Using Functional Active Objects to Enforce Privacy

Florian Kammüller (flokam@cs.tu-berlin.de)*

Abstract: In this paper we present an important step towards a language based modular assembly kit for security. This kit aims at supporting analysis of information flow security for distributed systems. As a distributed language we use functional active objects in ASP_{fun}. The contribution of the paper is an implementation concept based on ASP_{fun} for information flow control by hiding and an argument that this device enforces security. This hiding device illustrates a privacy enhancing technique for distributed languages and motivates the need for a systematic analysis of noninterference properties from a language based perspective. A language based property assembly kit for distributed systems seems a desirable means to that end. The hiding mechanism is one device for security enforcement.

Keywords: Active Objects, Noninterference, Type Systems

1 Introduction

A formal statement of confidentiality is noninterference. One might even say that noninterference is the natural way of formally encoding confidentiality in a system. Over the years there has been a vast amount of different notion of noninterference nicely summarized by Mantel's modular assembly kit for security (MAKS) [Man02].

The language ASP_{fun} [HK09b] is a small language intended to support the study of the foundations of active objects; we can consider ASP_{fun} as a calculus of functional active objects. The language and a classical type system of ASP_{fun} are formalized in the interactive theorem prover Isabelle/HOL [NPW02]; type safety is proved completely using this tool. By type safety we mean preservation and progress. For the special case of distributed objects this directly implies deadlock-freedom [HK09b]. We furthermore provide an ASP_{fun} implementation in Erlang [FK10] that serves as an environment for experimenting with privacy enhancing techniques, particularly for asynchronous languages based on *future* evaluation.

The contribution of this paper is an explicit hiding operator for ASP_{fun} and a proof that hiding implies information flow control. We furthermore provide a vision on how such an operator contributes to a global understanding of information flow properties.

In this paper we first introduce briefly the language ASP_{fun} (Section 2) giving its formal semantics. The application of the language is then illustrated on the classical Webtax example that has originally been used to motivate the decentralized label model (DLM) by Andrew Myers' [ML00] (Section 3). Next, we introduce the hiding operator for ASP_{fun} by means of example showing how it introduces privacy into the Webtax example

* Technische Universität Berlin
Institut für Softwaretechnik und Theoretische Informatik

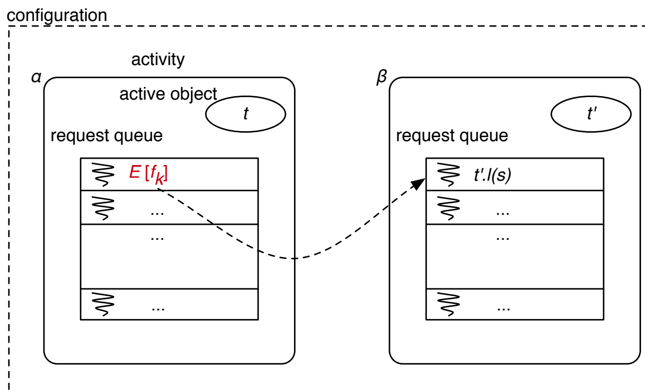

Fig. 1: $\mathrm{ASP}_{\mathrm{fun}}$: a configuration

(Section 4). After introducing the formal security notion of noninterference for $\mathrm{ASP}_{\mathrm{fun}}$, we show that hiding implies security of private data for $\mathrm{ASP}_{\mathrm{fun}}$ (Section 5). After considering some related work, we finally give a global picture of how the hiding operator fits into a global picture of information flow control and show future directions (Section 6).

2 Functional Active Objects in $\mathrm{ASP}_{\mathrm{fun}}$

The language $\mathrm{ASP}_{\mathrm{fun}}$ [HK09b] 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 labelled methods $[l_i = \varsigma(x, y)b]^{i \in 1..n}$ (attributes are considered as methods with no parameters). $\mathrm{ASP}_{\mathrm{fun}}$ 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 $\mathrm{ASP}_{\mathrm{fun}}$, 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. $\mathrm{ASP}_{\mathrm{fun}}$ is built as a conceptual simplification of ASP [CH05] – both languages support the Java API Proactive [Pro08].

