Self-Adaptation and Secure Information Flow in Multiparty Structured Communications: A Unified Perspective

Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Jorge A. Perez

Self-Adaptation and Secure Information Flow in Multiparty Structured Communications: A Unified Perspective

Ilaria Castellani
INRIA Sophia-Antipolis (FR)

Mariangiola Dezani-Ciancaglini
Università di Torino (IT)

Jorge A. Pérez
University of Groningen (NL)

We present initial results on a comprehensive model of structured communications, in which self-adaptation and security concerns are jointly addressed. More specifically, we propose a model of self-adaptive, multiparty communications with secure information flow guarantees. In this model, security violations occur when processes attempt to read or write messages of inappropriate security levels within directed exchanges. Such violations trigger adaptation mechanisms that prevent the violations to occur and/or to propagate their effect in the choreography. Our model is equipped with local and global mechanisms for reacting to security violations; type soundness results ensure that global protocols are still correctly executed, while the system adapts itself to preserve security.

1 Introduction

Large-scale distributed systems are nowadays conceived as heterogeneous collections of software artifacts. Hence, *communication* plays a central role in their overall behavior. In fact, ensuring that the different components follow the stipulated protocols is a basic requirement in certifying system correctness. However, as communication-centric systems arise in different computing contexts, system correctness can no longer be characterized solely in terms of protocol conformance. Several other aspects —for instance, security, evolvability/adaptation, explicit distribution, time— are becoming increasingly relevant in the specification of actual interacting systems, and should be integrated into their correctness analysis. Recent proposals have addressed some of these aspects, thus extending the applicability of known reasoning techniques over models of communication-based systems. In the light of such proposals, a pressing challenge consists in understanding whether known models and techniques, often devised in isolation, can be harmoniously integrated into unified frameworks.

As an example, consider the multiparty interaction between a user, his bank, a store, and a social network. All exchanges occur on top of a browser, which relies on plug-ins to integrate information from different services. For instance, a plug-in may announce in the social network that the user has just bought an item from the store. That is, agreed exchanges between the user, the bank, and the store may in some cases lead to a (public) message announcing the transaction. We would like to ensure that the buying protocol works as expected, but also to avoid that sensitive information, exchanged in certain parts of the protocol, is leaked —e.g., in a *tweet* which mentions the credit card used in the transaction. Such an undesired behavior should be corrected as soon as possible. In fact, we would like to stop relying on the (unreliable) participant in ongoing/future instances of the protocol. Depending on how serious the leak is, however, we may also like to react in different ways. If the leak is minor (e.g., because the user interacted incorrectly with the browser), then we may simply identify the source of the leak and postpone the reaction to a later stage, enabling unrelated participants in the choreography to proceed with their exchanges. Otherwise, if the leak is serious (e.g., when the plug-in is compromised by a malicious participant) we may wish to adapt the choreography as soon as possible, removing the plug-in and modifying the behavior of the involved participants. This form of reconfiguration, however,

Marco Carbone (Ed.): Third Workshop on
Behavioural Types (BEAT 2014)
EPTCS 162, 2014, pp. 9–18, doi:10.4204/EPTCS.162.2

should only concern the participants involved with the insecure plug-in; participants not directly affected by the leak should not be unnecessarily restarted. In our example, since the unintended tweet concerns only the user, the store and the social network, the update should not affect the behavior of the bank.

To analyze such choreographic scenarios, we propose a framework for self-adaptive, multiparty communications which ensures basic guarantees for access control and secure information flow. The framework consists of a language for processes and networks, global types, and runtime monitors. Runtime monitors are obtained as projections from global types onto individual participants. Processes represent code that will be coupled with monitors to implement participants. A network is a collection of monitored processes which realize a choreography as described by the global type.

Intuitively, a monitor defines the behavior of a single participant in the choreography. In our proposal, the monitor also defines a security policy by stipulating *reading and writing permissions*, represented by *security levels*. The reading permission is an upper bound for the level of incoming messages, and the writing permission is a lower bound for the level of outgoing messages. A reading or writing violation occurs when a participant attempts to read or write a message whose level is not allowed by the corresponding reading or writing permission. A monitored operational semantics for networks is given by a reduction relation which ensures that the reading/writing permissions are respected or, in case they are violated, that an appropriate adaptation mechanism is triggered to limit the impact of the violation.

We consider both *local* and *global* adaptation mechanisms, intended to handle minor and serious leaks, respectively. The local mechanism works as follows: in case of a reading violation, the behavior of the monitor is modified so as to omit the disallowed read, and a process compliant with the new monitor is injected; in case of a writing violation, we penalize the sender by decreasing the reading level of his monitor and the implementation for the receiver is replaced. (In any case, the culprit of a reading/writing violation is always considered to be the sender.[1]) The global mechanism relies on distinguished low-level values called *nonces*. When an attempt to leak a value is detected, the value is replaced in the communication with a fresh nonce. This avoids improperly communicating the value and allows the whole system to make progress, for the benefit of the participants not involved in the violation. The semantics may then trigger at any point a reconfiguration action which removes the whole group of participants that may propagate the nonce and replaces it with a new choreography (global type). Thus, in this form of adaptation, one part of the choreography is isolated and replaced.

2 Syntax

Our calculus is inspired by that of [6], where security issues were not addressed and adaptation was determined by changes of a global state, which is not needed for our present purposes. We consider networks with three active components: *global types*, *monitors*, and *processes*. A global type represents the overall communication choreography over a set of participants [5]. Moreover, the global type defines reading permissions for each participant, following [4]. By projecting the global type onto participants, we obtain monitors: in essence, these are local types that define the communication protocols of the participants. The association of a process with a "fitting" monitor, dubbed *monitored process*, incarnates a participant whose process implements the monitoring protocol. Notably, we exploit intersection types, union types and subtyping to make this "fitting" relation more flexible.

