Confinement for Active Objects

Florian Kammüller
Middlesex University, London UK and
Technische Universit¨at Berlin,
Germany

Abstract—**In this paper, we provide a formal framework for the security of distributed active objects. Active objects communicate asynchronously implementing method calls via futures. We base the formal framework on a security model that uses a semi-lattice to enable multi-lateral security crucial for distributed architectures. We further provide a security type system for the programming model ASPfun of functional active objects. Type safety and a confinement property are presented. ASPfun thus realizes secure down calls.**

Keywords*—*Distributed active objects, formalization, security type systems

I. Introduction

Formal models for actor systems become increasingly important for the security analysis of distributed applications. For example, models of organisational structures together with actors provide a basis for the analysis of insider threats, [31], [32].

Active objects define a programming model similar to actors [2] but closely related to object-orientation. An object is an *active object* if it serves as an access point to its own methods and associated (passive) objects and their threads. Consequently, every call to those methods will be from outside. These remote calls are collected in a list of requests. The unit comprising the object's methods and attributes and its current requests is called *activity*. The activity serves as a unit of distribution since it has a data space separate from its environment and can process requests independently. To enable asynchronous communication between distributed active objects, the concept of *futures* – promises for method call values – is used. Active objects are practically implemented in the Java API ProActive [6] developed by Inria and commercialized by its spin-off ActiveEON. Active objects are also a tangible abstraction for distributed information systems beyond just one specific language. ASP [7] is a calculus for active objects. ASP has been simplified into ASP$_{\text{fun}}$ – a calculus of *functional* active objects. ASP$_{\text{fun}}$ is formalized in Isabelle/HOL [18] thus providing a general automated framework for the exploration of properties of active objects.

In this paper, we use this framework to support security specification and analysis of active objects. The contributions of this paper are (a) the formalization of a novel security model for distributed active objects that supports multi-lateral security, (b) a type system for the static security analysis for ASP$_{\text{fun}}$ configurations, (c) preservation and the simple security property of confinement for well-typed configurations, (d) and an argument that secure down calls are possible for ASP$_{\text{fun}}$.

The novel security model [21] is tailored to active objects as it supports decentralized privacy specification of data in distributed entities. This is commonly known as multi-lateral security. To achieve it we break away from the classical dogma of lattices of security classes and use instead semi-lattices. In our model, we implement *confinement*. Every object can remotely access only public (L) methods of other activities. Methods can be specified as private (H) in an activity forbidding direct access. All other methods of objects are assumed to be L, partitioning methods locally into L and H. The security policy further forbids local information flow from H to L. To access an L-method remotely, the containing activity must also be visible to the calling activity in a configuration. In ASP$_{\text{fun}}$, this visibility relation is implemented by activity references. In other active object programming languages, visibility could be given alternatively by an import relation or a registry.

In this paper, we provide an implementation of this security model in the ASP$_{\text{fun}}$ framework to illustrate its feasibility and the applicability of the ASP$_{\text{fun}}$ framework.

We design a security type system for ASP$_{\text{fun}}$ that implements a type check for a security specification of active objects and visibility. We prove the preservation property for type safety of the type system guaranteeing that types are not changed by the evaluation of an ASP$_{\text{fun}}$ configuration. The specification of parts of an active object as confined, or private (or H), is possible at the discretion of the user. This specification is entered as a security assignment into the type system; by showing a general theorem that confinement is entailed in well-typedness, we thus know that a well-typed program provides confinement of private methods. Although the confinement property intuitively suffices for security, at this point, a formal security proof is still missing. Moreover, implicit flows may occur. We thus provide a definition of noninterference for active objects. Based on that, we prove that a well-typed configuration does not leak information to active objects below in the hierarchy of the security model, i.e., multi-lateral security holds for well-typed configurations.

Remote method calls in ASP$_{\text{fun}}$ have no side-effects. Hence, secure down calls can be made. Confinement provides that no private information is accessed remotely and side-effect freedom guarantees that through the call no information from the caller side is leaked. Side effects are excluded in our formal model ASP$_{\text{fun}}$ because it is functional but this can be implemented into the run-time system of other active object languages.

Overview

We first review the semi-lattice for multi-lateral security (Section II-A) and ASP$_{\text{fun}}$ (Section II-B) introducing a running example of private sorting (Section II-C). Next, we describe how the semi-lattice model can be applied to active

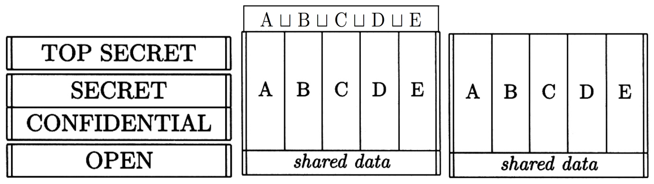

Fig. 1. Joins enable Top control in MLS models

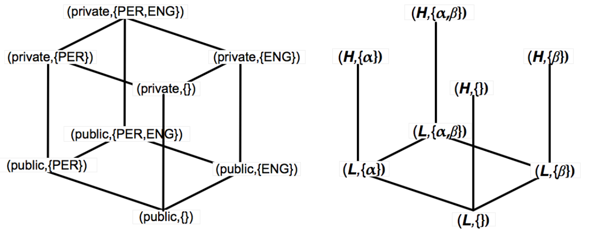

Fig. 2. Taking the top off MLS lattice (left) leads to semi-lattice (right).

objects by instantiating it for ASP$_{\text{fun}}$(Section III). We discuss secure down calls, a distinctive feature of ASP$_{\text{fun}}$ enabled by its functional nature and that moreover does not restrict common bi-directional communication patterns. To show the latter point, we present how to implement the Needham-Schroeder Public Key protocol in ASP$_{\text{fun}}$. We describe what we mean by security, i.e., the attacker model and the information flows between active objects through method calls (Section III-C) and illustrate their enforcement on the running example. Following that, we present a type system for the static analysis of a configuration of active objects in ASP$_{\text{fun}}$ (Section IV). Properties of this type system are presented (Section V): (a) preservation as a standard result of type safety and (b) confinement. We then define noninterference and multi-lateral security formally to present a soundness theorem, i.e., well-typed configurations are multi-lateral secure. We finish the paper with a related work section and also give some conclusions (Section VI). An Appendix contains sections A …E with formal details, more examples, and (full) proofs.

II. Prerequisites

A. Semi-Lattice Model for Privacy

We abstract the confinement property known from object oriented languages, e.g., private/public in Java, and use it as a blueprint for a model of privacy in distributed objects. Consider Figure 1: multi-*level* security models support strict hierarchies like military organization (left); multi-*lateral* security [3, Ch. 8] is intended to support a decentralized security world where parties A to E share resources without a strict hierarchy (right) thereby granting privacy at the discretion of each party. But lattice-based security models usually achieve the middle schema: since a lattice has joins, there is a security class A ⊔ B ⊔ C ⊔ D ⊔ E that has unrestricted access to all classes A to E. For a truely decentralized multi-lateral security model this top element is considered harmful. To realize confinement, we exclude the top element by excluding joins from the lattice. We thereby arrive at an algebraic structure called a semi-lattice in which meets always exist but not joins.

Semi-Lattice: The semi-lattice of security classes for active objects is a combination of global and local security lattices. The two lattices are used to classify the methods into groups and objects into hierarchies.

1) Local Classification: The local classification is used to control the information flow inside an object, where methods are called and executed. For every active object there is the public (L) and a private (H) level partitioning the set of this active object's methods. The order relation of the lattice for local classification is the relation $\leq$ defined on $\{L, H\}$ as $\{(L, L), (L, H), (H, H)\}$.

2) Global Classification: The purpose of the global classification is to control the course of information flows between methods of globally distributed objects and lead their information together in a common dominating activity. To remotely access active objects, the key is their identity (we use α, β to denote identities). As classes for the global lattice we use subsets of the set of all activity identities $\mathcal{I}$. These subsets of compartments build the lattice of global classes, the powerset lattice $\mathcal{P}(\mathcal{I})$ over activity identities $\mathcal{I}$.

$$(\mathcal{P}(\mathcal{I}), \cap, \cup, \subseteq, \varnothing, \mathcal{I})$$

In a concrete configuration, the global class label of an activity is the set of activity identities to which access is granted. For example, with respect to the Hasse diagram in Figure 2, an object at global level $\{\alpha, \beta\} \in \mathcal{P}(\mathcal{I})$ can access any part (method) of an object labeled as $\{\beta\}$ or $\{\alpha\}$ or $\{\}$ but only if this part is additionally labeled as L. Vice versa an object at level $\{\alpha\}$ can neither access L nor H parts of objects at level $\{\beta\}$ nor any parts at level $\{\alpha, \beta\}$ but only L parts at level $\{\}$. Thus the classification of parts of an active object needs to combine labels.

3) Combination of Lattices: The security model of the semi-lattice needs to combine the local and global classification scheme. As result, a *security class* is a pair of local and global class (S, δ). We want to impose confinement of methods in order to realize multi-lateral security with our model. Thus, we have to define the combination of the two constituting lattices such that its order relation corresponds to a multi-lateral information flow relation. I.e., private methods of an object are not accessible by any other than the object itself.

Consequently, the new order for security classes is defined as follows. The combined security class ordering for active objects is defined such that a method class (H, δ) dominates (L, δ) and also (L, δ') for all $\delta' \subseteq \delta$ but no other (X, δ_0) dominates (H, δ). The combination of local and global types into pairs gives a partial order

$$\mathit{CL} \equiv (\{L, H\} \times \mathcal{P}(\mathcal{I}), \sqsubseteq)$$

with

$$(S_0, I_0) \sqsubseteq (S_1, I_1) \equiv \begin{pmatrix} S_0 <_S S_1 \vee S_0 = S_1 = L \\ I_0 \subseteq I_1 \end{pmatrix}$$

where the vertical notation $\binom{\phi}{\xi}$ abbreviates $\phi \wedge \xi$ and $<_S = \{(L, H)\}$ denotes the strict ordering on the local security classes. Consequently, meets exist but no joins. The partial order *CL* is thus just a semi-lattice as illustrated by an example in Figure 2 (right).

B. Functional Active Objects: ASP$_{fun}$

ASP$_{\text{fun}}$ uses a slightly extended form of the simplest ς-calculus from the Theory of Objects [1] by distributing ς-calculus objects into activities. The calculus ASP$_{\text{fun}}$ is functional because method update is realized on a copy of the active object: there are no side-effects.

1) ς-calculus: Objects consist of a set of labeled methods $[l_i = \varsigma(y)b]^{i \in 1..n}$. Attributes are considered as methods not using the parameters. The calculus features method call $t.l(s)$ and method update $t.l := \varsigma(y)b$ on objects where ς is the binder for the method parameter y. Every method may also contain a "*this*" element representing the surrounding object. Note, that the "this" is usually [1] expressed as an additional parameter x to each method's ς scope but we use for this exposition literally *this* to facilitate the understanding. It is, however, important to bear in mind that formally *this* is a variable representing a copy of the current object and that this variable is scoped as a local variable for each object. The ς-calculus is Turing complete, e.g. it can simulate the λ-calculus. We illustrate the ς-calculus by our example below.

2) Syntax of ASP$_{fun}$: ASP$_{\text{fun}}$ is a minimal extension of the ς-calculus by one single additional primitive, the *Active*, for creating an activity. In the syntax (see Table I) we distinguish between underlined constructs representing the static syntax that may be used by a programmer, while futures and active object references are created at runtime. We use the naming convention s, t for ς-terms, α, β for active objects, f_k, f_j for futures, Q_α, Q_β for request queues.

$s, t ::= \underline{x}$	variable
$\mid \underline{this}$	generic object reference
$\mid \underline{[l_j = \varsigma(y_j)t_j]^{j \in 1..n}}$	$(\forall j,\ this \neq y_j)$ object definition
$\mid \underline{s.l_i(t)}$	$(i \in 1..n)$ method call
$\mid \underline{s.l_i := \varsigma(y)t}$	$(i \in 1..n, this \neq y)$ update
$\mid \underline{Active(s)}$	Active object creation
$\mid \alpha$	active object reference
$\mid f_i$	future

TABLE I. ASP$_{\text{FUN}}$ SYNTAX

3) Futures: A *future* can intuitively be described as a promise for the result of a method call. The concept of futures has been introduced in Multilisp [16] and enables asynchronous processing of method calls in distributed applications: on calling a method a future is immediately returned to the caller enabling the continuation of the computation at the caller side. Only if the method call's value is needed, a so-called wait-by-necessity may occur. Futures identify the results of asynchronous method invocations to an activity. Technically, we can see a future as a pair consisting of a future *reference* and a future *value*. The future reference points to the future value which is the instance of a method call in the request queue of a remote activity. In the following, we will use future and future *reference* synonymously for simplicity. Futures can be transmitted between activities. Thus different activities can use the same future.

4) Configuration: A *configuration* is a set of activities

$$C ::= \alpha_i[(f_j \mapsto s_j)^{j \in I_i}\ ,\ t_i]^{i \in 1..p}$$

where $\{I_i\}$ are disjoint subsets of $\mathbb{N}$. The unordered list $(f_j \mapsto s_j)^{j \in I_i}$ represents the request queue, t_i the active object, and $\alpha_i \in \mathrm{dom}(C)$ the activity reference. A configuration represents the "state"of a distributed system by the current parallel activities. Computation is now the state change induced by the evaluation of method calls in the request queues of the activities. Since ASP$_{\text{fun}}$ is functional, the *local* active object does not change – it is immutable – but the configuration is changed *globally* by the stepwise computation of requests and the creation of new activities.

