Privacy Enforcement and Analysis for Functional Active Objects

Florian Kammüller

Technische Universität Berlin
flokam@cs.tu-berlin.de

Abstract. In this paper we present an approach for the enforcement of privacy in distributed active object systems, illustrate its implementation in the language $\mathrm{ASP}_{\mathrm{fun}}$, and formally prove privacy based on information flow security.

1 Introduction

The language $\mathrm{ASP}_{\mathrm{fun}}$ is a calculus of active objects. It has been developed in the interactive theorem prover Isabelle/HOL – in a fully formal co-development style [HK09]. $\mathrm{ASP}_{\mathrm{fun}}$ is a functional language using the Theory of Objects by Abadi and Cardelli for local object calculation but adding a second layer of distributed *activities* that communicate asynchronously via *futures*.

Activities are a unification of objects and processes having – like objects – a local data space while representing a unit of distribution containing a queue of currently processed calls to methods of this activity's object.

Futures are promises to the results of method calls. They realize asynchronous communication because an activity that calls a method in another activity immediately receives a future as a result. Thus, the calling activity can continue its current calculation. Only when the calling activity needs the actual result of the method call, a so-called *wait-by-necessity* could occur. In Proactive, an imperative Java implementation of distributed active objects for Grid-computing, this is indeed the case. In our functional computation model $\mathrm{ASP}_{\mathrm{fun}}$, however, we allow the return of possibly not fully evaluated methods calls. We completely avoid wait-by-necessity and possibly resulting dead-lock situations.

Moreover, we can fully prove in our Isabelle/HOL formalization that $\mathrm{ASP}_{\mathrm{fun}}$ is non-blocking. This proof actually comes as a by-product of the proof of type safety we have conducted in Isabelle/HOL for our $\mathrm{ASP}_{\mathrm{fun}}$ type system [HK09]. The suitability of $\mathrm{ASP}_{\mathrm{fun}}$ as a privacy enforcement language has already been demonstrated [Kam10]: based on the semantic primitive of active object *update*, an economic and clean implementation of data hiding is possible.

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 [BNRNP08].

In ASP$_{\text{fun}}$, we get this privacy enhancing use of services for free via the flexibility provided through *functional* replies. This idea has been tested through practical applications with a prototypical ASP$_{\text{fun}}$ implementation in Erlang [FK10]. Now, we formalize this idea and prove its security using the well-established method of information flow analysis. More precisely, our contribution consists in the following steps.

– We provide a security model for ASP$_{\text{fun}}$ for formal security analysis, i.e. a formal definition of noninterference for functional active objects.
– Based on the construction of a function-like currying for our object calculus, we provide a generic way of implementing flexible parameterization.
– Using the formal notion of noninterference, we prove that flexible parameterization is secure, and thus provides privacy.

As a global methodology we aim at providing a double-sided method complementing privacy enforcement with an accompanying analysis. To round off the paper, we present a concept for a modular assembly kit for security for ASP$_{\text{fun}}$ (in the style of Mantel's successful approach) that enables a systematic representation of language based security properties for distributed active object systems, thereby modeling privacy.

2 Functional Active Objects with ASP$_{\text{fun}}$

The language ASP$_{\text{fun}}$ [HK09] is a computation model for functional active objects. Its local object language is a simple ς-calculus [AC96] 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 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. ASP$_{\text{fun}}$ is built as a conceptual simplification of ASP [CH05] – both languages support the Java API Proactive [Pro08].

2.1 Informal Semantics of ASP$_{\text{fun}}$

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

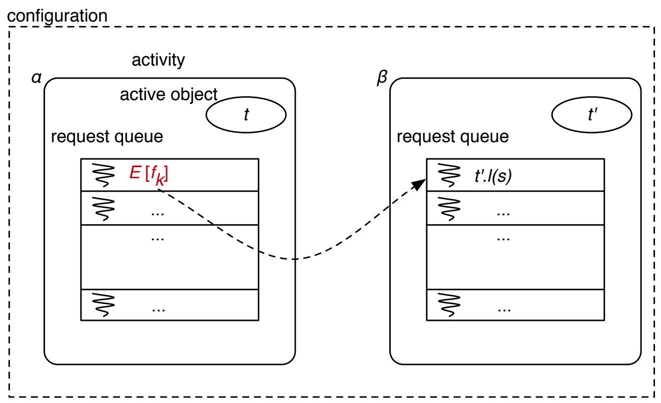


Fig. 1. ASP_{fun}: a configuration

- LOCAL: the local reduction relation $\to_\varsigma$ is based on the ς-calculus.
- ACTIVE: $\mathit{Active}(t)$ creates a new activity $\alpha[\varnothing, t]$ 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*).

