Under consideration for publication in Formal Aspects of Computing

Self-Adaptation and Secure Information Flow in Multiparty Communications

Ilaria Castellani[1] and Mariangiola Dezani-Ciancaglini[2] and Jorge A. Pérez[3]

[1]INRIA Sophia Antipolis, France
[2]Università di Torino, Turin, Italy
[3]University of Groningen, Groningen, The Netherlands

Abstract. We present a comprehensive model of structured communications in which self-adaptation and security concerns are jointly addressed. More specifically, we propose a model of multiparty, self-adaptive communications with access control and secure information flow guarantees. In our model, multiparty protocols (choreographies) are described as global types; security violations occur when process implementations of protocol participants 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 adaptation mechanisms for reacting to security violations of different gravity; type soundness results ensure that the overall multiparty protocol is still correctly executed while the system adapts itself to preserve the participants' security.

Keywords: concurrency, behavioural types, multiparty communication, self-adaptation, secure information flow.

1. Introduction

1.1. Context and Motivation

Large-scale distributed systems are nowadays conceived as heterogeneous collections of interconnected software artefacts. Hence, *communication* plays a central role in their overall behaviour. In fact, ensuring that the different components follow the stipulated communication 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 characterised solely in terms of protocol conformance. Several other aspects —for instance, security, adaptability, explicit distribution, time— are becoming increasingly relevant in the specification of interacting systems, and should therefore be integrated into their correctness analysis. In the context of analysis techniques based on *behavioural types* [HLV+16], recent proposals have addressed

Correspondence and offprint requests to: Jorge A. Pérez, e-mail: j.a.perez@rug.nl

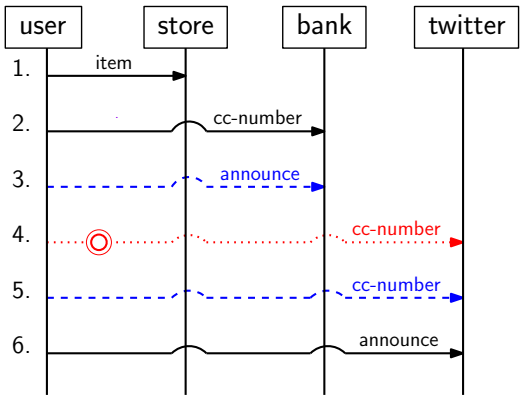

Fig. 1. Safe behaviour (solid lines in black), minor leak (dashed lines in blue), major leak (dotted lines in red).

some of these aspects (see, e.g., [BYY14, CCDC14, CCDC15, DP15]), thus extending the applicability of known reasoning techniques over models of communication-based systems. A pressing challenge consists in understanding whether such proposals, often devised in isolation, can be harmoniously integrated into unified frameworks.

As an example of a scenario in which protocol conformance falls short, consider a multiparty interaction (or *choreography*) supported by a web browser—this is a most common interface for accessing distributed services nowadays. Browsers rely on protocols (such as those for encryption) which are unknown to most users; such protocols are often sensitive to security threats of different magnitude. Suppose that a user, his bank, a store, and a social network use the browser to execute an e-commerce protocol in which the user purchases an item from the store, employing the bank to perform a payment subprotocol and the social network to publicly announce the purchase. The browser may rely on plug-ins to integrate information from these different services. 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* that reveals the credit card used in the transaction. Such an undesired behaviour should be corrected as soon as possible. In fact, we would like to exert a stronger control 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 behaviour of the involved participants. This form of reconfiguration, however, should only concern the participants involved with the insecure plug-in; other participants should not be unnecessarily restarted.

In our example, since the unintended tweet concerns only the user, the bank, and the social network, the update should not affect the behaviour of the store. A message sequence diagram showing these behaviours is given in Figure 1, where all communications are intended to be realised by means of a browser, and ◎ represents an unsafe plug-in.

1.2. Our Approach

To specify and analyse such choreographic scenarios, we propose a framework for self-adaptive, multiparty communications which provides basic guarantees for *access control* and *secure information flow*. Building upon multiparty sessions [HYC08, HYC16], our framework consists of a language for defining processes and networks, equipped with global types but also with run-time monitors. Global types offer an overall description of a structured protocol between two or more participants; since in our model they also contain

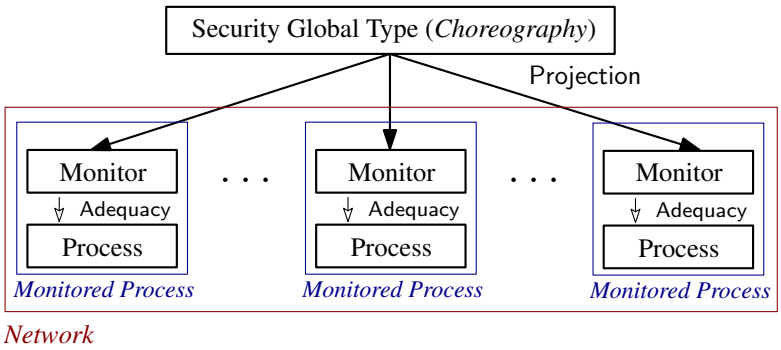

Fig. 2. Main ingredients of our model of multiparty, self-adaptive communications.

specifications of the intended security policies, we shall call them *security* global types. Run-time monitors are obtained as *projections* of a global type onto individual participants; they describe the global scenario from the local perspective of a single participant. Processes represent code that is coupled with monitors to implement participants; the compliance relation between processes and monitors is called *adequacy.* A (well-typed) network is a collection of monitored processes that realise the choreography described by the global type. Figure 2 gives a diagrammatical description of the model.

Monitors enable the communication behaviour of participants in the choreography, but also define their associated security policies. A distinctive aspect of our model of multiparty communication is that the monitor of each participant stipulates its *reading/writing permissions and boundaries for security violations*; these two pieces of information are represented by *security levels* (simply called "levels" in the following). While the reading permission is an upper bound for the level of incoming messages, the writing permission is a lower bound for the level of outgoing messages. The reading and writing boundaries extend the corresponding permissions, respectively upwards and downwards. Writing permissions may change as a result of message exchanges. Reading permissions may change when participants try to declassify values. Both permissions and boundaries can be modified through reconfiguration, as will be explained below. A security violation occurs when a participant attempts to read or write a message whose level is not allowed by the corresponding reading or writing permission. Security violations can be either *soft* or *hard*, depending on whether they involve levels within the corresponding boundaries or not:

- A reading violation is soft when the security level of the incoming value is less than or equal to the reading boundary, and hard otherwise.
- Dually, a writing violation is soft when the security level of the outgoing value is greater than or equal to the writing boundary, and hard otherwise.

The operational semantics for networks specifies the interaction between monitored processes, following an asynchronous (queue-based) communication discipline. It also couples security enforcement with adaptation mechanisms. To be precise, our semantics ensures that the reading/writing permissions are respected in protocol exchanges or, in case such permissions are violated, that an appropriate adaptation mechanism is triggered to limit the impact of the violation. We consider *local* and *global* adaptation mechanisms, intended to handle soft and hard violations, respectively:

- The **local adaptation mechanism** simply cancels the insecure action and continues with the rest of the protocol specified by the monitor. In case of a reading violation, the behaviour of the monitor is modified so as to skip the disallowed read, and the insecure receiver is replaced with a process compliant with the new monitor. The case of a writing violation is similar: the disallowed write is skipped, which entails modifying the behaviour of the monitor of the receiver, so as to ignore the attempted write.
- The **global adaptation mechanism** modifies the behaviour of all choreography participants involved in disallowed exchanges. To this end, this mechanism relies on *nonces*, distinguished dummy values generated only at run-time. 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 which removes the whole group of participants that

created the nonce or could propagate it, and replaces it with a new choreography (global type). Thus, in this form of adaptation, one part of the choreography is isolated and replaced. In order to keep track of the participants "tainted" with nonces, each session is equipped with a store that records the participants that created the nonces.

Returning to the e-commerce scenario discussed above, the interaction between the different participants and the browser would be specified using a global type. Such a type would provide a reference for the communication sequences to be performed by each participant (via the run-time monitors), but also would decree the security policies that the different message exchanges must observe (via the security levels associated to each monitor). Different ways of reacting to security violations can be specified by adjusting the security levels associated to each participant/monitor. Our model leaves unspecified the exact reaction in case of a reconfiguration. This is for the sake of generality, as different applications/scenarios may call for different criteria to replace a portion of the choreography, and for different alternative choreographies.