The constructor $Active(t)$ activates the object t by creating a new activity in which the object t becomes active object. Although the active object of an activity is immutable, an update operation on activities is provided. It performs an update on a freshly created copy of the active object placing it into a new activity with empty request queue; the invoking context receives the new activity reference in return. If we want to model operations that change active objects, we can do so using the update. Although the changes are not literally performed on the original objects, a state change can thus be implemented at the level of configurations (for examples see [18]). Efficiency is not the goal of ASP$_{\text{fun}}$ rather minimality of representation with respect to the main decisive language features of active objects while being fully formal.

5) Results, Programs and Initial Configuration: A term is a result, i.e., a totally evaluated term, if it is either an object (like in [1]) or an activity reference. We consider results as values.

In a usual programming language, a programmer does not write configurations but usual programs invoking some distribution or concurrency primitives (in ASP$_{\text{fun}}$ *Active* is the only such primitive). This is reflected by the ASP$_{\text{fun}}$ syntax given above. A "program" is a term s_0 given by this static syntax (it has no future or active object reference and no free variable). In order to be evaluated, this program must be placed in an initial configuration. The initial configuration has a single activity with a single request consisting of the user program:

$$initConf(s_0) = \alpha[f_0 \mapsto s_0, []]$$

Sets of data that can be used as *values* are indispensable if we want to reason about information flows. In ASP$_{\text{fun}}$, such values can be represented as results (see above) to any configuration either by explicit use of some corresponding object terms or by appropriate extension of the initial configuration that leads to the set-up of a data base of basic datatypes, like integers or strings.

6) Informal Semantics of ASP$_{fun}$: Syntactically, ASP$_{\text{fun}}$ merely extends the ς-calculus by a second parameter for methods (the first being *this*) and the *Active* primitive but the latter gives rise to a completely new semantic layer for the evaluation of distributed activities in a configuration.

Local semantics (the relation $\rightarrow_\varsigma$) and the *parallel* (configuration) semantics (the relation $\rightarrow_\parallel$) are given by the set of reduction rules informally described as follows (see Appendix C for the formal semantics).

- CALL, UPDATE, LOCAL: the local reduction relation $\rightarrow_\varsigma$ is based on the ς-calculus.
- ACTIVE: $Active(t)$ creates a new activity α, with t as its active object, global new name α, and initially no futures; in ASP$_{\text{fun}}$ notation this is $\alpha[\varnothing, t]$.
- REQUEST, SELF-REQUEST: a *method call* $\beta.l(t)$ creates a new future f_k for the method l of active object β placing the resulting future value onto β's request queue; the future f_k can be used to refer to the future value $\beta.l(t)$ at any time.
- REPLY: *returns result*, i.e., replaces future f_k by the referenced result term, i.e., the future value resulting from some $\beta.l(t)$.

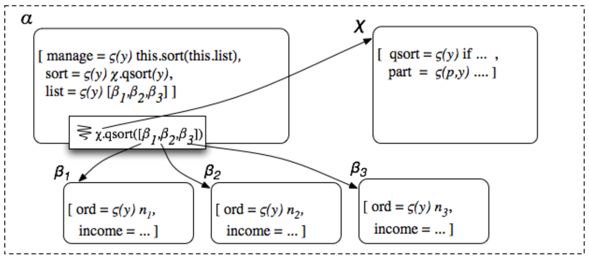

Fig. 3. Three active objects $\beta_1, \beta_2, \beta_3$ in controller α's list.

- UPDATE-AO: *active object update* creates a copy of the active object and updates the active object of the copy – the original remains the same (functional active objects are *immutable*).

C. Running Example: Private Sorting

As an example for a standard program consider the implementation of quicksort as an active object χ illustrated in Figure 3. The operations we use are :: for list cons, @ for list append, # for list length, hd for the list head, and a let construct (see [18] for details on their implementation).

$$
\begin{array}{l}
\chi\big[\varnothing, \\
\quad [\text{qsort} = \varsigma(y) \text{ if } y = [] \text{ then } [] \\
\qquad\qquad\qquad \text{else let } (a :: l) = y \\
\qquad\qquad\qquad\qquad (l_1, l_2) = \textit{this}.\text{part } (a, l) \\
\qquad\qquad\qquad\qquad l'_1 = \text{if } \#l_1 \le 1 \text{ then } l_1 \text{ else } \textit{this}.\text{qsort}(l_1) \\
\qquad\qquad\qquad\qquad l'_2 = \text{if } \#l_2 \le 1 \text{ then } l_2 \text{ else } \textit{this}.\text{qsort}(l_2) \\
\qquad\qquad\qquad\quad \text{in } l'_1 @ [a] @ l'_2 \\
\qquad\qquad\qquad \text{end}, \\
\quad\ \ \text{part} = \varsigma(p, y) \text{ if } y = [] \text{ then } ([], []) \\
\qquad\qquad\qquad \text{else let } (a{::}l) = y \\
\qquad\qquad\qquad\qquad (l_1, l_2) = \textit{this}.\text{part } (p, l) \\
\qquad\qquad\qquad\quad \text{in if } p < a.\text{ord then } (l_1, a{::}l_2) \text{ else } (a{::}l_1, l_2) \\
\qquad\qquad\qquad \text{end} \\
\quad \big]\big]
\end{array}
$$

The quick sort algorithm in χ is parametric over a method "ord", a numerical value, that is used in method "part". This method ord is assumed to be available uniformly in the target objects contained in the list that shall be sorted. We omit the parameter to calls of ord because it is unused, i.e., the empty object [].

The following controller object α holds a list of active objects (for example $[\beta_1, \beta_2, \beta_3]$ in Figure 3 but generally arbitrary thus represented as $\ldots$ below). Controller α uses the quick sort algorithm provided by χ to sort this list on execution of the manage method.

$$
\begin{array}{l}
\alpha\big[\varnothing, [\text{manage} = \varsigma(y)\textit{this}.\text{sort}(\textit{this}.\text{list}), \\
\qquad\ \ \text{sort} = \varsigma(y)\ \chi.\text{qsort}(y), \\
\qquad\ \ \text{list} = \ldots]\big]
\end{array}
$$

The target objects contained in $\alpha's$ list (omitted) are active objects of the kind of β below. Here, the n in the body of method ord is an integer specific to β and the field income shall represent some private confidential data in β.

$$\beta\big[\varnothing, [\text{ord} = \varsigma(y)n, \text{income} = \ldots]\big]$$

If active objects of the kind of β represent principals in the system, it becomes clear what is the privacy challenge: the controller object α should be able to sort his list of β-principals without learning anything about their private data, here income.

III. Semi-Lattice Model for ASP$_{\text{fun}}$

As a proof of concept, we show that the calculus of functional active objects ASP$_{\text{fun}}$ gives rise to a fairly straightforward implementation of the security semi-lattice by mapping the concepts of the security model onto language concepts as follows.

- The global class ordering on sets of activity identities corresponds to the sets of activity references that are accessible from within an activity. We name this accessibility relation visibility (see Definition 3.1). It is a consequence of the structure of a configuration thereby at the discretion of the configuration programmer.
- The local classification of methods into public L and private H methods is specified as an additional security assignment mapping method names to $\{L, H\}$ at the discretion of the user.
- Based on these two devices for specifying and implementing a security policy with active objects we devise as a practical verification tool a security type system for ASP$_{\text{fun}}$. The types of this type system correspond quite closely to the security classes of the semi-lattice defined in Section II-A: object types are pairs of security assignment maps and global levels.

A. Assigning Security Classes to Active Objects

Visibility: We define visibility as the "distributed part" of the accessibility within a configuration. It derives from the activity references and thus represents the global security specification as programmed into a configuration.

Definition 3.1 (Visibility): Let C be a configuration with a security specification *sec* partitioning the methods of each of C's active objects locally into H and L methods. Then, the relation $\le_{VI}$ is inductively defined on activity references by the following two cases.

$$
\begin{array}{lcl}
\begin{pmatrix} \beta[Q_\beta, [l_i = \varsigma(y)t_i]^{i \in 1..n}] \in C \\ sec(l_i) = L \wedge t_i = E[\alpha] \end{pmatrix} & \Rightarrow & \alpha \le_{VI} \beta \\
\begin{pmatrix} \beta[Q_\beta, [l_i = \varsigma(y)t_i]^{i \in 1..n}] \in C \\ sec(l_i) = L \wedge t_i = E[\gamma] \wedge \alpha \le_{VI} \gamma \end{pmatrix} & \Rightarrow & \alpha \le_{VI} \beta
\end{array}
$$

We use the vertical notation $\binom{\phi}{\xi}$ to abbreviate $\phi \wedge \xi$; for context variable E see Appendix C. We then define the relation called *visibility* $\sqsubseteq_C^{sec}$ as the *reflexive transitive closure* over $\le_{VI}$ for any C, *sec*. □

We denote the *visibility range* using Definition 3.1 as $VI_{sec}(\alpha, C) \equiv \{\beta \in \text{dom}(C) \mid \beta \sqsubseteq_C^{sec} \alpha\}$. The visibility relation extends naturally to a relation $\sqsubseteq$ on global levels: every activity $\alpha \in C$ may be assigned the global level corresponding to the union of all its visible activities $VI_{sec}(\alpha, C)$. This relation is a subrelation of the subset relation on the powerset of activity identities introduced before and thus also a partial order. We use it as the semantics of the subtype relation in Section IV.

Assigning Security Classes to Example: To illustrate how activities are labeled in the semi-lattice model using visibility, consider the running example above where we assume the list in controller α to contain various active object references

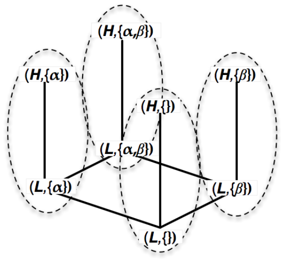

Fig. 4. Tentatively drawing in object classes as confinement zones.

$[\beta_0, \dots, \beta_n]$. We assign to each activity the global class containing its own identity and those of all its visible activities. For our example, the *global* class of controller α would be the following.

$$\delta_\alpha = \{\alpha\} \cup \delta_\chi \cup \bigcup_{i=0..n} \delta_{\beta_\mathrm{i}}$$

The global classes δ_{β_i} of the β_i objects and δ_χ in turn contain all their visible objects' classes. Thus, the global classes are ordered $\delta_{\beta_i} \subseteq \delta_\alpha$ for all i and $\delta_\chi \subseteq \delta_\alpha$. The security classification of methods assigns pairs of global classes and local levels to method names, for example, $\mathrm{ord}_{\beta_i} \mapsto (L, \delta_{\beta_i})$ and $\mathrm{income}_{\beta_i} \mapsto (H, \delta_{\beta_i})$.

Practical Classification of Objects: The pairs (S, δ) in the partial order *CL* (see Section II-A3) are the security classes for methods of active objects. The semi-lattice is actually defined as a partial order on object methods rather than objects. To classify objects we consider only the global part of the classification, i.e., the second δ component because all methods of an active object have this δ in common. Intuitively, this factorization corresponds to drawing objects as borders into the semi-lattice structure (see Figure 4). These borders represent the confinement zone of an active object.

Formally, we consider an object class to be the factorization $([l_i \mapsto S_i], \delta)$: a pair of a security assignment to $\{L, H\}$ for each method l_i of an object and the object's global class δ common for all parts. An activity contains one active object but may contain various passive objects. The security assignment of an active object must be defined for all contained objects (see rule SECASS SUBSUMPTION in Section IV).

B. Secure Down Calls

In a distributed system with a nontrivial security classification of communicating objects, secure down calls are not possible because they would violate the security policy of "no-down-flows" of information. In general, a method call represents an information flow to the remote object in the form of the request itself and the parameters passed; its response flows information back in the form of a reply. Therefore, secure method communication is trivially restricted to objects of one class – otherwise one direction would contradict the policy "no-down-flows". This catch-22 situation can be overcome if we exclude side-effects: the requests do not leave traces in the remote object. In ASP$_{\mathrm{fun}}$ this is given implicitly by the semantics because requests created by method calls in the remote object are not accessible by the remote object itself. However, the reply may flow information up. Thus, information does flow back, i.e. up.

As an overall result of the properties presented in this paper we can infer that secure down calls are possible. The reasoning is as follows. We assume as given a configuration together with a security specification *sec* partitioning a portion of the methods into public (L) and private (H). If this configuration can be type checked according to our type system, it is secure, i.e., we know it has confinement and is noninterfering as we are going to see in Section V. Therefore, futures can be securely used in higher security classes, i.e., method results may flow up but, since no implicit flows exist, information is not leaked in the process.

Side-effect freedom does permit to securely call down because the call leaves no visible trace. But does this not also exclude any mutual information exchange on the same level? It might seem so, but fortunately, if we have two activities that are in the same class, methods calls between them are possible permitting bidirectional information flow. As an example, an implementation of the Needham-Schroeder public key protocol is given next.

Needham Schroeder Public Key Protocol (NSPK) in ASP$_{fun}$

This example illustrates that inside one security class mutual information exchange is possible between different activities. The easiest way to illustrate this is to use a protocol. We use the corrected short form of the Needham Schroeder Public Key Protocol (NSPK) originally published by [28]. The originally published protocol missed out the B inside the encrypted message to A in step two thereby giving rise to the well-known attack of [22].

