Implementing Privacy with Erlang Active Objects

Andreas Fleck and Florian Kammüller
Technische Universität Berlin
Software Engineering Group
andreasfleck@web.de, flokam@cs.tu-berlin.de

Abstract

Functional active objects are a new paradigm for the implementation of services. They offer safe distributed evaluation with futures and immutable objects guaranteeing efficient implementation of privacy while offering verified quality assurance based on the functional paradigm and a development in an interactive theorem prover. In this paper, we present a novel and highly performant implementation of functional active objects in Erlang. Besides outlining the guiding principles of the interpreter, we show by concrete examples how secure services can be realized.

1. Introduction

The free lunch is over – as Sutter describes so vividly in his famous paper [12]. In all realms of modern computing, we need to distribute to keep up performance. Active objects combine the successful concept of object-orientation with the necessary concepts of concurrent processes to make them fit for this distributed world.

We present an implementation of the novel language ASP$_{\text{fun}}$ of *functional active objects* in the programming language Erlang. Besides the highly performant parallelization of Erlang, this approch supports privacy enhancing implementations for web-services. The main contributions of this novel implementation are as follows.

- Functional active objects enable a deadlock free evaluation that implies service invocation in a higher order fashion. That is, a customer can invoke a service without needing to provide his private data.
- The use of *futures* as the machinery behind method invocation enables a flexible reply to method requests. In particular, this reply mechanism supports the privacy enhancing result acquisition described in the previous point.
- Using Erlang as implementation language we present a novel future implementation concept where each future is represented as a process. Thereby, we can abstract from different future evaluation strategies; the Erlang ASP$_{\text{fun}}$ interpreter stays close to the original semantics (see Section 2.1): since it is functional it is not forced to sacrifice generality for the sake of operationality.

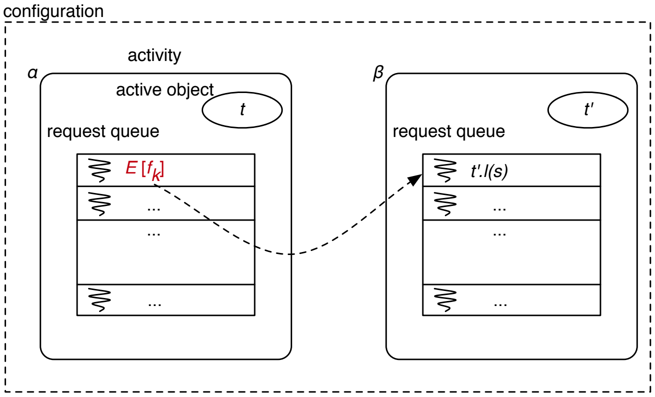

Figure 1. ASP$_{\text{fun}}$: a configuration

In this paper we first provide the prerequisites of this project: brief introductions to the language ASP$_{\text{fun}}$ and Erlang (Section 2). From there, we develop the concepts of our implementation of active objects in Erlang (Section 3). We then illustrate how the language can be efficiently used to implement secure services on three examples from privacy reflecting our contribution (Section 4). We finally offer conclusions and an outlook (Section 5).

2. Prerequisites

2.1. Functional Active Objects

The language ASP$_{\text{fun}}$ [6] is a computation model for functional active objects. Its local object language is a simple ς-calculus [1] featuring method call $t.l(s)$, and method update $t.l := \varsigma(x, y)b$ on objects (ς is a binder for the self x and method parameter y). Objects consist of a set of labelled methods $[l_i = \varsigma(x, y)b]^{i \in 1..n}$ (attributes are considered as methods with no parameters). ASP$_{\text{fun}}$ now simply extends this basic object language by a command *Active(t)* for creating an activity for an object t. A simple configuration containing just activities α and β within which are so-called active objects t and t' is depicted in Figure 1. This figure also illustrates *futures*, a concept enabling asynchronous communication. Futures are promises for the results of remote method calls, for example in Figure 1, f_k points to the location in activity β where at some point the result of the method evaluation $t'.l(s)$ can be retrieved from. Futures are first class citizen but they are not part of the *static* syntax of ASP$_{\text{fun}}$, that is, they cannot be used by a programmer. Similarly, activity references, e.g. α, β, in Figure 1, are references and not part of the static syntax. Instead, futures and activity references constitute the machinery for the computation of configurations of active objects.