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

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

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

Tab. 1: $\mathrm{ASP_{fun}}$ semantics

description with contexts E defined as usual.

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

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

The semantics of $\mathrm{ASP_{fun}}$ 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). In the following example (an extension of the motivating example of [HK09b]) a customer uses a hotel reservation

service provided by a broker.

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

The last step summarizes two reply rule applications: the broker first replies the future reference f_2 to the customer as a result of f_1; second, the hotel replies the value bookingref directly to customer as result of f_2 without passing it to broker.

The update for functional active objects in ASP_{fun} (see rule UPDATE-AO in Table 1) has been designed prolonging the central concept of a functional language by leaving the active object contained in an activity immutable. Otherwise, the semantics would create the possibility to use objects as a store and the update as an assignment. Since activities are persistent inside a configuration, their active objects need to be immutable, i.e. no change can ever happen. Intuitively, the active object inside an ASP_{fun} active object may be seen as a "local program code" that is only ever invoked on method calls. These calls invoked as new entries on the activities request queue may change. To accommodate at the same time some update also for active objects, we designed the ASP_{fun} semantics such that the update actual happens on a copy of the original active object; the original active object is preserved inside the original activity; a new activity with the updated active object is created and the reference in the calling activity that has invoked the update are replaced with the new activity reference.

This update might at first sight seem inefficient but it has its use in applications. We have already investigated its implications for concurrency by implementing Lamport's distributed consensus algorithm Paxos in ASP_{fun} [HK09a]. This experiment shows that in the language ASP_{fun} whenever there is a concurring situation – as for example two simultaneous update requests on one active object – the ASP_{fun} update implicitly resolves this by creating two copies corresponding to two possible outcomes of the concurring requests.

Using this feature, we motivate our ASP_{fun} update by examples from (Web) services where for various requests from customers individual service instances are created by a central server customizing these services to clients' data. This example relies on the mechanism of automatic copies created by active object updates for an efficient implementation of customization (this example is part of the newly extended version of [HK09b]).

In this paper we will show up another even more exciting use of the active object

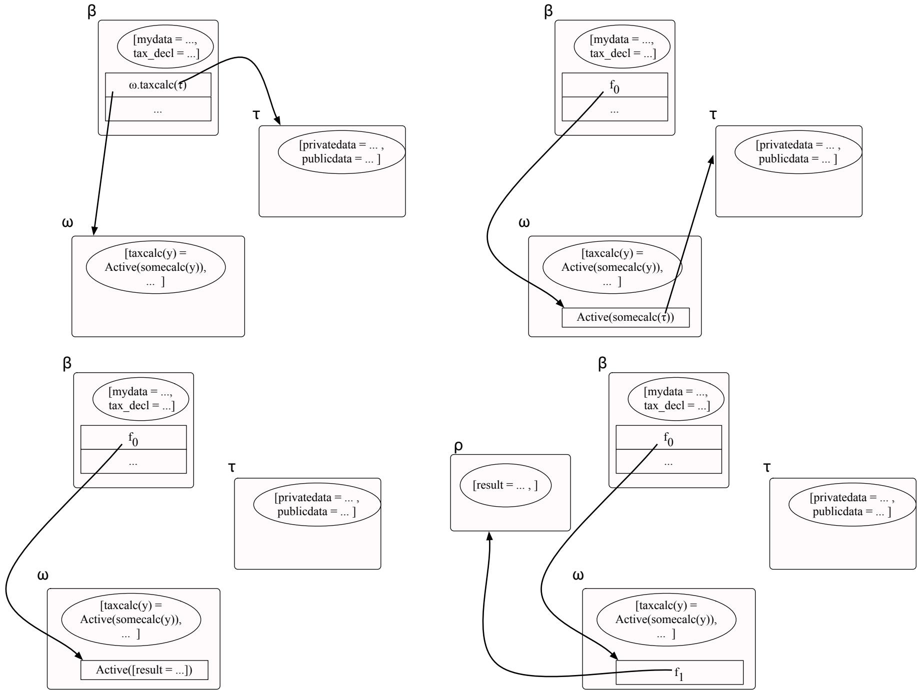

Fig. 2: The Webtax example in ASP_{fun} in a series from left top to bottom right.

update where it is used to overwrite by empty – and thus hide – data of an active object.