1.3. Contributions and Organisation

The main contribution of this paper is a formal model of multiparty communications in which monitored processes execute structured protocol interactions and enjoy access control and secure information flow guarantees. These guarantees are formally expressed by technical results which establish that (i) multiparty protocols are respected (*subject reduction*), (ii) messages are eventually delivered/received (*progress*), and (iii) communication actions (possibly in combination with adaptation mechanisms) respect security levels and boundaries (*access control* and *secure information flow*). To the best of our knowledge, ours is the first model that integrates concerns of communication correctness, self-adaptation, and access control/secure information flow in a unified way, building upon the approach of multiparty sessions [HYC08, HYC16].

The paper is structured as follows:

- § 2 presents the syntactic ingredients of our model: it defines security global types and monitors (which are obtained via projection), processes and networks (i.e., collections of monitored processes), and session types for processes. It also defines the adequacy relation between a process and its monitor, exploiting subtyping.
- § 3 defines the operational semantics of processes and networks, thus formalising the main novelties of our approach, namely the local and global adaptation mechanisms outlined above.
- § 4 illustrates further our framework with an example.
- § 5 extends typability from monitored processes to networks and presents the technical results of the paper. Namely, we prove that well-typed networks enjoy *subject reduction* and *progress* properties (Theorems 5.20 and 5.21, respectively). Moreover, we show that our operational semantics ensures compliance with reading/writing permissions and boundaries (Theorem 5.22).
- In § 6 we discuss related approaches and § 7 gives some concluding remarks.

This paper is a revised and extended version of the workshop paper [CDP14]. The current presentation includes additional technical details and proofs, clarifies the use of local and global adaptation mechanisms by introducing reading and writing boundaries, and offers additional examples (notably that of § 4).

2. Syntax

Our framework is inspired by that of [CDCV15], 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: *security global types*, *monitors* and *processes*. A security global type represents the overall communication choreography over a set of participants [HYC08, CHY12]. Moreover, it defines reading/writing permissions and reading/writing boundaries for each participant. The reading permissions, already used in [CCDC14], restrict the level of data a participant may receive from others. Similarly, the writing permissions restrict the level of data a participant may send to others. The reading/writing boundaries are used to set the limit between soft and hard violations of the corresponding permissions. 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 "matching" monitor, dubbed

	Read value v	Write value v
Safe behaviour	$lev(v) \sqsubseteq \mathsf{r}_p$	$\mathsf{w}_p \sqsubseteq lev(v)$
Soft violation	$lev(v) \not\sqsubseteq \mathsf{r}_p$ and $lev(v) \sqsubseteq \mathsf{r}_b$	$\mathsf{w}_p \not\sqsubseteq lev(v)$ and $\mathsf{w}_b \sqsubseteq lev(v)$
Hard violation	$lev(v) \not\sqsubseteq \mathsf{r}_b$	$\mathsf{w}_b \not\sqsubseteq lev(v)$

Table 1. Soft and hard security violations for reading and writing a value v.

monitored process, incarnates a participant whose process implements the monitoring protocol. Notably, we exploit intersection types, union types and subtyping to make this matching relation more flexible.

To deal with security, we assume as usual a finite lattice of *security levels* [Den76], denoted by $(\mathcal{L}, \sqsubseteq)$. We use $\ell, \ell', \ldots$ to range over elements of $\mathcal{L}$. We denote by $\sqcup$ and $\sqcap$ the join (least upper bound) and meet (greatest lower bound) operations on the lattice, respectively, and by $\bot$ and $\top$ its bottom and top elements.

2.1. Global Types and Monitors

Global types define overall schemes of labelled communications between session participants. In our setting, they also prescribe the *reading/writing permissions* and the *reading/writing boundaries* of participants. The *reading permission* is used for access control and represents, as usual, an upper bound for the level of data a participant is authorised to read (or input). Dually, the *writing permission* is used for secure information flow and represents a lower bound for the level of data a participant is authorised to write (or output). A reading/writing violation occurs when a participant attempts to read/write a value whose level does not fall within the corresponding bound. *Reading/writing boundaries* are a novelty of our approach. They are meant to relax the reading and writing permissions, respectively. According to this intuition, for each participant the reading boundary will be higher than the reading permission, and the writing boundary will be lower than the writing permission.[1]

We introduce boundaries to distinguish between two kinds of security violations, while sticking to a qualitative approach to security: soft security violations, in case the level of the improperly read/written value is comprised between the reading/writing permission and the corresponding boundary, and hard violations, otherwise.

Formally, we use $\mathsf{r}, \mathsf{r}', \ldots$ to range over pairs of levels of the form (reading permission, reading boundary), and $\mathsf{w}, \mathsf{w}', \ldots$ to range over pairs of the form (writing permission, writing boundary). The first and second element of r are denoted by r_p and r_b, and similarly for the elements of w, for which we use w_p and w_b. Given these notations for permissions and boundaries, Table 1 summarises our proposal for distinguishing soft and hard violations. In the table, we write $lev(v)$ to denote the *security level* of value v (see § 2.2).

We assume base sets of *participants*, ranged over by $\mathtt{p}, \mathtt{q}, \mathtt{k}, \ldots$; *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*, ranged over by S:

$$S ::= \mathsf{bool} \mid \mathsf{nat} \mid \ldots.$$

Our security global types, defined below, contain abstractions of structured protocols, as customary [HYC08]; they also include a pair of mappings that assigns to each protocol participant a pair $\mathsf{r} = (\mathsf{r}_p, \mathsf{r}_b)$, and a pair $\mathsf{w} = (\mathsf{w}_p, \mathsf{w}_b)$. All these levels are used to enforce security and allow adaptation at run-time.

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

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

We use $\mathsf{part}(\mathsf{G})$ to denote the set of participants in G, i.e., all senders $\mathtt{p}$ and receivers $\mathtt{q}$ occurring in G. Let Lr and Lw be mappings from $\mathsf{part}(\mathsf{G})$ to $\mathcal{L} \times \mathcal{L}$, assigning to each participant in G respectively a pair $\mathsf{r} = (\mathsf{r}_p, \mathsf{r}_b)$ and a pair $\mathsf{w} = (\mathsf{w}_p, \mathsf{w}_b)$ such that $\mathsf{r}_p \sqsubseteq \mathsf{r}_b$ and $\mathsf{w}_p \sqsupseteq \mathsf{w}_b$. Let $\mathsf{L} = (\mathsf{Lr}, \mathsf{Lw})$. Then the pair (G, L) is a *security global type*.

[1] The reader familiar with type systems for secure information flow will be probably reminded here of the security subtyping, which is covariant for expressions (whose type is a read level) and contra-variant for programs (whose type is a write level). However, the analogy should be taken with some care because our reading permissions refer to participants rather than to data, our writing permissions are dynamic (flow-sensitive), and boundaries are introduced to trespass the given bounds (permissions), rather than to comply with them.

$$(\mathtt{p} \to \mathtt{q} : \{\lambda_i(S_i).\mathsf{G}_i\}_{i\in I}) \upharpoonright \mathtt{k} = \begin{cases} \mathtt{p}?\{\lambda_i(S_i).\mathsf{G}_i \upharpoonright \mathtt{q}\}_{i\in I} & \text{if } \mathtt{k} = \mathtt{q} \\ \mathtt{q}!\{\lambda_i(S_i).\mathsf{G}_i \upharpoonright \mathtt{p}\}_{i\in I} & \text{if } \mathtt{k} = \mathtt{p} \\ \mathsf{G}_{i_0} \upharpoonright \mathtt{k} & \text{if } \mathtt{k} \neq \mathtt{p} \text{ and } \mathtt{k} \neq \mathtt{q} \text{ where } i_0 \in I \\ & \text{and } \mathsf{G}_i \upharpoonright \mathtt{k} = \mathsf{G}_j \upharpoonright \mathtt{k} \text{ for all } i, j \in I \end{cases}$$

$$\mathbf{t} \upharpoonright \mathtt{k} = \mathbf{t} \qquad (\mu\mathbf{t}.\mathsf{G}) \upharpoonright \mathtt{k} = \begin{cases} \mu\mathbf{t}.\mathsf{G} \upharpoonright \mathtt{k} & \text{if } \mathtt{k} \text{ occurs in } \mathsf{G} \\ \mathsf{end} & \text{otherwise} \end{cases} \qquad \mathsf{end} \upharpoonright \mathtt{k} = \mathsf{end}$$

Table 2. Projection of a global type onto a participant.

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 characterised 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* of global types onto individual participants, following the definition in [HYC08, CDCYP16] (see Table 2). 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 on 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 (cf. [HYC08]), in our model they have an active role in the dynamics of networks, since they guide and enable directed communications—see § 3.