As usual, we consider a finite lattice of *security levels* [8], ranged over by $\ell, \ell', \ldots$. We denote by $\sqcup$ and $\sqcap$ the join and meet operations on the lattice, and by $\bot$ and $\top$ its bottom and top elements. Also, we

[1] This is because the sender has an *active role* in producing and disseminating information through the system, while the receiver only has a passive role, and thus cannot be blamed for finding a sensitive value in the queue.

use $\mathsf{r}, \mathsf{r}', \ldots$ and $\mathsf{w}, \mathsf{w}', \ldots$ to range over levels denoting reading and writing permissions, respectively.

Global Types and Monitors. Global types define overall schemes of labeled communication between session participants. In our setting, they also prescribe the reading levels of the participants. We assume base sets of *participants*, ranged over by $\mathtt{p}, \mathtt{q}, \mathtt{r}, \ldots$; *labels*, ranged over by $\lambda, \lambda', \ldots$; and *recursion variables*, ranged over by $\mathbf{t}, \mathbf{t}', \ldots$. We also assume a set of basic *sorts* ($\mathsf{bool}, \mathsf{nat}, \ldots$), ranged over by S.

Definition 2.1 (Global Types and Security Global Types)**.** Global types *are defined by:*

$$\mathsf{G} \quad ::= \quad \mathtt{p} \to \mathtt{q} : \{\lambda_i(S_i).\mathsf{G}_i\}_{i\in I} \quad \mid \quad \boldsymbol{t} \quad \mid \quad \mu \boldsymbol{t}.\mathsf{G} \quad \mid \quad \mathsf{end}$$

We let $\mathsf{part}(\mathsf{G})$ *denote the set of participants in* G*, i.e., all senders* $\mathtt{p}$ *and receivers* $\mathtt{q}$ *occurring in* G*. A* security global type *is a pair* (G, L)*, where* G *is a global type and* L *maps each* $\mathtt{p}$ *in* $\mathsf{part}(\mathsf{G})$ *to a reading level* r*.*

A global type describes a sequence of value exchanges. Each value exchange is directed between a sender $\mathtt{p}$ and a receiver $\mathtt{q}$, and characterized by a label λ, which represents a choice among different alternatives. In writing $\mathtt{p} \to \mathtt{q} : \{\lambda_i(S_i).\mathsf{G}_i\}_{i\in I}$ we implicitly assume that $\mathtt{p} \neq \mathtt{q}$ and $\lambda_i \neq \lambda_j$ for all $i \neq j$. The global type end denotes the completed choreography. To account for recursive protocols, we consider recursive global types. As customary, we require guarded recursions and we adopt an equi-recursive view of recursion for all syntactic categories, identifying a recursive definition with its unfolding.

Monitors are obtained as *projections* from global types onto individual participants, following standard definitions [10, 1]. The projection of a global type G onto participant $\mathtt{p}$, denoted $\mathsf{G} \upharpoonright \mathtt{p}$, generates the monitor for $\mathtt{p}$. As usual, in order for $\mathsf{G} \upharpoonright \mathtt{p}$ to be defined, it is required that whenever $\mathtt{p}$ is not involved in some directed communication of G, it has equal projections in the different branchings of that communication. We say G is *well formed* if the projection $\mathsf{G} \upharpoonright \mathtt{p}$ is defined for all $\mathtt{p} \in \mathsf{part}(\mathsf{G})$. In the following we assume that all (security) global types are well formed.

Although monitors can be seen as local types, in our model they have an active role in the dynamics of networks, since they guide and enable/disable directed communications.

Definition 2.2 (Monitors)**.** *The set of* monitors *is defined by:*

$$\mathscr{M} \quad ::= \quad \mathtt{p}?\{\lambda_i(S_i).\mathscr{M}_i\}_{i\in I} \quad \mid \quad \mathtt{q}!\{\lambda_i(S_i).\mathscr{M}_i\}_{i\in I} \quad \mid \quad \boldsymbol{t} \quad \mid \quad \mu \boldsymbol{t}.\mathscr{M} \quad \mid \quad \mathsf{end}$$

An input monitor $\mathtt{p}?\{\lambda_i(S_i).\mathscr{M}_i\}_{i\in I}$ fits with a process that can receive, for each $i \in I$, a value of sort S_i, labeled by λ_i, and then continues as specified by $\mathscr{M}_i$. This corresponds to an external choice. Dually, an output monitor $\mathtt{q}!\{\lambda_i(S_i).\mathscr{M}_i\}_{i\in I}$ fits with a process which can send, for each $i \in I$, a value of sort S_i, labeled by λ_i, and then continues as prescribed by $\mathscr{M}_i$. As such, it corresponds to an internal choice.

Processes and Networks. We assume a set of *expressions*, ranged over by $\mathtt{e}, \mathtt{e}', \ldots$, which includes booleans and naturals (with operations over them) and a denumerable set $\mathit{Nonces} = \{\mathit{nonce}_i \mid i \geq 0\}$. An expression nonce_i —where i is fresh— is a dummy value that is generated at runtime to be used in place of some improperly sent value, in order to prevent security violations, see § 3. Each expression $\mathtt{e}$ is equipped with a security level, denoted $\mathit{lev}(\mathtt{e})$; for every i, $\mathit{lev}(\mathit{nonce}_i) = \bot$. Values are ranged over by v, v'; we use u to denote an *extended value*, which is either a value or a nonce.

We now define our set of processes, which represent code that will be coupled with monitors to implement participants. Our model, like [6]— but unlike other session calculi [9, 10, 2, 7]— uses processes that do not specify their partners in communication actions. It is the associated monitor which determines the partner in a given communication. Thus, processes represent flexible code that can be associated with different monitors to incarnate different participants. Communication actions are performed through *channels*. Each process owns a unique channel, which by convention is denoted by y in

the user code. At runtime, channel y will be replaced by a *session channel* $\mathsf{s}[\mathtt{p}]$, where s is the session name and $\mathtt{p}$ denotes the participant. We use c to stand for a user channel y or a session channel $\mathsf{s}[\mathtt{p}]$.