The protocol is usually written as follows using public keys K_A, K_B known globally and their secret counterparts K_A^{-1}, K_B^{-1} establishing nonces N_A, N_B in the process of authentication.

$$\begin{array}{rcl} A \to B & : & \{N_A, A\}_{K_B} \\ B \to A & : & \{N_A, N_B, B\}_{K_A} \\ A \to B & : & \{N_B\}_{K_B} \end{array}$$

In ASP$_{\mathrm{fun}}$, the protocol is implemented as a set of methods between two activities A and B. We omit details about decoding and keys because it is clear that they can be implemented and we want to highlight the communication process.

```
$A = [\varnothing,$
$[\mathrm{own}_{id} = \dots$
 $\mathrm{B}_{id} = \dots$
 $\mathrm{step}_1 = \varsigma(y)$
           let $N_A = \mathrm{new\_nonce}$
              $\mathit{reply} = B.\mathrm{step}_2(\{N_A, \mathit{this}.\mathrm{own}_{id}\}_{K_B})$
              $(N'_A, N'_B, B') = K_A^{-1}(\mathit{reply})$
           in if $B' = \mathit{this}.B_{id} \wedge N'_A = N_A$
              then $(\mathit{this}.\mathrm{knows} := N_B).\mathrm{NA} := N_A$
              else $\mathit{this}.\mathrm{knows} := \mathrm{error},$
 $\mathrm{step}_3 = \varsigma(y)$
           let $(N'_A, N_B, B'_{id}) = \{y\}_{K_A^{-1}}$
              if $N'_A = \mathit{this}.\mathrm{NA} \wedge B'_{id} = \mathit{this}.B_{id}$
              then $\{N_B\}_{K_B}$
              else $\mathit{this}.\mathrm{knows} := \mathrm{error},$
 $\mathrm{knows} = \dots,$
 $\mathrm{NA} = \dots]]$
```

The protocol can be executed by invoking method $A.\mathrm{step}_1$

which in turn invokes the step $B.\text{step}_2$ and $A.\text{step}_3$.

```
$B = [\varnothing,$
$[\text{own}_{id} = \ldots$
 $\text{A}_{id} = \ldots$
 $\text{step}_2 = \varsigma(y)$
            let $N_B = \text{new\_nonce}$
               $(N'_A, A'_{id}) = \{y\}_{K_B^{-1}}$
            in if $\textit{this}.A_{id} = A'_{id}$
               then let $\textit{reply} = A.\text{step}_3(\{N'_A, N_B, \textit{this}.\text{own}_{id}\}_{K_A})$
                        $N'_B = \{\textit{reply}\}_{K_B^{-1}}$
                  in if $N'_B = N_B$
                     then $(\text{this.knows} := N_A).\text{NB} := N_B$
                     else $\textit{this}.\text{knows} := \text{error},$
               else $\textit{this}.\text{knows} := \text{error},$
 $\text{knows} = \ldots,$
 $\text{NB} = \ldots]]$
```

In each of the steps the nonces are created, encrypted and tested between the method calls. If the communicated messages adhere to the protocol, i.e., the nonces and ids correspond to what has been sent in earlier steps, the own nonces are updated into the methods A.NA and B.NB and the other's nonces in the respective method "knows". Otherwise, the protocol failure is recorded as "error" in method knows. This protocol implementation illustrates that mutual information flows are possible locally within one security class. The type system that we present in the following Section IV accepts this configuration since the calls are of the same global level δ.

C. Security Analysis

In language based security, we may use the means provided by a language to enforce security. That is, we make use of certain security guarantees that correspond to implicit assumptions concerning the execution of programs. The language introduces a security perimeter because we assume that the language compilation and run-time system are respected (below the perimeter) while the language is responsible for the security above the perimeter by virtue of its semantics and other language tools, e.g. static analysis by type checking. We now describe the security goal of confidentiality addressed in this paper and elaborate on the attacker model for active objects.

Security Goal Confidentiality: A computation of active objects is an evaluation of a distributed set of mutually referencing activities. Principals that use the system can observe the system only by using the system's devices. We make the simplifying assumption that principals can be identified as activities. Principals, objects, programs and values are thus all contained in this configuration. There are no external inputs to this system – it is a closed system of communicating actors. We concentrate in this paper on confidentiality, i.e. activities should not learn anything about private parts of other activities neither directly nor indirectly. Integrity is the dual to this notion and we believe that it can simply be derived from our present work by inverting the order relation.

Attacker Model: As a further consequence to the language based approach to security, we restrict the attacker to only have the means of the language to make his observations. Consequently, we also consider the attacker – as any other principal – as being represented by an activity. The attacker's knowledge is determined by all active objects he sees, more precisely their public parts. If any of the internal computations in inaccessible parts of other objects leak information, the attacker can learn about them by noticing differences in different runs of the same configuration. Inaccessible parts of other objects must be their private methods or other objects that are referenced in these private parts. The language semantics and the additional static analysis must guarantee that under the assumption of the security perimeter an attacker cannot learn anything about private parts.

D. Information Flow Control

Information flow control [11] technically uses an *information flow policy* which is given by the specification of a set of *security classes* to classify information and a *flow relation* on these classes that defines allowed information flows. System entities that contain information, for example variables x, y, are bound to security classes. Any operation that uses the value of x to calculate that of y, creates a flow of information from x to y. This operation is only admissible if the class of y dominates the class of x in the flow relation, formally written $\delta_x \sqsubseteq \delta_y$ where δ_e denotes the class of entity e. The concept of information flow classically stipulates that the security classes together with the flow relation as an order relation on the classes are a lattice [10], [8]. We differ here since we only require a semi-lattice.

Information Flow Control for Active Objects: Information is contained in data values which are here either objects or activity references (see Section II-B). To apply the concept of information flow control to configurations of active objects, we need to interpret the above notions of security classes, their flow relation, and the entities that are assigned to the security classes: we identify the classes of our security model as the security classes of methods and the flow relation as the semi-lattice ordering on these classes (see Section II-A). Flows of information local to objects are generated by local method calls between neighboring methods of the same object. These are regulated by the local L/H-classification of an object's methods (H may call L *and* H – but L only L). Global flows result from remote method calls between objects' methods. The combined admissible flows have to be in accordance with a concrete configuration and its L/H specification.

E. Enforcing Legal Information Flows

To illustrate the task of controlling information flows, we first extend the intuition about information flow to configurations of active objects. An active object sees only other active objects that are directly referenced in its methods or those active objects that are indirectly visible via public methods of visible objects. From the viewpoint of one active object, information may flow into the object and out of the object. For each direction, there are two ways how information may flow: implicit or explicit (direct) flows. Information flows *explicitly into* an object by parameters passed to remote calls directed to the object's methods; it may also flow *implicitly into* the object simply if the choice of which method is called depends on the control flow of the calling object. Similarly, information flows *explicitly out* of an active object by parameters passed to remote method calls and *implicitly out* of it, if the choice depends on the object's own control flow. Some of these flows are illustrated on our running example next.

Running Example: Implicit Information Flow: We will now finally illustrate the security model on the running example showing implicit information flows of active objects introduced above in Section II-C. Let us assume that the implementation of the β-objects featuring in the controller's list had the following implementation.

$$\beta\big[\varnothing, \quad [\text{ord} = \varsigma(y) \text{ if } \textit{this}.\text{income}/10^3 \geq 1 \text{ then } 1 \text{ else } 0], \\ \text{income} = \ldots \quad \big]$$

Let us further assume that ord is again a public method and income again the private field of β. We have here a case of an implicit information flow. Since the guard of the if-command in ord depends on the private field income, effectively the order number of a β-object is 1 if the income of β is more than 1000 else 0. In our security model this control flow represents an illicit flow of information from a high level value in β to its public parts and is thus visible to the remote controller. This should not be the case since $H \sqsupset$ L. It should thus be detectable by an information flow control analysis. We will show next how to detect it statically by a security type system.

IV. Security Type System

Before formalizing security of active objects and defining a type system that implements rules for a static analysis, we summarize the security considerations so far and motivate the upcoming type system and proofs.

A. Intermediate Summary, Motivation, and Outlook

In a configuration of active objects we may have direct (explicit) and implicit information flows through method calls which are controlled differently.

- To guarantee only legal information flows on direct calls we rely on the labeling of methods by L and H and on the global hierarchy. This corresponds to the simple security property of *confinement*: remote method calls can refer only to low methods of visible objects. Confinement can be locally checked. It is decidable since it corresponds to merely looking up method labels in a security assignment.
- We will use a program counter PC that records the current security level of a method evaluation. Locally, within the confinement zone of an activity, accessing H-methods in L-methods may create implicit flows – as seen in the example. To detect such flows and protect the confidential information from flowing out of the confinement zone of the activity, the program counter records these dependencies by increasing to H. In combination with the method labels, the PC thereby allows associating the calling context with the called method. Implemented into type rules, this enables static checking and thus controlling of information flows in evaluations of configurations.

As a security enforcement mechanism of our multilateral security model for active objects, we propose a security type system, i.e., a rule set for static analysis of a configuration with respect to its methods' security assignment. The idea of a security type system is as follows. Not all possible programs in ASP$_{\text{fun}}$ are secure. In general, for example, any method can be accessed in an active object. The purpose of a type system for security is to supply a set of simple rules defining types of configurations enabling a static check (before run-time) whether those contain only allowed information flows.

The above described cases of information flows need to be implemented in the type rules such that the rules allow to infer a type just for secure configurations and otherwise reject them. The first direct case of information flow is intuitively simple, as it boils down to locally looking-up the security level of a method before deciding whether a remote call from up in the hierarchy can be granted. The "up in the hierarchy" is encoded in a subtype relation $\sqsubseteq$ encoding the global hierarchy described by the visibility relation. After the presentation of the type system in this section, we prove in the following Section V that confinement is a security property implied by it.

How to avoid and detect implicit flows, is more subtle: the combination of a program counter PC with the called method's security label grants us to combine the provenance of one call with the security level of the call context. However, this combination needs to adhere to the security specification for all runs of a program and thus all possible calls in a context. The appropriate notion of security for this is noninterference: in all runs the observable (low) parts of configurations need to look "the same". Therefore, we first introduce a notion of noninterference for active objects based on which we will then be able to express the absence of implicit flows and prove multilateral security. The definition of noninterference and proofs of properties are contained in Section V. We first introduce the type system.

B. Type System

Type Formation: We need to provide types for objects and for configurations of active objects; the latter by mapping names of futures and activities to object types. The two-dimensional classification of local and global security described above translates directly into the object types of the security type system. A type is a pair $([l_i \mapsto S_i]^{i=1..n}, \delta_\alpha)$ where $S_i \in \{L, H\}$. The first part $[l_i \mapsto S_i]^{i=1..n}$ provides the partition of methods into public (L) and private (H) methods for the object. The other element δ_α of an object type represents the global classification of an object. This global level corresponds to the classification of the object's surrounding activity α derived from its visibility. We adopt the following naming conventions for variables. δ stands for the global part of a type. We use A to denote security assignments, e.g. $A = [l_i \mapsto S_i]^{i=1..n}$. S_i, or simply S, stands for levels L or H. In general, we use indexed variables to designate result values of a function, e.g., S_i for the level value of method l_i – also expressed as $A(l_i)$. We use Σ for object types $\Sigma = ([l_i \mapsto S_i]^{i=1..n}, \delta)$. To map an object type Σ to its security assignment or its global part, respectively, we use the projections $ass(\Sigma)$ and $glob(\Sigma)$. We formally use a parameter *sec* as the parameter for the overall methods' security assignment of an entire configuration C. A triplet of maps is a configuration type $\langle \Gamma_{act}, \Gamma_{fut}, sec \rangle$ assigning types to all activities and futures of a configuration in addition containing the security assignment *sec*.

Typing Relations: A typing judgement $T; S \vdash x : ([l_i \mapsto S_i]^{i=1..n}, \delta)$ reads: given type assumptions in T, term x has type $([l_i \mapsto S_i]^{i=1..n}, \delta)$ in the context of a program counter $S \in \{L, H\}$. A program counter (PC) is a common technique in information flow control originating in Fenton's Data Mark Machine [12]. The PC encodes the highest security level that has been reached in all possible control flows leading to the current control state. In a functional language, like ASP$_{\text{fun}}$, this highest security level of all execution paths simply is the level of the evaluation context for the term x. Thus, the PC is some $S \in \{L, H\}$ denoting the security label of the local context. The type environment T contains types Σ for the parameter variables y and types for the parameter *this* both paired with the local security level S representing their local PC.

Subsumption Rules: Subsumption means that an element of a type also has the type of its super-type. It is responsible for making the partial order relation on global levels a subtype relation. Intuitively, GLOB SUBSUMPTION says that if a term can be typed in a low context it may as well be "lifted", i.e., considered as of higher global level thereby enforcing (together with TYPE CALL below) that only L-methods can be accessed remotely. This corresponds to the confinement property as formally shown in Section V-B. The local security class ordering is $L \leq H$ and features implicitly in the type system in the form of a second – the local – subsumption rule. Finally, the rule SECASS SUBSUMPTION allows the security assignment type of an object to be extended. This rule is necessary to consider an object also as a local object inside another (active) object adopting its security assignment.