2.2 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 ($\bullet$).

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

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

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

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

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

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

REQUEST

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

SELF-REQUEST

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

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

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

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

The semantics of $\mathrm{ASP}_{\mathrm{fun}}$ is built over the local semantics of the ς-calculus as a reduction relation $\to_{\parallel}$ that we call the parallel semantics (see Table 1).

2.3 Broker example

The following example illustrates the advantages of futures for the implementation of services. The three activities hotel, broker, and customer are composed by $\parallel$ into a configuration. The customer wants to make a hotel reservation in hotel. He uses a broker for this service by calling a method book provided in the active object of the broker. We omit the actual search of the broker in his database and instead hardwire the solution to always contact some hotel. That is, the method book is implemented as a call $\varsigma(x, \mathit{date})\mathrm{hotel.room}(\mathit{date})$ to a function room in the hotel. Also the internal administration of hotel is omitted; its method room just returns a constant bookingreference bookingref. The dependence of this bookingref on the parameter *date* exists only implicitly. Therefore, we denote it by $\mathrm{bookingref}_{\langle date \rangle}$ in instantiations. Initially, only the future list of the customer contains a request for a booking to broker; the future lists of broker and hotel are empty. The following steps of the semantic reduction relation $\to_{\parallel}$ illustrate how iterated application of reduction rules evaluates the program.

$$\begin{aligned}&\text{customer}[f_0 \mapsto \text{broker.book}(\mathit{date}), t]\\ \| \;&\text{broker}[\varnothing, [\text{book} = \varsigma(x, (\mathit{date}))\text{hotel.room}(\mathit{date}), \ldots]]\\ \| \;&\text{hotel}[\varnothing, [\text{room} = \varsigma(x,\mathit{date})\text{bookingref}, \ldots]]\end{aligned}$$

The following step of the semantic reduction relation $\rightarrow_{\|}^{*}$ creates the new future f_1 in broker by rule REQUEST, this call is reduced according to LOCAL, and the original call in the customer replaced by f_1.

$$\begin{aligned}&\text{customer}[f_0 \mapsto f_1, t]\\ \| \;&\text{broker}[f_1 \mapsto \text{hotel.room}(\mathit{date}), \ldots]\\ \| \;&\text{hotel}[\varnothing, [\text{room} = \varsigma(x,\mathit{date})\text{bookingref}, \ldots]]\end{aligned}$$

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

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

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.

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

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.

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

The abstract general semantics of ASP_{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_{fun} paradigm guarantees that flexible parameterization can be used to use services in a private manner.

3 Flexible parameterization

We use the example of the hotel broker again to show how flexible parameterization may be used to keep private data in one's secure local environment. As an informal security policy we assume that the client does not want to disclose his name to the broker. In Figure 2, we see the same scenario as in the previous section's example but with a slightly generalized program. Here, the function room in hotel has an additional parameter name besides date. However, room can be called just supplying the first date parameter. The broker still delegates the partially instantiated request to the hotel. 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. The broker can also access the resulting function but not the private data of the customer as it does not need to be transmitted. We have implemented this technique in our Erlang

