Privacy by Flexible Parameterization with Erlang Active Objects

Andreas Fleck
Software Engineering Group
Technische Universität Berlin, Germany
Email: andreasfleck@web.de

Florian Kammüller
Middlesex University, London UK and
Technische Universität Berlin, Germany
Email: f.kammueller@mdx.ac.uk

***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 how secure services can be realized based on the classical service triangle and prove its security based on a formal definition of information flow security for functional active objects.**

***Keywords*-Active object, future, Erlang, privacy, service computing**

I. Introduction

The free lunch is over – as Sutter describes so vividly in his famous paper [24]. 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$_{\text{fun}}$ for *functional active objects* in the programming language Erlang. ASP$_{\text{fun}}$ is an computation model that can be seen as a descendant of the actor model, or more precisely active objects. Its main specialty is the use of *futures* to avoid blocking states while invoking asynchronous methods. Since no data is shared between active objects, concurrent method invocation can be simply used without fear of racing. These features are very similar to the features of services. Hence, it is possible to formalize complex services in ASP$_{\text{fun}}$ and this fact allows us to transfer ASP$_{\text{fun}}$ properties to services. The Erlang implementation of ASP$_{\text{fun}}$ enables a prompt transfer from an ASP$_{\text{fun}}$ configuration to executable code and so the "real behavior" can be tested. Besides the highly performant parallelization of Erlang, this approach supports privacy enhancing implementations for services. The main contributions presented in this paper 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 update strategies; the Erlang ASP$_{\text{fun}}$ interpreter stays close to the original semantics (see Section II-A): since it is functional it is not forced to sacrifice generality for the sake of operationality.
- We offer a formal definition of information flow security and illustrate its use for the proof of security on the service triangle – our running example.

In this paper we first provide the prerequisites of this project: brief introductions to the language ASP$_{\text{fun}}$, currying in ASP$_{\text{fun}}$, and Erlang (Section II). From there, we develop the concepts of our implementation of active objects in Erlang (Section III). We then illustrate how the language can be efficiently used to implement secure services on three examples from privacy reflecting our contribution (Section IV). A formal security definition enables the proof of privacy for flexible parameterization (Section V). We finally offer conclusions, position our work, and give an outlook on further plans (Section VI). This paper extends the original conference contribution [13] by further details on the implementation concepts and a formal definition of security and proof of privacy for the service triangle using flexible parameterization.

II. Prerequisites

In this section, we present the formal definitions of the language ASP$_{\text{fun}}$ and a brief introduction in the concepts behind Erlang.

A. Functional Active Objects

The language ASP$_{\text{fun}}$ [12] 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

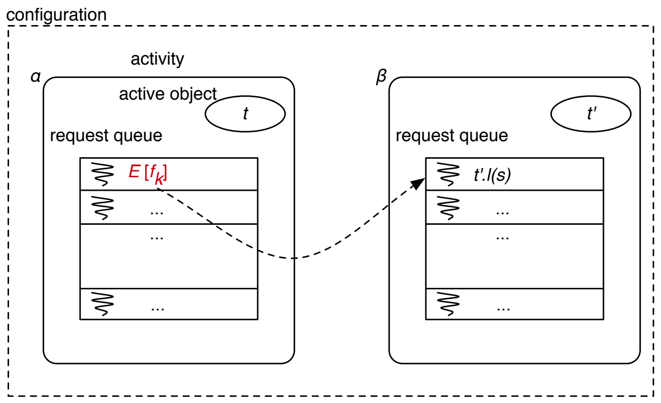

Figure 1. ASP$_{\text{fun}}$: a configuration

of labeled 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. Syntactically, we write configurations as $\alpha[R_\alpha, t_\alpha] \parallel \beta[R_\beta, t_\beta] \parallel \ldots$. For example, the configuration of Figure 1 would be syntactically expressed as $\alpha[f_0 \mapsto E[f_k] :: R_\alpha, t] \parallel \beta[f_k \mapsto t'.l(s) :: R_\beta, t']$.

B. Informal Semantics of ASP$_{fun}$

Local (ς-calculus) and *parallel* (configuration) semantics are given by a set of reduction rules informally described as follows.

- LOCAL: the local reduction relation $\rightarrow_\varsigma$ is based on the ς-calculus.
- ACTIVE: $Active(t)$ creates a new activity $\alpha[\emptyset, t]$ for new name α, empty request queue, and with t as active object.
- REQUEST: *method call* $\beta.l$ creates new future f_k in future-list of activity β (see Figure 1).
- REPLY: *returns result*, i.e. replaces future f_k by the referenced result term s (possibly not fully evaluated).
- UPNAME-AO: *activity upname* creates a copy of the activity and upnames the active object of the copy – the original remains the same (functional active objects are *immutable*).

C. Formal ASP$_{fun}$ semantics

We use a concise contextual description with contexts E defined as usual. Classically we define contexts as expressions with a single hole ($\bullet$).

$$E ::= \bullet \mid [l_i = \varsigma(x,y)E, l_j = \varsigma(x_j,y_j)t_j^{j\in(1..n)-\{i\}}] \mid E.l_i(t) \mid s.l_i(E) \mid E.l_i := \varsigma(x,y)s \mid s.l_i := \varsigma(x,y)E \mid Active(E)$$

$E[s]$ denotes the term obtained by replacing the single hole by s. The semantics of the ς-calculus for (local) objects is simply given by the following two reduction rules for calling and updating a method (or field) of an object.

CALL

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

UPDATE

$$\frac{l_i \in \{l_j\}^{j\in 1..n}}{\begin{array}{l}E\left[[l_j = \varsigma(x_j,y_j)b_j]^{j\in 1..n}.l_i := \varsigma(x,y)b\right] \rightarrow_\varsigma \\ E\left[[l_i = \varsigma(x,y)b, l_j = \varsigma(x_j,y_j)b_j^{j\in(1..n)-\{i\}}]\right]\end{array}} \tag{2}$$

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 parallel semantics (see Table I).