The semantics of the ς-calculus is simply given by the following two reduction rules for calling and updating a method (or field) of an object. We use a concise contextual description with contexts E defined as usual.

CALL

$$\frac{l_i \in \{l_j\}^{j \in 1..n}}{\begin{array}{l} E\left[[l_j = \varsigma(x_j, y_j)b_j]^{j \in 1..n}.l_i(b)\right] \rightarrow_\varsigma \\ E\left[b_i\{x_i \leftarrow [l_j = \varsigma(x_j, y_j)b_j]^{j \in 1..n}, y_j \leftarrow b\}\right] \end{array}}$$

UPDATE

$$\frac{l_i \in \{l_j\}^{j \in 1..n}}{\begin{array}{l} E\left[[l_j = \varsigma(x_j, y_j)b_j]^{j \in 1..n}.l_i := \varsigma(x, y)b\right] \rightarrow_\varsigma \\ E\left[[l_i = \varsigma(x, y)b, l_j = \varsigma(x_j, y_j)b_j^{j \in (1..n)-\{i\}}]\right] \end{array}}$$

The semantics of ASP$_{\text{fun}}$ is built over the local semantics of the ς-calculus as a reduction relation $\rightarrow_\parallel$ that we call the paralellel semantics (see Table 1). In the following example (an extension of the motivating example of [6]) a customer uses a hotel reservation service provided by a broker.

$$\begin{array}{ll}
 & \quad\text{customer}[f_0 \mapsto \text{broker.book}(name, date, limit), t] \\
 & \parallel \text{broker}[\varnothing, [\text{book} = \varsigma(x, (name, date, limit)) \\
 & \qquad\qquad\qquad\qquad \text{hotel.room}(name, date), \ldots]] \\
 & \parallel \text{hotel}[\varnothing, [\text{room} = \varsigma(x, name, date)\text{bookingref}, \ldots]] \\
\rightarrow_\parallel^* & (\text{REQUEST, LOCAL}) \\
 & \quad\text{customer}[f_0 \mapsto f_1, t] \\
 & \parallel \text{broker}[f_1 \mapsto \text{hotel.room}(name, date), \ldots] \\
 & \parallel \text{hotel}[\varnothing, [\text{room} = \varsigma(x, name, date)\text{bookingref}, \ldots]] \\
\rightarrow_\parallel^* & (\text{REQUEST, LOCAL}) \\
 & \quad\text{customer}[f_0 \mapsto f_1, t]] \\
 & \parallel \text{broker}[f_1 \mapsto f_2, \ldots] \\
 & \parallel \text{hotel}[f_2 \mapsto \text{bookingref}, \ldots] \\
\rightarrow_\parallel^* & (\text{REPLY}^*) \\
 & \quad\text{customer}[f_0 \mapsto \text{bookingref}, t] \\
 & \parallel \text{broker}[f_1 \mapsto f_2, \ldots] \\
 & \parallel \text{hotel}[f_2 \mapsto \text{bookingref}, \ldots]
\end{array}$$

The last step summarizes two reply rule applications: the broker first replies the future reference f_2 to the customer as a result of f_1; second, the hotel replies the value bookingref directly to customer as result of f_2 without passing it to broker. This example illustrates how privacy can be implemented through future evaluation. We see just one possible order of evaluation; the ASP$_{\text{fun}}$ semantics also allows others. The Erlang interpreter presented in this paper implements the semantics of ASP$_{\text{fun}}$ in its generality but offers at the same time the possibility to configure different evaluation strategies that may then – like in the above example – be used to realize privacy.