Definition 2.3 (Processes)**.** *The set of* processes *is defined by:*

$$P \quad ::= \quad \mathbf{0} \quad | \quad \mathsf{c}?\lambda(x).P \quad | \quad \mathsf{c}!\lambda(\mathsf{e}).P \quad | \quad X \quad | \quad \mu X.P \quad | \quad \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ P \quad | \quad P+P$$

The syntax of processes is rather standard: in addition to usual constructs for communication, recursion and conditionals, it includes the operator $+$, which represents external choice. For instance, $\mathsf{c}!\lambda.(\mathsf{e})P$ denotes a process which sends along c label λ and the value of the expression e and then behaves like P. We assume the following precedence among operators: prefix, external choice, recursion.

The previously introduced entities (global types, monitors, processes) are used to define *networks*. A network is a collection of monitored processes which realize a choreography as described by a global type. The choreography is initiated by the "new" construct applied to a security global type (G,L). This construct, akin to a *session initiator* [6], is denoted $\mathsf{new}(\mathsf{G},\mathsf{L})$. In carrying on a multiparty interaction, a process is always controlled by a monitor, which ensures that all its communications agree with the protocol prescribed by the global type. Each monitor is equipped with a reading permission r and a writing permission w. A monitored process, written $\mathscr{M}^{\mathsf{r},\mathsf{w}}[P]$, denotes a process P controlled by a monitor $\mathscr{M}$.

Data are exchanged among participants asynchronously, by means of *message queues*, ranged over by $h, h', \ldots$. There is one such queue for each active session. We denote by $\mathsf{s}:h$ the *named queue* associated with session s. The empty queue is denoted by $\emptyset$. Messages in queues are of the form $(\mathtt{p},\mathtt{q},\lambda(u))$, indicating that the label λ and the extended value u are communicated with sender $\mathtt{p}$ and receiver $\mathtt{q}$. Queue concatenation is denoted by "$\cdot$": it is associative and has $\emptyset$ as neutral element.

The parallel composition of session initiators, monitored processes, and runtime queues forms a network. Networks can be restricted on session names.

Definition 2.4 (Networks)**.** *The set of* networks *is defined by:*

$$N \ ::= \ \mathsf{new}(\mathsf{G},\mathsf{L}) \ | \ \mathscr{M}^{\mathsf{r},\mathsf{w}}[P] \ | \ \mathsf{s}:h \ | \ N \mid N \ | \ (\nu \mathsf{s})N$$

As mentioned above, annotations r and w in $\mathscr{M}^{\mathsf{r},\mathsf{w}}[P]$ represent reading and writing permissions for process P. While r acts as an upper bound for reading, w acts as a lower bound for writing. When the choreography is initialized, the reading level is set according to map L; the writing level is always set to $\bot$. The actions performed by the process determine dynamic modifications to these levels. In writing monitored processes we omit the levels when they are not used. Also, we shall sometimes write $\mathscr{M}_{\mathtt{p}}^{\mathsf{r},\mathsf{w}}[P]$ (or simply $\mathscr{M}_{\mathtt{p}}[P]$) to indicate that the channel in P is $\mathsf{s}[\mathtt{p}]$ for some s.

As in [6], process types (called *types* when not ambiguous) describe process communication behaviors. Types have prefixes corresponding to input and output actions. In particular, an *input type* (resp. *output type*) is a type whose prefix corresponds to an input (resp. output) action, while the *continuation* of a type is the type following its first prefix. A *communication type* is either an input or an output type. Intersection types are used to type external choices, since an external choice offers both behaviors of the composing processes. Dually, union types are used to type conditional expressions (internal choices).

To formally define types, we first give the more liberal syntax of *pre-types* and then we characterize process types by fixing some natural restrictions on pre-types.

Definition 2.5 (Pre-types)**.** *The set of* pre-types *is inductively defined by:*

$$T \ ::= \ ?\lambda(S).T \mid !\lambda(S).T \mid T \wedge T \mid T \vee T \mid \boldsymbol{t} \mid \mu \boldsymbol{t}.T \mid \mathsf{end}$$

where $\wedge$ and $\vee$ are considered modulo idempotence, commutativity, and associativity.

$$lin(?\lambda(S).T) = lout(!\lambda(S).T) = \{\lambda\}$$
$$lin(!\lambda(S).T) = lin(!\lambda) = lout(?\lambda(S).T) = lout(?\lambda) = \emptyset$$
$$lin(T_1 \wedge T_2) = lin(T_1 \vee T_2) = lin(T_1) \cup lin(T_2)$$
$$lout(T_1 \wedge T_2) = lout(T_1 \vee T_2) = lout(T_1) \cup lout(T_2)$$

Table 1: The mappings lin and $lout$, as required in Definition 2.6.