D. A Running Example from Service Computing

In the following example (an extension of the motivating example of [11]), a customer uses a hotel reservation service provided by a broker. This simple example is representative for service oriented architectures; we refer to it also as the *service triangle*. In this triangle, the three activities hotel, broker, and customer are composed by $\parallel$ into a configuration. To simplify this example the broker's search for a hotel is omitted and we always consider the same hotel and in addition we abstract from the computation in hotel that produces the booking reference for the customer. We concentrate on the message passing implemented in futures to highlight the actual flows of information in the following evaluation sequence.

$$\begin{array}{l}\text{customer}[f_0 \mapsto \text{broker.book}(date), t] \\ \parallel \text{broker}[\emptyset, [\text{book} = \varsigma(x, (date))\text{hotel.room}(date), \ldots]] \\ \parallel \text{hotel}[\emptyset, [\text{room} = \varsigma(x,date)\text{bookingref}, \ldots]]\end{array}$$

The following step of the semantic reduction relation $\rightarrow_\parallel^*$ creates the new future f_1 in broker following rule REQUEST. According to LOCAL, this call is reduced and the original call in the customer becomes f_1.

$$\begin{array}{l}\text{customer}[f_0 \mapsto f_1, t] \\ \parallel \text{broker}[f_1 \mapsto \text{hotel.room}(date), \ldots] \\ \parallel \text{hotel}[\emptyset, [\text{room} = \varsigma(x,date)\text{bookingref}, \ldots]]\end{array}$$

The parameter x representing the *self* is not used but the call to hotel's method room with parameter *date* creates again by rule REQUEST a new future in the request queue of the

$$\textsc{local}\quad \frac{s \to_\varsigma s'}{\alpha[f_i \mapsto s :: Q, t] :: C \to_\parallel \alpha[f_i \mapsto s' :: Q, t] :: C} \tag{3}$$

$$\textsc{active}\quad \frac{\gamma \notin (\mathrm{dom}(C) \cup \{\alpha\}) \qquad \mathit{noFV}(s)}{\alpha[f_i \mapsto E[\mathit{Active}(s)] :: Q, t] :: C \to_\parallel \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, s] :: C} \tag{4}$$

$$\textsc{request}\quad \frac{f_k \text{ fresh} \qquad \mathit{noFV}(s)}{\alpha\,[f_i \mapsto E[\beta.l(s)] :: Q, t] :: \beta[R, t'] :: C \to_\parallel \alpha\,[f_i \mapsto E[f_k] :: Q, t] :: \beta\,[f_k \mapsto t'.l(s) :: R, t'] :: C} \tag{5}$$

$$\textsc{self-request}\quad \frac{f_k \text{ fresh} \qquad \mathit{noFV}(s)}{\alpha\,[f_i \mapsto E[\alpha.l(s)] :: Q, t] :: C \to_\parallel \alpha\,[f_k \mapsto t.l(s) :: f_i \mapsto E[f_k] :: Q, t] :: C} \tag{6}$$

$$\textsc{reply}\quad \frac{\beta[f_k \mapsto s :: R, t'] \in \alpha[f_i \mapsto E[f_k] :: Q, t] :: C}{\alpha[f_i \mapsto E[f_k] :: Q, t] :: C \to_\parallel \alpha[f_i \mapsto E[s] :: Q, t] :: C} \tag{7}$$

$$\textsc{update-AO}\quad \frac{\gamma \notin (\mathrm{dom}(C) \cup \{\alpha\}) \qquad \mathit{noFV}(\varsigma(x,y)s) \qquad \beta[Q, t'] \in (\alpha[f_i \mapsto E[\beta.l := \varsigma(x,y)s] :: Q, t] :: C)}{\alpha[f_i \mapsto E[\beta.l := \varsigma(x,y)s] :: Q, t] :: C \to_\parallel \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, t'.l := \varsigma(x,y)s] :: C} \tag{8}$$

Table I
ASP$_{\text{FUN}}$ SEMANTICS

hotel activity that is immediately reduced due to LOCAL to bookingreference where the index indicates that *date* has been used.

$$\begin{aligned} &\text{customer}[f_0 \mapsto f_1, t]] \\ \parallel\ &\text{broker}[f_1 \mapsto f_2, \ldots] \\ \parallel\ &\text{hotel}[f_2 \mapsto \text{bookingref}_{\langle \mathit{date} \rangle}, \ldots] \end{aligned}$$

Finally, the result bookingreference is returned to the client by two REPLY-steps: first the future f_2 is returned from the broker to the customer and then this client receives the bookingreference via f_2 directly from the hotel.

$$\begin{aligned} &\text{customer}[f_0 \mapsto \text{bookingref}_{\langle \mathit{date} \rangle}, t] \\ \parallel\ &\text{broker}[f_1 \mapsto f_2, \ldots] \\ \parallel\ &\text{hotel}[f_2 \mapsto \text{bookingref}_{\langle \mathit{date} \rangle}, \ldots] \end{aligned}$$

This configuration can be considered as the final one; at least the service has been finished. From the perspective of privacy, it is actually here that we would like to end the evaluation. Unfortunately, the future f_2 is also available to the broker. So, in an final step the broker can serve himself the bookingreference as well.

$$\begin{aligned} &\text{customer}[f_0 \mapsto \text{bookingref}_{\langle \mathit{date} \rangle}, t] \\ \parallel\ &\text{broker}[f_1 \mapsto \text{bookingref}_{\langle \mathit{date} \rangle}, \ldots] \\ \parallel\ &\text{hotel}[f_2 \mapsto \text{bookingref}_{\langle \mathit{date} \rangle}, \ldots] \end{aligned}$$

The abstract general semantics of ASP$_{\text{fun}}$ allows this privacy breach.

We introduce now a general way of enforcing privacy by not disclosing private data in the first place. We show that relying on the ASP$_{\text{fun}}$ paradigm guarantees that flexible parameterization can be used to use services in a private manner.

E. Currying for ASP$_{fun}$

The contribution of this paper is a concept more generally useful for privacy: *flexible parameterization* – enabling the use of service functions while not supplying all parameters. For example, in the European project SENSORIA the COWS calculus has been designed as an extension to the Pi-calculus to realize *correlation*, a similarly dynamic service concept [4].

We have implemented this technique in our Erlang prototype for ASP$_{\text{fun}}$ (see Section III) as a pragmatic extension of the base language. However, as we will show now, this feature on flexible parameterization can be constructed conservatively in ASP$_{\text{fun}}$ using *currying*.

Currying is a well known technique in functional programming to render functions with several parameters partially applicable. That is, the parameters of a curried function may be supplied one after the other, in each step returning a new function.

Recall the definition of curry and its inverse uncurry in the λ-calculus.

$$\begin{aligned} \mathsf{curry} &\equiv \lambda\ f\ p.\ f(\mathsf{fst}\ p)(\mathsf{snd}\ p) \\ \mathsf{uncurry} &\equiv \lambda\ f\ a\ b.\ f(a,b) \end{aligned}$$

Here, (a, b) denotes the product and fst and snd the corresponding projections on a and b respectively. This datatype is itself definable in terms of simpler λ-terms as follows.

$$\begin{aligned} (a,b) &\equiv \lambda\ f.\ f\ a\ b \\ \mathsf{fst}\ p &\equiv (\lambda\ x\ y.\ x) \\ \mathsf{snd}\ p &\equiv (\lambda\ x\ y.\ y) \end{aligned}$$

We recall these classic definitions in order to prepare the less intuitive definition of currying for the ς-calculus and hence for ASP$_{\text{fun}}$. In comparison to the ς-calculus, the base objects of ASP$_{\text{fun}}$ differ in that we explicitly introduce a second parameter to each method in addition to the *self*-parameter x. Therefore, when we emulate functions in our version of the ς-calculus we profit from this parameter and avoid roundabout ways of encoding parameters.[1] As a prerequisite for having several parameters, we need products. Compared to the above presented encoding of pairs in the λ-calculus, pairs in the ς-calculus can make use of the natural *record* structure of objects thus rendering a more intuitive notion of product as follows.