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

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

An input monitor $\mathtt{p}?\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}$ matches a process that can receive, for each $i \in I$, a value of sort S_i, labelled by λ_i, and then continues as specified by $\mathcal{M}_i$. This corresponds to an external choice. Dually, an output monitor $\mathtt{q}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}$ matches a process which can send, for each $i \in I$, a value of sort S_i, labelled by λ_i, and then continues as prescribed by $\mathcal{M}_i$. As such, it corresponds to an internal choice. Monitors for recursive and completed protocols, denoted $\mu\mathbf{t}.\mathcal{M}$ and end, respectively, are as expected.

2.2. Processes and Networks

The syntax of processes relies on *expressions* and *values*. We assume a set $\mathcal{E}$ of expressions, ranged over by $\mathtt{e}, \mathtt{e}', \ldots$, which includes booleans and naturals (with operations over them), and a denumerable set $Nonces = \{nonce_i \mid i \geq 0\}$. Other expressions can be added without problems according to the considered framework. A term $nonce_i$ —where i is fresh— is a dummy value, generated at run-time; it is to be used in place of some improperly sent value to prevent security violations—see § 3. We will use v, v' to range over ordinary *values*, and u, u' to denote *extended values*, which are either values or nonces. Expressions evaluate to extended values: in particular, expressions containing nonces evaluate nondeterministically to one of these nonces, as will be explained in more detail in § 3. Constants are assumed to be decorated with security levels: thus, for instance, $\mathsf{true}^{\bot}$ and $\mathsf{true}^{\top}$ are two distinct values. Each ordinary expression $\mathtt{e}$, not containing nonces, is equipped with a security level given by the function $lev : \mathcal{E} \to \mathcal{L}$: if v is a constant, then $lev(\mathtt{e})$ is simply the level that decorates it, otherwise $lev(\mathtt{e})$ is the join of the levels of all subexpressions of $\mathtt{e}$. Nonces have no security level, since they carry no information.

We now define our set of processes, which represent code that will be coupled with monitors to implement participants. In our model, like in [CDCV15]— but unlike in other session calculi [HVK98, HYC08, BCD$^+$13, CDCYP16]— processes 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 run-time, channel y will be replaced by a *session channel* $\mathsf{s}[\mathsf{p}]$, where s is the session name and p denotes the participant. We use c to stand for a user channel y or a session channel $\mathsf{s}[\mathsf{p}]$.

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

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

The syntax of processes is rather standard: in addition to usual constructs for communication, recursion, conditionals, and inactive behaviour, 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. In the following we shall omit trailing $\mathbf{0}$'s in processes.

The previously introduced entities (global types, monitors, processes) are used to define *networks*. A network is a collection of monitored processes which realise 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* [CDCV15], 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 process communications agree with the protocol prescribed by the global type. Importantly, each monitor is equipped with a reading pair r and a writing pair w. The elements of the reading pair are the reading permission and the reading boundary, the elements of the writing pair are the writing permission and the writing boundary. A monitored process, written $\mathcal{M}^{\mathsf{r},\mathsf{w}}[P]$, denotes a process P controlled by a monitor $\mathcal{M}$.

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. The empty queue is denoted by $\emptyset$. Messages in queues are of the form $(\mathsf{p}, \mathsf{q}, \lambda(u))$, indicating that the label λ and the extended value u are communicated with sender p and receiver q. Queue concatenation is denoted by "$\cdot$": it is associative and has $\emptyset$ as neutral element.

Each session is equipped with a store σ that records nonce creation, i.e.,

$$\sigma ::= \emptyset \quad | \quad \sigma, (\mathsf{p}, \mathit{nonce}_i)$$

We dub *named buffer* the pair of the queue and the store associated with session s (notation $\mathsf{s} : \langle h; \sigma \rangle$).

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

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

$$N \quad ::= \quad \mathsf{new}(\mathsf{G}, \mathsf{L}) \quad | \quad \mathcal{M}^{\mathsf{r},\mathsf{w}}[P] \quad | \quad \mathsf{s} : \langle h; \sigma \rangle \quad | \quad N \mid N \quad | \quad (\nu \mathsf{s})N$$

As mentioned above, annotations r and w in $\mathcal{M}^{\mathsf{r},\mathsf{w}}[P]$ represent reading/writing permissions and boundaries for process P. When the choreography is initialised, these levels are set according to the mappings Lr and Lw. The actions performed by the process may determine dynamic modifications to the permissions, while the boundaries may only change as a result of global adaptation. Since reading permissions can only be lowered and writing permissions can only be raised, the order relations between permissions and boundaries are preserved. In writing monitored processes we omit the levels when they are not used. Also, we shall sometimes write $\mathcal{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P]$ (or simply $\mathcal{M}_{\mathsf{p}}[P]$) to indicate that the channel in P is $\mathsf{s}[\mathsf{p}]$ for some s. Similarly P_{p} means that the channel in P_{p} is $\mathsf{s}[\mathsf{p}]$. Only in the premise of Rule Init (see Table 6) the channel in P_{p} is y, but y will be replaced with $\mathsf{s}[\mathsf{p}]$ in the conclusion of the rule.

2.3. Session Types for Processes

We now introduce session types for processes and specify the relationship between processes, types, and monitors. As in [CDCV15], process types (called *types* when not ambiguous) describe process communication behaviours. Types have prefixes corresponding to input and output actions. In particular, an *input type* (resp. *output type*) has a prefix corresponding to an input (resp. output) action, followed by another type called its *continuation*. A *communication type* is either an input or an output type. Inspired by [Pad11], we use intersection and union types instead of standard branching and selection [HYC08], to take advantage from the subtyping induced by subset inclusion.

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

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

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{bool} \quad \Gamma \vdash P_1 \rhd \mathsf{c} : \mathsf{T}_1 \quad \Gamma \vdash P_2 \rhd \mathsf{c} : \mathsf{T}_2}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \rhd \mathsf{c} : \mathsf{T}_1 \vee \mathsf{T}_2} \text{IF} \qquad \frac{\Gamma \vdash P_1 \rhd \mathsf{c} : \mathsf{T}_1 \quad \Gamma \vdash P_2 \rhd \mathsf{c} : \mathsf{T}_2}{\Gamma \vdash P_1 + P_2 \rhd \mathsf{c} : \mathsf{T}_1 \wedge \mathsf{T}_2} \text{CHOICE}$$

Table 3. Typing rules for processes.

$$[\text{SUB-END}]\ \mathsf{T} \leq \mathsf{end} \qquad [\text{SUB-IN}]\ \frac{\forall i \in I : \quad \mathsf{T}_i \leq \mathsf{T}'_i}{\bigwedge_{i \in I \cup J} ?\lambda_i(S_i).\mathsf{T}_i \leq \bigwedge_{i \in I} ?\lambda_i(S_i).\mathsf{T}'_i} \qquad [\text{SUB-OUT}]\ \frac{\forall i \in I : \quad \mathsf{T}_i \leq \mathsf{T}'_i}{\bigvee_{i \in I} !\lambda_i(S_i).\mathsf{T}_i \leq \bigvee_{i \in I \cup J} !\lambda_i(S_i).\mathsf{T}'_i}$$

Table 4. Subtyping on process types.

The grammar of session types, ranged over by T, is then

$$\mathsf{T} \quad ::= \quad \textstyle\bigwedge_{i \in I} ?\lambda_i(S_i).\mathsf{T}_i \quad \mid \quad \bigvee_{i \in I} !\lambda_i(S_i).\mathsf{T}_i \quad \mid \quad \mu \mathbf{t}.\mathsf{T} \quad \mid \quad \mathbf{t} \quad \mid \quad \mathsf{end}$$

where we require that $\lambda_i \neq \lambda_j$ for all $i, j \in I$ such that $i \neq j$.

Intersection types are used to type external choices, since an external choice offers both behaviours of the composing processes. Dually, union types are used to type conditional expressions (internal choices).

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 3. We assume that expressions are typed by sorts, as usual, and a nonce has all sorts.