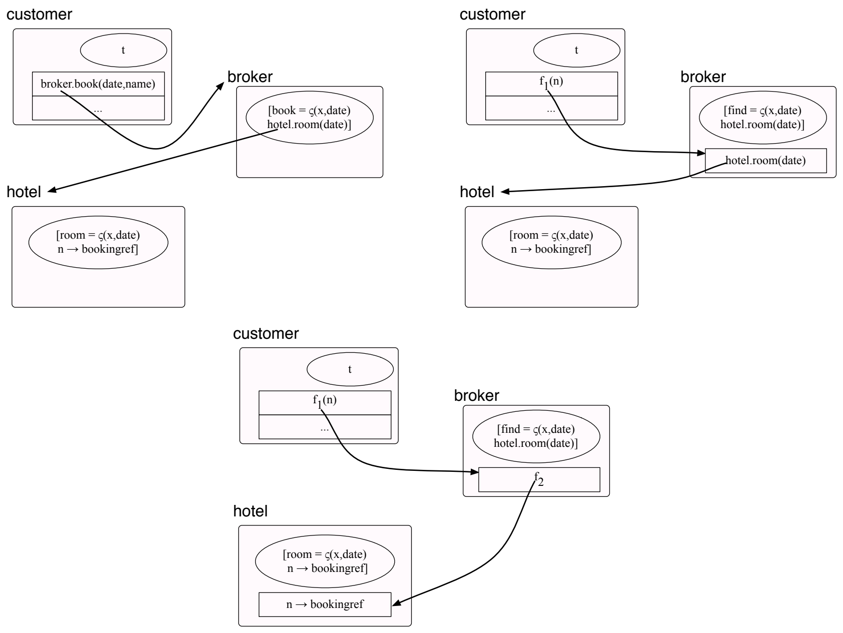

customer
t
broker.book(date,name)
...
broker
[book = ς(x,date) hotel.room(date)]
hotel
[room = ς(x,date) n → bookingref]
customer
t
f1(n)
...
broker
[find = ς(x,date) hotel.room(date)]
hotel.room(date)
hotel
[room = ς(x,date) n → bookingref]
customer
t
f1(n)
...
broker
[find = ς(x,date) hotel.room(date)]
f2
hotel
[room = ς(x,date) n → bookingref]
n → bookingref

Fig. 2. Flexible parameterization: delegation to partially instantiatable room method

prototype for $\mathrm{ASP}_{\mathrm{fun}}$ [FK10] as a pragmatic extension of the base language. However, as we will show now, this feature on flexible parameterization can be constructed conservatively in $\mathrm{ASP}_{\mathrm{fun}}$ 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$_{\text{fun}}$.

3.1 Currying in 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.

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

[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.

3.2 Hiding

We introduce formal definitions of hiding and restrictions that will be used later for the formal definition of strong noninterference. Hiding, at the object level, is an operation that takes a ς-object and a label and hides the methods that are identified by the label. Practically, it is realized by overwriting the field labeled h of a ς-object with the empty object $\langle\rangle$.

Definition 1 (Hiding for ς-terms).

$$\begin{aligned}
(o.m)(p) \setminus l &\equiv \ \text{if } m = l \text{ then } \langle\rangle \text{ else } (o \setminus l).m(p \setminus l) \\
[l_j = \varsigma(x_j, y_j)b_j]^{j \in 1..n} \setminus l &\equiv \begin{cases} [l_i = \varsigma(x, y)\langle\rangle, l_j = \varsigma(x_j, y_j)b_j]^{j \in 1..n-\{i\}}, & \text{if } l_i = l \\ [l_j = \varsigma(x_j, y_j)(b_j \setminus l)]^{j \in 1..n}, & \text{else} \end{cases} \\
(o.m := f) \setminus l &\equiv \ \text{if } m = l \text{ then } o.m := \langle\rangle \text{ else } (o \setminus l).m := f
\end{aligned}$$

To lift the hiding operator to configurations, we just have to introduce name spacing with respect to configuration names and map that to the previous hiding definition.

Definition 2 (Hiding for ASP$_{\text{fun}}$ terms). *For any configuration C hiding is defined as follows.*

$$(\alpha[Q, t] :: C) \setminus \Lambda.l \equiv \begin{cases} \alpha[Q, t] :: (C \setminus \Lambda.l) & \text{if } \alpha \neq \Lambda \\ \alpha[Q, t \setminus l] :: (C \setminus \Lambda.l) & \text{else} \end{cases}$$

In order to define a notion of security for ASP$_{\text{fun}}$, we define observational equivalence. This means informally that programs are considered to be secure if an attacker cannot tell apart the outcomes of the program when observing only those parts that are visible to him according to a security policy. For simplicity, we assume this security policy for ASP$_{\text{fun}}$ programs to be $S : C \to \{\Delta, \nabla\}$. That is, we assume that all activities of a configuration C can be divided into only two groups: the secure part represented by Δ (usually called high H) and the insecure part represented by Δ's complement $\nabla \equiv \text{dom}(C) - \Delta$ (or low L). [2] In general, this is not such an unrealistic assumption if one takes the viewpoint of the secure part: the rest of the world is either part of the secure domain or not.