$$\Gamma \vdash \mathbf{0} \triangleright \mathsf{c} : \mathsf{end}\ \text{END} \qquad \Gamma, X : \mathsf{T} \vdash X \triangleright \mathsf{c} : \mathsf{T}\ \text{RV}$$

$$\frac{\Gamma, X : \mathsf{T} \vdash P \triangleright \mathsf{c} : \mathsf{T}}{\Gamma \vdash \mu X.P \triangleright \mathsf{c} : \mathsf{T}}\ \text{REC} \qquad \frac{\Gamma, x : S \vdash P \triangleright \mathsf{c} : \mathsf{T}}{\Gamma \vdash \mathsf{c}?\lambda(x).P \triangleright \mathsf{c} :?\lambda(S).\mathsf{T}}\ \text{RCV} \qquad \frac{\Gamma \vdash P \triangleright \mathsf{c} : \mathsf{T} \qquad \Gamma \vdash \mathsf{e} : S}{\Gamma \vdash \mathsf{c}!\lambda(\mathsf{e}).P \triangleright \mathsf{c} :!\lambda(S).\mathsf{T}}\ \text{SEND}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{bool} \quad \Gamma \vdash P_1 \triangleright \mathsf{c} : \mathsf{T}_1 \quad \Gamma \vdash P_2 \triangleright \mathsf{c} : \mathsf{T}_2 \quad \mathsf{T}_1 \vee \mathsf{T}_2 \in \mathscr{T}}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \triangleright \mathsf{c} : \mathsf{T}_1 \vee \mathsf{T}_2}\ \text{IF}$$

$$\frac{\Gamma \vdash P_1 \triangleright \mathsf{c} : \mathsf{T}_1 \quad \Gamma \vdash P_2 \triangleright \mathsf{c} : \mathsf{T}_2 \quad \mathsf{T}_1 \wedge \mathsf{T}_2 \in \mathscr{T}}{\Gamma \vdash P_1 + P_2 \triangleright \mathsf{c} : \mathsf{T}_1 \wedge \mathsf{T}_2}\ \text{CHOICE}$$

Table 2: Typing Rules for Processes.

In writing pre-types and types we assume that '.' has precedence over '$\wedge$' and '$\vee$'.

In order to define types for processes, we have to avoid intersection between input types with the same first label, which would represent an ambiguous external choice: indeed, the types following a same input prefix could be different and this would lead to a communication mismatch. For the same reason, process types cannot contain intersections between output types with the same label. Since we have to match types with monitors, where internal choices are always taken by participants sending a label, we force unions to take as arguments output types (possibly combined by intersections or unions). Therefore, we formalize the above restrictions by means of two mappings from pre-types to sets of labels (Table 1) and then we define types by using these mappings.

Definition 2.6 (Process Type)**.** *A* (process) type *is a pre-type satisfying the following constraints modulo idempotence, commutativity and associativity of unions and intersections:*

- *all occurrences of the shape $T_1 \wedge T_2$ are such that $lin(T_1) \cap lin(T_2) = lout(T_1) \cap lout(T_2) = \emptyset$.*
- *all occurrences of the shape $T_1 \vee T_2$ are such that $lin(T_1) = lin(T_2) = lout(T_1) \cap lout(T_2) = \emptyset$.*

We use T to range over types and $\mathscr{T}$ to denote the set of types.

For instance, $(T \wedge T) \vee T$ is a type, whenever T is a type, since types are considered modulo idempotence.

We now introduce the type system for processes. An *environment* Γ is a finite mapping from expression variables to sorts and from process variables to types:

$$\Gamma ::= \emptyset \mid \Gamma, x : S \mid \Gamma, X : \mathsf{T}$$

where the notation $\Gamma, x : S$ (resp. $\Gamma, X : \mathsf{T}$) means that x (resp. X) does not occur in Γ.

Typing rules for processes are given in Table 2. We assume that expressions are typed by sorts, as usual, and a nonce has all sorts. In rules IF and CHOICE we require that the applications of union and intersection on two types form a type (cf. conditions $\mathsf{T}_1 \vee \mathsf{T}_2 \in \mathscr{T}$ and $\mathsf{T}_1 \wedge \mathsf{T}_2 \in \mathscr{T}$).

We define $\leq$ as the minimal reflexive and transitive relation on $\mathscr{T}$ such that:

$$\mathbf{t} \leq \mathbf{t} \qquad \mathsf{T} \leq \mathsf{end} \qquad \mathsf{T}_1 \wedge \mathsf{T}_2 \ \leq \ \mathsf{T}_i \qquad \mathsf{T}_i \leq \mathsf{T}_1 \vee \mathsf{T}_2 \quad (i = 1,2)$$
$$\mathsf{T}_1 \leq \mathsf{T}_2 \text{ implies } !\lambda(S).\mathsf{T}_1 \leq !\lambda(S).\mathsf{T}_2 \text{ and } ?\lambda(S).\mathsf{T}_1 \leq ?\lambda(S).\mathsf{T}_2$$
$$\mathsf{T} \leq \mathsf{T}_1 \text{ and } \mathsf{T} \leq \mathsf{T}_2 \text{ imply } \mathsf{T} \ \leq \ \mathsf{T}_1 \wedge \mathsf{T}_2$$
$$\mathsf{T}_1 \leq \mathsf{T} \text{ and } \mathsf{T}_2 \leq \mathsf{T} \text{ imply } \mathsf{T}_1 \vee \mathsf{T}_2 \leq \mathsf{T}$$
$$(\mathsf{T}_1 \vee \mathsf{T}_2) \wedge \mathsf{T}_3 \leq \mathsf{T} \ \text{ iff } \ \mathsf{T}_1 \wedge \mathsf{T}_3 \leq \mathsf{T} \text{ and } \mathsf{T}_2 \wedge \mathsf{T}_3 \leq \mathsf{T}$$
$$\mathsf{T} \leq (\mathsf{T}_1 \wedge \mathsf{T}_2) \vee \mathsf{T}_3 \ \text{ iff } \ \mathsf{T} \leq \mathsf{T}_1 \vee \mathsf{T}_3 \text{ and } \mathsf{T} \leq \mathsf{T}_2 \vee \mathsf{T}_3$$
$$\mu\mathbf{t}.\mathsf{T} \leq \mu\mathbf{t}.\mathsf{T}' \ \text{ iff } \ \mathsf{T} \leq \mathsf{T}'$$

Table 3: Subtyping on Process Types.