The compliance between process types and monitors (*adequacy*) is made flexible by using the *subtyping* relation on types, denoted $\leq$. Intuitively, $\mathsf{T}_1 \leq \mathsf{T}_2$ means that a process with type T_1 has all the behaviours required by type T_2 but possibly more. Subtyping is monotone, for input/output prefixes, with respect to continuations, and it follows the usual set theoretic inclusion of intersection and union. The top type is end, since it has no requirement. Table 4 gives the subtyping rules: the double line in rules indicates that the rules are interpreted *coinductively* [Pie02, 21.1]. Subtyping can be easily decided, see for example [GH05]. For reader convenience, below we give the procedure $\mathcal{S}(\Sigma, \mathsf{T}, \mathsf{T}')$, where Σ is a set of subtyping judgments.

$$\mathcal{S}(\Sigma, \mathsf{T}, \mathsf{T}') = \begin{cases} \mathsf{true} & \text{if } \mathsf{T} \leq \mathsf{T}' \in \Sigma \text{ or } \mathsf{T}' = \mathsf{end} \\ \&_{i \in I}\, \mathcal{S}(\Sigma \cup \{\mathsf{T} \leq \mathsf{T}'\}, \mathsf{T}_i, \mathsf{T}'_i) & \text{if } (\mathsf{T} = \bigwedge_{i \in I \cup J} ?\lambda_i(S_i).\mathsf{T}_i \text{ and } \mathsf{T}' = \bigwedge_{i \in I} ?\lambda_i(S_i).\mathsf{T}'_i) \text{ or} \\ & \text{if } (\mathsf{T} = \bigvee_{i \in I} !\lambda_i(S_i).\mathsf{T}_i \text{ and } \mathsf{T}' = \bigvee_{i \in I \cup J} !\lambda_i(S_i).\mathsf{T}'_i) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

This procedure is a decision procedure for the subtyping ordering. It terminates since unfolding of session types generates regular trees, so Σ cannot grow indefinitely and we have only a finite number of subtyping judgments to consider. Clearly $\mathcal{S}(\emptyset, \mathsf{T}, \mathsf{T}')$ is equivalent to $\mathsf{T} \leq \mathsf{T}'$.

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

Definition 2.5 (Adequacy). Let the mapping $|\cdot|$ from monitors to types be defined as

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

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

In the following we will omit security levels, labels, brackets, unions and intersections whenever suitable.

3. Semantics

In this section, we define the operational semantics for monitors, processes, and networks. The semantics of networks makes use of a *collection $\mathcal{P}$ of pairs* (P, T) made of a *user process* P (namely, a process without session channels) together with its type T. This collection will be used in session initialisations, in local adaptations, and in reconfigurations in order to provide processes whose types match the new monitors.

The semantics of monitors and processes is given by labelled transition systems (LTS); relying on these two LTSs, the semantics of networks, parametrised on the collection $\mathcal{P}$, is given in the style of a reduction semantics. Local and global adaptation mechanisms are therefore formalised as reduction steps.

A monitor guides the communications of a process by choosing its partners in labelled 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 formalises the expected intuitions:

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

The LTS for processes is also fairly simple. It is given in Table 5, omitting symmetric rules. 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. We use $\mathtt{e} \downarrow u$ to indicate that expression $\mathtt{e}$ evaluates to the extended value u, assuming that each expression containing nonces evaluates nondeterministically to one of these nonces and that $u \downarrow u$. For example, $\mathit{nonce}_1 \& \mathit{nonce}_2 \downarrow \mathit{nonce}_1$ and $\mathit{nonce}_1 \& \mathit{nonce}_2 \downarrow \mathit{nonce}_2$. In order to track information flow, when reducing a conditional we record the level of the tested expression $\mathtt{e}$. If the tested expression evaluates to a nonce, the conditional behaves as an external choice. In this way the monitors select the branch and the process will not be stuck. There are two rules for external choice, which specify that a choice may be resolved only via a communication action by one of the summands. As long as these perform only internal computations, the choice remains available. In other words, choices are tied to communication actions and are not affected by the testing activity that goes on in conditional expressions. Notice that nonces are generated only when networks are reduced.

We now describe the reduction semantics for networks. It relies on a structural equivalence for which the parallel operator is commutative and associative:

$$N_1 \mid N_2 \equiv N_2 \mid N_1 \qquad (N_1 \mid N_2) \mid N_3 \equiv N_1 \mid (N_2 \mid N_3)$$

and session restriction behaves as usual:

$$(\nu\mathsf{s})N_1 \mid N_2 \equiv (\nu\mathsf{s})(N_1 \mid N_2) \qquad (\nu\mathsf{s})(\nu\mathsf{s}')N \equiv (\nu\mathsf{s}')(\nu\mathsf{s})N \qquad (\nu\mathsf{s})(\mathsf{s} : \langle\emptyset;\sigma\rangle) \mid N \equiv N$$

Moreover, the structural equivalence erases any monitored process with end monitor, since it is idle:

$$\mathsf{end}[P] \mid N \;\equiv\; N$$

For message queues, the structural equivalence is very simple. Two consecutive messages in a queue are *dependent* if they have the same sender and the same receiver; they are *independent* otherwise. The structural equivalence for queues allows independent messages to be commuted.

$$h \cdot (\mathtt{p}, \mathtt{q}, \lambda(u)) \cdot (\mathtt{p}', \mathtt{q}', \lambda'(u')) \cdot h' \equiv h \cdot (\mathtt{p}', \mathtt{q}', \lambda'(u')) \cdot (\mathtt{p}, \mathtt{q}, \lambda(u)) \cdot h' \qquad \text{if } \mathtt{p} \neq \mathtt{p}' \text{ or } \mathtt{q} \neq \mathtt{q}'$$

Note that the structural equivalence allows a request message from $\mathtt{p}$ to $\mathtt{q}$ and the reply from $\mathtt{q}$ to $\mathtt{p}$ to be commuted. This is because the message queue may be viewed as a merge of all participants' reading queues, which are all independent from each other. Here the request message from $\mathtt{p}$ to $\mathtt{q}$ belongs to the reading queue of $\mathtt{q}$, while the reply from $\mathtt{q}$ to $\mathtt{p}$ belongs to the reading queue of $\mathtt{p}$.

The equivalence on message queues induces an equivalence on named buffers in the expected way:

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

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

$$\frac{P \xrightarrow{\alpha} P'}{\mathsf{if\ e\ then}\ P\ \mathsf{else}\ Q \xrightarrow{\alpha} P'} \quad \mathsf{e}\downarrow \mathit{nonce}_i \qquad \frac{Q \xrightarrow{\alpha} Q'}{\mathsf{if\ e\ then}\ P\ \mathsf{else}\ Q \xrightarrow{\alpha} Q'} \quad \mathsf{e}\downarrow \mathit{nonce}_i$$

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

Table 5. LTS of processes.

$$h \equiv h' \ \text{ implies } \ \mathsf{s} : \langle h;\sigma\rangle \equiv \mathsf{s} : \langle h';\sigma\rangle.$$

The rules of the reduction semantics rely on some auxiliary definitions, which we detail next. The following two definitions are handy to formalise the local adaptation mechanism.

Given a message queue and two participants $\mathtt{p}$ and $\mathtt{q}$, the function # below computes the list of labels of messages from $\mathtt{p}$ to $\mathtt{q}$ in the queue. We use Λ to range over lists of labels. By a slight abuse of notation, we shall use $\emptyset$ and $\cdot$ to denote the empty list and list concatenation, respectively.

Definition 3.1 (Function #). Given a queue h and two participants $\mathtt{p}$, $\mathtt{q}$, we define $\#(h,\mathtt{p},\mathtt{q})$ by induction on h:

$$\#(\emptyset,\mathtt{p},\mathtt{q}) = \emptyset$$

$$\#((\mathtt{p}',\mathtt{q}',\lambda(u))\cdot h',\mathtt{p},\mathtt{q}) = \begin{cases} \lambda\cdot\#(h',\mathtt{p},\mathtt{q}) & \text{if } \mathtt{p}=\mathtt{p}' \text{ and } \mathtt{q}=\mathtt{q}',\\ \#(h',\mathtt{p},\mathtt{q}) & \text{otherwise.}\end{cases}$$

The second definition "strips off" from a monitor $\mathcal{M}$ the occurrence of an input prefix which refers to a given participant $\mathtt{p}$ and follows a given list Λ of labels. The branchings with sender $\mathtt{p}$ and labels alternative to Λ are also erased.

