

A New Risk Management Approach Deployed over a Client/Server Distributed Functional Architecture*

Víctor M. Gulías, Carlos Abalde, Laura M. Castro, Carlos Varela
LFCIA - MADS Group. Computer Science Department
University of A Coruña (Spain)
{gulias,carlos,cvarela,laura}@dc.fi.udc.es

Abstract

This work shows our experience in the development of a large, scalable and reliable client/server risk management information system. It was developed using the distributed functional language Erlang for describing the domain logic and the imperative language Java for the thin-client interface. The robust system architecture, and the extensibility and adaptability of the framework to any business model, are the main novelties of this work in the field of risk management information systems. This paper provides a semi-formal and non-technical explanation about the system internals.

1 Introduction

In this paper, our experience in the development of a large, scalable and reliable client/server risk management application using the distributed functional [7] language Erlang [3] for describing the domain logic and the imperative language Java for the thin-client interface is presented. The application (ARMISTICE, *Advanced Risk Management Information System: Tracking Insurances, Claims and Exposures* [4], <http://mads.lfcia.org/armistice>) is a risk management information system designed for a large company.

The main novelty of this system, in the field of traditional management software, is that it has been partly developed using a declarative language (Erlang). On the other hand, the main novelty in the field of risk management information systems (RMISs) is the new approach used to manage the different kind of heterogeneous business objects. ARMISTICE provides an abstract framework that can be easily adapted to any organisation. It represents a step forward compared with other popular RMIS products like [2] or [1].

The paper is structured as follows. First, some background knowledge about the whole architecture is presented. Next, section 3 provides a semi-formal and non-technical explanation about the system internals. Finally, section 4 shows some conclusions.

2 Mixing Functional and Imperative Worlds

ARMISTICE has a client/server architecture structured in layers using two well-known architectural patterns: Layers and Model-View-Controller [6]. Figure 1 provides an overview.

* This work has been partially supported by Spanish MCyT TIC 2002-02859 and by the company Alfa21 Outsourcing S.L. (FUAC R&D project 2/79).

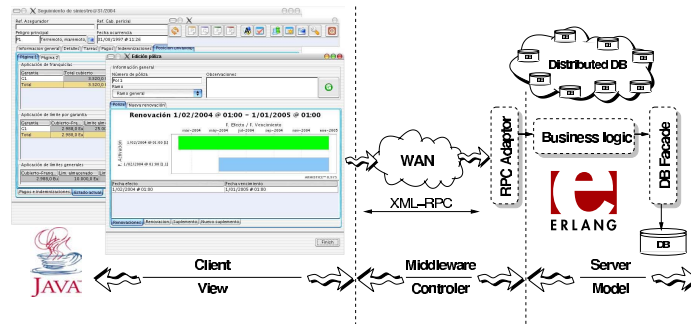


Figure 1. ARMISTICE three tier architecture

The user side is a lightweight Swing/Java client which only makes remote procedure calls to the server (XML-RPC) and has no associated logic. The server side, completely developed in Erlang, supports the model and all the business logic. In [4], some additional information about other technical notes on object handling in a functional language and about the integration between client and server can be found.

Erlang is a distributed and concurrent functional programming language developed by Ericsson for telecommunication applications. It is one of the notable success exceptions of functional languages in real world applications [7]. What makes Erlang different from other functional languages is its support for concurrency and distribution. Furthermore, all the concurrency constructs are semantically equivalent in a distributed framework.

So, the use of Erlang and its programming philosophy to deal with the risk management domain complexity has considerably simplified the server development process, reducing implementation time cycles thanks to the higher abstraction level it provides. Moreover, Erlang platform allowed us to easily reach key features such as server scalability and high availability at a very low cost.

3 Insurance Policy Model

The system is splitted into three large subsystems: *risks*, *policies* and *claims*. Thus, the *risk subsystem*, the most abstract and dynamic one, models a set of basic concepts used throughout the whole application. The *policies subsystem*, which is closer to the domain, uses those concepts to define *insurance policies* based upon formulas and applicability constraints. Finally, the *claims subsystem* uses the insurance policy models as the input to drive user decisions and manage all the claims. This subsystem is the subset of ARMISTICE used everyday by non-expert users. They can manage all the tasks around a claim: add it to the system, search for coverage using a decision assistant support system, etc.