2.2. Erlang

Erlang is a concurrent-oriented functional programming platform for open distributed telecommunication (OTP) systems developed by Ericsson corporation. It implements the actor paradigm by providing message passing as strategy for communication between several actors implemented as processes. Processes run fully parallel in Erlang. Each process has a mailbox where arriving messages are stored. The programmer can use pattern matching for message selection. Hence, the behavior of an actor is controllable. If a process needs an answer its process identifier (PID) has to be passed through the message. Since memory sharing does not exist, neither locks nor mutexes are necessary. The code is grouped in modules which are refered to by their name. So `modulname:functionname(args).` starts a function from a specific module.

A process is created by the `spawn`-command supplying it with the process' function and initial arguments. Erlang supports also named processes. Using `register(Name,PID)` the PID is registered in a global process registry and the process can be called by its name.

```
PID = spawn(Func,Args),
PID!Message,
Func(Args)...
receive
 Pattern1 [when Guard1] -> Expression1;
 Pattern2 [when Guard2] -> Expression2;
  ...
end.
```

Above, we show the basics of distribution in Erlang. First, we start a new process which runs the function `Func`. Then, we send a `Message` to the new process which is identified by `PID`. The function `Func` implements several patterns for incoming messages. Now, the system tries to match the arrived message against `Pattern1` (and the guard if exist). In case of success, `Expression1` is evaluated. If the first pattern fails, the second will be used, and so on. Another fundamental feature of Erlang is the single assignment, as in algebra, meaning that Erlang variables are immutable.

The main data types are (untyped) lists and records, called tuple, for example {`green, apple`} and atoms which represent different non-numerical constant values. Any lower case name is interpreted as an atom, any higher case name is a variable. In addition, there are modules for interoperability to other programming languages like C, Java or databases.

3. An ASP$_{\text{fun}}$ Interpreter in Erlang

Active objects bridge the gap between parallel processes and object-orientation. Intuitively, we want an object to be a process at the same time; unfortunately the two concepts are not identical. Hence, *activities* are introduced as a new notion of process containing an active object together with its current method execution(s).

LOCAL

$$\frac{s \to_{\varsigma} s'}{\alpha[f_i \mapsto s :: Q, t] :: C \to_{\parallel} \alpha[f_i \mapsto s' :: Q, t] :: C}$$

ACTIVE

$$\frac{\gamma \notin (\mathrm{dom}(C) \cup \{\alpha\}) \qquad \mathit{noFV}(s)}{\alpha[f_i \mapsto E[\mathit{Active}(s)] :: Q, t] :: C \to_{\parallel} \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, s] :: C}$$

REQUEST

$$\frac{f_k \text{ fresh} \qquad \mathit{noFV}(s)}{\alpha\,[f_i \mapsto E[\beta.l(s)] :: Q, t] :: \beta[R, t'] :: C \to_{\parallel} \alpha\,[f_i \mapsto E[f_k] :: Q, t] :: \beta\left[f_k \mapsto t'.l(s) :: R, t'\right] :: C}$$

SELF-REQUEST

$$\frac{f_k \text{ fresh} \qquad \mathit{noFV}(s)}{\alpha\,[f_i \mapsto E[\alpha.l(s)] :: Q, t] :: C \to_{\parallel} \alpha\,[f_k \mapsto t.l(s) :: f_i \mapsto E[f_k] :: Q, t] :: C}$$

REPLY

$$\frac{\beta[f_k \mapsto s :: R, t'] \in \alpha[f_i \mapsto E[f_k] :: Q, t] :: C}{\alpha[f_i \mapsto E[f_k] :: Q, t] :: C \to_{\parallel} \alpha[f_i \mapsto E[s] :: Q, t] :: C}$$

UPDATE-AO

$$\frac{\gamma \notin (\mathrm{dom}(C) \cup \{\alpha\}) \qquad \mathit{noFV}(\varsigma(x,y)s) \qquad \beta[Q, t'] \in (\alpha[f_i \mapsto E[\beta.l := \varsigma(x,y)s] :: Q, t] :: C)}{\alpha[f_i \mapsto E[\beta.l := \varsigma(x,y)s] :: Q, t] :: C \to_{\parallel} \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, t'.l := \varsigma(x,y)s] :: C}$$

Table 1. ASP$_{\text{fun}}$ semantics

In this section we describe how the concepts of activities, active objects and futures are realized on the infrastructure of Erlang; each concept resides in a separate module.