$$\begin{aligned}(a,b) &\equiv [\,\mathsf{fst} = \varsigma(x,y)a,\ \mathsf{snd} = \varsigma(x,y)b\,]\\ \mathsf{fst}\,p &\equiv p.\mathsf{fst}\\ \mathsf{snd}\,p &\equiv p.\mathsf{snd}\end{aligned}$$

We simulate currying of a method f of an object o that expects a pair p of type $\alpha \times \beta$ as second parameter, i.e.

$$o = [\,f = \varsigma(x,p).t\,]$$

by extending this object o with a second method f_C as follows.

$$\begin{aligned}\mathsf{curry}\,o \equiv\ & [\,f = \varsigma(x,p)o.f(p),\\ & f_C = \varsigma(x,a)[f' = \varsigma(y,b)x.f(a,b)]\,]\end{aligned}$$

F. 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 referred 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 it exists). 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.

III. 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).

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.

A. 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 [6]. All requests are executed in parallel and run in individual processes. In our Erlang interpreter, the *activity* is implemented as a separate process. Following the ASP$_{\text{fun}}$ semantics, an activity contains a request-queue and the functional active object which are dedicated to the process. 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 III-C) – message passing is the correct foundation for asynchronous communication with futures.

To keep the activity alive, the process is called again after each receive. Any request has to be sent as a tuple {`Caller_PID,request,RequestFunction,Args`} where

[1] In the ς-calculus the parameter has to be simulated by updating a separate field in an objects and that consequently needs to be attached to each object.

`Caller_PID` is the `PID` or registered name of the calling process (see Section III-C), 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.

B. Functional Active Object

The second module specifies the functionality of *functional active objects*. The active object stores the ς-calculus methods in an list, where they can identify by a given name. 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 also a process which communicates with its activity by message passing. The activity requests a function using its name and the functional active object replies the function to the activity 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.

C. Future

The last module represents the implementation of *futures*. Futures act traditionally as placeholder for later results calculated in parallel [6]. In the ASP$_{\text{fun}}$ computation model there is no need to describe the fashion of updating a future with the calculated value. Furthermore, the evaluation of a future is possible at any time. This means, the result can be an value or the current state of the function evaluation. These facts have to be considered while implementing ASP$_{\text{fun}}$. For example, the ”on demand evaluation” can be implemented in Erlang by a watching process for each calculation which stores the current state. This is not very efficient and in most cases unnecessary. For the future updating process there exist several approaches, such as message-based or forward-based updates [16]. These strategies have in common that they need to store the relations between activities and futures [27]. We decided to expand the functionality by implementing the future as a process which also stores the final value. This concept makes allows a complete separation from the activity in a parallel manner and presents several advantages. First, the future is more active and allows the use of message passing and the distribution by its network identifier. So a future is unique in the entire configuration and therefore, there is no need to plan the update-process because other activities can call the future directly by its network identifier to get the value. The second advantage 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 by using the future as a kind of “proxy for communication” between 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 [5]). In some existing approaches [6], a future is created and immediately returned by the called activity (in pseudo-code).

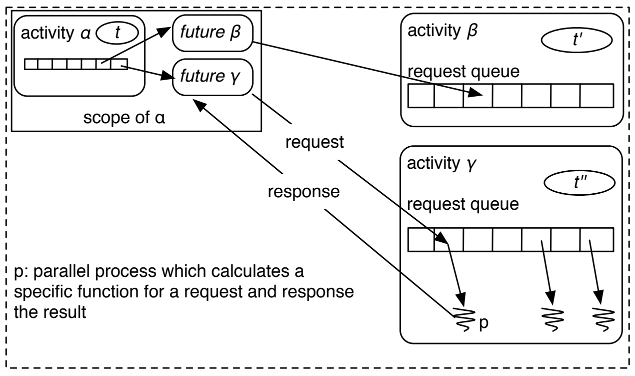

Figure 2. ASP$_{\text{fun}}$-Erlang: communication flow

```
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 therefore use messages instead of return-values [5]. 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 III-D). The final value of this local function is the process identifier of the future. A next advantage is the enforcement of security policies at the point of communication. The circumstance that the future is created by the calling activity and stores the final value (see Section IV-D) allows to enforce security policies of this activity for requests and responses at one single point.

D. Function Execution and Evaluation

All functions which are invoked in activities are running completely parallel in their own processes. By contrast, imperative active object systems like ProActive or others [9] 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 time span 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 get the result, stores it and waits for the evaluation by the activity 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 evaluate this future and returns this result.

IV. 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 II. 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 is passed to the customer so he can communicate directly with the hotel without passing private 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[2] 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.

A. Classic Service Evaluation

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