3.1 Risks Subsystem

3.1.1 Risks Situations and their Attributes

The key concept in this subsystem is the *risk situation* (RS). A RS models the state of any element that can be included on the coverage of a policy (e.g. a shop or a vehicle). Every RS is an instance of a *risks group* (RG). A RG is a template, that is, a meta-description of all the important attributes (e.g. address, vehicle registration number, warehouse area or

cubic capacity) which describe the state of any element that can be insured (e.g. commercial locals or industrial vehicles). We denote the set of attributes as \mathcal{A} .

$$\begin{aligned} \text{Attribute} &\equiv (\text{name}, \text{type}) \text{ where } \text{name} \in \text{String} \wedge \text{type} \in \text{Types} \wedge \\ \text{Types} &= \{\text{String}, \text{Boolean}, \text{Timestamp}, \text{Money}, \dots\} \end{aligned}$$

Over these elements, the set of risks groups \mathcal{RG} is defined. Each member $\mathcal{RG}_i \in \mathcal{RG}$ is a set like follows:

$$\mathcal{RG}_i = \{a_0, a_1, \dots, a_{n-1}\} \forall x \in [0, n-1] a_x \in \mathcal{A}$$

Therefore a RG is a set of attributes which provides a meta-description of any property that can be important from the insurance's point of view.

Finally, using the \mathcal{RG} set, the set of risk situations \mathcal{RS} is defined. A RS \mathcal{RS}_j is an instance of a RG \mathcal{RG}_i ($\mathcal{RS}_j : \mathcal{RG}_i \longrightarrow \cup \text{Types}$). Every attribute in the RG will have a concrete value assigned in the RS.

$$\mathcal{RS}_j = \{p_0, p_1, \dots, p_{n-1}\} \forall x \in [0, n-1] p_x \equiv (a, v) \wedge a \in \mathcal{RG}_i \wedge v \in \text{Type}(a)$$

Then, a RS represents the state of any element that can be insured. For short, we will use $\mathcal{RS}_j.a$ to denote the value of attribute a of the RS \mathcal{RS}_j .

3.1.2 Versions and Revisions of RSs

Using the previous tools, it is possible to model the state of every insurable element (i.e. RSs). However, business requirements demand a temporal representation of the RSs. Based on the Fowler patterns [5], a temporal tracking of the \mathcal{RS} set is performed. The evolution of a RS is modelled since its creation as a set of *versions* and *revisions* (two-dimensional temporal modelling). A *version* represents a new state of a RS as a result of a business event. However, a *revision* represents a new state of a RS as a result of a mistake or any other business external event. This behaviour is provided modifying the original definition of a RS as follows:

$$\begin{aligned} \mathcal{RS}_j &= \{\mathcal{RS}_j^{v_0}, \dots, \mathcal{RS}_j^{v_{\alpha-1}}\} \forall x \in [0, \alpha-1] v_x \in \text{Timestamp} \wedge v_x < v_{x+1} \wedge \\ \mathcal{RS}_j^{v_x} &= \{\mathcal{RS}_j^{v_x, r_0}, \dots, \mathcal{RS}_j^{v_x, r_{\beta-1}}\} \wedge \forall y \in [0, \beta-1] r_y \in \text{Timestamp} \wedge r_0 = v_x \leq r_y < r_{y+1} \wedge \\ \mathcal{RS}_j^{v_x, r_y} &= \{p_0, p_1, \dots, p_{n-1}\} \wedge \forall z \in [0, n-1] p_z \equiv (a, v) \wedge a \in \mathcal{RG}_i \wedge v \in \text{Type}(a) \end{aligned}$$

Therefore, the expression $\mathcal{RS}_j.a$ must be changed by $\mathcal{RS}_j.a[vDate][rDate]$. With this new model questions like “At the moment in time $rDate \in \text{Timestamp}$, what was the value we thought attribute a (of RS \mathcal{RS}_j) had at date $vDate \in \text{Timestamp}$, and which one we do know it is now?” can be answered. Figure 2 shows a schematic view of the evolution of the versions and revisions of a RS \mathcal{RS}_j through time. In particular, the evolution of the RS state in the time interval $[v1, v2]$ is shown. Here the evolution is caused by the correction of a hypothetical mistake. Before temporal point $r1$, the number of employees in the RS was 7, but after $r1$ a correction was made updating that value to 8. The tracking of these events is a basic requirement of the policies and claims subsystems.