The compliance between process types and monitors (*adequacy*) is made flexible by using the *subtyping* relation on types, denoted $\leq$ and defined in Table 3. Subtyping is monotone, for input/output prefixes, with respect to continuations and it follows the usual set theoretic inclusion of intersection and union. Notice that we use a weaker definition than standard subtyping on intersection and union types, since it is sufficient to define subtyping on types. Intuitively, $\mathsf{T}_1 \leq \mathsf{T}_2$ means that a process with type T_1 has all the behaviors required by type T_2 but possibly more.

An input monitor naturally corresponds to an external choice, while an output monitor naturally corresponds to an internal choice. Thus, intersections of input types are adequate for input monitors and unions of output types are adequate for output monitors. Formally, *adequacy* is defined as follows:

Definition 2.7 (Adequacy)**.** *Let the mapping $|\cdot|$ from monitors to types be defined as*

$$|\mathtt{p}?\{\lambda_i(S_i).\mathscr{M}_i\}_{i\in I}| = \textstyle\bigwedge_{i\in I} ?\lambda_i(S_i).|\mathscr{M}_i| \qquad |\mathtt{q}!\{\lambda_i(S_i).\mathscr{M}_i\}_{i\in I}| = \textstyle\bigvee_{i\in I} !\lambda_i(S_i).|\mathscr{M}_i|$$
$$|\boldsymbol{t}| = \boldsymbol{t} \qquad |\mu\boldsymbol{t}.\mathscr{M}| = \mu\boldsymbol{t}.|\mathscr{M}| \qquad |\mathsf{end}| = \mathsf{end}$$

We say that type T is adequate *for a monitor $\mathscr{M}$, notation $\mathsf{T} \propto \mathscr{M}$, if $\mathsf{T} \leq |\mathscr{M}|$.*

3 Semantics

The semantics of monitors and processes is given by labeled transition systems (LTS), while that of networks is given in the style of a reduction semantics.

A monitor guides the communications of a process by choosing its partners in labeled exchanges, and by allowing only some actions among those offered by the process.

The LTS for monitors uses labels $\mathtt{p}?\lambda$ and $\mathtt{p}!\lambda$, and formalizes the expected intuitions:

$$\mathtt{p}?\{\lambda_i(S_i).\mathscr{M}_i\}_{i\in I} \xrightarrow{\mathtt{p}?\lambda_j} \mathscr{M}_j \qquad \mathtt{q}!\{\lambda_i(S_i).\mathscr{M}_i\}_{i\in I} \xrightarrow{\mathtt{q}!\lambda_j} \mathscr{M}_j \quad j \in I$$

The LTS for processes, given in Table 4, is also fairly simple. It relies on labels $\mathtt{s}[\mathtt{p}]?\lambda(u)$ (input), $\mathtt{s}[\mathtt{p}]!\lambda(u)$ (output), and ℓ (security levels for expressions). The labels $\mathtt{s}[\mathtt{p}]?\lambda(u)$ and $\mathtt{s}[\mathtt{p}]!\lambda(u)$ are ranged over by α, β. We use $\mathtt{e} \downarrow u$ to indicate that expression $\mathtt{e}$ evaluates to the extended value u, assuming $\mathit{nonce}_i \downarrow \mathit{nonce}_i$. When reducing a conditional we record the level of the tested expression in order to track information flow. The rules for sum specify that choices are performed by the communication actions, while internal computations are transparent.

The reduction of networks assumes a collection $\mathscr{P}$ of pairs (P, T) of processes together with their types. It uses a rather natural structural equivalence $\equiv$ which erases monitored processes with end monitor and commutes independent messages (with different senders or different receivers) in queues [4].

$$\mathsf{s}[\mathsf{p}]?\lambda(x).P \xrightarrow{\mathsf{s}[\mathsf{p}]?\lambda(u)} P\{u/x\} \qquad \mathsf{s}[\mathsf{p}]!\lambda(\mathsf{e}).P \xrightarrow{\mathsf{s}[\mathsf{p}]!\lambda(u)} P \quad \mathsf{e}\downarrow u$$

$$\text{if } \mathsf{e} \text{ then } P \text{ else } Q \xrightarrow{lev(\mathsf{e})} P \quad \mathsf{e}\downarrow \mathsf{true} \qquad \text{if } \mathsf{e} \text{ then } P \text{ else } Q \xrightarrow{lev(\mathsf{e})} Q \quad \mathsf{e}\downarrow \mathsf{false}$$

$$P \xrightarrow{\alpha} P' \;\Rightarrow\; P+Q \xrightarrow{\alpha} P' \qquad P \xrightarrow{\ell} P' \;\Rightarrow\; P+Q \xrightarrow{\ell} P'+Q$$

Table 4: LTS of processes. Symmetric rules are omitted.

$$\frac{\mathscr{M}_{\mathsf{p}} = \mathsf{G}\upharpoonright\mathsf{p} \qquad \forall \mathsf{p}\in\mathsf{part}(\mathsf{G}).\ (P_{\mathsf{p}},\mathsf{T}_{\mathsf{p}})\in\mathscr{P}\ \&\ \mathsf{T}_{\mathsf{p}}\propto\mathscr{M}_{\mathsf{p}}}{\mathsf{new}(\mathsf{G},\mathsf{L}) \longrightarrow (\nu\,\mathsf{s})\prod_{\mathsf{p}\in\mathsf{part}(\mathsf{G})}(\mathscr{M}_{\mathsf{p}}^{\mathsf{L}(\mathsf{p}),\perp}[P_{\mathsf{p}}\{\mathsf{s}[\mathsf{p}]/y\}] \mid \mathsf{s}:\emptyset)}\ \textsc{Init}$$