3 The Webtax Example

Privacy enforcement for object oriented programming has been revolutionized by Andrew Myers' work on the Decentralized Label Model (DLM) [ML00]. We give a very short description only to motivate the scenario. We then pick up one of his motivating examples and show how ASP_{fun} can be used to implement it.

The Decentralized Label Model (DLM) enables a role based approach to enforcing security in programs [ML00]. The main idea in DLM is to have explicit labels in the program modelling the actors that have access to labelled program parts. Owners and readers can be specified for each data item enabling a fine tuned control over distributed entities.

Myers uses the motivating example of an Webtax program: Bob wants to make his tax declaration; Preparer offers a Web application called Webtax that can be applied to Bob's data to produce as output the tax declaration [ML00]. However, Bob as well as Preparer have their security anxieties. Bob naturally wants to protect his private data while Preparer wants to prevent that any information about his secret algorithm is shed onto the output generated in from of a tax declaration.

As an illustration of the scenario and to show how ASP_{fun} is applied consider the steps depicted in Figure 2, where activity β represents Bob, activity τ his (tax) data, and ω the Webtax program supplied by Preparer.

4 Hiding Information with ASP_{fun}

In this section we describe how the Webtax example can be implemented in a slightly different way in ASP_{fun} in order to hide the private information. We illustrate by this second version the special feature of active object update.

The implementation of Webtax in ASP_{fun} seen in the previous section does not support privacy at all: any object may access any data of any other active object. On the other hand it is fairly simple to protect access to objects by prescribing access rights on the object level. Since the access to objects is only possible via method calls we just need to use access control mechanism to protect data contained in objects. However, if we need to give away data objects to others, like in the above example, we must give access to the objects and its content otherwise we cannot use the service. Naturally, we could now employ a finer grained policy – similar to Myers' DLM – to assign a policy that differentiates private from public parts of an object. But, why don't we just delete the private data? Normally we refrain from doing this to prevent loss. However, in ASP_{fun} we can just use an update that deletes the private information and then proceeds as before. Figure 3 shows how the adapted version works. Here, the initial call to Webtax uttered by Bob, contains the method invocation $\tau.\texttt{privatedata} = []$ on the tax data object τ passed as parameter to `taxcalc` of ω. Now, due to the semantics of update the overwrite automatically creates a copy τ' of the original data object τ. It is this copy that is passed on to ω instead of τ – thus, intuitively quite clear, there is no private information revealed.

The procedure used to protect the information is some kind of *hiding* because the original object is preserved and thus no data is lost. We can generalize this procedure into a hiding operator and we can then generally show that it provides noninterference (see following section). Based on the syntax of ASP_{fun} it is not difficult to generalize the cancellation by method overwrite to all objects by a primitive recursive definition of a hiding operator $\backslash$. In brief, $t \setminus \Delta$ passes through the ASP_{fun} term t and performs the above seen operation $x = []$ on all x that are contained in Δ.

5 Hiding Enables Information Flow Control

In order to generally justify that the hiding operator provides security we first introduce a formal notion of information flow for ASP_{fun}: noninterference.

Intuitively, noninterference 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_{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_{fun} has a safe type system [HK09b] that can serve here but is omitted for brevity [HK09a].

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

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

(where T is given by an activity type $\Gamma_{act}(a)$ or a future type $\Gamma_{fut}(a)$ respectively).