Definition 3 (Security for ASP$_{\text{fun}}$). *Two configurations C_0, C_1 are equivalent for the attacker $C_0 =_\nabla C_1$ if their visible parts in ∇ are equal modulo renaming and local reduction with $\to_\varsigma$. A configuration C_0 is now called secure, if for any other configuration C_1 that is equivalent $C_0 =_\nabla C_1$, if $C_0 \to_\parallel^* C_0'$ and $C_1 \to_\parallel^* C_1'$ such that C_0', C_1' are values, then $C_0' =_\nabla C_1'$.*

[2] We use here $-$ instead of $\setminus$ for symmetric set difference because $\setminus$ is now used for hiding.

Equality up to renaming and local reduction needs a word of explanation. We assume two configurations to be equal from an attackers perspective if the “low” parts in ∇ are isomorphic: we cannot assume the names to be equal as they are generated in each run. However, given a bijection on these names, we can identify any two configurations in two runs of an ASP_{fun} program if their low parts are isomorphic. We further normalize the low equality relation module local reduction as two configurations should not be distinguishable if they differ only in the local evaluation of request queue entries and are otherwise isomorphic. Overloading syntax, we simply write $t \setminus \Delta$ to denote hiding the list of labels of all activities that are in Δ, i.e. that are considered to be high. Hiding is not a usual term operation: it takes labels as second argument, which are terms. Hence, the confluence result for terms, or the context rules [HK09], respectively, do not apply. That is, from $t \rightarrow_{\parallel} t'$ does not follow $t \setminus \Delta \rightarrow_{\parallel} t' \setminus \Delta$. In fact, this is the case if and only iff hiding is secure as we will see in the following section.

4 Noninterference via Hiding

4.1 Language Based Modular Assembly Kit for Security (LB-MAKS)

In this section we want to introduce a conceptual framework – MAKS [Man00] – enabling the comparison and proof of security properties. MAKS is based on labeled transition systems, a model too abstract for the consideration of certain security critical situations — like blockings.

MAKS is based on a labeled transition system model for systems. It defines a set of six *basic security predicates* defined as closures over set properties [Man02]. *Backwards strong deletion* (BSD) and *backwards strong insertion* (BSI) are two basic security predicates. The strongest security property in MAKS is given as the conjunction of *backwards strong deletion* and *backwards strong insertion* for the flow policy $\mathcal{V}_L^{LH}$ defined as $(L, \varnothing, H)$ – the classical policy: “information may flow from L to H but not vice versa”. This “root” property is not named but it forms the root of the tree formed by implications between the identified properties. On one branch from this root lie *separability*, *nondeducibility on inputs*, *perfect security property*, *noninference* (note this is *not* nonin*ter*ference) and on the other branch lie *forward correctability* and *generalized noninterference*. Both branches unite in the same property *generalized noninference* (without “ter”). The positions in this “implication” tree are implicitly proved since all these properties are expressed as conjunctions of basic security predicates.

Focardi and Gorrieri compare security properties based on trace semantics but already consider algebraic characterizations using their process algebra SPA [FG95]. Like us, motivated by the fact that bisimulation based approaches are more fine grained, they provide algebraic definitions of security properties. For example, the simple characterization of *strong nondeterministic noninterference* (SNNI) in [FG95] is as follows, where $|_\Delta$ is restriction and $\setminus\Delta$ hiding.

$$E \mid_\Delta \overset{\sim}{=} E \setminus \Delta \qquad (1)$$

We adapt the original syntax to avoid confusion. This relation $\overset{\sim}{=}$ can either represent a trace set equality or – for a more refined view – a bisimulation relation. This notion of strong nondeterministic noninterference is very concise but also a strong predicate. It corresponds to $BSD_{\mathcal{V}_L^{LH}}(Tr) \wedge BSI_{\mathcal{V}_L^{LH}}(Tr)$, the root of the MAKS tree [Man00]. The open question is: how does our language based characterization of security relate to the classical notions given by Mantel, Focardi and Gorrieri, and McLean? In the following, we show that we can enforce noninterference based on hiding in our language based model.