$$\frac{P \xrightarrow{\ell} P'}{\mathscr{M}^{\mathsf{r},\mathsf{w}}[P] \longrightarrow \mathscr{M}^{\mathsf{r},\mathsf{w}\sqcup\ell}[P']}\ \textsc{UpLev} \qquad \frac{\mathscr{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}?\lambda} \widehat{\mathscr{M}_{\mathsf{p}}} \qquad P \xrightarrow{\mathsf{s}[\mathsf{p}]?\lambda(u)} P' \qquad lev(u)\le\mathsf{r}}{\mathscr{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathsf{s}:(\mathsf{q},\mathsf{p},\lambda(u))\cdot h \longrightarrow \widehat{\mathscr{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}}[P'] \mid \mathsf{s}:h}\ \textsc{In}$$

$$\frac{\mathscr{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}!\lambda} \widehat{\mathscr{M}_{\mathsf{p}}} \quad P \xrightarrow{\mathsf{s}[\mathsf{p}]!\lambda(u)} P' \quad u\in \mathit{Nonces} \text{ or } (u=v \text{ and } \mathsf{w}\le lev(v))}{\mathscr{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathsf{s}:h \longrightarrow \widehat{\mathscr{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}}[P'] \mid \mathsf{s}:h\cdot(\mathsf{p},\mathsf{q},\lambda(u))}\ \textsc{Out}$$

$$\frac{\mathscr{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}?\lambda} \widehat{\mathscr{M}_{\mathsf{p}}} \quad \mathit{nonce}_i = \mathsf{next}(\mathit{Nonces}) \quad P \xrightarrow{\mathsf{s}[\mathsf{p}]?\lambda(\mathit{nonce}_i)} P' \quad lev(v)\not\le\mathsf{r}}{\mathscr{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathsf{s}:(\mathsf{q},\mathsf{p},\lambda(v))\cdot h \longrightarrow \widehat{\mathscr{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}}[P'] \mid \mathsf{s}:h}\ \textsc{InGlob}$$

$$\frac{\mathscr{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}!\lambda} \widehat{\mathscr{M}_{\mathsf{p}}} \quad P \xrightarrow{\mathsf{s}[\mathsf{p}]!\lambda(v)} P' \quad \mathit{nonce}_i = \mathsf{next}(\mathit{Nonces}) \quad \mathsf{w}\not\le lev(v)}{\mathscr{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathscr{M}_{\mathsf{q}}^{\mathsf{r}',\mathsf{w}'}[Q] \mid \mathsf{s}:h \longrightarrow \widehat{\mathscr{M}_{\mathsf{p}}^{\mathsf{r}\sqcap\mathsf{r}',\mathsf{w}}}[P'] \mid \mathscr{M}_{\mathsf{q}}^{\mathsf{r}',\mathsf{w}'}[Q] \mid \mathsf{s}:h\cdot(\mathsf{p},\mathsf{q},\lambda(\mathit{nonce}_i))}\ \textsc{OutGlob}$$

$$\frac{\mathscr{A}(\{P_{\mathsf{p}} \mid \mathsf{p}\in\Pi\}, \mathit{nonce}_i) = \Pi' \quad F(\{P_{\mathsf{p}} \mid \mathsf{p}\in\Pi'\}) = (\mathsf{G},\mathsf{L})}{(\nu\,\mathsf{s})\,(\prod_{\mathsf{p}\in\Pi}\mathscr{M}_{\mathsf{p}}[P_{\mathsf{p}}] \mid \mathsf{s}:h) \longrightarrow (\nu\,\mathsf{s})\,(\prod_{\mathsf{p}\in\Pi-\Pi'}\mathscr{M}_{\mathsf{p}}[P_{\mathsf{p}}] \mid \mathsf{s}:h\setminus\Pi') \mid \mathsf{new}(\mathsf{G},\mathsf{L})}\ \textsc{Refresh}$$

$$\frac{\mathscr{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}?\lambda} \widehat{\mathscr{M}_{\mathsf{p}}} \quad (P',\mathsf{T})\in\mathscr{P} \quad \mathsf{T}\propto\widehat{\mathscr{M}_{\mathsf{p}}} \quad lev(v)\not\le\mathsf{r}}{\mathscr{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathsf{s}:(\mathsf{q},\mathsf{p},\lambda(v))\cdot h \longrightarrow \widehat{\mathscr{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}}[P'] \mid \mathsf{s}:h}\ \textsc{InLoc}$$

$$\frac{\mathscr{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}!\lambda} \widehat{\mathscr{M}_{\mathsf{p}}} \quad P \xrightarrow{\mathsf{s}[\mathsf{p}]!\lambda(v)} P' \quad \widehat{\mathscr{M}_{\mathsf{q}}} = \mathscr{M}_{\mathsf{q}}\backslash?(\mathsf{p},\lambda) \quad (Q',\mathsf{T})\in\mathscr{P} \quad \mathsf{T}\propto\widehat{\mathscr{M}_{\mathsf{q}}} \quad \mathsf{w}\not\le lev(v)}{\mathscr{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathscr{M}_{\mathsf{q}}^{\mathsf{r}',\mathsf{w}'}[Q] \longrightarrow \widehat{\mathscr{M}_{\mathsf{p}}^{\mathsf{r}\sqcap\mathsf{r}',\mathsf{w}}}[P'] \mid \widehat{\mathscr{M}_{\mathsf{q}}^{\mathsf{r}',\mathsf{w}'}}[Q'\{\mathsf{s}[\mathsf{q}]/y\}]}\ \textsc{OutLoc}$$

$$\frac{N_1\equiv N_1' \quad N_1'\longrightarrow N_2' \quad N_2\equiv N_2'}{N_1\longrightarrow N_2}\ \textsc{Equiv} \qquad \frac{N\longrightarrow N'}{\mathscr{E}[N]\longrightarrow\mathscr{E}[N']}\ \textsc{Ctx}$$

Table 5: Reduction Rules for Networks.