Definition 3.2 (Monitor with Skipped Input). Given a monitor $\mathcal{M}$ and a list of labels Λ, we define $<\mathcal{M},\Lambda> \backslash ?(\mathtt{p},\lambda)$ inductively as follows:

$$<\mathtt{q}?\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I},\Lambda> \backslash ?(\mathtt{p},\lambda) = \begin{cases} \mathcal{M}_i & \text{if } \mathtt{p}=\mathtt{q} \text{ and } \lambda=\lambda_i \text{ and } \Lambda=\emptyset,\\ \mathtt{p}?\lambda'(S_i).(<\mathcal{M}_i,\Lambda'> \backslash ?(\mathtt{p},\lambda)) & \text{if } \mathtt{p}=\mathtt{q} \text{ and } \lambda'=\lambda_i \text{ and } \Lambda=\lambda'\cdot\Lambda',\\ \mathtt{q}?\{\lambda_i(S_i).(<\mathcal{M}_i,\Lambda> \backslash ?(\mathtt{p},\lambda))\}_{i\in I} & \text{if } \mathtt{p}\neq\mathtt{q}.\end{cases}$$

$$<\mathtt{q}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I},\Lambda> \backslash ?(\mathtt{p},\lambda) = \mathtt{q}!\{\lambda_i(S_i).(<\mathcal{M}_i,\Lambda> \backslash ?(\mathtt{p},\lambda))\}_{i\in I}$$

$$<\mu\mathbf{t}.\mathcal{M},\Lambda> \backslash ?(\mathtt{p},\lambda) = \mu\mathbf{t}.(<\mathcal{M},\Lambda> \backslash ?(\mathtt{p},\lambda))$$

Notice that $<\mathcal{M},\Lambda> \backslash ?(\mathtt{p},\lambda)$ is defined only when $\mathcal{M}$ prescribes the reception from participant $\mathtt{p}$ of messages with labels in $\Lambda\cdot\lambda$ in the specified order.

Let us now introduce a couple of notations that will be useful for defining the global adaptation mechanism. We first define an operation that eliminates from a queue all messages whose receiver belongs to a given set of participants:

Definition 3.3 (Restriction of a Queue wrt a Set of Participants).
Let Π be a set of participants. Given the message queue h, we define $h \setminus \Pi$ as follows:

$$\emptyset \setminus \Pi = \emptyset$$

$$(\mathtt{q},\mathtt{p},\lambda(u))\cdot h \setminus \Pi = \begin{cases} h\setminus\Pi & \text{if } \mathtt{p}\in\Pi\\ (\mathtt{q},\mathtt{p},\lambda(u))\cdot(h\setminus\Pi) & \text{otherwise.}\end{cases}$$

We now define the "range of contamination" of a given nonce, namely the set of participants that can

contain that nonce, plus the participant that generated that nonce. A participant $\mathtt{p}$ can be contaminated by a participant $\mathtt{q}$ only if $\mathtt{p}$ communicates with $\mathtt{q}$ and $\mathtt{q}$ in turn can be contaminated.

Definition 3.4 (Set of Participants Affected by a Nonce).
Given a set of monitored processes $\{\mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{p} \in \Pi\}$, a store σ, and a nonce nonce_i, we define

$$\mathcal{A}(\{\mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{p} \in \Pi\}, \sigma, \mathit{nonce}_i) = \bigcup_{j \in \mathbb{N}} \mathcal{A}_j(\{\mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{p} \in \Pi\}, \sigma, \mathit{nonce}_i)$$

where $\mathcal{A}_j$ is defined as:

$$\mathcal{A}_0(\{\mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{p} \in \Pi\}, \sigma, \mathit{nonce}_i) = \{\mathtt{p} \in \Pi \mid \mathit{nonce}_i \in P_{\mathtt{p}}\} \cup \{\mathtt{q} \mid (\mathtt{q}, \mathit{nonce}_i) \in \sigma\}$$
$$\mathcal{A}_{j+1}(\{\mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{p} \in \Pi\}, \sigma, \mathit{nonce}_i) = \{\mathtt{p} \in \Pi \mid \exists \mathtt{q} \in \mathcal{A}_j(\{\mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{p} \in \Pi\}, \sigma, \mathit{nonce}_i).\, \mathtt{q} \in \mathcal{M}_{\mathtt{p}}\}$$

Finally, to define the semantics of networks we will use *evaluation contexts*, as standard:

Definition 3.5 (Evaluation Contexts). *Evaluation contexts* are defined as follows:

$$\mathcal{E} ::= [\,] \mid \mathcal{E} \mid N \mid (\nu s)\mathcal{E}$$

The reduction of networks is denoted $N \longrightarrow_{\mathcal{P}} N'$, where $\mathcal{P}$ is the collection of pairs (P, T) used to find processes with adequate types for monitors (cf. Definition 3.7). We also write $N \longrightarrow^*_{\mathcal{P}} N'$ with the expected meaning. The reduction rules for networks are given in Table 6. We briefly describe them. The first four and the last two rules in the table describe the "normal" execution of a network:

- Rule Init initialises a choreography denoted by global type G. The network $\mathsf{new}(\mathsf{G}, \mathsf{L})$ evolves into a composition of monitored processes and a fresh named buffer. For each $\mathtt{p} \in \mathsf{part}(\mathsf{G})$, type $\mathsf{T}_{\mathtt{p}}$ must be adequate for the monitor obtained by projecting G onto $\mathtt{p}$, and there must be a pair $(P_{\mathtt{p}}, \mathsf{T}_{\mathtt{p}})$ in the collection $\mathcal{P}$. Then process $P_{\mathtt{p}}$ (where channel y has been replaced by $s[\mathtt{p}]$) is coupled with the corresponding monitor, whose security permissions and boundaries are initialised using the mapping L. Lastly, the empty named buffer $\mathtt{s} : \langle \varnothing; \emptyset \rangle$ is created, and the name $\mathtt{s}$ is restricted.
- Rule UpLev modifies the writing permission of a monitor whose associated process tests a value. Indeed, by the semantics of processes (Table 5), we know that the label ℓ is the level associated to a conditional expression. The writing permission is updated to the join of ℓ and the current writing permission (noted w_p, assuming $\mathsf{w} = (\mathsf{w}_p, \mathsf{w}_b)$). This is to prevent classical implicit information leaks in conditionals.
- Rule In defines the secure input of an extended value u present in the queue; this action must be enabled by the monitor. If u is a proper value v, we further require its associated level to be lower than or equal to the reading permission r_p of the monitor. Nothing is required in case u is a nonce, since nonces provide no information.
- Rule Out defines the secure output of an extended value u by adding it to the queue. 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 writing permission w_p of the monitor. Nothing is required in case u is a nonce, as in the case of input.
- Rules Equiv and Ctx are standard: they allow the interplay of reduction with structural congruence and enable the reduction within evaluation contexts, respectively.

The remaining rules in Table 6 concern the novel local and global adaptation mechanisms:

- Rule InLoc defines the *local adaptation* mechanism for a *soft reading violation*, i.e., when the level of the value in the queue is not less than or equal to permission r_p, but it is still less than or equal to boundary r_b. To deal with such an "admissible" reading violation, this rule ignores the insecure input: the message is removed from the queue and the process implementation is replaced with new code, where the input action is not present. This code replacement considers the monitor resulting after the input action (noted $\widehat{\mathcal{M}}_{\mathtt{p}}$ in the rule), and picks up a process P' in the collection $\mathcal{P}$ that agrees with this monitor.
- Rule OutLoc defines the *local adaptation* mechanism for a *soft writing violation*, i.e., when the level of the sent value is not greater than or equal to permission w_p, but it is still greater than or equal to boundary w_b. As in InLoc, the monitor of the receiver is modified and the implementation is replaced with one that conforms to the modified monitor. In the rule, the monitor $<\mathcal{M}_{\mathtt{q}}, \#(h, \mathtt{p}, \mathtt{q})> \backslash ?(\mathtt{p}, \lambda)$ is obtained from $\mathcal{M}_{\mathtt{q}}$ by erasing the input action dual to the shown output and choosing the corresponding

$$\frac{\mathcal{M}_{\mathsf{p}} = \mathsf{G}\upharpoonright\mathsf{p} \qquad \forall \mathsf{p} \in \mathsf{part}(\mathsf{G}).\ (P_{\mathsf{p}}, \mathsf{T}_{\mathsf{p}}) \in \mathcal{P}\ \&\ \mathsf{T}_{\mathsf{p}} \propto \mathcal{M}_{\mathsf{p}}}{\mathsf{new}(\mathsf{G},\mathsf{L}) \ \longrightarrow_{\mathcal{P}}\ (\nu\,\mathsf{s}) \prod_{\mathsf{p}\,\in\,\mathsf{part}(\mathsf{G})} (\mathcal{M}_{\mathsf{p}}^{\mathsf{Lr}(\mathsf{p}),\mathsf{Lw}(\mathsf{p})}[P_{\mathsf{p}}\{\mathsf{s}[\mathsf{p}]/y\}] \mid \mathsf{s} : \langle \varnothing; \emptyset \rangle)}\ \textsc{Init}$$

$$\frac{P \xrightarrow{\ell} P'}{\mathcal{M}^{\mathsf{r},\mathsf{w}}[P] \longrightarrow_{\mathcal{P}}\ \mathcal{M}^{\mathsf{r},(\mathsf{w}_p \sqcup \ell, \mathsf{w}_b)}[P']}\ \textsc{UpLev}$$