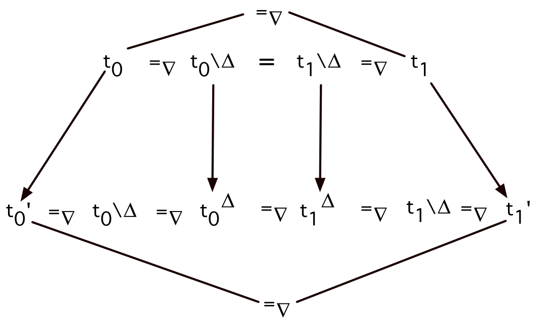
Hiding (see Section 3.2) can be used to enforce information flow control. In [Kam10], we showed that in some special cases hiding implies security. These cases use that sometimes hiding has no effect, i.e. $t \rightarrow_{\|}^* t'$ and $t \setminus \Delta \rightarrow_{\|}^* t'$.

4.2 Language based security characterization

The characterization of security by hiding we have given in [Kam10] somehow implicitly assumes that $t' = t \setminus \Delta$ This assumption expresses that in secure programs the hiding operation is compositional, i.e. $t \rightarrow_{\|} t' \Rightarrow t' \setminus \Delta = t \setminus \Delta$. In an attempt to transfer the hiding based noninterference by Focardi and Gorrieri to ASP$_{\mathrm{fun}}$, we derive the following theorem.

Theorem 1 (NI by Hiding). *Let t be an ASP$_{\mathrm{fun}}$ term representing a program and Δ be a security policy. If $t \rightarrow_{\|}^* t'$ implies $t \setminus \Delta \rightarrow_{\|}^* t^{\Delta}$ with $t^{\Delta} =_{\nabla} t' \setminus \Delta$, i.e. $\setminus$ is (low)-compositional for Δ, then t is secure in the sense of Definition 3.*

Proof. By definition of $=_{\nabla}$, $t =_{\nabla} t \setminus \Delta$ for any term t. Let $t_0 =_{\nabla} t_1$, $t_0 \rightarrow_{\|}^* t_0'$ and $t_1 \rightarrow_{\|}^* t_1'$ such that t_0', t_1' are values according to the hypotheses in the Security Definition 3.



$$\begin{tikzcd}
t_0 \arrow[rr, phantom, "=_\nabla"] \arrow[d] \arrow[rrrrr, dash, bend left, "=_\nabla"] & & t_0\setminus\Delta \arrow[r, phantom, "="] \arrow[d] & t_1\setminus\Delta \arrow[rr, phantom, "=_\nabla"] \arrow[d] & & t_1 \arrow[d] \\
t_0' \arrow[r, phantom, "=_\nabla"] \arrow[rrrrr, dash, bend right, "=_\nabla"'] & t_0\setminus\Delta \arrow[r, phantom, "=_\nabla"] & t_0^{\Delta} \arrow[r, phantom, "=_\nabla"] & t_1^{\Delta} \arrow[r, phantom, "=_\nabla"] & t_1\setminus\Delta \arrow[r, phantom, "=_\nabla"] & t_1'
\end{tikzcd}$$

Since $t_0 =_{\nabla} t_1$, by definition, $t_0 \setminus \Delta = t_1 \setminus \Delta$ up to isomorphism due to renaming. Since $t_0 \rightarrow_{\|}^* t_0'$ and $t_1 \rightarrow_{\|}^* t_1'$, we can apply the assumption to obtain $t_0 \setminus \Delta \rightarrow_{\|}^* t_0^{\Delta}$, $t_0^{\Delta} =_{\nabla} t_0' \setminus \Delta$ and $t_1 \setminus \Delta \rightarrow_{\|}^* t_1^{\Delta}$ and $t_1^{\Delta} =_{\nabla} t_1' \setminus \Delta$. Now, $t_0 \setminus \Delta = t_1 \setminus \Delta$ implies $t_0^{\Delta} =_{\rightarrow_{\|}^*} t_1^{\Delta}$ because of confluence of ASP$_{\mathrm{fun}}$which gives $t_0^{\Delta} =_{\nabla} t_1^{\Delta}$. Since $t_0^{\Delta} =_{\nabla} t_0' \setminus \Delta$ and $t_1^{\Delta} =_{\nabla} t_1' \setminus \Delta$ we get in turn by transitivity of $=_{\nabla}$ that $t_0' \setminus \Delta =_{\nabla} t_1' \setminus \Delta$ under the same renaming as before (or possibly a conservative extension due to creation of new low elements). This corresponds to $t_0' =_{\nabla} t_1'$ by definition of $=_{\nabla}$ and we are finished. □