The reduction rules for networks are given in Table 5. We briefly describe them:

1. Rule INIT initializes a choreography denoted by global type G. A network $\mathsf{new}(\mathsf{G},\mathsf{L})$ evolves in a reduction step into a composition of monitored processes and a session queue. For each $\mathtt{p} \in \mathsf{part}(\mathsf{G})$, there must be a pair $(P_{\mathtt{p}}, \mathsf{T}_{\mathtt{p}})$ in the collection $\mathscr{P}$. The type $\mathsf{T}_{\mathtt{p}}$ must be adequate for the monitor obtained as projection of G onto $\mathtt{p}$. Then the process (where the channel y has been replaced by $s[\mathtt{p}]$) is coupled with the corresponding monitor and the empty queue $\mathtt{s} : \emptyset$ is created. The security levels of the monitors are instantiated at runtime: while the initial reading level is obtained from the mapping L, the initial writing level is $\bot$. Lastly, the name $\mathtt{s}$ is restricted.

2. Rule UPLEV updates the current writing level $\mathtt{w}$ of the monitor with the least upper bound (join) of $\mathtt{w}$ and the level ℓ of the conditional expression tested by the process (by the semantics of processes, we know that ℓ is the level of a conditional expression). This is to prevent usual information leaks.

3. Rule IN defines the input of an extended value u. The input action must be enabled by the monitor. We further require that the level associated with u in the queue be lower than or equal to the reading level $\mathtt{r}$ of the monitor. Notice that this check is nontrivial only for values, as nonces have all level $\bot$.

4. Rule OUT defines the output of an extended value u. If u is a proper value v, we require that the output be allowed by the monitor, i.e., that the level associated with v be higher than or equal to the level $\mathtt{w}$ of the monitor. Nothing is required in case u is a nonce, since nonces provide no information.

5. Rule INGLOB defines the *global* reconfiguration mechanism for reading violations, which creates nonces. A reading violation occurs when the level associated with the value in the queue is not lower than or equal to the reading level of the monitor. A reduction is still enabled, but since the monitored process is not allowed to input the provided value, an adaptation is realized by: (a) inputting a fresh nonce instead of the value, and (b) removing the unreadable value from the queue. In this rule and in the next one the function $\mathsf{next}(\mathit{Nonces})$ is used to obtain a fresh nonce in the set Nonces.

6. Rule OUTGLOB defines the *global* reconfiguration mechanism for writing violations. Such a violation occurs when the level of the sent value v (no writing violation may occur with nonces) is not greater than or equal to the writing level $\mathtt{w}$ of the monitor controlling the sender (noted $\mathtt{p}$ in the rule). Also in this case a reduction is enabled; adaptation is realized by: (a) adding a fresh nonce to the queue and (b) updating the reading permission $\mathtt{r}$ attached to the monitor of $\mathtt{p}$. Indeed, to formalize the fact that $\mathtt{p}$ is responsible for the writing violation, by trying to "declassify" value v from its original level to the reading level $\mathtt{r}'$ of the monitor controlling the receiver (noted $\mathtt{q}$ in the rule), we update its current reading level $\mathtt{r}$ to the greatest lower bound (meet) of $\mathtt{r}$ and $\mathtt{r}'$. Hence, the reading level of $\mathtt{p}$ is downgraded to that of $\mathtt{q}$ (or lower), accounting for the fact that $\mathtt{p}$ attempted to leak information to $\mathtt{q}$. This is intended to counter any possible "recidivism" in $\mathtt{p}$'s offending behaviour, by preventing new sensitive values to be received by $\mathtt{p}$ and then leaked again to $\mathtt{q}$.

7. Rule REFRESH goes hand-in-hand with rules INGLOB and OUTGLOB. It extracts the set of participants whose processes can send nonce_i, which are the processes that contain nonce_i, and all those which (transitively) communicate with them. This set is obtained using the mapping $\mathscr{A}$. For the participants affected by nonce_i, a new global type is obtained via a function F. This function is left unspecified, for we are interested in modelling the mechanism of adaptation, and not the way in which the new security global type is chosen. Notice that the new security global type may involve other participants than those affected by nonce_i. The reduction step then consists in (a) starting the new choreography and (b) continuing the execution of the unaffected participants. For (b) we must erase from the queue all messages involving affected participants; we denote by $h \setminus \Pi'$ the resulting queue.

8. Rule INLOC defines the *local* reconfiguration mechanism triggered in the case of a reading violation. Intuitively, this rule defines adaptation by "ignoring" the forbidden input: the message is removed from the queue and the implementation of the monitored process is replaced with new code where the input action is not present. This code replacement is formalized simply by considering the monitor that results from the reduction (noted $\widehat{\mathscr{M}_{\mathtt{p}}}$ in the rule), and picking a process P' that agrees with it.
9. Rule OUTLOC defines the *local* reconfiguration mechanism for writing violations. As for INLOC, the monitor is modified and a new implementation that conforms to the modified monitor is injected. The monitor $\mathscr{M}_{\mathtt{q}}\backslash?(\mathtt{p},\lambda)$ is obtained from $\mathscr{M}_{\mathtt{q}}$ by erasing the input action $?(\mathtt{p},\lambda)$ and choosing the corresponding branch. The reading permission of the sender monitor is modified as in rule OUTGLOB.
10. Rules EQUIV and CTX are standard: they allow the interplay of reduction with structural congruence and enable the reduction within evaluation contexts (defined as expected), respectively.

The reduction of networks is clearly nondeterministic, in contrast with standard session calculi. Nondeterminism arises at every security violation, which can be treated either by generating nonces (rules INGLOB and OUTGLOB) or by modifying the receiver's monitor and process, just skipping "wrong" message receptions (rules INLOC and OUTLOC). On top of these alternatives, rule REFRESH can always be applied, resulting in the splitting of the choreography between a part affected by a fixed nonce (arbitrarily chosen) and an unaffected part. As a result, the affected participants are adapted using some (unspecified) adaptation function, while unaffected participants remain unaware of this adaptation.