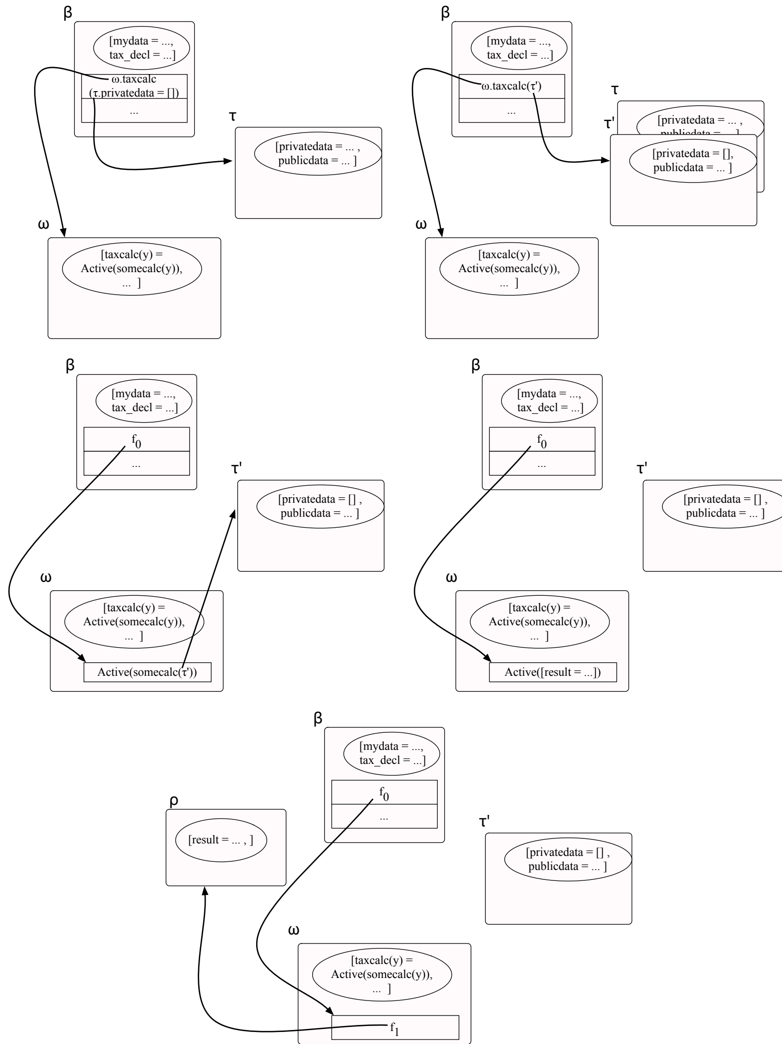

Fig. 3: Similar procedure as in Figure 2 – but now hiding the private information.

The intuition behind typed bijections is that $\mathrm{dom}(\sigma)$ 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 bijection are typed because they relate activities and futures that are all of type L. 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 $T_{\sigma,\tau}$ we denote the term (or type) T where all occurrences of activity names a or futures f are replaced by their counterparts $\sigma(a)$ or $\tau(f)$, respectively, given they are in the domain, otherwise unchanged.

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. The types are indicated implicitly by variable names, i.e. t, t' for base terms, α_i for activity references, f_j for futures, R_{α_k} for request lists, and finally C_n for configurations.*

$$
\begin{array}{rcl}
t \sim_{\sigma,\tau} t' & \equiv & t_{\sigma,\tau} = t' \\
\alpha_0 \sim_{\sigma,\tau} \alpha_1 & \equiv & \tau(\alpha_0) = \alpha_1 \\
f_k \sim_{\sigma,\tau} f_j & \equiv & \sigma(f_k) = f_j \\
[R_{\alpha_0}, t_{\alpha_0}] \sim_{\sigma,\tau} [R_{\alpha_1}, t_{\alpha_1}] & \equiv & R_{\alpha_0} \sim_{\sigma,\tau} R_{\alpha_1} \wedge t_{\alpha_0} \sim_{\sigma,\tau} t_{\alpha_1} \\
R_{\alpha_0} \sim_{\sigma,\tau} R_{\alpha_1} & \equiv & \begin{array}{l} \mathrm{dom}(\sigma) \subseteq \mathrm{dom}(R_{\alpha_0}) \wedge \mathrm{ran}\ (\sigma) \subseteq \mathrm{dom}(R_{\alpha_1}) \wedge \\ \forall\, f_k, f_j\ .f_k \sim_{\sigma,\tau} f_j \Longrightarrow R_{\alpha_0}(f_k) \sim_{\sigma,\tau} R_{\alpha_1}(f_j) \end{array} \\
C_0 \sim_{\sigma,\tau} C_1 & \equiv & \begin{array}{l} \mathrm{dom}(\tau) \subseteq \mathrm{dom}(C_0) \wedge \mathrm{ran}\ (\tau) \subseteq \mathrm{dom}(C_1) \wedge \\ \forall\, \alpha_0, \alpha_1.\ \alpha_0 \sim_{\sigma,\tau} \alpha_1 \Longrightarrow C_0(\alpha_0) \sim_{\sigma,\tau} C_1(\alpha_1) \end{array}
\end{array}
$$