LOC SUBSUMPTION
$$\frac{T; L \vdash x : (A, \delta)}{T; H \vdash x : (A, \delta)}$$

SECASS SUBSUMPTION
$$\frac{T; S \vdash x : (A, \delta) \qquad A \subseteq A'}{T; S \vdash x : (A', \delta)}$$

GLOB SUBSUMPTION
$$\frac{T; L \vdash x : (A, \delta) \qquad \delta \sqsubseteq \delta'}{T; L \vdash x : (A, \delta')}$$

TABLE II. SUBSUMPTION RULES, $A = [l_i \mapsto S_i]^{i \in 1..n}, S \in \{L, H\}$

VAL SELF
$$\mathit{this} : \Sigma :: T; \sqcup_{i \in 1..n} S_i \vdash \mathit{this} : \Sigma$$

VAL LOCAL
$$x : \Sigma :: T; S \vdash x : \Sigma$$

TYPE OBJECT
$$\frac{\begin{array}{c}\forall i \in 1..n.\\ \mathit{this} : \Sigma :: y : \Sigma :: T; S_i \vdash t_i : \Sigma\end{array}}{T; \sqcup_{i \in 1..n} S_i \vdash [l_i = \varsigma(y) t_i]^{i \in 1..n} : \Sigma}$$

TYPE CALL
$$\frac{\begin{array}{c}T; S \vdash o : \Sigma\\ j \in 1..n \qquad T; S_j \vdash t : \Sigma\end{array}}{T; S_j \vdash o.l_j(t) : \Sigma}$$

TYPE UPDATE
$$\frac{\begin{array}{c}T; S \vdash o : \Sigma\\ j \in 1..n \qquad \mathit{this} : \Sigma :: y : \Sigma :: T; S_j \vdash t : \Sigma\end{array}}{T; S \vdash o.l_j := \varsigma(y) t : \Sigma}$$

TABLE III. TYPE RULES FOR OBJECTS; $\Sigma = ([l_i \mapsto S_i]^{i \in 1..n}, \delta)$

Object Typing: The object typing rules in Table III describe how object types are derived for all possible terms of ASP$_{\text{fun}}$. The VAL-rules state that type assumptions stacked on the type environment T left of the turnstile $\vdash$ can be used in type judgments. These rules apply to the two kinds of environment entries for *this* and for the y-parameter. Since the *this* represents the entire object value itself, its PC is derived as the supremum of all security levels assigned to methods in it. We express this supremum as the join over all levels $\sqcup_{i \in 1..n} S_i$. The other rules are explained as follows. TYPE OBJECT: if every method l_i of an object is typeable with some local type $S_i \in \{L, H\}$ assigned to it by the assignment component A of Σ, then the object comprising these methods is typeable with their maximal local type. Thus, objects that contain H methods cannot themselves be contained in other L-methods. Otherwise, local objects containing confidential parts could be typed with GLOB SUBSUMPTION at higher levels (see the Appendix for a "borderline example" illustrating this point). Only objects that are purely made from L-methods can be accessed remotely in their entirety. Albeit this strong restriction, the CALL rule permits selectively accessing L methods of such objects (see below). The *PC* guarantees that all method bodies t_i are typeable on their given privacy level S_i. The rule TYPE CALL is the central rule enforcing that only L methods can be called in any object – locally or remotely. Initially, a call $o.l_j(t)$ can only be typed as $\Sigma = ([l_i \mapsto S_i]^{1=1..n}, \delta)$ for the δ of the surrounding object o. Although the PC in the typing of o is (by TYPE OBJECT) the maximal level of all methods, we may still call L-methods on objects that are typed with PC as H. The PC in the typing of the resulting call $o.l_j(t)$ is coerced to S_j, i.e., the security level assigned to the called method. This prevents H methods from being callable remotely while admitting to call methods on objects that are themselves typed in a H-*PC*. Because of the rule GLOB SUBSUMPTION any method call $o.l_j(t) : (A, \delta)$ can also be interpreted as $o.l_j(t) : (A, \delta')$ for $\delta \sqsubseteq \delta'$ but this is restricted to L contexts: a method call typeable in an H context cannot be "lifted", i.e., it cannot be interpreted as well-typed with δ'; to prevent this, the *PC* in GLOB SUBSUMPTION is L thus excluding CALL instantiations for methods l_j with $S_j = H$. UPDATE: an update of an object's method is possible but conservatively, i.e., the types must remain the same.

Configuration Typing: The rules for configurations (see Table IV) use the union of all futures of a configuration.

Definition 4.1 (Future Domain): Let C be a configuration. We define the domain of all futures of C.

$$dom_{\mathit{fut}}(C) \equiv \bigcup \{\text{dom}(Q) \mid \exists\ \alpha, a.\ \alpha[Q, a] \in C\} \qquad \Box$$

The rules for configurations anticipate two semantic properties of futures in well formed ASP$_{\text{fun}}$ configurations. We use well-formedness of ASP$_{\text{fun}}$ configurations as defined in [18]; in brief: there are no dangling references.

Property 4.2 (Unique Future Home Activity): Every future is defined in the request queue of one unique activity.

$$\forall f_k \in \text{dom}_{\mathit{fut}}(C).\ \exists! \alpha[Q, a] \in C.\ f_k \in \text{dom}(Q)$$

We denote this unique activity α as $\mathit{futact}_C(f_k)$. $\Box$

Next, every future f_k in a well formed configuration C is created by a call to a unique label in its home activity.

Property 4.3 (Unique Future Label): Let $\alpha[Q, a] \in C$ be the unique $\mathit{futlab}_C(f_k)$. Then,

$$\forall f_k \in \text{dom}_{\mathit{fut}}(C).\ \exists! l \in \text{dom}(a).\ \exists t.\ a.l(t) \rightarrow_{\parallel}^{*} Q(f_k).$$

We denote this unique method label as $\mathit{futlab}_C(f_k)$. $\Box$

We omit the configuration C for the previous two operators if it is clear from context.

The configuration type rules link up types for activities and futures with the local types of terms in active objects and request lists (see Table IV).

$$\textsc{Type Active}\quad \frac{\langle \Gamma_{\mathit{act}}, \Gamma_{\mathit{fut}}, \mathit{sec}\rangle, T; S \vdash a : \Sigma}{\langle \Gamma_{\mathit{act}}, \Gamma_{\mathit{fut}}, \mathit{sec}\rangle, T; S \vdash \mathit{Active}(a) : \Sigma}$$

$$\textsc{Type Active Object Reference}\quad \frac{\beta \in \mathrm{dom}(\Gamma_{\mathit{act}})}{\langle \Gamma_{\mathit{act}}, \Gamma_{\mathit{fut}}, \mathit{sec}\rangle, T, M_\beta \vdash \beta : \Gamma_{\mathit{act}}(\beta)}$$

$$\textsc{Type Future Reference}\quad \frac{f_k \in \mathrm{dom}(\Gamma_{\mathit{fut}})}{\langle \Gamma_{\mathit{act}}, \Gamma_{\mathit{fut}}, \mathit{sec}\rangle, T; \mathit{ass}(\Gamma_{\mathit{fut}}(f_k))(\mathit{futlab}(f_k)) \vdash f_k : \Gamma_{\mathit{fut}}(f_k)}$$

$$\textsc{Type Configuration}\quad \frac{\begin{array}{c}\mathrm{dom}(\Gamma_{\mathit{act}}) = \mathrm{dom}(C) \qquad \mathrm{dom}(\Gamma_{\mathit{fut}}) = \mathrm{dom}_{\mathit{fut}}(C) \qquad \bigcup_{\alpha \in \mathrm{dom}(C)} A_\alpha \subseteq \mathit{sec} \\ \forall\, \alpha[Q, a] \in C.\ \begin{cases} \langle \Gamma_{\mathit{act}}, \Gamma_{\mathit{fut}}, \mathit{sec}\rangle, \varnothing; M_\alpha \vdash a : \Gamma_{\mathit{act}}(\alpha) \wedge \\ \forall\, f_k \in \mathrm{dom}(Q).\ \begin{cases} \Gamma_{\mathit{act}}(\alpha) = \Gamma_{\mathit{fut}}(f_k) \wedge \\ \langle \Gamma_{\mathit{act}}, \Gamma_{\mathit{fut}}, \mathit{sec}\rangle, \varnothing; A_\alpha(\mathit{futlab}(f_k)) \vdash Q(f_k) : \Gamma_{\mathit{fut}}(f_k) \end{cases} \end{cases}\end{array}}{\vdash C : \langle \Gamma_{\mathit{act}}, \Gamma_{\mathit{fut}}, \mathit{sec}\rangle}$$

TABLE IV. TYPING CONFIGURATIONS; $M_\alpha = \bigsqcup_{i \in 1..n_\alpha} A_\alpha(i)$; $A_\alpha = \mathit{ass}(\Gamma_{\mathit{act}}(\alpha))$, WHERE $\mathit{ass}([l_i \mapsto S_i]^{i=1..n}, \mathit{delta}) = [l_i \mapsto S_i]^{i=1..n}$.

TYPE ACTIVE allows to transfer the type of an object term to its activation which coerces the types of activities and activity references to coincide with the types of their defining objects. This is achieved together with TYPE ACTIVE OBJECT REFERENCE and the clause $\langle \Gamma_{\mathit{act}}, \Gamma_{\mathit{fut}}, \mathit{sec}\rangle, \varnothing \vdash a : \Gamma_{\mathit{act}}(\alpha)$ of TYPE CONFIGURATION.

TYPE FUTURE REFERENCE similarly assigns the types for the future references in Γ_{fut}. For a given activity α, this rule further coerces the PC for the typing of f_k to coincide with $A_\alpha(\mathit{futlab}(f_k))$, i.e., α's security assignment applied to the label that leads to the instance of f_k.

The rule TYPE CONFIGURATION ensures consistency between the type maps Γ_{fut}, Γ_{act}, and the overall security assignment *sec*. It looks rather complex but it essentially only scoops up what has been prepared by the other rules. The first two clauses ensure that the domains of activities coincide with the configuration domain and similarly for futures that the future type map Γ_{fut} is defined over precisely all futures in all activities. The third clause integrates the security specification *sec* to be respected by the individual security assignments of activities. The last large clause of TYPE CONFIGURATION specifies first that the activity types assigned to activity references by Γ_{act} coincide with their active object types. The second part of that clause addresses the future types in Γ_{fut}. Note, that in the context of this clause we may assume $\alpha = \mathit{futact}(f_k)$ by Property 4.2. The clause ensures that the types assigned by Γ_{fut} coincide with the ones assigned by Γ_{act} to their home activity. Additionally, this final clause ensures that the request $Q(f_k)$ must have the type assigned by the future map Γ_{fut} for this future f_k with the PC that corresponds to the PC assigned by the security assignment in the home activity.

C. Running Example: Type System Checks Example

For the sake of argument, we illustrate the application of the type system with an inconsistent constraint on the assignment *sec* for the example in Section II-C. The extended implementation as discussed in Section III-E contains the following changed ord function (we repeat the code here for convenience).

$$\beta\big[\varnothing,\quad [\mathrm{ord} = \varsigma(y)\ \text{if } \mathit{this}.\mathrm{income}/10^3 \geq 1 \text{ then } 1 \text{ else } 0],\\ \mathrm{income} = \ldots\quad\big]$$

If we specify income as private, this extended version of the running example may contain an implicit illegal information flow. Any security assignment *sec* that fulfills the constraint must be fallacious since the call β_i.ord in the manager object α reveals information about the confidential (H) value of income. The type system rejects any such *sec* since no consistent type can be inferred for the configuration in this case as we illustrate next. The failed type checking thus proves that for the extended configuration all specifications would have to specify income $\mapsto L$ because the assumption income $\mapsto H$ was inconsistent.

The global classification is derived according to the visibility relation (Definition 3.1) from the example's configuration as $\delta_{\beta_i} \sqsubseteq \delta_\alpha$ for all i. To be able to type the call to β_i.ord in manager object α this method must be an L-method according to TYPE CALL. Hence, we need to have the following extended constraint on *sec*.

$$\mathit{sec} \supseteq \quad \{\mathrm{ord} \mapsto L, \mathrm{income} \mapsto H\}$$

The third clause of TYPE CONFIGURATION, i.e., $\bigcup_{\alpha \in \mathrm{dom}(C)} \mathit{ass}(\Gamma_{\mathit{act}}(\alpha)) \subseteq \mathit{sec}$, gives us the constraint $\mathit{ass}(\Gamma_{\mathit{act}}(\beta_i)) \supseteq \{\mathrm{ord} \mapsto L, \mathrm{income} \mapsto H\}$ since *sec* and $\mathit{ass}(\Gamma_{\mathit{act}}(\beta_i))$ are both functions.