```
Room=fun(Self,{Date})->
BookingRef= database:any_database_call(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,{Date})->
... find a hotel by Date -> return an activity hotel
FutBookingRef=future:start(hotel,room,{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,{Mydate}),
BookingRef = future:evaluate(FutBookHotelRoom)
```

The arguments is the date on which he wants to book a room. 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

[2]For the sources `http://www.users.cs.tu-berlin.de/~flokam`

calls an `evaluate` at `FutBookingRef` which returns the `BookingRef`. These two steps represent the second application of the rule REPLY in the ASP_{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_{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 that he can retrieve it.

B. Private Customer Negotiation

In the first extension, the customer has an additional parameter `Name`, which should not be shared with the `broker`. So in the relation between this special customer and the untrusted broker our goal is to prevent the untrusted `broker` to read the private data. For another customer, the same broker could be a trusted partner. This behavior can be defined in individual security policies. To make these scenarios possible we change the `book` function and add a case analysis.

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

So, if the argument `Name` is missing, the function returns the network identification of the activity `hotel`. Now, the `customer` can communicate directly with the services of the `hotel`.

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

The evaluation of the future `FutBookHotelRoom` returns now an activity. The customer uses this activity to call the function `Room` directly with his private data. In this example, the `broker` cannot read the private argument of the `customer`. This strategy is simple and intuitive but needs the knowledge of the inner structure of the web-service. So the programmer needs to know the interfaces of actually hidden services. The difficulty grows with the complexity of the web-service.

C. Privacy by Partial Services

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

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

D. Subsumption

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. The concept of partial services is the base of a new kind of using complex partially trusted services. The fact that the name is just shared with hotels (or one of them) and that the date is public can be specified by the customer in an individual security policy. The implementation of an active future, the evaluation behavior of futures and the idea of currying allow to enforce these policies by using flexible parameterization in the future. The future knows the current communication partner and can share the arguments on the basis of the actual security policy. If the current call is untrusted, private data will not be shared.

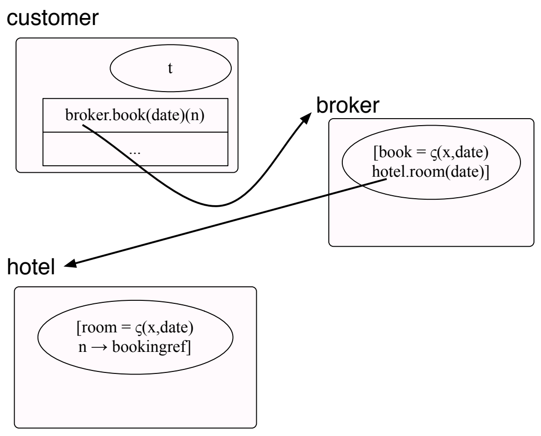

Figure 3. Private service scenario using currying

Then the behavior of currying will take effect and the result is a new function of a new communication partner and the description of them. So, the evaluation will run again and call the function with the required arguments, depending on the policy. For the case of the evaluation of a future by another activity, the future can also check the policy before returning the result. Another challenge is to avoid an information flow between requests inside an activity. This is prevented by Erlang through the no data sharing concept and in ASP$_{\text{fun}}$ through the rule UPDATE. 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 [11].

E. Service Triangle with Currying

To prepare a rigorous privacy analysis, we generalize the service triangle in an abstract fashion summarizing the various possible extensions seen above to a "bare bones" privacy scenario. Now, the function room in hotel has one additional parameter n for *name* besides date. This is to emphasize the privacy issue; wishing to keep your name n private seems natural; date, by contrast, can be considered as irrelevant, i.e. low. See Figure 3 for the setup configuration of this privacy scenario.

Now in the curried version, room can be called just supplying the first date parameter. The broker still delegates the partially instantiated request to the hotel (see Figure 4). Thereby, the customer can then directly access a function in hotel – via the futures f_1 and f_2 – that calculates his bookingref on supplying the missing parameter name (see Figure 5).

This intuitive idea of a curried version looks in the ASP$_{\text{fun}}$ representation as follows.

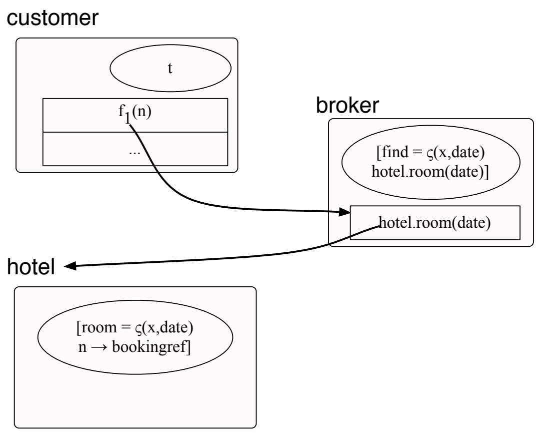

Figure 4. Partially instantiated request is delegated.

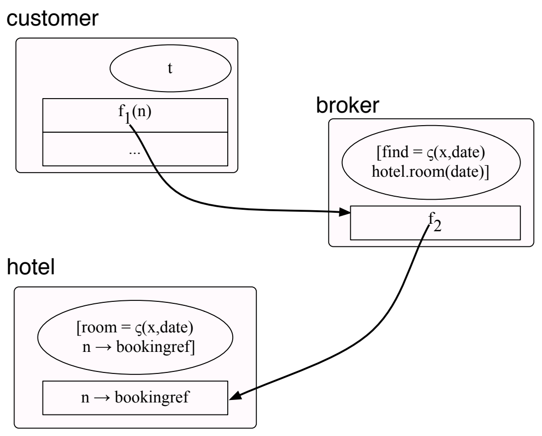

Figure 5. Customer may retrieve "bookingref"-function.

$$\begin{array}{l}\text{data}[\varnothing, [\text{id} = \text{hugo}]] \\ \parallel \text{customer}[f_0 \mapsto \text{broker.book}_C(d).\text{room'}(\text{data.id}), t] \\ \parallel \text{broker}[\varnothing, [\text{book}_C = \varsigma(x,d)\text{hotel.room}_C(d), \ldots]] \\ \parallel \text{hotel}[\varnothing, [\text{room} = \varsigma(x,(d,n))\text{bookingref}, \\ \qquad \text{room}_C = \varsigma(x,d)[\text{room'} = \varsigma(x',n)x.\text{room}(d,n)]]]\end{array}$$

It can reduce according to the semantics as shown in the subsequent configurations.[3] First, f_1 is created in data.

$$\begin{array}{l}\text{data}[f_1 \mapsto \text{hugo}, [\text{id} = \text{hugo}]] \\ \parallel \text{customer}[f_0 \mapsto \text{broker.book}_C(d).\text{room'}(f_1), t] \\ \parallel \text{broker}[\varnothing, [\text{book}_C = \varsigma(x,d)\text{hotel.room}_C(d), \ldots]] \\ \parallel \text{hotel}[\varnothing, [\text{room} = \varsigma(x,(d,n))\text{bookingref}, \\ \qquad \text{room}_C = \varsigma(x,d)[\text{room'} = \varsigma(x',n)x.\text{room}(d,n)]]]\end{array}$$

The call to book$_C$ creates future f_2,

[3]Clearly, there are also other possibilities to reduce, e.g. instead of evaluating data.id in the first step we could first reduce the call to book$_C$, thereby duplicating data.id. Since ASP$_{\text{fun}}$ is confluent up to identical copies, we always end with the same result.

$$\begin{array}{l}\text{data}[f_1 \mapsto \text{hugo}, [\text{id} = \text{hugo}]] \\ \| \text{ customer}[f_0 \mapsto f_2.\text{room'}(f_1), t] \\ \| \text{ broker}[f_2 \mapsto \text{hotel.room}_C(d), \\ \qquad [\text{book}_C = \varsigma(x,d)\text{hotel.room}_C(d), \ldots]] \\ \| \text{ hotel}[\varnothing, [\text{room} = \varsigma(x,(d,n))\text{bookingref}, \\ \qquad \text{room}_C = \varsigma(x,d)[\text{room'} = \varsigma(x',n)x.\text{room}(d,n)]]]\end{array}$$

which in turn produces a future f_3 in hotel where A is an abbreviation for the active object of hotel.