3.1. Activity

The first module describes the functionality of an *activity*. An *activity* encapsulates a functional active object to prevent direct access to it and manage requests simultaneously. In a functional language there is no need to make a sequential plan for execution in contrast to imperative active objects [4]. All requests are executed in parallel and run in individual processes. In our Erlang interpreter, the *activity* is implemented as a separate process which calls a run-function in a loop. Following the ASP$_{\text{fun}}$ semantics, an activity contains a request-queue and the functional active object which are also the arguments to the run-function. We use the Erlang built-in functionality for send and receive. This comes in quite naturally to model asynchronous communication of activities. In fact – as we will see when implementing futures (see Section 3.3) – message passing is the correct foundation for asynchronous communication with futures.

To hold the activity alive, the run-function is called again after each receive. Any request has to be sent as a tuple {Caller_PID,request,RequestFunction,Args} where Caller_PID is the PID or registered name of the calling process (see Section 3.3), the constant request which is used as pattern in the receive evaluation, the name of a requested function, and optionally the arguments as tuple or nil. An activity is now started in Erlang by

```
ActiveObject_PID = activeObject:start(),
activity:start(Identifier,ActiveObject_PID).
```

where ActiveObject_PID is the process identifier of the functional active object to be introduced next.

3.2. Functional Active Object

The second module specifies the functionality of *functional active objects*. The active object stores the ς-calculus methods. Deviating from the original notion of immutable objects of ASP$_{\text{fun}}$, our Erlang implementation is a dynamic ASP$_{\text{fun}}$-interpreter: ς-calculus methods can be added or deleted on the fly. Methods can also be declared at runtime or even be specified in separate modules. In our implementation, a functional active object is a process-in-the-loop which communicates with its activity by message passing. The activity requests a function using its name and the functional active object returns the function to the activity as a reply if it exists. This fact allows additionally separate distribution of the activity and the active object. To start an empty functional active object in our Erlang active object interpreter, we just call the following function.

```
ActiveObject_PID = activeObject:start().
```

To add functionality to this initial functional active object one can define own functions or use existing Erlang modules.

```
Foo = fun(Self,{arg1,arg2,...})
  -> some calculation, return value
end.
ActiveObject_PID!{add_func,functionname, Foo}.
```

where functionname is the name which one can use in other activities. To enable functions as return values, it is important to add a Self-Parameter. This parameter is set automatically by our system when distributing functions.

3.3. Future

The last module represents the implementation of *futures*. Futures act traditionally as placeholder for later results calculated in parallel [4]. We decided to expand the functionality by implementing active messaging. This fact allows us to use the future also as a kind of "proxy for communication" between activities. In Erlang, the future is represented by a separate process and therefore it is possible to communicate with the future easily by using the built-in messaging functionality to make the future active. This happenstance allows a complete separation from the activity

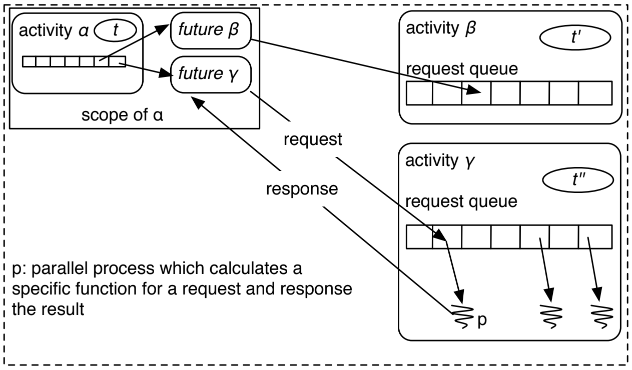

Figure 2. ASP$_{\text{fun}}$-Erlang: communication flow

in a parallel manner and presents several advantages. The first is the location of future creation. In our implementation, the future is created by the enquiring activity and not by the requested activity. This augments the privacy of activities. No activity has to announce itself to others when remote calculation is needed. The future asks the remote activity for the requested calculation and waits for the response: it is a "proxy for communication". In Figure 2, we illustrate this communication flow. In our opinion, this approach is the consequent continuation of an asynchronous communication concept (cf [11]).