3.1.3 Dangers that Affect RSs

In short, the risks subsystem allows users to manage the sets \mathcal{A} , \mathcal{RG} and \mathcal{RS} . Therefore, the framework provided by ARMISTICE can be easily adapted to a big variety of business

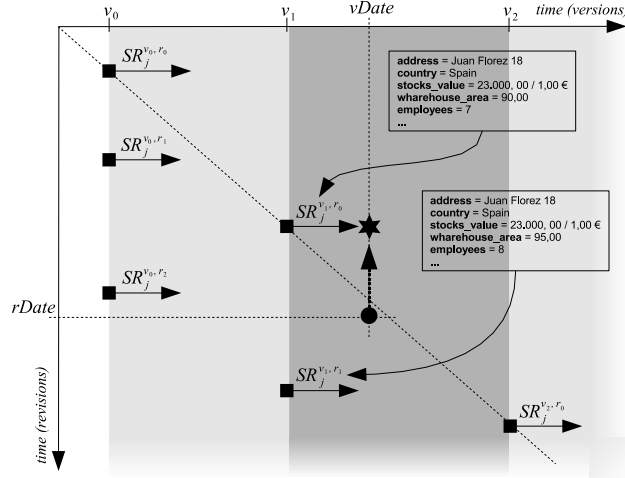


Figure 2. Versions and revisions of a RS

models. However, the risks subsystem is not only a tool for modelling the state on any insurable element. That is the key feature, but there is an additional and very important concept: the *dangers*. A *danger* can act over a RS causing a damage or accident. This issues need to be modelled too. Therefore, the system allows the management of a set of dangers \mathcal{D} (e.g. fire, explosion or terrorism). Each business model must make a standard list of its set of dangers adapting it to its particularities, to ease policy modelling.

3.2 Policies

Policies are the most complex element in the system. They group and link every business object. ARMISTICE works over a set of policies \mathcal{P} , where each policy \mathcal{P}_i is modelled as a set of renewals \mathcal{P}_i^{rj} . A *renewal* represents a new policy created to provide coverage to a new set of RSs in a new temporal interval. At the same time, a renewal is composed by a set of supplements $\mathcal{P}_i^{rj, sk}$. The first supplement in a renewal ($\mathcal{P}_i^{rj, s0}$) is called *emission supplement*. After the creation of the emission supplement, the system allows the creation of other supplements. A *supplement* represents a revision of the policy to change its coverage, to change its contractual clauses, to change its relevant dates, etc.

A supplement represents the minimal element that can be used to give coverage to a claim. Here, a temporal modelling is applied, like in section 3.1.2, but with three dimensions or temporal axis: each supplement $\mathcal{P}_i^{rj, sk}$ is parametrised by starting and ending validity dates, an activation date and a timestamp.

$$\begin{aligned} \mathcal{P}_i &\equiv (\text{Number}, \{\mathcal{P}_i^{r0}, \mathcal{P}_i^{r1}, \dots, \mathcal{P}_i^{r\alpha-1}\}) \\ \mathcal{P}_i^{rj} &= (\text{ReceiptModel}, \text{Receipts}, \{\mathcal{P}_i^{rj, s0}, \mathcal{P}_i^{rj, s1}, \dots, \mathcal{P}_i^{rj, s\beta-1}\}) \\ \mathcal{P}_i^{rj, sk} &\equiv (\text{Start}, \text{End}, \text{Activation}, \text{Timestamp}, \text{Coverage}, \text{Condicional}) \\ \text{Start}, \text{End}, \text{Activation}, \text{Timestamp} &\in \text{Timestamp} \wedge \text{Coverage} \subseteq \mathcal{RS} \end{aligned}$$

3.2.1 Conditional Clauses

A conditional is the key object to implement the ARMISTICE help decision support system. It is used when a user is looking for coverage to any claim. The supplement

conditional is a model of the contractual clauses of a specific policy. Thus, it is linked to a supplement $\mathcal{P}_i^{r_j, s_k}$. In particular, the system is only interested in the model of the policy coverage, that is to say, in the model of the policy *warranties* (e.g. the warranty against natural risks, its application preconditions, its franchise and its limits). Then, a conditional \mathcal{C}_i can be represented as,

$$\mathcal{C}_i = \{g_0, g_1, \dots, g_n\} \forall x \in [0, n - 1] g_x \equiv (Res, Fra, GrlL, PSinL, PSitL, AgrL)$$

Where *Res* is an applicability precondition (section 3.2.4) and *Fra*, *GrlL*, *PSinL*, *PSitL* and *AgrL* are formulas (section 3.2.3) used to calculate the franchise and limits (general, per sinister, per RS and aggregated). Restrictions and formulas are used when the system looks for coverage to a claim. These expressions can access to the context where they are evaluated and they can use temporal information about RSs, policies and other claims.