Theorem 1 proves noninterference according to Definition 3. This is a less restrictive security as the notion of strong nondeterministic noninterference, as used by Focardi and Gorrieri [FG95]. But the criteria characterizing security in Theorem 1 corresponds to the original definition (1) [FG95]. What is more, ours is a real language based security notion for active objects in $\mathrm{ASP}_{\mathrm{fun}}$ and not abstract event systems. Conceptually, the correspondence provides an important link between the elegant world of event systems and the more tedious but fairer language based models. It marks the first important stepping stone for LB-MAKS, our envisaged security tool kit for $\mathrm{ASP}_{\mathrm{fun}}$. In the following section, we will evaluate our notion on the running example of the hotel broker.

4.3 Application example

In this section, we demonstrate how the security characterization given in Theorem 1 can be practically useful to statically verify security. We reconsider the hotel broker example from Section 2.3 to prove that it is insecure; we then show using the same method that the improved version using flexible parameterization by currying (Section 3) is secure. To model private data, we refine the initial configuration by introducing some private information. There is a data object containing the customer's identity "name" that he wants to keep private from the broker.

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

In several evaluation steps this program reduces to the following. Other than the insertion of the identity, the evaluation is little changed with respect to the earlier version in Section 2.3, apart from the fact that bookingref now also depends on the customer's id.

$$t' \equiv \begin{array}{l} \mathrm{data}[f_1 \mapsto \mathrm{id}, [\mathrm{name} = \mathrm{id}]] \\ \parallel \mathrm{customer}[f_0 \mapsto \mathrm{bookingref}_{\langle d,\mathrm{id}\rangle}, [\ldots]] \\ \parallel \mathrm{broker}[f_2 \mapsto \mathrm{bookingref}_{\langle d,\mathrm{id}\rangle}, [\ldots]] \\ \parallel \mathrm{hotel}[f_3 \mapsto \mathrm{bookingref}_{\langle d,\mathrm{id}\rangle}, [\ldots]] \end{array}$$

We want to prove security of enforcement via hiding for the example configuration given the security policy $\Delta = \{\mathrm{customer}, \mathrm{data}\}, \nabla = \{\mathrm{hotel}, \mathrm{broker}\}$. According to Theorem 1, we need to show compositionality of hiding with respect to private data labeled data.name since this is (for simplicity) the only data in Δ.

Applying hiding of customer's identity to the initial configuration t, i.e. $t \setminus \Delta$, erases the customer's name in data.

$$\left(\begin{array}{l} \quad \text{data}[\varnothing, [\text{name} = \text{id}]] \\ \parallel \text{customer}[f_0 \mapsto \text{broker.book}(\text{data.name}), t] \\ \parallel \text{broker}[\varnothing, [\text{book} = \varsigma(x,(d,n))\text{hotel.room}(d,n), \ldots]] \\ \parallel \text{hotel}[\varnothing, [\text{room} = \varsigma(x,(d,n))\text{bookingref}, \ldots]] \end{array}\right) \setminus \text{data.name} =$$

$$\begin{array}{l} \quad \text{data}[\varnothing, [\text{name} = \langle\rangle]] \\ \parallel \text{customer}[f_0 \mapsto \text{broker.book}(\text{data.name}), t] \\ \parallel \text{broker}[\varnothing, [\text{book} = \varsigma(x,(d,n))\text{hotel.room}(d,n), \ldots]] \\ \parallel \text{hotel}[\varnothing, [\text{room} = \varsigma(x,(d,n))\text{bookingref}, \ldots]] \end{array}$$

Finally, this $t \setminus \Delta$ reduces to the following t^{Δ}.