In some existing approaches [4], a future is created and immediately returned by the called activity (in pseudo-code).

```
Future localfuture = activity_anywhere_in_www.foo()
```

However, this call is not really asynchronous because the remote activity might not respond immediately or the message is lost. As a result, the local activity also blocks. A really asynchronous solution must therfore use messages instead of return-values [11]. Our approach works as follows.

```
Future localfuture = new Future,
localfuture.start(activity_anywhere_in_www.foo())
```

First, the future is created in the scope of a calling activity and, then, the communication with the remote activity starts through the future by messages in an asynchronous manner. A new future-activity-request in Erlang may be started as follows.

```
newfuture = future:start(activity,functionname,args)
```

This function call creates a new future, sends a request message with the identifier of the new future, the function's name and the arguments to `activity` (see Section 3.4). The final value of this local function is the process identifier of the future.

Another benefit of our approach is that we can distribute a future by its network identifier so it is unique in the complete configuration. Therefore, there is no need to plan the update-process because it is a one to one relation between the called activity and the corresponding future.

3.4. Function Execution and Evaluation

All functions which are calls through activities are runing completely parallel in their own processes. By contrast, imperative active object systems like ProActive or others [5] use a sequentializing process and the execution runs in the thread of the activity. A further benefit is that activities do not freeze in case a function execution blocks. If a function blocks, only the future evaluation blocks. Therefore, we have built a second argument into our evaluation function representing the maximal evaluation time. After that timespan the evaluation returns nil. If and when the result is ready, the requested activity, that is, the calculating process, sends a message with the result to the calling future. The future takes the result as a new argument for the loop-function and waits for evaluation which starts by

```
Result = future:evaluate(Future, 10).
```

Since the evaluation can occur at any time, we have implemented two different cases:

- if the result is finished, it will be returned,
- and, if the result is not ready, the evaluation-process blocks until the future is updated (wait-by-necessity and finished after the update).

The result for self can be an ordinary value (tuple, atom, variable, etc.), a function (higher-order), an activity (a `PID` or name) or again a future. In the first three cases, the result is returned. If the result is itself a future, the evaluation function calls the evaluation function of the future and returns this result.

4. Secure Services in ASP$_{\text{fun}}$

In this section, we will now come back to the running example of a hotel-broker-service and show that our Erlang active object interpreter can model different possible scenarios. Note, that these scenarios are consistent with the ASP$_{\text{fun}}$ semantics given in Section 2. They define just different strategies corresponding to various privacy goals. We first show the classical evaluation order where service results flow back via the invocation structure to the customer. We then additionally sketch two refinements, where first the actual service result is passed privately to the customer so he can communicate directly with the hotel without passing data through the broker. Next, we show that our Erlang active object interpreter makes full use of the functional support: a customer can use a service by only providing partial information. Thereby, he can guard private information and still get some (information about the) service.

Thus, our Erlang active object interpreter represents an implementation of ASP$_{\text{fun}}$ in its broadest sense. Various different more "operational" semantics corresponding to different security policies can be easily implemented in our Erlang active object framework. For professional use, the basic machinery presented in this paper needs to be equipped with a mechanism for a simplified control over the different strategies (see Section 5).