We show now that β_i (for an arbitrary i in the configuration) cannot be typed with this type constraint. The final step in a type inference to arrive at a type $\Gamma_{act}(\beta_i)$ for β_i can only be an instance of TYPE OBJECT which looks as follows.

$$\textsc{Instance Type Object}\quad \frac{\begin{array}{c}\mathit{this}{:}\Sigma_{\beta_i} :: [\,]{:}\Sigma_{\beta_i} :: \varnothing; A(\mathrm{ord}) \vdash t_{\mathrm{ord}} : \Sigma_{\beta_i} \\ \mathit{this}{:}\Sigma_{\beta_i} :: [\,]{:}\Sigma_{\beta_i} :: \varnothing; A(\mathrm{income}) \vdash t_{\mathrm{income}} : \Sigma_{\beta_i}\end{array}}{\begin{array}{c}\varnothing; \sqcup\{A(\mathrm{ord}), A(\mathrm{income})\} \vdash \\ [\mathrm{ord} = \varsigma(y)\ t_{\mathrm{ord}}, \mathrm{income} = \varsigma(y)\ t_{\mathrm{income}}] : \Sigma_{\beta_i}\end{array}}$$

We write Σ_{β_i} for $\Gamma_{act}(\beta_i)$, $A = \mathit{sec}(\Sigma_{\beta_i})$, and $t_{\mathrm{ord}} =$ if *this*.income$/10^3 \geq 1$ then 1 else 0. In fact, a more technical definition of t_{ord} is

$$t_{\mathrm{ord}} = \quad (((\mathit{true}.\mathrm{if} := (\mathit{this}. > 0(\mathit{this}.\mathrm{div}_{10^3}(\mathit{this}.\mathrm{income})))\\ ).\mathrm{then} := 1).\mathrm{else} := 0).\mathrm{if}([\,]).$$

where *true* is a boolean object containing methods if, then, and else. The details of this boolean object and its typing as well as the details of the following abridged reasoning are contained in Appendices A and B. The main point that we can see from this implementation is that the type A(ord) is coerced by the type A(if), i.e., it must hold that $A(\mathrm{ord}) = A(\mathrm{if})$ in Σ_{β_i}. This is the case, because t_{ord} is a call to the method if. According to the rule TYPE CALL, the PC must thus be S_{if} which corresponds here to A(if) and coincides with the PC A(ord) in the above instance of TYPE OBJECT, i.e., $A(\mathrm{ord}) = A(\mathrm{if})$. Now, the remaining argument just shows that A(if) must be H. In short form, the reasoning for the latter goes as follows. By assumption, A(income) must be H. Thus according to TYPE CALL and VAL SELF, *this*.income is typeable only with PC as H. We must apply TYPE CALL twice, to type $\mathit{this}. > 0(\mathit{this}.\mathrm{div}_{10^3}(\mathit{this}.\mathrm{income}))$. The PC for typing this is H each time because it must be the same as the PC (named S_j) in typing the parameter (named t in the rule TYPE CALL) and the previous typing of the parameter *this*.income has a H-PC. The PC in the application of the rule TYPE UPDATE is then also coerced to H in the typing of the newly inserted body method l_j, here "if". Hence, this update coerces the PC S_{if} to be H, i.e., $A(\text{if})$ to be H. The two following updates do not change the security type of the method if. We are finished since as we have seen above $A(\text{if}) = A(\text{ord})$. Thus, $A(\text{ord})$ must be H and cannot be typed L as would be necessary to call this method remotely in α. The typing fails. We have a contradiction to the initially required specification that income be private. Since this was the only assumption, if follows by contraposition that income must be L to make the configuration typeable.

This illustrates the correctness of the type system by example: the configuration C of our running example cannot be typed with the constraint income $\mapsto H$ since any attempt to infer a type $\langle \Gamma_{act}, \Gamma_{fut}, sec\rangle$ for it fails. The type inference reveals the dependency between ord and income: a security leak because it would enable implicit information flows from β_i's private part to χ.

In the following, we provide general proofs showing that the type system is sound, i.e., it generally implies security not just for the example.

V. PROPERTIES

A. Preservation

Type safety includes always a preservation theorem: if a program can be typed, the type has to be preserved by the evaluation of the program – otherwise the guarantees encoded in the types would be lost. In our case, since configurations dynamically change during the evaluation with the reduction relation $\rightarrow_{\parallel}$, the preservation has a slightly unusual form as the configuration type actually changes. But this change is conservative, i.e., dynamically created new elements are assigned new types but old types persist, as represented below by $\subseteq$. Alongside the configuration types, also the security class lattice is extended likewise in a conservative way by extension of the visibility relation.

Theorem 1 (Preservation):

$$\left(\begin{array}{l} \vdash C : \langle \Gamma_{act}, \Gamma_{fut}, sec\rangle \\ C \rightarrow_{\parallel} C' \end{array}\right) \Rightarrow \exists\, \Gamma'_{act}, \Gamma'_{fut}.\ \vdash C' : \langle \Gamma'_{act}, \Gamma'_{fut}, sec\rangle$$

where $\Gamma_{act} \subseteq \Gamma'_{act}$ and $\Gamma_{fut} \subseteq \Gamma'_{fut}$.

The proof of this theorem has two parts. The first part shows a local preservation property for the part of the type system that describes secure method calls at the level of objects, i.e., the rules depicted in Tables II and III. The second part of the proof addresses the typing rules at the global level, i.e., the configuration typing rules depicted in Table IV. Both proofs are straightforward using the induction schemes corresponding to the inductive rule definitions of the type rule definitions. Albeit the relatively small size of the computation model ASP$_{\text{fun}}$, these rules are fairly complex. Hence to avoid mistakes in these proofs we have formalized them in Isabelle/HOL. The Isabelle/HOL sources can be found at https://sites.google.com/site/floriankammueller/home/resources.

B. Confinement

Confinement is the property of our type system encoding the principal idea of the security model: if a method of an object can be called remotely, it must be a public L method. As a preparation to proving confinement, we present next a chain of lemmas that lead up to it. Let o be an object and T be an arbitrary type environment throughout the following formal statements.

The type rules for subsumption allow that types of objects can be "lifted", i.e., objects can have more than one type. We lose uniqueness of type judgments. To overcome this, we use a well-known trick (already been used in the Hindley-Milner type system for ML to accommodate polymorphic types) to regain some kind of uniqueness: minimal types.

Definition 5.1 (Minimal Type): Define the minimal type in the PC context of $S \in \{L, H\}$ as follows.

$$T; S \vdash_{\text{ML}} o : (A, \delta) \equiv \left\{ \begin{array}{l} T; S \vdash o : (A, \delta) \wedge \\ \forall S', A', \delta'. \\ T; S' \vdash o : (A', \delta') \Rightarrow \left(\begin{array}{c} S \leq S' \\ \delta \sqsubseteq \delta' \\ A \subseteq A' \end{array} \right) \end{array} \right.$$

This provides at least that minimal types of local typings are unique.

Lemma 5.2 (Minimal Type Uniqueness): Let $S \in \{L, H\}$. If $T; S \vdash_{\text{ML}} o : (A, \delta)$ and $T; S \vdash_{\text{ML}} o : (A', \delta')$, then $\delta = \delta'$.

A slightly stronger form of that previous lemma exists for H PCs.

Lemma 5.3 (High PC Uniqueness): If $T; H \vdash o : (A, \delta)$ and $T; H \vdash_{\text{ML}} o : (A, \delta')$, then $\delta = \delta'$.

Using slight generalization and contraposition, the previous lemma can be strengthened to the following key lemma for confinement.

Lemma 5.4 (Abstract Confinement): If $T; S \vdash o : (A, \delta)$ and $\delta_0 \sqsubset \delta$ and $T; S_0 \vdash_{\text{ML}} o : (A, \delta_0)$, then $S_0 = L$.

The following key fact, about the minimal type for futures provides the anchor to apply Abstract Confinement and arrive at Confinement.

Proposition 5.5 (Minimal Future Type): Let $\vdash C : \langle \Gamma_{act}, \Gamma_{fut}, sec\rangle$, $f_k \in \text{dom}(\Gamma_{fut})$, and $\alpha = \mathit{futact}(f_k)$ the home activity of f_k. Then

$$\langle \Gamma_{act}, \Gamma_{fut}, sec\rangle, A_\alpha(\mathit{futlab}(f_k)) \vdash_{\text{ML}} f_k : \Gamma_{act}(\alpha)\,.$$

Theorem 2 (Confinement): If a future f_k is typeable with an arbitrary PC S as of type δ strictly larger than the global level of f_k's home activity α, then f_k has been initially generated from a call to an L method of α. Formally, let $\vdash C : \langle \Gamma_{act} \cup sec, \Gamma_{fut}\rangle$, $\alpha[Q, a] \in C$, and $f_k \in \text{dom}(Q)$ with

$$\langle \Gamma_{act}, \Gamma_{fut}, sec\rangle, T; S \vdash f_k : (A_x, \delta) \text{ where } \Gamma_{act}(\alpha) \sqsubset \delta\,.$$

Then

$$A_\alpha(\mathit{futlab}(f_k)) = L\,.$$

The proof of confinement is basically just a combination of Lemma 5.4 and Proposition 5.5. The chain of lemmas and confinement have been proved in Isabelle/HOL as well.

C. Noninterference

Confinement can be considered as a simple security property because it is similar to a safety property: confinement is preserved on every trace of execution of a configuration. Intuitively it seems to imply confidentiality of private parts but this is only true for direct information flows. Confidentiality necessitates that no information is leaked to an outsider even considering implicit information flows as described in Section III-C. Based on those observations, we define the general property of confidentiality as *noninterference*, informally meaning that an attacker cannot learn anything despite his ability to observe configurations on all runs while comparing values that he can see: a difference in the value of the same call allows deductions about a change in hidden parts. The formal definition of noninterference for active objects in general [21] is a bisimulation over the indistinguishability relation $\sim_\alpha$ on configurations. We omit the rather technical definition of indistinguishability referring to Appendix D. Essentially, indistinguishability says that C and C_1 appear equal to the attacker α's viewpoint even if they differ in secret parts; noninterference means that this appearance is preserved by the evaluation of configurations.

Definition 5.6 (α-Noninterference): If configuration C is indistinguishable to any C_1 for α with respect to *sec* and remains so under the evaluation of configurations $\rightarrow_\parallel$, then C is α-noninterfering. Formally, we define α-noninterference C *sec* as follows.

$$\begin{pmatrix} C \rightarrow_\parallel C' \\ C \sim_\alpha C_1 \end{pmatrix} \Longrightarrow \exists\, C_1'. \begin{pmatrix} C_1 \rightarrow_\parallel^* C_1' \\ C' \sim_\alpha C_1' \end{pmatrix}$$

A main result for our security type system is *soundness*: a well-typed configuration is secure; α-noninterference holds for the configuration, i.e., it does not leak information.

Theorem 3 (Soundness): For any well-typed configuration C, we have noninterference with respect to $\alpha \in C$, i.e.,

$$\begin{pmatrix} \vdash C : \langle \Gamma_{act}, \Gamma_{fut}, sec \rangle \\ \alpha[Q, a] \in C \end{pmatrix} \Rightarrow \alpha\text{-noninterference } C \; sec\,.$$

The proof of this theorem is a case analysis distinguishing the cases where a reduction step of the configuration has happened in the α-visible part or outside it. In the latter case, a difference in the visible part would mean a breach of confinement. Within the visible part, a straightforward case analysis shows that what is possible in one configuration must also be possible in the other, indistinguishable, one, since those parts are isomorphic; hence the same reduction rules apply. We have formalized the definitions of indistinguishability, noninterference, and multi-lateral security, as well as the statements of the theorems in Isabelle/HOL – only the soundness proof is not yet formalized but a detailed paper proof is contained in Appendix E.

The parameterization of the attacker as an active object α grants the possibility to adapt the noninterference predicate. If we universally quantify α in our definition of noninterference, we obtain a predicate where each object could be the attacker corresponding to multi-lateral security.

Definition 5.7 (Multi-Lateral Security): If a configuration C is α-noninterfering for all $\alpha \in \mathrm{dom}(C)$ then multi-lateral security holds for C.

Since no α is fixed in the type statement, the soundness theorem holds for any α if the configuration is well-typed. Hence, well-typing implies immediately multi-lateral security.

VI. Related Work and Conclusions

The main difference of our approach is that we specifically address functional active objects. We also use a non-standard security model [21] for multi-lateral security tailored to distributed active objects. Other work on actor security, e.g. [20], is based on message passing models different to our high level language model. The paper [4] addresses only direct information flows in active objects. The priority program Reliably Secure Software Systems (RS3) of the German Research Foundation (DFG) [23] addresses in its part project MoVeSPAcI [29] security of actor systems using an event based approach without futures.

The Distributed Information Flow Control (DIFC) approach [27] provides support for Java programs (Jif) to annotate programs with labels "Alice" and "Bob" for information flow control. In this approach objects are not first class citizen. The formal model [37] uses a lambda calculus λ_{DSec} to accommodate the rich hierarchy of labels but (Java) objects are not in the calculus. They use an elegant approach to prove noninterference of a type system for labels pioneered by [30] Pottier and Simonet. This approach does not apply to parallel languages since the evaluation order of parallel processes is not deterministic. The language based approach offers the first model of language based information flow control for concurrency [36], later refined by Boudol and Castellani addressing scheduling problems and related timing leaks. Many works have followed this methodology (see [35] for an overview). However, most works consider imperative while languages with various extensions like multi threading. Sabelfeld and Mantel consider message passing in distributed programs [34], [24]. These works use the secure channel abstraction, i.e. connecting remote processes of the same security class via secure channels integrating security primitives.

Distributed security has also been considered in many works in the setting of process algebras most prominently using pi calculus by [26] (see [15] and [33] providing overviews). Commonalities of process algebra based security to our work are the bisimulation notion of noninterference and asynchronous communication. There is a line of research on mobile calculi that use purely functional concurrent calculi. A few representative papers are by [19] on the pi calculus and [17] for the security pi calculus. An impressive approach on information flows for distributed languages with mobility and states [25] first introduces declassification. Similar work is by [5] also studying noninterference for distribution and mobility for Boxed Ambients. In common with these works are modeling distributed system by a calculus but none of the pi calculus related work focuses on active objects while we do not consider cryptographic primitives. An interesting perspective would be to investigate the relationship between confinement and effects of cryptographic primitives. We also use a bisimulation-based equivalence relation to express noninterference. In the applied pi calculus, for example, the notion of a static equivalence, similar to our indistinguishability is used in addition to observational equivalence that corresponds to our notion of noninterference (see e.g. [9]).