$$t^{\Delta} \equiv \begin{array}{l} \quad \text{data}[f_1 \mapsto \text{id}, [\text{name} = \langle\rangle]] \\ \parallel \text{customer}[f_0 \mapsto \text{bookingref}_{\langle d,\langle\rangle\rangle}, [\ldots]] \\ \parallel \text{broker}[f_2 \mapsto \text{bookingref}_{\langle d,\langle\rangle\rangle}, [\ldots]] \\ \parallel \text{hotel}[f_3 \mapsto \text{bookingref}_{\langle d,\langle\rangle\rangle}, [\ldots]] \end{array}$$

If we apply hiding "data.name" to the *reduced* configuration t' instead, we still erase the identity of the customer in data but do not erase the id that is now implicit in bookingref.

$$\left(\begin{array}{l} \quad \text{data}[f_0 \mapsto \text{id}, [\text{name} = \text{id}]] \\ \parallel \text{customer}[f_1 \mapsto \text{bookingref}_{\langle d,\text{id}\rangle}, [\ldots]] \\ \parallel \text{broker}[f_2 \mapsto \text{bookingref}_{\langle d,\text{id}\rangle}, [\ldots]] \\ \parallel \text{hotel}[f_3 \mapsto \text{bookingref}_{\langle d,\text{id}\rangle}, [\ldots]] \end{array}\right) \setminus \text{data.name} =$$

$$\begin{array}{l} \quad \text{data}[f_0 \mapsto \text{name}, [\text{name} = \langle\rangle]] \\ \parallel \text{customer}[f_1 \mapsto \text{bookingref}_{\langle d,\text{id}\rangle}, [\ldots]] \\ \parallel \text{broker}[f_2 \mapsto \text{bookingref}_{\langle d,\text{id}\rangle}, [\ldots]] \\ \parallel \text{hotel}[f_3 \mapsto \text{bookingref}_{\langle d,\text{id}\rangle}, [\ldots]] \end{array}$$

In this example program, hiding data.name is not preserved by the program evaluation. The program thus does not meet the prerequisites of Theorem 1. It is also intuitively clear, that this program is *not* secure as the broker has access to the private booking result of the customer.

Example with currying

By contrast, in the program that uses the curried version, hiding is respected by the program evaluation and therefore secure according to our Theorem 1.

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

This configuration reduces to the following, where H abbreviates the active object of hotel.

$$\begin{array}{l} \quad \text{data}[f_1 \mapsto \text{id}, [\text{name} = \text{id}]] \\ \| \text{ customer}[f_0 \mapsto f_3.\text{room'}(\text{id}), t] \\ \| \text{ broker}[f_2 \mapsto f_3, [\ldots]] \\ \| \text{ hotel}[f_3 \mapsto [\text{room'} = \varsigma(x,n)\text{H.room}(d,n)], [\ldots]]] \end{array}$$

Replying the semi-evaluated function via f_3 to customer and broker and reducing locally we get the personalized bookingref in customer's request list.

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

The crucial test is now to apply hiding first to the initial configuration t'.

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

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

Reducing this, we reach the following configuration t^{Δ}.

$$t^{\Delta} \equiv \begin{array}{l} \quad \text{data}[f_1 \mapsto \langle\rangle, [\text{name} = \langle\rangle]] \\ \| \text{ customer}[f_0 \mapsto \text{bookingref}_{\langle d, \langle\rangle \rangle}), t] \\ \| \text{ broker}[f_2 \mapsto [\text{room'} = \varsigma(x,n)\text{H.room}(d,n)], [\ldots]] \\ \| \text{ hotel}[f_3 \mapsto [\text{room'} = \varsigma(x,n)\text{H.room}(d,n)], [\ldots]]] \end{array}$$

Now, if we consider hiding Δ in the reduction of the previous configuration, i.e. $t' \setminus \Delta$ we get this configuration.

$$t' \setminus \text{data.name} \equiv \begin{array}{l} \quad \text{data}[f_1 \mapsto \langle\rangle, [\text{name} = \langle\rangle]] \\ \| \text{ customer}[f_0 \mapsto \text{bookingref}_{\langle d, \text{id} \rangle}), t] \\ \| \text{ broker}[f_2 \mapsto [\text{room'} = \varsigma(x,n)\text{H.room}(d,n)], [\ldots]] \\ \| \text{ hotel}[f_3 \mapsto [\text{room'} = \varsigma(x,n)\text{H.room}(d,n)], [\ldots]]] \end{array}$$

At first sight, this seems odd because $t^{\Delta} \neq t \setminus \Delta$ (they differ in f_0) but we have $t^{\Delta} =_{\nabla} t \setminus \Delta$. This suffices for the conclusion of Theorem 1. We have thus shown that this program is secure for Δ.