4.1. Classic Service Evaluation

First, we show how the ASP$_{\text{fun}}$ example from Section 2 looks in "standard" form. Therefore, we define a function `room` where we use the ordinary Erlang syntax including the named specifications.

```
Room=fun(Self,{Name,Date})->
BookingRef= database:any_database_call(Name,Date),
BookingRef
end.
```

This function calls a function at a local database module. This can take some time. Next, we create a new active object, add the created function, and instantiate an activity named `hotel` which encapsulates the active object.

```
AO_Hotel = activeObject:start(),
AO_Hotel!add_func,room,Room,
activity:start(ActHotel,AOHotel),
```

Thereafter, we define the `broker` in the same manner, with the exception that the `book` function uses the `room` function of `hotel` and returns a future.

```
Book = fun(Self,{Name,Date,Limit})->
... find a hotel by Limit -> return an activity hotel
FutBookingRef=future:start(hotel,room,{Name,Date}),
FutBookingRef
end,
```

The newly created future sends a message to `hotel` and waits for an answer with result. Finally, we define the `customer`'s wish.

```
FutBookHotelRoom =
  future:start(broker,book,{myname,mydate,mylimit}),
BookingRef = future:evaluate(FutBookHotelRoom)
```

The arguments are the customer's name, the date on which he wants to book a room and the price limit. The created future `FutBookHotelRoom` sends a message to the activity `broker` which runs the function `book`. The function `book` also creates a future `FutBookingRef` to communicate with `hotel` and returns this to the future of `customer` – similar to the first application of the rule REPLY in the ASP$_{\text{fun}}$ example. After the `room` function has finished, the future `FutBookingRef` is updated. The evaluation of `FutBookHotelRoom` additionally calls an `evaluate` at `FutBookingRef` which returns the `BookingRef`. These two steps represent the second application of the rule REPLY in the ASP$_{\text{fun}}$ example. In the case that `FutBookingRef` is not updated yet, a wait-by-necessity occurs until `FutBookingRef` is ready. As in the original ASP$_{\text{fun}}$ example, the broker can evaluate the future `FutBookingRef` too because it is in the same scope. This means that the result is not passed directly to the broker but there is a potential risk (see Section 5).

4.2. Private Customer Negotiation

In the first extension the customer is only prepared to relinquish his limit and wants to keep his name and the date hidden from the `broker`. To make this possible we change the `book` function and add a case analysis.

```
Book = fun(Self,{Name,Date,Limit})->
...find a hotel by Limit -> return an activity hotel
case (Name == nil || Date==nil) of
 true ->
  whereis(hotel);
 false ->
  FutBookingRef=future:start(hotel,room,{Name,Date}),
  FutBookingRef
end
end,
```

So, if one of the arguments `Name` or `Date` is missing, the function returns the network identification of the activity `hotel`. The `customer` can now communicate with `hotel` directly.

```
FutBookHotelRoom =
    future:start(broker,book,{nil,nil,mylimit}),
ActHotel =future:evaluate(FutBookHotelRoom),
FutBookingRef =
    future:start(ActHotel,room,{myname,mydate}),
BookingRef = future:evaluate(FutBookingRef),
```

The evaluation of the future `FutBookHotelRoom` returns now an activity. The customer uses this activity to call the function `room` with his private data. In this example, the `broker` cannot read the private arguments of the `customer`.

4.3. Privacy by Partial Services

In the second extension, we show another way to implement privacy, now with distributed functions. This time, the `customer` shares the limit and the date. In the definition of `room`, we use a local database function. However, this fact does not allow to distribute this function. To make it again possible, we change the code slightly using our `Self`-operator. The case analysis is necessary because Erlang does not implement currying. Using existing implementations of currying functors, the following code could be further improved.

```
Room=fun(Self,{Name,Date})->
case Name == nil of
true ->
    NewFun = fun(MissingName) ->
            Args ={MissingName,Date},
            future:start(Self,room,Args)
            end;
false ->
   BookingRef= database:any_database_call(Name,Date),
   BookingRef
end
end.
```

In case argument `Name` is missing, a new function is defined which uses the existing argument `Date` and needs the missing `Name`. This function returns a new future which communicates also with hotel and uses both arguments the private Name and the public Date. The customer's wish looks now as follows.

```
FutBookHotelRoom =
    future:start(broker,book,{nil,mydate,mylimit}),
FunctionRoomByName=future:evaluate(FutBookHotelRoom),
FutBookingRef = FunctionRoomByName(myname),
BookingRef = future:evaluate(FutBookingRef),
```

The evaluation of FutBookHotelRoom returns the function NewFun which is defined in room and awaits Name as argument. The execution of this function returns a new future which is evaluated by customer.