This work presented a formal framework for the security of active objects based on the semi-lattice security model that propagates confinement. We presented a safe security type system, that verifies the confinement property and is sound, i.e., checks security, with respect to a dedicated formal notion of noninterference, or more generally, multi-lateral security. ASP$_{\text{fun}}$ makes secure down-calls possible and is still applicable bi-directionally as illustrated by implementing the NSPK protocol. The proofs have been in large parts formalized in Isabelle/HOL. An implementation of functional active objects is given by Erlang Active Objects in [13]. providing a simple extension by a run-time monitor for confinement. [14].

References

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

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "Towards a theory of actor computation (extended abstract)," in *CONCUR'92: Proceedings of the Third International Conference on Concurrency Theory*, W. R. Cleaveland, Ed. Berlin, Heidelberg: Springer-Verlag, 1992, pp. 565–579.

[3] R. Anderson, *Security Engineering – A Guide to Building Dependable Distributed Systems*. Wiley, 2001.

[4] I. Attali, D. Caromel, L. Henrio, and F. L. D. Aguila, "Secured information flow for asynchronous sequential processes," *Electr. Notes Theor. Comput. Sci.*, vol. 180, no. 1, pp. 17–34, 2007.

[5] M. Bugliesi, G. Castagna, and S. Crafa, "Access control for mobile ambients: the calculus of boxed ambients," *ACM Transactions on Programming Languages and Systems*, vol. 26, no. 1, pp. 57–124, 2004.

[6] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton, "ProActive: an integrated platform for programming and running applications on grids and P2P systems," *Computational Methods in Science and Technology*, vol. 12, no. 1, pp. 69–77, 2006.

[7] D. Caromel, L. Henrio, and B. P. Serpette, "Asynchronous and deterministic objects," in *Proceedings of the 31st ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM Press, 2004, pp. 123–134.

[8] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order (2. ed.)*. Cambridge University Press, 2002.

[9] S. Delaune, S. Kremer, and M. D. Ryan, "Symbolic bisimulation for the applied pi calculus," *Journal of Computer Security*, vol. 18, no. 2, pp. 317–377, 2010.

[10] D. E. Denning, "Lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–242, 1976.

[11] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, 1977.

[12] J. S. Fenton, "Information protection systems," Ph.D. dissertation, University of Cambridge, 1973.

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

[14] ——, "A security model for functional active objects with an implementation in erlang," in *Computational Informatics, Social Factors and New Information Technologies: Hypermedia Perspectives and Avant-Garde Experiences in the Era of Communicability Expansion*. Blue Herons, 2011.

[15] R. Focardi and R. Gorrieri, "A taxonomy of security properties for process algebras," *Journal of Computer Security*, vol. 3, no. 1, pp. 5–34, 1995.

[16] R. H. Halstead, Jr., "Multilisp: A language for concurrent symbolic computation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 4, pp. 501–538, 1985.

[17] M. Hennessy and J. Riely, "Information flow vs. resource access in the asynchronous pi-calculus," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 5, pp. 566–591, 2002.

[18] L. Henrio, F. Kammüller, and B. Lutz, "Aspfun: A typed functional active object calculus," *Science of Computer Programming*, vol. 77, no. 7-8, pp. 823–847, 2012.

[19] K. Honda, V. T. Vasconcelos, and N. Yoshida, "Secure information flow as typed process behaviour," in *ESOP*, ser. Lecture Notes in Computer Science, G. Smolka, Ed., vol. 1782. Springer, 2000, pp. 180–199.

[20] D. Hutter, H. Mantel, I. Schaefer, and A. Schairer, "Security of multi-agent systems: A case study on comparison shopping," *J. Applied Logic*, vol. 5, no. 2, pp. 303–332, 2007.

[21] F. Kammüller, "A semi-lattice model for multi-lateral security," in *Data Privacy Management, DPM'12, Seventh International Workshop. Co-located with ESORICS'12*, ser. LNCS, Security and Cryptology, vol. 7731. Springer, 2013.

[22] G. Lowe, "An attack on the needham-schroeder public-key authentication protocol," *INFORMATION PROCESSING LETTERS*, vol. 56, pp. 131–133, 1995.

[23] H. Mantel, D. Hutter, G. Snelting, and T. Nipkow, "Reliably secure software systems – Priority Program of the German Research Foundation (DFG)," 2010, http://www.reliably-secure-software-systems.de.

[24] H. Mantel and A. Sabelfeld, "A unifying approach to the security of distributed and multi-threaded programs," *J. Computer Security*, vol. 11, p. 2003, 2002.

[25] A. A. Matos and J. Cederquist, "Non-disclosure for distributed mobile code," *Mathematical Structures in Computer Science*, vol. 21, no. 6, pp. 1111–1181, 2011.

[26] R. Milner, *Communication and concurrency*, ser. PHI Series in computer science. Prentice Hall, 1989.

[27] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proceedings of the sixteenth ACM symposium on Operating systems principles*, ser. SOSP '97. New York, NY, USA: ACM, 1997, pp. 129–142. [Online]. Available: http://doi.acm.org/10.1145/268998.266669

[28] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Communications of the ACM*, no. 21, 1978.

[29] A. Poetzsch-Heffter and C. Feller, "Modular verification of security properties in actor implementations (movespaci)," 2011, project of the German Research Foundation, Priority Program RS3. [Online]. Available: http://softech.informatik.uni-kl.de/Homepage/MoVeSPAcI

[30] F. Pottier and V. Simonet, "Information flow inference for ml," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 1, pp. 117–158, 2003.

[31] C. W. Probst and R. R. Hansen, "An extensible analysable system model," *Information Security Technical Report*, vol. 13, no. 4, pp. 235–246, Nov. 2008.

[32] C. W. Probst, J. Hunker, D. Gollmann, and M. Bishop, Eds., *Insider Threats in Cybersecurity*. Springer, 2010.

[33] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe, *The Modelling and Analysis of Security Protocols*. Addison-Wesley, 2000.

[34] A. Sabelfeld and H. Mantel, "Static confidentiality enforcement for distributed programs." in *Static Analysis Symposium, SAS'02*, ser. LNCS, vol. 2477. Springer, 2002.

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

[36] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," in *POPL'98*. ACM, 1998.

[37] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," *International Journal of Information Security*, vol. 6, no. 2–3, 2007.

Appendix

A: Booleans and conditional in the ς-calculus and their security types

To prepare for the type inference, we need the implementation of the boolean datatype and the *if-then-else* in the ς-calculus, i.e. in the local calculus of ASP$_{\text{fun}}$.

$$\begin{aligned}
\text{true} &= [\,\text{if} = \varsigma(y)\textit{this}.\text{then}(y),\\
&\qquad \text{then} = \varsigma(y)[\,], \text{else} = \varsigma(y)[\,]\,]\\
\text{false} &= [\,\text{if} = \varsigma(y)\textit{this}.\text{else}(y),\\
&\qquad \text{then} = \varsigma(y)[\,], \text{else} = \varsigma(y)[\,]\,]\\
\text{if } b \text{ then } c \text{ else } d &=\\
&\quad ((b.\text{then} := \varsigma(y)c).\text{else} := \varsigma(y)d).\text{if}([\,])
\end{aligned}$$

In the third line above, $\textit{this}, y \notin FV(c) \cup FV(d)$; [] denotes the empty object. The definition shows how – similar to λ-calculus – the functionality of the constructor is encoded in the elements of the datatype: when b is true its method if delegates to the method then, filled with term c, when false, if delegates to else, executing term d. Typing of the *if-then-else* construct is a base test for an information flow type system as this construct is the basic example that gives rise to implicit information flows. We will thus here illustrate how the security type rules presented in this paper establish that the guard of the *if-then-else* construct, the if, must be typed with the same PC as the branches, i.e., then and else. Then it immediately follows that if the method if has $H\text{-}PC$ then the branches must have $H\text{-}PC$ as well. The reasoning instantiates type rules showing the constraints that follow for the security assignment in the security type Σ_{ifte}. A condition b in the method if of an *if-then-else* object evaluates to either $true$ or $false$. We consider those two possibilities and infer their types and the resulting constraints. To type *true*, we initially type *this* which can be done only by rule VAL SELF leading to the following typing where $\Sigma_{\text{ifte}} = (A_{\text{ifte}}, \delta_{\text{ifte}})$ is the security type for the *if-then-else* object and $M_{\text{ifte}} = \sqcup\{A_{\text{ifte}}(\text{if}), A_{\text{ifte}}(\text{then}), A_{\text{ifte}}(\text{else})\}$. We use the arbitrary set of additional type assumptions T provided by the rule to integrate the type assumption for y already here. It is needed further down for typing the object but only formally.

INSTANCE VAL SELF
$$\textit{this}:\Sigma_{\text{ifte}} :: (y:\Sigma_{\text{ifte}}) :: T; M_{\text{ifte}} \vdash \textit{this}:\Sigma_{\text{ifte}}$$

We then apply the rule TYPE CALL to infer a type for *this*.then. The following instance of that rule sets the parameters such that it can be applied to the previous INSTANCE VAL SELF.

INSTANCE TYPE CALL
$$\frac{\begin{array}{c}\textit{this}:\Sigma_{\text{ifte}} :: y:\Sigma_{\text{ifte}} :: T; M_{\text{ifte}} \vdash \textit{this}:\Sigma_{\text{ifte}}\\ \textit{this}:\Sigma_{\text{ifte}} :: y:\Sigma_{\text{ifte}} :: T; A_{\text{ifte}}(\text{then}) \vdash [\,]:\Sigma_{\text{ifte}}\end{array}}{\textit{this}:\Sigma_{\text{ifte}} :: y:\Sigma_{\text{ifte}} :: T; A_{\text{ifte}}(\text{then}) \vdash \textit{this}.\text{then}([\,]):\Sigma_{\text{ifte}}}$$

Now, to type the *true* object including its fields then and else we next need an instance of TYPE OBJECT.

INSTANCE TYPE OBJECT
$$\frac{\begin{array}{c}\textit{this}:\Sigma_{\text{ifte}} :: y:\Sigma_{\text{ifte}} :: T; A_{\text{ifte}}(\text{if}) \vdash \textit{this}.\text{then}([\,]) : \Sigma_{\text{ifte}}\\ \textit{this}:\Sigma_{\text{ifte}} :: y:\Sigma_{\text{ifte}} :: T; A_{\text{ifte}}(\text{then}) \vdash [\,] : \Sigma_{\text{ifte}}\\ \textit{this}:\Sigma_{\text{ifte}} :: y:\Sigma_{\text{ifte}} :: T; A_{\text{ifte}}(\text{else}) \vdash [\,] : \Sigma_{\text{ifte}}\end{array}}{T; M_{\text{ifte}} \vdash \text{true} = [\,\text{if} = \varsigma(y)\textit{this}.\text{then}(y), \text{then} = \varsigma(y)[\,], \text{else} = \varsigma(y)[\,]\,] : \Sigma}$$

The main observation is that the following constraint must hold for A_{ifte}

$$A_{\text{ifte}}(\text{if}) = A_{\text{ifte}}(\text{then})$$

because this is necessary for the first proviso of the above instance to be matched with the previous type derivation for *this*.then([]) by INSTANCE TYPE CALL.

With a very similar argument for typing *false*, i.e., *this*.else([]), we arrive at a similar constraint.

$$A_{\text{ifte}}(\text{if}) = A_{\text{ifte}}(\text{else})$$

Since for an arbitrary *if-then-else* guard b we have to allow both values *true* and *false* as possible outcome we have to combine the constraints and conclude for A_{ifte} the following overall constraint.

$$A_{\text{ifte}}(\text{if}) = A_{\text{ifte}}(\text{then}) = A_{\text{ifte}}(\text{else})$$

The update of the methods then and else does not change the PC and thus preserves the security assignment and the constraints. This constraint is what we expect for information flow security. If the guard of an *if-then-else* can only be typed in a $H\text{-}PC$ then its branches must also be "lifted" to H. Only if the guard can be typed in a $L\text{-}PC$, can the branches also be typed in $L\text{-}PC$.

Note on typing constants

In the above type rule instances we have used typings for constants, for example, the empty object [] as granted and did not refine them any further.

$$\textit{this}:\Sigma_{\text{ifte}} :: y:\Sigma_{\text{ifte}} :: T; A_{\text{ifte}}(\text{else}) \vdash [\,] : \Sigma_{\text{ifte}}$$

A word is in order to explain how these are constructed and their types are derived. A simple way to integrate the empty object into an activity is to add a method empty and then replace all [] by *this*.empty. The security assignment should be $A(\text{empty}) = L$. Then, we can use TYPE CALL to have $\ldots L \vdash \textit{this}.\text{empty} : \Sigma_{\text{ifte}}$ and from there derive the above $\ldots; A_{\text{ifte}}(\text{else}) \vdash \textit{this}.\text{empty} : \Sigma_{\text{ifte}}$. However, [] (and other commonly used plain objects) can more practically be considered as *activities without any H methods* that are included as a "data base" in a configuration. Then, an occurrence of [] is literally the activity named "empty object", i.e., [] is an activity reference. For the typing, the natural type of the empty object is given as the empty security assignment $\varnothing$, i.e., the partial function that is undefined for all inputs, and the bottom element $\bot$ of the visibility semi-lattice which corresponds to the empty set of activity names.

$$\langle \Gamma_{act}, \Gamma_{fut}, sec\rangle; L \vdash [\,] : (\varnothing, \bot)$$

By definition this typing with $L\text{-}PC$ as $(\varnothing, \bot)$ for the empty object enables typing [] "into" any other activity type (A, δ) because $\varnothing \subseteq A$ and $\bot \sqsubseteq \delta$. Thus – by SECASS SUBSUMPTION and GLOB SUBSUMPTION – $\ldots; L \vdash [\,] : (A, \delta)$.