5 Conclusions

5.1 Related Work

Myers and Liskow augmented the DLM model with the idea of information flow control as described in the papers [ML00]. Further works by Myers have been mostly practically oriented, foundations only considered later [ZM07]. Initially, he implemented a Java tool package called JFlow, nowadays JIF, that implements his Decentralzied Label Model (DLM) based on information flow control [Mye99]. In more recent work, still along the same lines, Zheng and Myers [ZM07] have gone even further in exploring the possibilities to dynamically assign security labels while still trying to arrive at static information flow control. 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.

One very popular strand of research towards verification of security has been static analysis of noninterference using type checking. Briefly, this means that type systems are constructed that encode a noninterference property. When a program passes the type check for that system then we know that it has a certain security type. The security type can be for example an assignment of program data to security classes. The type check then guarantees that the information flows are only those allowed by the assignment of the data to the security classes. A good survey giving an introduction to the matter and comparing various activities is given by Sabelfeld and Myers [SM03]. A constructive way to support noninterference analysis is by providing a set of basic properties that can be combined to build various forms of noninterference. H. Mantel's work since his PhD has mainly focused on providing such a logical tool box [Man00,Man02].

Focardi and Gorrieri's work has strongly influenced our current approach. We take their method of algebraic characterization of information flow security further in applying them to the distributed object calculus ASP_{fun} thereby addressing real language issues.

5.2 Discussion and Outlook

We aim at using functional active objects as a language calculus and build a logical tool set for compositional properties. Since already on the simpler trace models the definition and theory development for a MAKS is an intrinsically complex task, we additionally employ Isabelle/HOL as a verification environment to support us. The derived LB-MAKS security properties shall be transformed into security type systems – in the sense as described in the previous section – to derive practically useful static analysis tools from our mechanized

LB-MAKS. The presented currying concept is a first important step towards such an enforcement library. A further important stepping stone is the definition of hiding for ASP_{fun}. It helps bridging the gap between abstract trace based semantics and more detailed language based models as illustrated in this paper.

To our knowledge no one has considered the use of futures in combination with confinement given by objects as a means to characterize information flow. The major advantage of our approach is that we are less abstract than event systems while being abstract enough to consider realistic distributed applications.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer-Verlag, New York, 1996.

[BNRNP08] 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.

[CH05] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects.* Springer-Verlag New York, Inc., 2005.

[FG95] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.

[FK10] A. Fleck and F. Kammüller. Implementing privacy with erlang active objects. In *5th International Conference on Internet Monitoring and Protection, ICIMP'10.* IEEE, 2010.

[HK09] L. Henrio and F. Kammüller. Functional active objects: Typing and formalisation. In *8th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA'09*, ENTCS. Elsevier, 2009. Also invited for journal publication in Science of Computer Programming, Elsevier.

[Kam10] F. Kammüller. Using functional active objects to enforce privacy. In *5th Conf. on Network Architectures and Information Systems Security, SAR-SSI 2010*, 2010.

[Man00] H. Mantel. Possibilistic definitions of security – an assembly kit. In *Computer Security Foundations Workshop*, pages 185–199. IEEE, 2000.

[Man02] H. Mantel. On the composition of secure systems. In *Symposium on Security and Privacy*, 2002.

[ML00] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9:410–442, 2000.

[Mye99] A. C. Myers. Jflow: Practical mostly-static information flow control. In *26th ACM Symposium on Principles of Programming Languages, POPL'99*, 1999.

[Pro08] ProActive API and environment, 2008. Available at `http://www.inria.fr/oasis/proactive` (under LGPL).

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications*, 21:5–19, 2003.

[ZM07] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2–3), 2007.