3.2.2 ARMISTICE as a Help Decision Support System

From the point of view of Artificial Intelligence, a mapping between the elements of a classic productions system and the components of ARMISTICE can be done. First, we identify the *knowledge base*, composed by the database of rules (i.e. applicability restrictions of the warranties of the conditionals) and the *facts* (i.e. claims, state of RSs and policies, both with its associated temporal tracking). The combinations of facts and rules selects the relevant (i.e. activated) rules. Next, the *active memory* contains the intermediate results obtained during execution, together with external world information introduced by the user, activated rules, etc. In ARMISTICE, a rule can be activated, not activated at all or activated partially under requiring additional information from the user. Finally, the *inference engine* (i.e. formulas and restrictions analysers) drives the whole process of suggesting coverage, franchises and limits.

3.2.3 Formulas

Formulas, together with the RSs meta-description (i.e. RGs) and the set of dangers \mathcal{P} , are key elements to build a flexible and adaptable RMIS framework. The formulas language contains basic, conditional and grouping operators, constructors to access properties of the RSs considering temporal restrictions, facilities to access the internal structure of policies and to access internal counters, user parameters, composition facilities using libraries of elementary formulas, etc. These functionalities allow users to model receipts, franchises and limits.

Next, a formula modelling the franchise of a warranty is shown. The value of the franchise is equal to the cost of the sinister if it is less than 300€. Otherwise, the franchise is the 20% of the cost, with an upper limit equal to the value of the average stocks in the affected RSs. In this example, two input user parameters can be identified: `#min`, which will be equal to 300 and `#pct` which will be equal to 20. This abstraction is useful to reuse the same formula in different scenarios (i.e. pattern). A grouping operator (`avg`), a counter (`sin.covered`) and an accessor to the value of a property of a set of RSs (stocks) with a temporal restriction (`sr.pol.sup.add`, which selects the last revision of the version at policy creation date) can also be identified.

```
if(sin.covered < #min, sin.covered, min(#pct * sin.covered, avg(sr.pol.sup.add(%stocks))))
```

3.2.4 Restrictions

As has been explained earlier (section 3.2.2), restrictions are the conditions to be hold by a warranty (i.e. inference rule) to be activated. Therefore, restrictions are a basic tool for modelling policies behaviour. The restrictions language is a superset of the formulas language, where logical operators and the concept of *nuance* are added. A *nuance* is a building block of a restriction. It is expressed using natural language and can only be evaluated by a human user. Then, the evaluation process of a restriction can be seen as a simplification of the tree which encodes a contractual clause: the system removes all the nuances as far as possible (that is to say, using the context information). The result will be a tree with a logical combination of nuances that must be evaluated by a human user.

The following example shows a trivial restriction. It contains two elements from the dangers set \mathcal{P} and two nuances. In a context where the activated danger would be a fire, the expression will be simplified to a question to be answered by the user: "Arson?".

`(fire and "Arson") or (earthquake and "Intensity > 4")`

4 Conclusions

In this paper we have presented the ARMISTICE kernel. We have stressed the extensibility and adaptability to any business model of the proposed framework. We have left out other features like the integration of the framework in a global document and report management system and its fault tolerant and scalable architecture.

Nowadays, ARMISTICE is being successfully deployed in the risk management department of a large holding enterprise. Its adaptability has been proved as it has been possible to model easily all the insurable elements, policies and dangers. Moreover, the formalism introduced by the need of that modelling has made possible to detect small mistakes in the original composition of the policy clauses. To sum up, the ARMISTICE adaptability and modelling tasks are a differential factor regarding other RMIS systems. Finally, the deployment process has revealed some framework improvements like the definition of dynamic properties in the RGs or the definition of a standardised language to encode all the behaviour of a policy.

References

- [1] Aon, 2005. <http://www.aon.com>.
- [2] STARS, 2005. <http://www.starsinfo.com>.
- [3] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
- [4] David Cabrero, Carlos Abalde, Carlos Varela, and Laura Castro. ARMISTICE: An experience developing management software with erlang. *PLI'03. 2nd ACM SIGPLAN Erlang Workshop*, pages 23–28, August 2003.
- [5] Martin Fowler. *Design patterns*, 2005. <http://www.martinfowler.com>.
- [6] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.
- [7] Philip Wadler. Functional programming: An angry half dozen. *SIGPLAN Notices*, 33(2):25–30, February 1998.