$$\begin{array}{l}\text{data}[f_1 \mapsto \text{hugo}, [\text{id} = \text{hugo}]] \\ \| \text{ customer}[f_0 \mapsto f_2.\text{room'}(f_1), t] \\ \| \text{ broker}[f_2 \mapsto f_3, [\text{book}_C = \varsigma(x,d)\text{hotel.room}_C(d), \ldots]] \\ \| \text{ hotel}[f_3 \mapsto [\text{room'} = \varsigma(x',n)\text{A.room}(d,n)], \\ \qquad [\text{room} = \varsigma(x,(d,n))\text{bookingref}, \\ \qquad \text{room}_C = \varsigma(x,d)[\text{room'} = \varsigma(x',n)x.\text{room}(d,n)]]]\end{array}$$

Now, we choose to reply first f_3 to customer,

$$\begin{array}{l}\text{data}[f_1 \mapsto \text{hugo}, [\text{id} = \text{hugo}]] \\ \| \text{ customer}[f_0 \mapsto f_3.\text{room'}(f_1), t] \\ \| \text{ broker}[f_2 \mapsto f_3, [\text{book}_C = \varsigma(x,d)\text{hotel.room}_C(d), \ldots]] \\ \| \text{ hotel}[f_3 \mapsto [\text{room'} = \varsigma(x',n)\text{A.room}(d,n)], \\ \qquad [\text{room} = \varsigma(x,(d,n))\text{bookingref}, \\ \qquad \text{room}_C = \varsigma(x,d)[\text{room'} = \varsigma(x',n)x.\text{room}(d,n)]]]\end{array}$$

whereby the customer can serve himself by f_3 the resulting object containing the room' method from hotel.

$$\begin{array}{l}\text{data}[f_1 \mapsto \text{hugo}, [\text{id} = \text{hugo}]] \\ \| \text{ customer}[f_0 \mapsto [\text{room'} = \\ \qquad \varsigma(x',n)\text{A.room}(d,n)].\text{room'}(f_1), t] \\ \| \text{ broker}[f_2 \mapsto f_3, [\text{book}_C = \varsigma(x,d)\text{hotel.room}_C(d), \ldots]] \\ \| \text{ hotel}[f_3 \mapsto [\text{room'} = \varsigma(x',n)\text{A.room}(d,n)], \\ \qquad [\text{room} = \varsigma(x,(d,n))\text{bookingref}, \\ \qquad \text{room}_C = \varsigma(x,d)[\text{room'} = \varsigma(x',n)x.\text{room}(d,n)]]]\end{array}$$

Finally, the customer receives by the rule REPLY the id hugo contained in f_1 (see Table I),

$$\begin{array}{l}\text{data}[f_1 \mapsto \text{hugo}, [\text{id} = \text{hugo}]] \\ \| \text{ customer}[f_0 \mapsto [\text{room'} = \\ \qquad \varsigma(x,n)\text{A.room}(d,n)].\text{room'}(\text{hugo}), t] \\ \| \text{ broker}[f_2 \mapsto f_3, [\text{book}_C = \varsigma(x,d)\text{hotel.room}_C(d), \ldots]] \\ \| \text{ hotel}[f_3 \mapsto [\text{room'} = \varsigma(x',n)\text{A.room}(d,n)], \\ \qquad [\text{room} = \varsigma(x,(d,n))\text{bookingref}, \\ \qquad \text{room}_C = \varsigma(x,d)[\text{room'} = \varsigma(x',n)x.\text{room}(d,n)]]]\end{array}$$

and we locally reduce the future f_0 in several steps with the rule for CALL of the ς-reduction (see Section II-A).

$$\begin{array}{l}\text{data}[f_1 \mapsto \text{hugo}, [\text{id} = \text{hugo}]] \\ \| \text{ customer}[f_0 \mapsto \text{bookingref}_{\langle\text{hugo}\rangle}, t] \\ \| \text{ broker}[f_2 \mapsto f_3, [\text{book}_C = \varsigma(x,d)\text{hotel.room}_C(d), \ldots]] \\ \| \text{ hotel}[f_3 \mapsto [\text{room'} = \varsigma(x',n)\text{A.room}(d,n)], \\ \qquad [\text{room} = \varsigma(x,(d,n))\text{bookingref}, \\ \qquad \text{room}_C = \varsigma(x,d)[\text{room'} = \varsigma(x',n)x.\text{room}(d,n)]]]\end{array}$$

It is crucial to observe that, even if the broker would have served himself the content of future f_3 he would not be able to produce the result bookingref$_{\langle\text{hugo}\rangle}$. Thus, the program with currying is secure with respect to a security assignment that marks broker as low, or an outsider, and thus protects the privacy of customer. Although the acquired privacy effect might now seem obvious, it is necessary to find a way to ascertain it rigorously. Therefore, we introduce in the following section a formal notion of security for ASP$_{\text{fun}}$ and use it to analyze the security of the server triangle.

V. Formal Security Proof

In this section, we present a formal analysis of security for the service triangle. We show that the classical solution is insecure while the one with currying is secure. These formal proofs apply a formal security definition for ASP$_{\text{fun}}$ to show that no information flows from the private domain of the client to the public server.

A. Noninterference Definition for ASP$_{fun}$

Intuitively, noninterference [26] means that an attacker cannot learn anything about private data by regarding public parts of a program. To arrive at a formal expression of this idea for ASP$_{\text{fun}}$, we first define a relation of indistinguishability, often also called L-equivalence because in this relation L-terms have to be equal.

We use here the notion of types informally because this suffices to disambiguate the following bijection. Indeed, ASP$_{\text{fun}}$ has a safe type system [11] that can serve here but is omitted for brevity.

Definition 5.1 (Typed Bijection): A typed bijection is a finite partial function σ on activities α (or futures f_k respectively) such that

$$\forall\, a : \text{dom}(\sigma).\ \vdash a : T \ \Rightarrow \vdash \sigma(a) : T$$

(where T is given by an activity type $\Gamma_{act}(\alpha)$ or a future type $\Gamma_{fut}(f_k)$ respectively).
The intuition behind typed bijections is that $\text{dom}(\sigma)$ designates all those futures or activity references that are or have been visible to the attacker. We cannot assume the names in different runs of programs, even for low elements, to be the same. Hence, we relate those names via a pair of bijections. These bijections are typed because they relate activities and futures that might need to be structurally equivalent, in case they are low. The following definition of indistinguishability uses the typed bijection in this sense.

We define (low)-indistinguishability as a relation $\sim_{\sigma,\tau}$ parameterized by two typed bijections one over activity names and one over futures. It is a heterogeneous relation as it ranges over elements of different types, for example activities and request queues. We leave out the types as they are indicated by our notational convention. By $t =_{\sigma,\tau} t'$ we denote the equality of terms up to replacing all occurrences of activity names α or futures f_k by their counterparts $\tau(\alpha)$

or $\sigma(f_k)$, respectively. The local reduction with $\to_\varsigma$ of a term t to a value t_e (again up to future and activity references) is written as $t \Downarrow t_e$.

Definition 5.2 (Indistinguishability): An indistinguishability relation is a heterogeneous relation $\sim_{\sigma,\tau}$, parameterized by two isomorphisms σ and τ whose differently typed subrelations are as follows.

$$\begin{array}{rcl} t \sim_{\sigma,\tau} t' & \equiv & t \Downarrow t_e \wedge t' \Downarrow t'_e \\ & & \wedge\ t_e =_{\sigma,\tau} t'_e \\ \alpha \sim_{\sigma,\tau} \beta & \equiv & \tau(\alpha) = \beta \\ f_k \sim_{\sigma,\tau} f_j & \equiv & \sigma(f_k) = f_j \\ [R_\alpha, t_\beta] \sim_{\sigma,\tau} [R_\alpha, t_\alpha] & \equiv & R_\beta \sim_{\sigma,\tau} R_\beta \wedge t_\alpha \sim_{\sigma,\tau} t_\beta \end{array}$$