We repeat here the remark made by the designers of this kind of indistinguishability definition [BN03]: "The above exploits the convention that equations involving partial functions are interpreted as false when the function is undefined." Thus, α_0, α_1 (or f_k, f_j, respectively) are in the relation $\sim_{\sigma,\tau}$ if and only if (α_0, α_1) is in τ (or $(f_k, f_j) \in \sigma$, respectively) because otherwise $(\tau(\alpha_0) = \alpha_1) = \mathit{false}$ (or $(\sigma(f_k) = f_j) = \mathit{false}$). In case of $\alpha_0 \notin \mathrm{dom}(\tau)$, for example, $C(\alpha_0) \sim_{\sigma,\tau} C(\alpha_1)$ could be true or false illustrating the partiality of the definition of indistinguishability. The entire high part of the program is not relevant for L-indistinguishability and thus not recorded at all in the corresponding typed bijections. That is, "H-indistinguishability" really corresponds to "indistinguishability not defined".

Based on the notion of indistinguishability we can now define noninterference as "low" indistinguishability preservation. This is equivalent to saying that the indistinguishability relation is a *bisimulation* over the semantics.

Definition 5.3 (Noninterference) *Indistinguishability of terms is preserved by the semantics, i.e. any two terms of the same security level according to policies σ and τ that are indistinguishable remain so under evaluation. Formally, noninterference holds for a security policy represented by σ, τ, if for any two ASP*$_{\text{fun}}$ *terms t_0 and t_1 such that $t_0 \sim_{\sigma,\tau} t_1$ and $t_0 \rightarrow_{\parallel} t'_0$ there exists t'_1 such that $t_1 \rightarrow^*_{\parallel} t'_1$ and $t'_0 \sim_{\sigma,\tau} t'_1$.*

Given the hiding operator defined in the previous section we can prove that for ASP$_{\text{fun}}$ programs hiding enforces security. The following theorem shows that hiding according to a security policy implies noninterference.

Theorem 1 *Let t and Δ be arbitrary ASP*$_{\text{fun}}$ *terms representing a program and some private data. Let furthermore, σ, τ be policies such that Δ is equal to all data assigned as private by σ, τ. For some value t', if $t \rightarrow^*_{\parallel} t'$ and $t \setminus \Delta \rightarrow^*_{\parallel} t'$ then noninterference holds for σ, τ.*

Proof. Unfolding Definition 5.3 for noninterference, the proof is a straightforward case analysis over the semantics after applying induction. □

The theorem assumes that $t \setminus \Delta$ terminates and that it terminates with the same value as t. A way to automatically verify this condition is given by static analysis: classical type safety implies that a well typed program does not get stuck. Thus, the ASP$_{\text{fun}}$ type system we proved to be type safe [HK09b] can be used to statically ascertain the conditions for security of hiding.

Corollary 5.4 *Let t and Δ be arbitrary well-typed ASP*$_{\text{fun}}$ *terms, such that $t \setminus \Delta$ is well-typed like t, then for any security policy σ, τ, in which Δ is equal H, noninterference for σ, τ holds.*

Other generalizations of Theorem 1 are not theorems: not for all programs t that are secure, i.e. noninterfering with respect to some suitable policy σ, τ can we delete Δ. The program might crash or its semantics changed that it produces other outputs than t.