$$\frac{\mathcal{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}?\lambda} \widehat{\mathcal{M}}_{\mathsf{p}} \qquad P \xrightarrow{\mathsf{s}[\mathsf{p}]?\lambda(u)} P' \qquad u \in \mathit{Nonces} \text{ or } (u = v \text{ and } \mathit{lev}(v) \sqsubseteq \mathsf{r}_p)}{\mathcal{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathsf{s} : \langle (\mathsf{q},\mathsf{p},\lambda(u)) \cdot h; \sigma \rangle \longrightarrow_{\mathcal{P}}\ \widehat{\mathcal{M}}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P'] \mid \mathsf{s} : \langle h; \sigma \rangle}\ \textsc{In}$$

$$\frac{\mathcal{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}!\lambda} \widehat{\mathcal{M}}_{\mathsf{p}} \qquad P \xrightarrow{\mathsf{s}[\mathsf{p}]!\lambda(u)} P' \qquad u \in \mathit{Nonces} \text{ or } (u = v \text{ and } \mathsf{w}_p \sqsubseteq \mathit{lev}(v))}{\mathcal{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathsf{s} : \langle h; \sigma \rangle \longrightarrow_{\mathcal{P}}\ \widehat{\mathcal{M}}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P'] \mid \mathsf{s} : \langle h \cdot (\mathsf{p},\mathsf{q},\lambda(u)); \sigma \rangle}\ \textsc{Out}$$

$$\frac{\mathcal{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}?\lambda} \widehat{\mathcal{M}}_{\mathsf{p}} \qquad P \xrightarrow{\mathsf{s}[\mathsf{p}]?\lambda(v)} P' \qquad \mathsf{T} \propto \widehat{\mathcal{M}}_{\mathsf{p}} \qquad (P', \mathsf{T}) \in \mathcal{P} \qquad \mathit{lev}(v) \not\sqsubseteq \mathsf{r}_p \qquad \mathit{lev}(v) \sqsubseteq \mathsf{r}_b}{\mathcal{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathsf{s} : \langle (\mathsf{q},\mathsf{p},\lambda(v)) \cdot h; \sigma \rangle \longrightarrow_{\mathcal{P}}\ \widehat{\mathcal{M}}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P'] \mid \mathsf{s} : \langle h; \sigma \rangle}\ \textsc{InLoc}$$

$$\frac{\begin{array}{c}\mathcal{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}!\lambda} \widehat{\mathcal{M}}_{\mathsf{p}} \qquad P \xrightarrow{\mathsf{s}[\mathsf{p}]!\lambda(v)} P' \qquad \mathsf{w}_p \not\sqsubseteq \mathit{lev}(v) \qquad \mathsf{w}_b \sqsubseteq \mathit{lev}(v) \\ \widehat{\mathcal{M}}_{\mathsf{q}} = \mathcal{M}_{\mathsf{q}} \mathbin{<\!\!|} \#(h,\mathsf{p},\mathsf{q}) \mathbin{|\!>} \backslash ?(\mathsf{p},\lambda) \qquad \mathsf{T} \propto \widehat{\mathcal{M}}_{\mathsf{q}} \qquad (Q', \mathsf{T}) \in \mathcal{P}\end{array}}{\mathcal{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathcal{M}_{\mathsf{q}}^{\mathsf{r}',\mathsf{w}'}[Q] \mid \mathsf{s} : \langle h; \sigma \rangle \longrightarrow_{\mathcal{P}}\ \widehat{\mathcal{M}}_{\mathsf{p}}{}^{\mathsf{r},\mathsf{w}}[P'] \mid \widehat{\mathcal{M}}_{\mathsf{q}}{}^{\mathsf{r}',\mathsf{w}'}[Q'\{\mathsf{s}[\mathsf{q}]/y\}] \mid \mathsf{s} : \langle h; \sigma \rangle}\ \textsc{OutLoc}$$

$$\frac{\mathcal{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}?\lambda} \widehat{\mathcal{M}}_{\mathsf{p}} \qquad \mathit{nonce}_i = \mathsf{next}(\sigma) \qquad P \xrightarrow{\mathsf{s}[\mathsf{p}]?\lambda(\mathit{nonce}_i)} P' \qquad \mathit{lev}(v) \not\sqsubseteq \mathsf{r}_b}{\mathcal{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathsf{s} : \langle (\mathsf{q},\mathsf{p},\lambda(v)) \cdot h; \sigma \rangle \longrightarrow_{\mathcal{P}}\ \widehat{\mathcal{M}}_{\mathsf{p}}{}^{\mathsf{r},\mathsf{w}}[P'] \mid \mathsf{s} : \langle h; \sigma, (\mathsf{p}, \mathit{nonce}_i) \rangle}\ \textsc{InGlob}$$

$$\frac{\mathcal{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}!\lambda} \widehat{\mathcal{M}}_{\mathsf{p}} \qquad P \xrightarrow{\mathsf{s}[\mathsf{p}]!\lambda(v)} P' \qquad \mathit{nonce}_i = \mathsf{next}(\sigma) \qquad h' = h \cdot (\mathsf{p},\mathsf{q},\lambda(\mathit{nonce}_i)) \qquad \mathsf{w}_b \not\sqsubseteq \mathit{lev}(v)}{\mathcal{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathcal{M}_{\mathsf{q}}^{\mathsf{r}',\mathsf{w}'}[Q] \mid \mathsf{s} : \langle h; \sigma \rangle \longrightarrow_{\mathcal{P}}\ \widehat{\mathcal{M}}_{\mathsf{p}}{}^{(\mathsf{r}_p \sqcap \mathsf{r}'_p, \mathsf{r}_b), \mathsf{w}}[P'] \mid \mathcal{M}_{\mathsf{q}}^{\mathsf{r}',\mathsf{w}'}[Q] \mid \mathsf{s} : \langle h'; \sigma, (\mathsf{p}, \mathit{nonce}_i) \rangle}\ \textsc{OutGlob}$$

$$\frac{\mathcal{A}(\{\mathcal{M}_{\mathsf{p}}[P_{\mathsf{p}}] \mid \mathsf{p} \in \Pi\}, \sigma, \mathit{nonce}_i) = \Pi' \qquad F(\{\mathcal{M}_{\mathsf{p}}[P_{\mathsf{p}}] \mid \mathsf{p} \in \Pi'\}, \sigma) = (\mathsf{G},\mathsf{L})}{(\nu\,\mathsf{s})\,(\prod_{\mathsf{p}\in\Pi} \mathcal{M}_{\mathsf{p}}[P_{\mathsf{p}}] \mid \mathsf{s} : \langle h; \sigma \rangle) \longrightarrow_{\mathcal{P}}\ (\nu\,\mathsf{s})\,(\prod_{\mathsf{p}\in\Pi\setminus\Pi'} \mathcal{M}_{\mathsf{p}}[P_{\mathsf{p}}] \mid \mathsf{s} : \langle h \setminus \Pi'; \sigma \setminus \mathit{nonce}_i \rangle) \mid \mathsf{new}(\mathsf{G},\mathsf{L})}\ \textsc{Reconf}$$

$$\frac{N_1 \equiv N_1' \qquad N_1' \longrightarrow_{\mathcal{P}}\ N_2' \qquad N_2 \equiv N_2'}{N_1 \longrightarrow_{\mathcal{P}}\ N_2}\ \textsc{Equiv} \qquad \frac{N \longrightarrow_{\mathcal{P}}\ N'}{\mathcal{E}[N] \longrightarrow_{\mathcal{P}}\ \mathcal{E}[N']}\ \textsc{Ctx}$$

Table 6. Reduction rules for networks.

branch (cf. Definitions 3.1 and 3.2). This is achieved by taking into account the fact that the queue h can contain messages of the shape $(\mathsf{p},\mathsf{q},\lambda(u))$. We illustrate this rule by means of an example.