$$R_\alpha \sim_{\sigma,\tau} R_\beta \quad \equiv \quad \begin{array}{l} \mathrm{dom}(\sigma) \subseteq \mathrm{dom}(R_\alpha) \wedge \mathrm{ran}(\sigma) \subseteq \mathrm{dom}(R_\beta) \\ \wedge \forall\, f_k \in \mathrm{dom}(\sigma).\ R_\alpha(f_k) \sim_{\sigma,\tau} R_\beta(\sigma(f_k)) \end{array}$$

$$C_0 \sim_{\sigma,\tau} C_1 \quad \equiv \quad \begin{array}{l} \mathrm{dom}(\tau) \subseteq \mathrm{dom}(C_0) \wedge \mathrm{ran}(\tau) \subseteq \mathrm{dom}(C_1) \\ \wedge \forall\, \alpha \in \mathrm{dom}(\tau).\ C_0(\alpha) \sim_{\sigma,\tau} C_1(\tau(\alpha)) \end{array}$$

The high part of the program is ignored for the above L-indistinguishability. That is, it is not part of the typed bijections σ and τ. Indistinguishability is for H-elements really something like "indistinguishability undefined".

Using indistinguishability we define now noninterference as *preservation* of "low"-indistinguishability between pairs of configurations. This is equivalent to saying that the indistinguishability relation is a (weak low)-*bisimulation* [19] over the configuration semantics. "Low" is the set of all elements (activities and futures) identified as low by the security assignment and hence in the domain of σ and τ. The definition of security of an ASP$_{\text{fun}}$ configuration is given with the following definition of noninterference.

Definition 5.3 (Noninterference): Two ASP$_{\text{fun}}$ configurations C_0 and C_1 are called *non-interfering* with respect to a security assignment *sp* represented by σ, τ, if whenever they are indistinguishable, i.e. $C_0 \sim_{\sigma,\tau} C_1$ and $C_0 \to_\parallel C'_0$ there exists a configuration C'_1 that $C_1 \to^*_\parallel C'_1$ and $C'_0 \sim_{\sigma,\tau} C'_1$. A configuration C is now called *secure* for *sp* if C and C_1 are non-interfering for all configurations C_1 with $C \sim_{\sigma,\tau} C_1$.

B. Classical Service Triangle is Insecure

Let us now show how the formal definition of information flow security, i.e. noninterference, is applied by reconsidering our running example of a service triangle.

$$C_0 \equiv \begin{array}{l} \mathrm{data}[\varnothing, [\mathrm{id} = \mathrm{hugo}]] \\ \parallel \mathrm{customer}[f_0 \mapsto \mathrm{broker.book}(d, \mathrm{data.id}), t], \\ \parallel \mathrm{broker}[\varnothing, [\mathrm{book} = \varsigma(x, (d, n)) \\ \qquad\qquad \mathrm{hotel.room}(d, n), \ldots]] \\ \parallel \mathrm{hotel}[\varnothing, [\mathrm{room} = \varsigma(x, (d, n))\mathrm{bookingref}, \ldots]] \end{array}$$

We want to protect the customer's privacy against the broker as reflected in the following security assignment *sp*.

$$sp \equiv (\{\mathrm{data}, \mathrm{customer}, \mathrm{hotel}\} \mapsto H, \mathrm{broker} \mapsto L\}, \{f_0 \mapsto L\})$$

According to the above assumption, the name `hugo` in `data` is thus confidential. To show that the above configuration, say C_0, is secure with respect to *sp*, we have to prove according to Definition 5.3, that all other configurations C_1 that are indistinguishable with respect to σ, τ containing sp, remain so under evaluation – or – fail in the attempt. In fact, as this example is insecure, we must fail. Let us consider an arbitrary configuration C_1 with $C_0 \sim_{\sigma,\tau} C_1$ as follows.

$$C_1 \equiv \begin{array}{l} \delta[\varnothing, [\mathrm{id} = \mathrm{ianos}]] \\ \parallel \gamma[g_0 \mapsto \beta.\mathrm{book}(d, \delta.\mathrm{id}), t], \\ \parallel \beta[\varnothing, [\mathrm{book} = \varsigma(x, (d, n)) \\ \qquad\qquad \alpha.\mathrm{room}(d, n), \ldots]] \\ \parallel \alpha[\varnothing, [\mathrm{room} = \varsigma(x, (d, n))\mathrm{bookingref}, \ldots]] \end{array}$$

Since C_1 is low-indistinguishable to C_0 with respect to *sp*, we can define σ and τ as a bijection of the low future and activity references of *sp*.

$$\begin{array}{rcl} \tau & \equiv & \{\texttt{broker} \mapsto \beta\} \\ \sigma & \equiv & \{f_0 \mapsto g_0\} \end{array}$$

Now, in three steps of evaluation of C_0 we reach the following configuration C'_0.

$$C'_0 \equiv \begin{array}{l} \mathrm{data}[f_1 \mapsto \mathrm{hugo}, [\mathrm{id} = \mathrm{hugo}]] \\ \parallel \mathrm{customer}[f_0 \mapsto f_2, \mathrm{hugo}), t], \\ \parallel \mathrm{broker}[f_2 \mapsto \mathrm{hotel.room}(d, \mathrm{hugo}), \\ \qquad [\mathrm{book} = \varsigma(x, (d, n))\ \alpha.\mathrm{room}(d, n), \ldots]] \\ \parallel \mathrm{hotel}[\varnothing, [\mathrm{room} = \varsigma(x, (d, n))\mathrm{bookingref}, \ldots]] \end{array}$$

A configuration C'_1 can be reached similarly from C_1.

$$C'_1 \equiv \begin{array}{l} \delta[g_1 \mapsto \mathrm{ianos}, [\mathrm{id} = \mathrm{ianos}]] \\ \parallel \gamma[g_0 \mapsto g_2, t], \\ \parallel \beta[g_2 \mapsto \alpha.\mathrm{room}(d, \mathrm{ianos}), \\ \qquad [\mathrm{book} = \varsigma(x, (d, n))\alpha.\mathrm{room}(d, n), \ldots]] \\ \parallel \alpha[\varnothing, [\mathrm{room} = \varsigma(x, (d, n))\mathrm{bookingref}, \ldots]] \end{array}$$

Since the future f_0 and g_0 are low, the new futures f_2 and g_2 are low, while the call to the high object data and δ has created the new futures f_1, g_1 as high. Now, σ can only be extended to $\sigma' = \sigma \cup (f_2 \mapsto g_2)$ since it must remain a bijection of the low futures. However, differing from the definition of indistinguishability we have for the request queues

$$R_{C'_0}(f_2) = \mathrm{hotel.room}(d, \mathrm{hugo}) \neq R_{C'_1}(g_2) = \alpha.\mathrm{room}(d, \mathrm{ianos})$$

whereby $\neg(C'_0 \sim_{\sigma,\tau} C'_1)$. Since no further reduction of C'_1 can remedy this, we know by Definition 5.3 that the configuration C_0 is *not secure*.

C. Curried Service Triangle is Secure

Now, we reconsider the same example but this time using the curried call to the booking function in the broker activity. We use the same security assignment $sp = (\{\text{data}, \text{customer}, \text{hotel}\} \mapsto H, \text{broker} \mapsto L\}, \{f_0 \mapsto L\})$ and abbreviate again *date* by d.

$$C_0 \equiv \begin{array}{l} \text{data}[\varnothing, [\text{id} = \text{hugo}]] \\ \parallel \text{customer}[f_0 \mapsto \text{broker.book}_C(d).\text{room'}(\text{data.id}), t] \\ \parallel \text{broker}[\varnothing, [\text{book}_C = \varsigma(x,d)\text{hotel.room}_C(d), \ldots]] \\ \parallel \text{hotel}[\varnothing, [\text{room} = \varsigma(x,(d,n))\text{bookingref}, \\ \qquad \text{room}_C = \varsigma(x,d)[\text{room'} = \varsigma(x',n)x.\text{room}(d,n)]]] \end{array}$$

Now, any other indistinguishable configuration, say C_1, would have at least elements corresponding to the low elements f_0 and broker of C_0 because otherwise the bijections σ and τ with $f_0 \in \text{dom}(\sigma)$ and broker $\in \text{dom}(\tau)$ were undefinable. In addition, the methods that are called in the low parts of C_1 must be defined even if they are contained in its high part. So, the least structure we have in an indistinguishable configuration C_1 is up to renaming as follows, unknown parts marked by dots.