Nevertheless, as we have seen, in combination with classical safe typing, hiding enables noninterference enforcement for a given policy on an ASP$_{\text{fun}}$ program.

6 Conclusion

In this paper we have presented a hiding operator for ASP$_{\text{fun}}$ programs, illustrated its use on a privacy critical example, and justified its security properties formally.

6.1 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. Initially, he implemented a Java tool package called JFlow that implements the DLM and information flow control [Mye99]. The extensions given by the DLM to the standard noninterference notions lead to undecidable typing. That is, typing cannot be completely performed at compile time but has to be partly done at run-time – which is costly and risky. 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. Whereas Myers earlier publication are rather pragmatic solution to designing a decentralized model of security classes, he concentrates on concepts of downgrading values. This is, however, nothing else than "casting" values to a subtype. It is only later that Myers and his colleagues identify type systems as the appropriate model for static analysis with the DLM [ZM07].

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].

For distributed systems the noninterference property becomes more difficult. Already Volpano and Smith show that concurrency produces new security risks. They resolve these risks by strong restrictions of the type system [SV98]. Even though later some generalisations could soften up these strong restrictions, concurrency and distribution remain a challenge to information flow control [BC01, MBC07].

A great advantage of type systems is that their support with interactive theorem provers is fairly advanced these days, e.g. [Kam08], enabling reliable analysis and derivation of prototype type checkers. Our next goal is in fact to apply the static analysis technique of type systems to support a language based framework for security – as described next.

6.2 Language Based Modular Assembly Kit for Security

As we have seen in the previous section, formal notions of security are complex and difficult to understand. Therefore, it is indispensable to have systematic support. 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]. One very important goal is compositionality: if two parts of a system have each individually some security property, what is the security property of the composed system? Does it stay the same, or is it at least also some composite property of the individual components properties. Another possibility are emergent system properties, i.e. properties that – when systems are composed – become augmented.

Mantel provides in his work a precise characterization of the most important known properties and categorizes them in his assembly kit. For example, the classical security property noninterference as first defined by Goguen and Meseguer [GM82] is a deterministic property that has a naturally been followed by the definition of *nondeterministic noninterference*. The quest for compositionality motivated the notion of *restrictiveness* which turned out to be a specialisation of nondeterministic noninterference. However, later the security property *correctability* has been found to be another compositional security property between restrictiveness and nondeterministic noninterference. The great innovation of Mantel's work is the identification of basic security predicates that enable the construction of many existing information flow properties. Thereby, he arrives at comparing

these properties systematically and supporting decomposition and proof of corresponding theory, like unwinding theorems and zipping lemmata, that support a systematic analysis.

Mantel's work is based on the formal system model of event systems. All the properties are based on abstract trace models. Naturally, this view abstracts from more refined differences between system behaviours disabling the contemplation of security at a finer scale. We want to use a language based approach and the more fine grained bisimulation view to support information flow control. 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. We are only at the beginning of this project. The presented hiding device of ASP_{fun} is a first member that will feature in our LB-MAKS toolbox as an enforcement method for noninterference.

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.

[BC01] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *ICALP*, volume 2076 of *Lecture Notes in Computer Science*, pages 382–395. Springer, 2001.

[BN03] A. Banerjee and D. A. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15(2), 2003.

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

[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.

[GM82] J. Goguen and J. Meseguer. Security policies and security models. In *Symposium on Security and Privacy, SOSP'82*, pages 11–22. IEEE Computer Society Press, 1982.

[HK09a] L. Henrio and F. Kammüller. Functional active objects: Noninterference and distributed consensus. Technical Report 2009/19, Technische Universität Berlin, 2009.

[HK09b] 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.

[Kam08] F. Kammüller. Formalizing non-interference for a small bytecode-language in coq. *Formal Aspects of Computing*, 20(3):259–275, 2008.

[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.

[MBC07] Ana Almeida Matos, Gérard Boudol, and Ilaria Castellani. Typing noninterference for reactive programs. *J. Log. Algebr. Program.*, 72(2):124–156, 2007.

[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.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[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.

[SV98] Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, pages 355–364, 1998.

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