Example 3.6. Consider the following network (where, according to our convention, we omit the level of the constant true in the queue, since this level does not matter here):

$$\mathsf{q}!\lambda(\mathsf{nat}).\mathsf{end}[\mathsf{s}[\mathsf{p}]!\lambda(5^{\ell})] \mid \mathsf{p}?\lambda(\mathsf{bool}).\mathsf{p}?\lambda(\mathsf{nat}).\mathsf{end}[\mathsf{s}[\mathsf{q}]?\lambda(x).\mathsf{s}[\mathsf{q}]?\lambda(y)] \mid \mathsf{s} : \langle (\mathsf{p},\mathsf{q},\lambda(\mathsf{true})); \sigma \rangle$$

If $\mathsf{Lw}(\mathsf{q}) = (\ell_1, \ell_2)$ and $\ell_1 \not\sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell$, then Rule OutLoc applied to the network can give

$$\mathsf{end}[\mathbf{0}] \mid \mathsf{p}?\lambda(\mathsf{bool}).\mathsf{end}[\mathsf{s}[\mathsf{q}]?\lambda(x)] \mid \mathsf{s} : \langle (\mathsf{p},\mathsf{q},\lambda(\mathsf{true})); \sigma \rangle.$$

Assuming the same security levels, another interesting example is the application of this rule to the net-

work:

$$\mathsf{q}!\lambda(\mathsf{nat}).\mathsf{end}[\mathsf{s}[\mathsf{p}]!\lambda(5^{\ell})] \mid \mathsf{p}?\{\lambda'(\mathsf{bool}).\mathsf{p}?\lambda(\mathsf{nat}).\mathsf{end}, \lambda(\mathsf{nat}).\mathsf{end}\}[\mathsf{s}[\mathsf{q}]?\lambda'(x).\mathsf{s}[\mathsf{q}]?\lambda(y) + \mathsf{s}[\mathsf{q}]?\lambda(z)] \mid \mathsf{s} : \langle(\mathsf{p}, \mathsf{q}, \lambda'(\mathsf{true})); \sigma\rangle$$

resulting in

$$\mathsf{end}[\mathbf{0}] \mid \mathsf{p}?\lambda'(\mathsf{bool}).\mathsf{end}[\mathsf{s}[\mathsf{q}]?\lambda'(x) + \mathsf{s}[\mathsf{q}]?\lambda(z)] \mid \mathsf{s} : \langle(\mathsf{p}, \mathsf{q}, \lambda'(\mathsf{true})); \sigma\rangle.$$

- Rule InGlob defines the *global adaptation* mechanism for a *hard reading violation*, i.e., when the level associated with the value in the queue is not lower than or equal to boundary r_b (and a fortiori not lower than or equal to permission r_p). As already explained, this mechanism is based on nonce creation: a reduction is still enabled, but since the monitored process is not allowed to input the provided value, an adaptation is realised 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 $\mathsf{next}(\sigma)$ is used to obtain a nonce not occurring in σ. Observe that the store is updated to account for the newly created nonce.
- Rule OutGlob defines the *global adaptation* mechanism for a *hard writing violation*, i.e., when the level of the sent value v is not greater than or equal to boundary w_b (and a fortiori not greater than or equal to permission w_p). Observe that no writing violation may occur with nonces. As in the previous rule, in this case a reduction is enabled; adaptation is realised by adding a fresh nonce to the queue. To formalise the fact that p is responsible for the hard writing violation, by trying to "declassify" value v from its original level to the reading permission r'_p of the monitor controlling the receiver (noted q in the rule), we update its current reading permission r_p to the meet of r_p and r'_p. Hence, the reading permission of p is downgraded to that of q (or lower), accounting for the fact that p attempted to leak information to q. This is intended to counter any possible "recidivism" in p's offending behaviour, by preventing new values of level $\ell \not\sqsubseteq \mathsf{r}'_p$ to be received by p and then leaked again to q. As in the previous case, the store is updated to account for the new nonce.
- Rule Reconf defines a reconfiguration and goes hand-in-hand with Rules InGlob and OutGlob. It chooses a particular nonce (nonce_i) and it adapts the whole set of participants affected by this nonce. This choice can be guided by various criteria, such as the number of occurrences of nonces, the number of affected participants etc. The reconfiguration is realised by extracting the creator of nonce_i (using the store) and the set of participants whose processes can send nonce_i, which are the processes that contain nonce_i and all those which (transitively) may communicate with them. This set is obtained as the least fixed point of the mapping $\mathcal{A}$ of Definition 3.4. For the set of participants affected by nonce_i, a new global type is obtained via a function F which considers these participants and the store. 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. The reduction step starts the new choreography and continues the execution of the unaffected participants. To avoid orphan messages we must erase from the queue all messages with affected participants as receivers, using the mapping $h \setminus \Pi$ of Definition 3.3. In the store we erase the unique pair whose second component is nonce_i; we denote by $\sigma \setminus \mathit{nonce}_i$ the resulting store.

Note that, although the choice between local and global adaptation is deterministic, network reduction can be nondeterministic, since Rule Reconf can be applied at any point of execution. We could render our semantics more deterministic by introducing some alert threshold below which adaptation mechanisms are applied, and starting from which reconfiguration is applied instead.

Rules Init, InLoc, OutLoc and Reconf take processes from a *complete collection* $\mathcal{P}$:

Definition 3.7 (Complete Collection). A collection $\mathcal{P}$ of processes and types is *complete for a given set $\mathcal{G}$ of global types* if, for every $\mathsf{G} \in \mathcal{G}$, there are processes in $\mathcal{P}$ whose types are adequate for the monitors obtained by projecting G onto its participants and then possibly applying the input erasure of Definition 3.2 to all sub-monitors.

The existence of processes with types adequate to the monitors (obtained by projecting global types) ensures that each session initiator can reduce (Rules Init and Reconf). The other condition ensures that we can find suitable processes when Rules InLoc and OutLoc are applied.

It is interesting to notice that in the present calculus we can reduce without problems networks which would not be typable in the type system of [CCDC14] and which would produce an error in the monitored

semantics of [CCDC15]. For example, let $\mathsf{G} = \mathtt{p} \to \mathtt{q} : \mathsf{bool}.\mathtt{q} \to \mathtt{p} : \mathsf{bool}$ and $\mathsf{Lr}(\mathtt{q}) = (\top, \top)$ and $\mathsf{Lw}(\mathtt{q}) = (\bot, \bot)$. Then $\mathsf{new}(\mathsf{G}, \mathsf{L})$ can reduce to the network:

$$(\nu \mathsf{s})(\mathtt{q}!\mathsf{bool}.\mathtt{q}?\mathsf{bool}.\mathsf{end}[\mathtt{s}[\mathtt{p}]!(\mathsf{true}^{\top}).\mathtt{s}[\mathtt{p}]?(x)] \mid \mathtt{p}?\mathsf{bool}.\mathtt{p}!\mathsf{bool}.\mathsf{end}[\mathtt{s}[\mathtt{q}]?(z).\mathtt{s}[\mathtt{q}]!(\mathsf{false}^{\bot})])$$

which reduces in a number of steps to $\mathsf{end}[\mathbf{0}]$. The process representing participant $\mathtt{p}$ is not typable in the system of [CCDC14], since it first receives a secret value and then sends a public value, thus realising a "level drop". For the same reason, the monitored semantics of [CCDC15] raises an error while reducing the whole process. We give further comparisons with [CCDC14] and [CCDC15] in § 6.

Looking back at the web browser example in § 1, we could think of the dotted red line in Figure 1 as representing a hard writing violation due to the fact that the malicious plug-in tests some personal information of the user before attempting to send his cc-number, thereby making the writing level of the user monitor exceed its writing boundary.

4. Example: A Travel Agency Network

In this section, we illustrate the main features of our approach with a simple network example. We first informally describe the expected behaviour of the network, as well as the soft and hard security violations that could occur. We then show how these different violations can be formalised using reading and writing boundaries, and how the corresponding adaptation mechanisms are enforced by our semantics. Finally, we describe some of the processes that implement the network. To start with, we only mention the reading permissions of participants. Writing permissions and boundaries will be introduced later, in a gradual way.

Consider a *Travel Agency* with reading permission $\top$, which employs two sales agents $Agent_1$ and $Agent_2$ whose reading permissions are $high_1$ and $high_2$. These agents are in charge of handling travel requests coming from single clients $client_i$ and corporate clients (groups of clients) $gclient_j$, respectively. Assume there are n clients and m corporate clients, with reading permissions $priv_1, \ldots, priv_n$ and $gpriv_1, \ldots, gpriv_m$, all pairwise incomparable. Each $client_i$ is assigned two numbers: his client number $cnumber_i$ of level $priv_i$, and his $status_i \in \{1, 2, 3, 4\}$ of level low_1 (we may interpret the numbers $1, 2, 3, 4$ as typical client fidelity statuses like ivory, silver, gold and platinum). A corporate client $gclient_j$ is given similar numbers $gcnumber_j$ of level $gpriv_j$ and $gstatus_j \in \{1, 2, 3, 4\}$ of level low_2. Finally, there are two services $StatServ_1$ and $StatServ_2$, with reading permissions low_1 and low_2, which compute weekly statistics about the trips organised by the agency, for single and corporate clients respectively. These services do not have access to the private information of clients (contained in their client numbers), but only to their statuses. Both these services compute also annual statistics, where the distinction between single and corporate clients disappears, as well as their status information. These annual statistics are then transmitted to a public statistics office *StatOff*, whose reading permission is $\bot$.