$$C_1 \equiv \begin{array}{l} \delta[\ldots, [\text{id} = \ldots]] \\ \parallel \ldots [g_0 \mapsto \beta.\text{book}_C(d).\text{room'}(\delta.\text{id}), \ldots] \\ \parallel \beta[\varnothing, [\text{book}_C = \varsigma(x,d)\alpha.\text{room}_C(d), \ldots]] \\ \parallel \alpha[\ldots, [\ldots, \text{room}_C = \varsigma(x,d)[\text{room'} = \varsigma(x',n)\ldots]] \end{array}$$

The typed bijections are now $\sigma = \{f_0 \mapsto g_0\}$ and $\tau = \{\text{broker} \mapsto \beta\}$. The configuration C_0 can now make the steps we have seen in Section IV-E arriving at the following configuration C_0'.

$$C_0' \equiv \begin{array}{l} \text{data}[f_1 \mapsto \text{hugo}, [\text{id} = \text{hugo}]] \\ \parallel \text{customer}[f_0 \mapsto [\text{room'} = \\ \qquad \varsigma(x',n)\text{A.room}(d,n)].\text{room'}(\text{hugo}), t] \\ \parallel \text{broker}[f_2 \mapsto f_3, [\text{book}_C = \varsigma(x,d)\text{hotel.room}_C(d), \ldots]] \\ \parallel \text{hotel}[f_3 \mapsto [\text{room'} = \varsigma(x',n)\text{A.room}(d,n)], \\ \qquad [\text{room} = \varsigma(x,(d,n))\text{bookingref}, \\ \qquad \text{room}_C = \varsigma(x,d)[\text{room'} = \varsigma(x',n)x.\text{room}(d,n)]]] \end{array}$$

For any arbitrary C_0-indistinguishable configuration C_1 described above we know that we can arrive in a similar configuration C_1', where $\hat{\alpha}$ denotes α's active object.

$$C_1' \equiv \begin{array}{l} \delta[g_1 \mapsto \text{“value of id”} :: \ldots, [\text{id} = \ldots]] \\ \parallel \ldots [g_0 \mapsto [\text{room'} = \varsigma(x',n)\ldots].\text{room'}(g_1), \ldots] \\ \parallel \beta[g_2 \mapsto g_3, [\text{book}_C = \varsigma(x,d)\hat{\alpha}.\text{room}_C(d), \ldots]] \\ \parallel \alpha[g_3 \mapsto [\text{room'} = \varsigma(x',n)\ldots] :: \ldots, \\ \qquad [\ldots, \text{room}_C = \varsigma(x,d)[\text{room'} = \varsigma(x',n)\ldots]] \end{array}$$

This parallel reduction $C_1 \rightarrow_{\parallel} C_1'$ might have taken some more steps than $C_0 \rightarrow_{\parallel} C_0'$ but this is legal for a weak bisimulation. The bijection τ is updated to $\tau' \equiv \tau \cup \{f_2 \mapsto g_2\}$; $\sigma' = \sigma$. Since

$$R_{C_0'}(f_2) = f_3 \sim_{\sigma',\tau'} g_3 = R_{C_1'}(g_2)$$

we see that the resulting configurations are low-bisimilar, i.e.

$$C_0' \sim_{\sigma',\tau'} C_1'.$$

That is, the service triangle based on flexible parameterization using currying is secure with respect to the security assignment *sp*, i.e. preserves the privacy of customer's identity from the service broker.

D. Discussion

Concerning the enforcement of privacy policies in distributed application, the most successful and well-known approach is the Decentralized Label Model (DLM) of A. C. Myers [21]. It enables role based enforcement of program security. In the DLM, explicit labels are used to annotate elements of programs. These labels specify the *principals* that own those program elements as well as who has access to them. The DLM model is founded as our approach on the idea of information flow control; the labels serve as types in a noninterference type systems for static analysis of the allowed information flows. The main criticism to the DLM is that it *assumes that all principals respect the DLM*. We also consider this as a weakness in particular in distributed applications where assumptions about remote parties seems inappropriate. To illustrate this difference: in our example above the DLM would have assumed that the customer's call of book to the broker would also be high and thus be treated confidentially. Contrarily to this strong assumption of the DLM, we do *not make any assumptions about the low site*. In particular the customer can see everything in his request queue, be it marked high or low.

Following the earlier paper [15], we follow the philosophy that private information must not leave the trusted site. However, this is often too strong an assumption. Consider again our example as analyzed in this section. The additional parameter d for the *date* is considered to be low, thus not relevant to the security analysis. However, to avoid invalidating our security analysis, *date* needs to be constant in all applications! Otherwise, the broker would note a difference between changes on the high data. It might seem from this example that our notion of low-indistinguishability is too strict. However, in principle, it is correct to reject the application example if *date* can differ: since it is a parameter set by high, its visibility in broker represents a flow of information from high to low. To overcome this problem, we should consider a possibility to mark certain terms from high as “don't care” in our security model. That is, they are treated as low values, but their actual value is abstracted to make them admissible to low-equality. This represents a kind of down-grading of high values; in our example a “real” *date* of arbitrary value would be downgraded by anonymizing it to a default constant d before passing it on to the broker (this only for the analysis, the program remains unchanged). Then the call would pass as before as indistinguishable.

VI. 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: using flexible parameterization by currying, we could prevent illegal information flows of private data. This claim has been formally justified using a formal definition for noninterference for ASP$_{\text{fun}}$ functional active objects.

A. Related work

In earlier work, we have used Erlang to support privacy for data enquiries [15]. The current work follows the same basic philosophy used in this earlier paper as a naïve solution to the privacy problem in distributed settings: never let the private data leave the trusted home environment, instead import all data and search at home. We already discussed in [15] the implications for the more general setup of distributed active objects – now finally treated here.

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 [5], 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).

As already discussed in Section IV, 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. Concepts similar to Myers' Decentralized Label Model (DLM) [20] and the related Java implementation JIF based on type systems for information flow control are an adequate means to specify security policies for Java programs and make them amenable for (mostly) static analysis. However, as a prerequisite, a formal proof that typing implies the semantical notion of security, usually noninterference, is necessary. Myers has only lately come up with (partial) proof of soundness of JIF [25].