5. Conclusions

We have introduced functional active objects, their implementation in Erlang, and how the Erlang active object framework can be employed to support privacy in web-services. In earlier work, we have used Erlang to support privacy for data enquiries [7]. The current work spells out an alternative approach already discussed in this earlier paper.

Our implementation of futures is – in comparison – the most natural as we base it on message passing. Similar to the ideas recently expressed in Ambient Talk [11], the future is created by the asynchronous send. In other implementations, the future is the result of a remote method invocation and therefore not completely asynchronous: blocking can occur. The next difference to other future implementations is the fact that our future is more active. This means that the future is the active communicator between activities. In addition, this augments privacy: in the example above, the customer is always invisible for broker and hotel. In our current implementation, we decided to declare the future explicitly to show the concrete communication and information flows. Although possible for little examples, it represents a source of fault for complex programs. The idea to hide the complete asynchronous communication can be implemented as a further step. For the time being, it should be seen as a playground for evaluating different strategies. We believe our functional parallel approach even allows us to run activities with circular references without deadlock (because the circle is formed to a helix).

The three different examples show how privacy can be implemented by using futures and active objects. As shown above there are the possibilities to use intermediary activities which return futures of others requests. Furthermore, the result can be an activity allowing to break up the communication flow. In the last example, we show how functionality can be transferred/delegated. These basic concepts allow – in the context of web-services – to implement private services in a new manner. For example, the use of one generic service which can be cloned and loaded with private data by clients allows to create a one-to-one relation between a client and his "private" web service. The generic service and other private services are then excluded from the communication of one specific service client relationship [6].

In the first example in Section 4, we show a potential risk in the current implementation. This risk can be solved by implementing additional logic which redirects futures. We decide to implement just the basic form to show the given insecurity by using web services. In this example, the customer trusts the hotel but not the broker.

As already discussed in Section 4, when presenting our different strategies to support privacy, there is need to enable users to specify these strategies and consequently to enforce these privacy requirements based on our implementation. An interesting continuation of our project is to build some modular kit for constructing such specification, using basic concepts similar to Myers' Decentralized Label Model (DLM) [8]. Thereby, we could label ASP$_{\text{fun}}$ programs with (parts of) security policies that could then be enforced by our privacy strategies.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer, New York, 1996.

[2] J. Armstrong. *Programming Erlang – Software for a Concurrent World.* The Pragmatic Bookshelf, 2007.

[3] H. Baker and C. Hewitt. The Incremental Garbage Collection of Processes. *Symposium on Artificial Intelligence Programming Languages.* SIGPLAN Notices 12, 1977.

[4] D. Caromel and L. Henrio. *A Theory of Distributed Objects.* Springer-Verlag, 2005.

[5] T. Gurrock. *A Concurrency Abstraction Implemented for C# and .NET.* Bachelor Thesis. Univ. Paderborn, 2007.

[6] L. Henrio and F. Kammüller. Functional Active Objects: Typing and Formalisation. *FOCLASA'09.* ENTCS, 2009.

[7] F. Kammüller and R. Kammüller. Enhancing Privacy Implementations of Database Enquiries. *Internet Monitoring and Protection.* IEEE, 2009.

[8] A. C. Myers and B. Liskov. Protecting Privacy using the decentralized label model. *TOSEM*, **9**:410–442, 2000.

[9] L. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1995.

[10] R. G. Lavender and D. C. Schmidt *An Object Behavioral Pattern for Concurrent Programming* http://www.cs.wustl.edu/ schmidt/PDF/Act-Obj.pdf

[11] E. Boix, T. Van Cutsem, J. Vallejos, W. De Meuter, and T. D'Hondt. A Leasing Model to Deal with Partial Failures in Mobile Ad Hoc Networks *TOOLS, 2009.*

[12] H. Sutter. The Free Lunch Is Over – A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, **30**(3), 2005.