A similar type and subtyping argumentation goes for other constants, for example 0 or 1, used in the running example. Similar to Church numerals simple term representation can be given to them in ASP$_{\text{fun}}$. Such constant activities η must have their methods all assigned to L, i.e., their security assignment A maps all method names of η to L. Then the PC of the activity η is also L because it is given as $\sqcup\{L\}$ according to the rule TYPE CONFIGURATION. The global level of a constant activity like η is defined as the set $\{eta\}$. If the constant η is used by referencing it in other activities of the configuration, the name η becomes part of the other activities' global levels.

B: Running Example – Details on Typing

The following shows why the example configuration presented as running example cannot be typed with income $\mapsto H \in sec$.

Implementation: The quicksort function is described in Section II-C. The manager activity that controls the ordering of a list and the sorting object χ that calls the ord method in β-objects are repeated here for convenience of the reader.

$$\begin{aligned}
\alpha\big[\varnothing, [&\text{manage} = \varsigma(y)\textit{this}.\text{sort}(\textit{this}.\text{list}),\\
&\text{sort} = \varsigma(y)\ \chi.\text{qsort}(y),\\
&\text{list} = \ldots]\big]
\end{aligned}$$

```
χ[∅,
  [qsort = ς(y) if y = [] then []
                 else let (a :: l) = y
                          (l_1, l_2) = this.part (a, l)
                          l'_1 = if #l_1≤1 then l_1 else this.qsort(l_1)
                          l'_2 = if #l_2≤1 then l_2 else this.qsort(l_2)
                      in l'_1@[a]@l'_2
                 end,
   part = ς(p, y) if y = [] then ([], [])
                  else let (a::l) = y
                           (l_1, l_2) = this.part (p, l)
                       in if p<a.ord then (l_1, a::l_2) else (a::l_1, l_2)
                  end
  ]]
```

The extended method ord that bears a dependency between ord and income,

$$\beta\big[\varnothing,\quad \begin{array}{l}[\text{ord} = \varsigma(y) \text{ if } \textit{this}.\text{income}/10^3{\geq}1 \text{ then } 1 \text{ else } 0],\\ \text{income} = \ldots \quad \big]\end{array}$$

is not typeable for any security assignment *sec* that imposes the constraint that method income $\mapsto H$. The following type inference elaborates that the type system rejects any security assignment that contains the constraint income $\mapsto H$. It illustrates how the security assignment $A_{\beta_i} = \textit{ass}(\Gamma_{\textit{act}}(\beta_i))$ is inferred.

Typing remote call implies ord $\mapsto L$

Since the method ord is called remotely in χ via α we need that ord $\mapsto L$, which cannot be possible because of the dependency in the above implementation. To be able to type the call to β_i.ord in the object χ this method must be an L-method according to TYPE CALL and GLOB SUBSUMPTION. More precisely, let $(A_{\beta_i}, \delta_{\beta_i}) = \Gamma_{\textit{act}}(\beta_i)$. We have that $\langle\Gamma_{\textit{act}}, \Gamma_{\textit{fut}}, \textit{sec}\rangle, \varnothing; M_{\beta_i} \vdash \beta_i : (A_{\beta_i}, \delta_{\beta_i})$ because of TYPE ACTIVE OBJECT REFERENCE and $\beta_i \in \text{dom}(C)$. The PC is $M_{\beta_i} = \sqcup_{j\in \text{dom}A_{\beta_i}} A_{\beta_i}(j)$ where A_{β_i} needs to be inferred in the process. We can use next TYPE CALL to type $\langle\Gamma_{\textit{act}}, \Gamma_{\textit{fut}}, \textit{sec}\rangle, \varnothing; A_{\beta_i}(\text{ord}) \vdash \beta_i.\text{ord} : (A_{\beta_i}, \delta_{\beta_i})$. However, to type β_i.ord in the context of the object χ it needs to be typed as $(A_{\beta_i}, \delta_{\chi})$ with global type component δ_χ. This upgrading of the call can only be achieved by application of rule GLOB SUBSUMPTION which requires that $\delta_{\beta_i} \sqsubseteq \delta_\chi$ which is true but also requires that the PC of the typing $\langle\Gamma_{\textit{act}}, \Gamma_{\textit{fut}}, \textit{sec}\rangle, \varnothing; A_{\beta_i}(\text{ord}) \vdash \beta_i.\text{ord} : (A_{\beta_i}, \delta_{\beta_i})$, i.e., $A_{\beta_i}(\text{ord})$, is L.

Typing β_i.ord at global level δ_{β_i} only with H-PC

The next part of the argument states that the only type that can be inferred for a call β_i.ord is $T; H \vdash \beta_i.\text{ord} : (A_{\beta_i}, \delta_{\beta_i})$, i.e., with H-PC. This is because types for calls can only be inferred by rule CALL and $A_{\beta_i}(\text{ord}) = H$ which coerces the PC according to rule CALL to H. For clarity of the exposition, we omit in the following $\langle\Gamma_{\textit{act}}, \Gamma_{\textit{fut}}, \textit{sec}\rangle$ in front of the typings. Since we want to arrive at $\vdash C : \langle\Gamma_{\textit{act}}, \Gamma_{\textit{fut}}, \textit{sec}\rangle$, by the inversion principle of inductive type definitions, all provisos of TYPE CONFIGURATION have to be true. Since

$$\beta_i[\varnothing, [\text{ord} = \varsigma(y)\ t_{\text{ord}}, \text{income} = \varsigma(y)\ t_{\text{income}}]] \in C,$$

the fourth proviso, first clause, of TYPE CONFIGURATION yields

$$\varnothing, M_{\beta_i} \vdash [\text{ord} = \varsigma(y)\ t_{\text{ord}}, \text{income} = \varsigma(y)\ t_{\text{income}}] : (A_{\beta_i}, \delta_{\beta_i}).$$

The coercion of $A_{\beta_i}(\text{ord})$ to H is a consequence of the typing of the object β_i with an instance of rule TYPE OBJECT.

INSTANCE TYPE OBJECT

$$\frac{\begin{array}{c}\textit{this}:\Sigma_{\beta_i} :: []:\Sigma_{\beta_i} :: \varnothing; A_{\beta_i}(\text{ord}) \vdash t_{\text{ord}} : \Sigma_{\beta_i}\\ \textit{this}:\Sigma_{\beta_i} :: []:\Sigma_{\beta_i} :: \varnothing; A_{\beta_i}(\text{income}) \vdash t_{\text{income}} : \Sigma_{\beta_i}\end{array}}{\varnothing; M_{\beta_i} \vdash [\text{ord} = \varsigma(y)\ t_{\text{ord}}, \text{income} = \varsigma(y)\ t_{\text{income}}] : \Sigma_{\beta_i}}$$

This instance enforces $A(\text{ord})$ to be the same as the PC in the typing of

$$\begin{array}{rcl} t_{\text{ord}} &=& \text{if } \textit{this}.\text{income}/10^3{\geq}1 \text{ then } 1 \text{ else } 0\\ &=& (((\textit{true}.\text{if} := (\textit{this}. > 0(\textit{this}.\text{div}_{10^3}(\textit{this}.\text{income})))\\ && \quad).\text{then} := 1).\text{else}{:=}\ 0).\text{if}([]). \end{array}$$

The only way to arrive at a type for t_{ord} is by an application of TYPE CALL as in the following instance.

INSTANCE TYPE CALL

$$\frac{\begin{array}{c}T; S \vdash (((\textit{true}.\text{if} := (\textit{this}. > 0(\textit{this}.\text{div}_{10^3}(\textit{this}.\text{income}))))\\ .\text{then} := 1).\text{else}{:=}\ 0) : \Sigma\\ T; A_{\beta_i}(\text{if}) \vdash [] : \Sigma\end{array}}{T; A_{\beta_i}(\text{if}) \vdash t_{\text{ord}} : \Sigma}$$

In order to match the conclusion of the above with the first proviso of the earlier INSTANCE TYPE OBJECT, the security assignment of ord is coerced to that of if

$$A_{\beta_i}(\text{ord}) = A_{\beta_i}(\text{if}).$$

We only need to show that $A_{\beta_i}(\text{if}) = H$ and we are finished.

Typing implies that A_{β_i}(if) $= H$

The following chain of steps shows how a type for the body of ord and thus $A_{\beta_i}(\text{ord})$ must be inferred detailing how the security assignment parameter $A_{\beta_i}(\text{if})$ needs to be instantiated to H. The chain of reasoning starts from the one specified security assignment income $\mapsto H$ in *sec* and shows that then also ord $\mapsto H$ which contradicts the above ord $\mapsto L$. Hence, no type can exist with the constraint income $\mapsto H$ for this configuration.

$A_{\beta_i}(\text{income})$ is H by constraint on *sec* and thus A_{β_i}. According to VAL SELF with

$$M = \sqcup\{A_{\beta_i}(\text{income}), A_{\beta_i}(\text{ord}), \ldots\} = H$$

we get the following typing for *this*.

$$T; H \vdash \textit{this} : (A_{\beta_i}, \delta_{\beta_i})$$

According to TYPE CALL, *this*.income is typeable only with PC as H since $\{\text{income} \mapsto H\} \subseteq A_{\beta_i}$.

$$T; H \vdash \textit{this}.\text{income} : (A_{\beta_i}, \delta_{\beta_i})$$

The previous typing feeds into rule TYPE CALL again but this time for the parameter t. Since the PC S_j matches with H we get again a H-PC coercing the method div_{10^3} also to be assigned to H in A_{β_i}.

$$T; H \vdash \textit{this}.\text{div}_{10^3}(\textit{this}.\text{income}) : (A_{\beta_i}, \delta_{\beta_i})$$

In the same fashion, the previous considered as a parameter typing TYPE CALL consequently coerces $A_{\beta_i}(> 0)$ also to H:

$$T; H \vdash \textit{this}. > 0(\textit{this}.\text{div}_{10^3}(\textit{this}.\text{income})) : (A_{\beta_i}, \delta_{\beta_i})$$

We instantiate UPDATE as follows.

INSTANCE TYPE UPDATE

$$\frac{\begin{array}{c}\varnothing; S \vdash \text{true} : (A_{\beta_i}, \delta_{\beta_i})\\ \textit{this}:\Sigma_{\beta_i} :: []:\Sigma_{\beta_i} :: \varnothing; A_{\beta_i}(\text{if}) \vdash\\ \textit{this}. > 0(\textit{this}.\text{div}_{10^3}(\textit{this}.\text{income})) : (A_{\beta_i}, \delta_{\beta_i})\end{array}}{\begin{array}{c}\varnothing; \sqcup\{A_{\beta_i}(\text{if}), \ldots\}) \vdash\\ \text{true.if}[] := \textit{this}. > 0(\textit{this}.\text{div}_{10^3}(\textit{this}.\text{income})) : (A_{\beta_i}, \delta_{\beta_i})\end{array}}$$

The first proviso, the typing for *true* can be inferred as shown in the previous section, using rule SEC ASS SUBSUMPTION in addition to embed it into β_i. We spell out some portion of $M_{\beta_i} = \sqcup\{A_{\beta_i}(\text{if}), \ldots\}$ above to emphasize that $A_{\beta_i}(\text{if})$ is part of the PC. The dots stand for $A_{\beta_i}(\text{ord})$ and $A_{\beta_i}(\text{income})$ etc. To match the previous derivation above of $\varnothing; H \vdash \textit{this}. > 0(\textit{this}.\text{div}_{10^3}(\textit{this}.\text{income}))$:

$(A_{\beta_i}, \delta_{\beta_i})$ to the second proviso in the above instance it is necessary to coerce

$$A_{\beta_i}(\text{if}) = H.$$

We are finished here already because we have already shown above that $A_{\beta_i}(\text{if}) = A_{\beta_i}(\text{ord})$ which thus is H contradicting the earlier requirement to be L.

For completeness, we continue the derivation of the body of t_{ord}. From the previous step above, we get the conclusion

$$\varnothing; H \vdash \text{true.if}[] := \textit{this}. > 0(\textit{this}.\text{div}_{10^3}(\textit{this}.\text{income})) : (A_{\beta_i}, \delta_{\beta_i}).$$

We can again instantiate the update rule.

$$\textsc{Instance Type Update}\quad \frac{\begin{array}{c}\varnothing; H \vdash \text{true.if}[] := \textit{this}. > 0(\textit{this}.\text{div}_{10^3}(\textit{this}.\text{income})) : (A_{\beta_i}, \delta_{\beta_i}) \\ \textit{this}: (A_{\beta_i}, \delta_{\beta_i}) :: y : (A_{\beta_i}, \delta_{\beta_i}) :: \varnothing; H \vdash 1 : (A_{\beta_i}, \delta_{\beta_i})\end{array}}{\varnothing; H \vdash \textit{this}. > 0(\textit{this}.\text{div}_{10^3}(\textit{this}.\text{income})).\text{then} := 1 : (A_{\beta_i}, \delta_{\beta_i})}$$

And a second time we instantiate TYPE UPDATE for 0 to finally obtain

$$\varnothing; H \vdash ((\textit{this}. > 0(\textit{this}.\text{div}_{10^3}(\textit{this}.\text{income}))).\text{then} := 1).\text{else} := 0 : (A_{\beta_i}, \delta_{\beta_i})$$

Running Example: Typing Summary

The coercions revealed in the above steps determine the parameter A_{β_i} in summary as follows.

$$A_{\beta_i} = [\text{income} \mapsto H,\ \text{div}_{10^3} \mapsto H,\ > 0 \mapsto H \text{ if} \mapsto H,\ \text{ord} \mapsto S_{\text{if}}]$$

I.e., the only possible instantiation for $A_{\beta_i}(\text{ord})$ is H. We cannot meet the required constraint $A_{\beta_i}(\text{ord}) = L$ necessary to call it from the outside in χ as explained initially. Therefore, the example configuration cannot be typed with the constraint income $\mapsto H$.