Main Results. As in [6], well-typed networks enjoy *subject reduction* and *progress* properties. Moreover, reduction of well-typed networks always respects reading and writing permissions:

Theorem 3.1. *Let N be a network.*

1. *If $N = \mathscr{M}_{\mathtt{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathtt{s} : (\mathtt{q},\mathtt{p},\lambda(v)) \cdot h \longrightarrow \widehat{\mathscr{M}_{\mathtt{p}}^{\mathsf{r},\mathsf{w}}}[P'] \mid \mathtt{s} : h$, then either $lev(v) \leq \mathsf{r}$ or P' is not obtained by consuming the message $(\mathtt{q},\mathtt{p},\lambda(v))$.*
2. *If $N = \mathscr{M}_{\mathtt{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathtt{s} : h \longrightarrow \widehat{\mathscr{M}_{\mathtt{p}}^{\mathsf{r},\mathsf{w}}}[P'] \mid \mathtt{s} : h \cdot (\mathtt{p},\mathtt{q},\lambda(v))$, then $\mathsf{w} \leq lev(v)$.*

Theorem 3.1(1) says that if the reading permission of a monitor is not respected, then the disallowed value is never read from the queue—by virtue of the runtime mechanisms implemented by rules INGLOB and INLOC. Analogously, Theorem 3.1(2) says that if a value is added to a session queue, then it is always the case that this is allowed by the writing permission of the given monitor. Here again, it is worth observing that adaptation mechanisms defined by rules OUTGLOB and OUTLOC can always be triggered to handle the situations in which the sender attempts to transgress his monitor's writing permission.

4 Concluding Remarks

Our work builds on [6], where a calculus based on global types, monitors and processes similar to ours was introduced. There are two main points of departure from that work. First, the calculus of [6] relied on a global state, and global types describe only finite protocols; adaptation was triggered after the execution of the communications prescribed by a global type, in reaction to changes of the global state. Second, adaptation in [6] involved all participants in the choreography. In sharp contrast, in our calculus reconfigurations are triggered by security violations, and reconfiguration may be either local or global. Therefore, we may consider our adaptation mechanism as more flexible than that of [6] in two respects. First, adaptation is triggered as a reaction to security violations (whose occurrence is hard to predict)

rather than at fixed, prescribed computation points. Second, adaptation may be restricted to a subset of participants (those involved in the security violation), thus resulting in a less disruptive procedure.

Our approach based on monitored processes (as defined in [6]) relies on rather elementary assumptions on the nature of processes. In particular, we assume that processes are well typed with respect to a rather simple discipline (based on intersection and union types) which does not mention security permissions. In fact, runtime information on permissions is handled by the monitor of the process; the relationship between typed processes and monitors is formalized by the notion of adequacy. This degree of independence between typed processes and security annotations distinguishes our approach from previous works on security issues for multiparty session typed processes (see, e.g. [3, 4]).

Acknowledgments. We are grateful to the anonymous reviewers for their useful remarks. This work was supported by COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems via a Short-Term Scientific Mission grant (to Pérez). Dezani was also partially supported by MIUR PRIN Project CINA Prot. 2010LHT4KM and Torino University/Compagnia San Paolo Project SALT.

References

[1] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini & Nobuko Yoshida (2008): *Global Progress in Dynamically Interleaved Multiparty Sessions*. In Pierpaolo Degano, Rocco De Nicola & José Meseguer, editors: *CONCUR'08, LNCS* 5201, Springer, pp. 418–433. Available at `http://dx.doi.org/10.1007/978-3-540-85361-9_33`.

[2] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda & Nobuko Yoshida (2013): *Monitoring Networks through Multiparty Session Types*. In Dirk Beyer & Michele Boreale, editors: *FMOODS/FORTE'13, LNCS* 7892, Springer, pp. 50–65. Available at `http://dx.doi.org/10.1007/978-3-642-38592-6_5`.

[3] Sara Capecchi, Ilaria Castellani & Mariangiola Dezani-Ciancaglini (2014): *Information Flow Safety in Multiparty Sessions*. *Mathematical Structures in Computer Science*. To appear.

[4] Sara Capecchi, Ilaria Castellani & Mariangiola Dezani-Ciancaglini (2014): *Typing Access Control and Secure Information Flow in Sessions*. *Information and Computation*. Available at `http://dx.doi.org/10.1016/j.ic.2014.07.005`.

[5] Marco Carbone, Kohei Honda & Nobuko Yoshida (2012): *Structured Communication-Centered Programming for Web Services*. *ACM Transactions on Programming Languages and Systems* 34(2), pp. 8:1–8:78. Available at `http://doi.acm.org/10.1145/2220365.2220367`.

[6] Mario Coppo, Mariangiola Dezani-Ciancaglini & Betti Venneri (2014): *Self-Adaptive Monitors for Multiparty Sessions*. In Marco Aldinucci, Daniele D'Agostino & Peter Kilpatrick, editors: *PDP'14*, IEEE, pp. 688–696. Available at `http://doi.ieeecomputersociety.org/10.1109/PDP.2014.18`.

[7] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida & Luca Padovani (2014): *Global Progress for Dynamically Interleaved Multiparty Sessions*. *Mathematical Structures in Computer Science*. To appear.

[8] Dorothy E. Denning (1976): *A Lattice Model of Secure Information Flow*. *Commun. ACM* 19(5), pp. 236–243. Available at `http://doi.acm.org/10.1145/360051.360056`.

[9] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Disciplines for Structured Communication-based Programming*. In Chris Hankin, editor: *ESOP'98, LNCS* 1381, Springer, pp. 22–138. Available at `http://dx.doi.org/10.1007/BFb0053567`.

[10] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types*. In George C. Necula & Philip Wadler, editors: *POPL'08*, ACM Press, pp. 273–284. Available at `http://doi.acm.org/10.1145/1328438.1328472`.