B. Positioning

For open communication systems, such as web-application and web-services where data and the computation are commonly distributed, there exist several approaches to enforce privacy, in particular the protection of private data. In the scope of local computation, *information flow control* [8], [7] – an approved method to protect data sharing in operating systems – is used more an more. This method, originally a centralized form of mandatory access control, has more recently been extended by decentralized aspects to concern distributed data. A practical implementation is Myer's JIF [21] that tags variables by labels representing owners and readers. The information of a variable is governed by the security policy that is expressed by these labels. A special compiler enforces the security policy by verifying that a program respects its policy. Between verified programs privacy is secured. As pointed out in Section V, we doubt that in a distributed setting it is reasonable to assume that activities stick to the rules. Therefore, our security model uses much weaker assumptions, and still enforces privacy using computational methods, like currying, to enforce data protection. The formal model of currying presented in Section II-E has already been presented in [14] but the notion of security presented there has been further refined in this paper. Here, it is a precise general bisimulation.

Another approach, which goes one step further, is the use of cryptographic protocols such as blind signatures and zero knowledge proof to hide or masquerade real private data. So data can be protected not just against outsider attacks but also against attacks caused by the communication partners. To confirm the cryptographic token a centralized verifier is needed. We consider this technique an important step forward because it can be used in addition to techniques as we use them to support the kind of "downgrading" we consider as necessary based on the security analysis in Section V. In that respect, work on integration of typing techniques for a static security analysis with cryptographically masked flows, e.g. [18], [17] serves the same purpose as our approach of protecting confidential parameters via flexible parameterization.

Complex distributed systems and the provided services are characterized by being often not completely verified. A centralized instance to enforce policies is not always possible or desired and sometimes sharing of private data is necessary. In the scope of intercommunication between the parts of a service, there are trusted and untrusted parts. Of course, programmers can differentiate between trusted and untrusted parts and split the communication process. But with the complexity level of the system, this task becomes more complicated. Our approach tries to implement privacy without a centralized policy instance, splitting communication processes but sharing private data with trusted parties. For the enforcement of these policies *inside* the local computation we can rely on existing components like JIF.

Nevertheless, being based on a formal computation model, i.e. ASP$_{\text{fun}}$, even distributed systems might be amenable to statically provable security as illustrated in a first approximation in this paper. With this goal in mind, our further plans are to define a security type system enabling static analysis of privacy of an ASP$_{\text{fun}}$ program as seen in this paper. The semantic security definition for ASP$_{\text{fun}}$ presented in this paper can be used as the basis for proving the correctness of this type system.

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] J. Bauer, F. Nielsen, H. Ries-Nielsen, and H. Pilegaard. Relational analysis of correlation. In *Static Analysis, 15th International Symposium, SAS'08*, volume 5079 of *LNCS*, pages 32–46. Springer, 2008.

[5] 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.*

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

[7] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 1977.

[8] D. E. Denning. Lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.

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

[10] L. Henrio and F. Kammüller. Functional active objects: Noninterference and distributed consensus. Technical Report 2009/19, Technische Universität Berlin, 2009.

[11] L. Henrio and F. Kammüller. Functional Active Objects: Typing and Formalisation. *8th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA'09*. Satellite to ICALP'09. ENTCS **255**:83–101, Elsevier, 2009.

[12] L. Henrio, F. Kammüller, and B. Lutz. ASP$_{\text{fun}}$: A Functional Active Object Calculus. *Science of Computer Programming*, Elsevier. In print, 2011.

[13] A. Fleck and F. Kammüller. Implementing privacy with Erlang active objects. *The 5th International Conference on Internet Monitoring and Protection, ICIMP'10*. IEEE, 2010.

[14] F. Kammüller. Privacy Enforcement and Analysis for Functional Active Objects. *Fifth International Workshop on Data Privacy Management, DPM'10.* Satellite to ESORICS'10. LNCS 6514, Springer, 2011.

[15] F. Kammüller and R. Kammüller. Enhancing Privacy Implementations of Database Enquiries. *The Fourth International Conference on Internet Monitoring and Protection.* IEEE, 2009. Extended version: Security Analysis of Private Data Enquiries in Erlang. *Int. Journal on Advances in Security*, **2**(2+3), 2009.

[16] M. Uzair Khan and L. Henrio. First class futures: a study of update strategies. Research Report RR-7113, INRIA, 2009.

[17] P. Laud. On the computational soundness of cryptographically masked flows. *35th Symposium on Principles of Programming languages, POPL'08*. ACM 2008.

[18] P. Laud and V. Vene. A type system for computationally secure information flow. *FCT'05*, volume 3623 of *LNCS*, Springer, 2005.

[19] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersey, 1989. SU Fisher Research 511/24.

[20] A. C. Myers. Jflow: Practical mostly-static information flow control. In *26th ACM Symposium on Principles of Programming Languages, POPL'99*, 1999.

[21] A. C. Myers and B. Liskov. Protecting Privacy using the decentralized label model. *Transaction on Software Engineering and Methodology, TOSEM* **9**:410–442, IEEE 2000.

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

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

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

[25] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2–3), 2007.

[26] J. Goguen and J. Meseguer. Security Policies and Security Models. *Proceedings of Symposium on Security and Privacy, SOSP'82*, pages 11-22. IEEE Computer Society Press, 1982.

[27] Muhammad Uzair Khan. *A Study of First Class Futures: Specification, Formalisation, and Mechanised Proofs* University of Nice, PhD Thesis, 2011.