by circles, and others represented by diamonds. Circle states are states were services are offered and ready to be consummed by clients; on diamonds states feedback is given back to the caller.

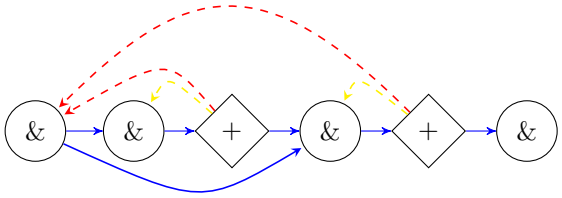


Figure 3.7: `WelcomeActor` behavior described by a *session type*.

Login. In the login state the actor expects a message wrapping the user username and password. Input validation, storage, and error are reported in state $ValidateLogin$. Upon successful login, the user can enter the gates of Akkaria in state U.

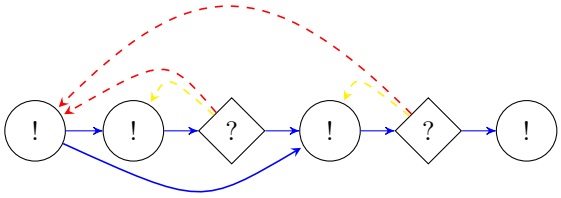
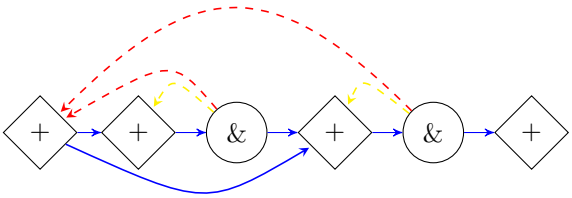


Figure 3.8: `WelcomeActor` behavior described by a *session type*.



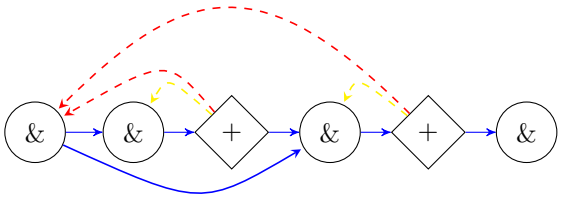


Figure 3.9: `WelcomeActor` behavior described by a *session type*.