Borderline Example for Confinement

The confinement property states that remote calls can only be addressed to L methods. But does this simple security property guarantee that no hidden H methods can be returned with the reply to such a call? Consider the following example

$$\alpha[\varnothing, [\text{leak} = \varsigma(y)\textit{this}, \text{key} = \varsigma(y)n]]$$

where n is an integer representing a secret key. Let the security assignment for α be $\{\text{leak} \mapsto L, \text{key} \mapsto H\}$. One may think that an activity β could contain a call α.leak.key since the method leak is L enabling the remote call to α.leak. Once the call result is returned into β, it would evaluate to the active object of α inside β (since *this* represents this active object of α). Since we are now already in β, it might seem possible to apply the method key to extract the key.

How does the security type system prevent this? Since the typing for *this* inside α is only possible with the PC as $M_\alpha = \sqcup_{i \in \{\text{leak, key}\}} A_\alpha(i) = H$ (since $A_\alpha(\text{key}) = H$), the typing for *this* yields

$$\textsc{Instance Val Self} \qquad \textit{this}: \Sigma_\alpha :: T; H \vdash \textit{this} : (A_\alpha, \delta_\alpha).$$

Typing the object α must use the following instance of TYPE OBJECT.

$$\textsc{Instance Type Object}\quad \frac{\begin{array}{c}\textit{this}: \Sigma_\alpha :: y : \Sigma_\alpha :: \varnothing; A_\alpha(\text{leak}) \vdash \textit{this} : \Sigma_\alpha \\ \textit{this}: \Sigma_\alpha :: y : \Sigma_\alpha :: \varnothing; A_\alpha(\text{key}) \vdash n : \Sigma_\alpha\end{array}}{\varnothing; \sqcup\{A_\alpha(\text{leak}), A_\alpha(\text{key})\} \vdash [\text{leak} = \varsigma(y)\textit{this}, \text{key} = \varsigma(y)n] : \Sigma_\alpha}$$

Now, matching the instance of VAL SELF for *this* with the first proviso of the instance of TYPE OBJECT coerces $A_\alpha(\text{leak})$ to H contradicting the initial specification. I.e., the method leak is forced to be H and cannot be called remotely.

C: Formal Semantics of ASP$_{fun}$

For a concise representation of the operational semantics, we define contexts as expressions with a single hole ($\bullet$). A context $E[t]$ denotes the term obtained by replacing the single hole by t.

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

This notion of context is used in the formal semantics of ASP$_{\text{fun}}$ in Table V and also in the definition of visibility (see Definition 3.1).

D: Indistinguishability

In ASP$_{\text{fun}}$ active objects are created by activation, futures by method calls. Names of active objects and futures may differ in evaluations of the same configuration but this does not convey any information to the attacker. We use “partial bijections” to express the equality of the visible parts of a configuration.

Definition 6.1 (Typed Bijection): A typed bijection is a finite partial function σ on activities α (or futures f_k respectively) such that for a type T

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

By $t[\sigma, \tau] =|_{sec}\ t'$ 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, restricted to the label names in *sec*, i.e., in the object terms t and t' we exempt those parts of the objects that are private. 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 6.2 (Equality up to Name Isomorphism): An equality up to name isomorphism is a family of equivalence relations on ASP$_{\text{fun}}$ terms indexed by two typed bijections $(\sigma, \tau) := R$ and security assignment *sec* consisting of the following differently typed sub-relations; the sub-relation’s types are indicated by the naming convention: t for ς-terms, α, β for active objects, f_k, f_j for futures, Q_α, Q_β for request queues.

$$\begin{array}{rcl} t =_R t' & \equiv & t \Downarrow t_e \wedge t' \Downarrow t'_e \wedge\ t_e[\sigma, \tau] =|_{sec}\ t'_e \\ \alpha =_R \beta & \equiv & \sigma(\alpha) = \beta \\ f_k =_R f_j & \equiv & \tau(f_k) = f_j \\ Q_\alpha =_R Q_\beta & \equiv & \left(\begin{array}{l} \text{dom}(\tau) \supseteq \text{dom}(Q_\alpha) \\ \text{ran }(\tau) \supseteq \text{dom}(Q_\beta) \\ \forall\, f_k \in \text{dom}(Q_\alpha). \\ Q_\alpha(f_k) =_R Q_\beta(\tau(f_k)) \end{array}\right) \\ \alpha[Q_\alpha, t_\beta] =_R \beta[Q_\beta, t_\alpha] & \equiv & \alpha =_R \beta \wedge Q_\alpha =_R Q_\beta \wedge t_\alpha =_R t_\beta \end{array}$$

Such an equivalence relation defined by two typed bijections σ and τ may exist between given sets V_0, V_1 of active object names in C, C_1. If V, V_1 correspond to the viewpoints of attacker α in C and its counterpart in C_1 we call this equivalence relation indistinguishability. In the following, we use the *visibility range* based on Definition 3.1 as $VI_{sec}(\alpha, C) \equiv \{\beta \in \text{dom}(C) \mid \beta \sqsubseteq_C^{sec} \alpha\}$.

Definition 6.3 (Indistinguishability): Let C, C_1 be arbitrary configurations, well-typed with respect to a security specification *sec*, active object $\alpha \in \text{dom}(C)$ and $\alpha \in \text{dom}(C_1)$ (we exempt α from renaming for simplicity). Configurations C and C_1 are called indistinguishable with respect to α and *sec*, if α’s visibility ranges are the same in both up to name isomorphism.

$$C \sim_\alpha C_1 \equiv \exists\, \sigma, \tau. \left(\begin{array}{l} VI_{sec}(\alpha, C) = \text{dom}(\sigma) \\ VI_{sec}(\alpha, C_1) = \text{ran }(\sigma) \\ \forall\, \beta \in VI_{sec}(\alpha, C). \\ \quad C(\beta) =_{\sigma,\tau} C_1(\sigma(\beta)) \end{array}\right)$$

As an example for α-indistinguishable configurations consider the running example. In the original (non-fallacious) form, β_1.income could be 42 in configuration C and it could be 1042 in configuration

CALL
$$\frac{l_i \in \{l_j\}^{j\in 1..n}}{E\left[[l_j = \varsigma(y_j)s_j]^{j\in 1..n}.l_i(t)\right] \to_\varsigma E\left[s_i\{\mathit{this} \leftarrow [l_j = \varsigma(y_j)s_j]^{j\in 1..n}, y_i \leftarrow t\}\right]}$$

UPDATE
$$\frac{l_i \in \{l_j\}^{j\in 1..n}}{E\left[[l_j = \varsigma(y_j)s_j]^{j\in 1..n}.l_i := \varsigma(y)t\right] \to_\varsigma E\left[[l_i = \varsigma(y)t, l_j = \varsigma(y_j)s_j^{j\in(1..n)-\{i\}}]\right]}$$

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

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

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

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

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

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

TABLE V. ASP$_{\text{FUN}}$ SEMANTICS

C'. Since income is specified as H those two configurations can be considered as α-indistinguishable (with respect to β_1). Attacker α sees no difference between the two. In the fallacious example, however, he'd notice a difference when calling the quicksort algorithm that implicitly drafts information from income through ord: here, C and C' would be distinguishable since β_1.ord is 0 in C and 1 in C'.

E: Noninterference Proof

Theorem 2 (Soundness) For any well-typed configuration, we have noninterference with respect to $\alpha \in C$, i.e.,

$$\left(\begin{array}{c} \vdash C : \langle \Gamma_{act}, \Gamma_{fut}, sec \rangle \\ \alpha[Q, a] \in C \end{array}\right) \Rightarrow \alpha\text{-noninterference } C\ sec\,.$$

Proof:
Let C_1 be another arbitrary but fixed configuration such that $C \sim_\alpha C_1$. This means that for any $\beta \in \mathrm{dom}(C)$, if $\beta \in$ visibility range of α – we have that $\sigma(\beta) \in \mathrm{dom}(C_1)$ and $C(\beta) =_{\sigma,\tau} C_1(\sigma(\beta))$ for some τ and σ. That is, aside differently named futures (and active object references) these two activities are structurally the same and contain the same values. For the sake of clarity of the proof exposition, we leave the naming isomorphism implicit, i.e. use the same names, e.g., β, f_k, for both sides, i.e., for $\beta, \sigma(\beta)$ and $f_k, \tau(f_k)$. Note, that the type of the configurations C' and C_1' is in some cases an extension to the types of C and C_1, as described in Preservation (Theorem 1). The proof is an induction over the reduction relation combined with a case analysis whether an arbitrary $\beta \in \mathrm{dom}(C)$ is in the visibility range of α or not. If, for the first case, $C \to_\parallel C'$ by some reduction according to the semantics rules in the part of the configuration that is *not visible* to α, then for most cases trivially no change becomes visible by the transition to C': for any local reduction, this is the case since the visibility relation is unchanged. Hence, "invisible" objects remain invisible. If $C \sim_\alpha C_1$ and $C \to_\parallel C'$, then also $C' \sim_\alpha C_1$ whereby we trivially have the conclusion since $C_1 \to_\parallel^* C_1$ (in zero steps). This observation is less trivial for the rules REQUEST and ACTIVE where new elements, futures and activities, respectively, are created. In the case of REQUEST, let $\beta[f_k \mapsto E[\gamma.l(t)] :: Q_\beta, t_\beta] \in C$ and $\gamma[Q_\beta, t_\gamma] \in C$ with β in the α-invisible part and γ visible to α. The fact that there are no side effects provides that request f_m created in the request step in γ, i.e. $\gamma[f_m \mapsto t_\gamma.l(t) :: Q_\gamma, t_\gamma]$ in C', is not α-visible. Similarly, if a new activity γ is created from a method in β according to ACTIVE, then γ will not be in the visibility range of α since β was not visible to α by Definition 3.1 of the visibility relation. Thus, for β not visible to α, if $C \sim_\alpha C_1$ and $C \to_\parallel C'$, then also $C' \sim_\alpha C_1$ and the conclusion holds again because $C_1 \to_\parallel^* C_1$. This closes the case of *non-α-visible* reductions. If $C \to_\parallel C'$ by some reduction in the *α-visible* part, we need to consider all cases individually as given by the induction according to the semantics rules. If $C \to_\parallel C'$ by semantics rule REQUEST then C must have contained $\beta[f_k \mapsto E[\gamma.l(t)] :: Q_\beta, t_\beta]$ and $\gamma[Q_\gamma, t_\gamma]$ for some β, f_k, and γ. Hence, $\beta[f_k \mapsto E[f_m] :: Q_\beta, t_\beta]$ and $\gamma[f_m \mapsto t_\gamma.l(t) :: Q_\gamma, t_\gamma]$ in C' for some new future f_m. Since β is α-visible, so is γ by definition of visibility (since f_k was created from t_β, t_β must have an L-method containing γ). By confinement, f_m has global level δ_β and l is L. Since $C \sim_\alpha C_1$, and β, f_k, γ visible to α, we have (up to isomorphism of names) that $\beta[f_k \mapsto E[\gamma.l(t)] :: Q_\beta, t_\beta]$ and $\gamma[Q_\gamma, t_\gamma]$ in C_1. Therefore, we can equally apply the rule REQUEST to C_1 to obtain that C_1' contains $\beta[f_k \mapsto E[f_m] :: Q_\beta, t_\beta]$ and $\gamma[f_m \mapsto t_\gamma.l(t) :: Q_\gamma, t_\gamma]$. In C_1, γ is also α-invisible and l is typed L as well. Now, the α-visible parts of C_1' are equal to the ones of C' apart from the new future f_m. However, based on the future bijection τ that exists due to indistinguishability between C and C_1 we can extend this for f_m to a bijection τ'. In addition, by preservation, C' as well as C_1' are well-typed whereby finally $C' \sim_\alpha C_1'$ and this finishes the REQUEST-case. Another, also less obvious case for new elements in the α-visible part, is the one for ACTIVE. However, here we have a very similar situation as in the REQUEST case. If, in C, there is some $\beta[f_k \mapsto E[Active(t)] :: Q_\beta, t_\beta]$, we also have β alike in C_1, whereby we get in the next step – according to rule ACTIVE – $\beta[f_k \mapsto E[\gamma] :: Q_\beta, t_\beta] \parallel \gamma[\varnothing, t]$ in C' replacing the previous β. We can also apply ACTIVE in C_1 so that $\beta[f_k \mapsto E[\gamma] :: Q_\beta, t_\beta] \parallel \gamma[\varnothing, t]$ in C' and C_1' as well instead of just the old β. Indistinguishability is preserved since a bijection σ' exists as extension to σ to the new activity γ and by preservation again C' and C_1 remain well-typed. We are finished with the case for ACTIVE since $C' \sim_\alpha C_1'$. The other cases, corresponding to the remaining semantics rules, are of very a similar nature. Thus, the second part of α-visible parts of the configurations C and C_1 is also finished and completes the proof of the theorem. $\square$