Assume that $\sqcup_i priv_i = high_1$, $\sqcup_j gpriv_j = high_2$, $high_1 \sqcup high_2 = \top$, $\sqcap_i priv_i = low_1$, $\sqcap_j gpriv_j = low_2$ and $low_1 \sqcap low_2 = \bot$. Thus the lattice of security levels (which happens to coincide with that of reading permissions) may be pictured as in Figure 3.

We only describe some parts of the network behaviour, namely those that are relevant to illustrate our approach. After returning from a trip, each $client_i$ is invited to rate the destination and the service provided by the agency, by sending to $Agent_1$ a report $report_i$ of level low_i. Now $Agent_1$ forwards this report, preceded by $cnumber_i$, to the travel agency with $client_i$ in cc. She also forwards the report to $StatServ_1$, preceded this time by $status_i$ (in place of $cnumber_i$). So the agency receives a named report, while the statistics service receives an anonymised report together with a status.

A similar interaction goes on between $gclient_j$, $Agent_2$ and $StatServ_2$.

Assuming each $client_i$ and $gclient_j$ have writing permission $\bot$, and $Agent_1$ and $Agent_2$ have writing permissions low_1 and low_2, the described behaviour is in keeping with both reading and writing permissions of the involved participants.

Let us see now what could go wrong if this behaviour were slightly altered, and how our different adaptation mechanisms could be of help. Suppose that $Agent_i$ has writing boundary low_i (equal to its writing permission) and $StatServ_1$ has reading boundary $high_1$, while $StatServ_2$ has reading boundary low_2 (equal to its reading permission). The reason for restricting $StatServ_2$ more than $StatServ_1$ is that it is more damageable to unduly receive private information about a whole group of clients than about a single client.

- **Soft reading violation.** Suppose that $Agent_1$ sends to $StatServ_1$ the number $cnumber_i$ instead of the

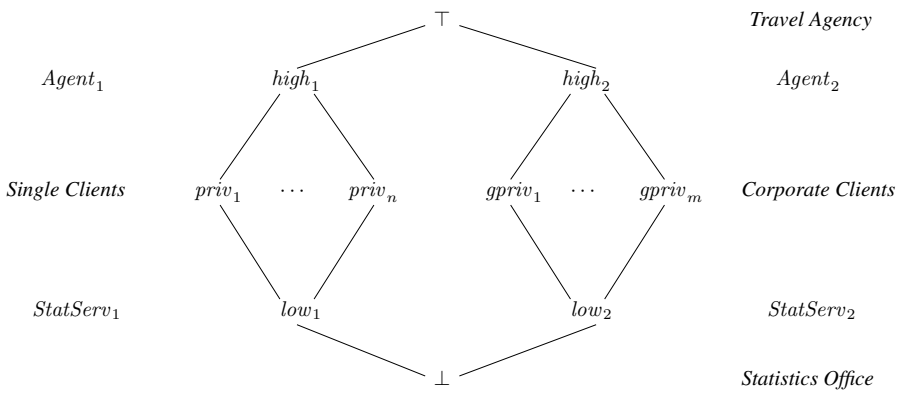

Fig. 3. Lattice of security levels for the Travel Agency example of § 4.

number $status_i$. In this case there is no writing violation for $Agent_1$ but there is a reading violation for $StatServ_1$. This is a soft reading violation since $priv_i \sqsubseteq high_1$, so it is handled by local adaptation (Rule InLoc), by cancelling $Agent_1$'s dangerous message from the queue and advancing by one step the monitor of $StatServ_1$. Indeed, $StatServ_1$ continues to work even without the missing status information, although in slightly degraded mode, since it will not be able to compute the exact number of reports for each status, but only a lower bound for it.

- **Hard reading violation.** Suppose it is now $Agent_2$ who sends $gcnumber_j$ instead of $gstatus_j$ to $StatServ_2$. Again, there is no writing violation for $Agent_2$ but there is a reading violation for $StatServ_2$. This time, however, the reading violation is hard, because $gpriv_j \not\sqsubseteq low_2$. Therefore it is handled by global adaptation (Rule InGlob), turning $Agent_2$'s dangerous message into some $nonce_k$.
- **Soft writing violation.** Suppose now that $Agent_1$, instead of sending $cnumber_i$ to the agency and $client_i$, sends them some general information $geninfo_i$ of level $priv_i$, allegedly concerning only $client_i$, but computed after testing all clients' reports. For instance, $geninfo_i$ could be: "$client_i$ is the first client from New Zealand to travel to Madagascar". In this case there is a writing violation for $Agent_1$, because the level of the tested expression is the join of all levels $priv_k$, thus the writing permission of $Agent_1$ is raised to $high_1$, while the sent information $geninfo_i$ is of level $priv_i$. On the other hand, this is a soft writing violation since the writing boundary of $Agent_1$ is low_1 and $low_1 \sqsubseteq priv_i$. Hence this violation is handled by local adaptation (Rule OutLoc).
- **Hard writing violation.** Suppose finally that $Agent_2$ owns some information $info_i$ about a single $client_i$ (this could happen, for instance, if $client_i$ also belongs to some group $gclient_j$) and, instead of sending $gcnumber_j$ to the agency and $gclient_j$, she sends them the information $info_i$ of level $priv_i$. Again, this is a writing violation since $low_2 \not\sqsubseteq priv_i$. However, in contrast to the previous case, now the writing violation is hard because the writing boundary is also low_2. Therefore, Rule OutGlob is used. Since $Agent_2$ is deemed responsible for this hard writing violation, her reading level is downgraded to $gpriv_j$. This will reduce the nuisance capacity of $Agent_2$ in further interactions, by preventing her from leaking again to the corporate client $gclient_j$ any information about the other clients. Indeed, $Agent_2$ will not be able to receive any new information from the other corporate clients, and therefore only a reconfiguration of the system (via Rule Reconf) will restore its expected functionality.

Showing only the required communications, a network producing the hard reading violation above could be:

$$\begin{aligned} N = {} & StatServ_2!report(\mathsf{string} \times \mathsf{nat}).\mathcal{M}[\mathsf{s}[Agent_2]!report(\text{"nice travel"}, 25).\cdots] \\ & \mid Agent_2?report(\mathsf{string} \times \mathsf{nat}).\mathcal{M}'[\mathsf{s}[StatServ_2]?report(x,y).P] \\ & \mid \mathsf{s} : \langle h; \sigma \rangle \mid \ldots \end{aligned}$$

where 25 is a client number of level $priv_{25}$.

By Rule Out, N reduces to the following (where we omit some terms):

$$\begin{aligned} & Agent_2?report(\mathsf{string} \times \mathsf{nat}).\mathcal{M}'[\mathsf{s}[StatServ_2]?report(x,y).P] \\ & \mid \mathsf{s} : \langle (Agent_2, StatServ_2, report(\text{"nice travel"}, 25)) \cdot h; \sigma \rangle \mid \ldots \end{aligned}$$

Then Rule InGlob must be applied, yielding:

$$\mathcal{M}'[P\{\mathit{nonce}_3/(x,y)\}] \mid \mathsf{s} : \langle h; \sigma, (\mathit{StatServ}_2, \mathit{nonce}_3)\rangle \mid \ldots$$

where $\mathit{nonce}_3 = \mathsf{next}(\sigma)$.

Consider now the case of a soft writing violation. If Agent_1 tests an expression involving more than one client:

$$\begin{array}{l}\mathit{Travel\ Agency}!\mathit{report}(\mathsf{string} \times \mathsf{nat}).\\ \quad \mathcal{M}[\mathsf{if}\ \mathsf{e}(\mathit{client}_1, \ldots, \mathit{client}_n)\ \mathsf{then}\ \mathsf{s}[\mathit{Agent}_1]!\mathit{report}(\text{"first NZ-MG"}, 1).P\ \mathsf{else}\ \ldots]\\ \mid\ \mathit{Agent}_1?\mathit{report}(\mathsf{string} \times \mathsf{nat}).\mathcal{M}'[\mathsf{s}[\mathit{Travel\ Agency}]?\mathit{report}(x,y).Q] \mid \ldots\end{array}$$

then her writing permission is raised to high_1. As a consequence the network obtained by evaluating the conditional:

$$\begin{array}{l}\mathit{Travel\ Agency}!\mathit{report}(\mathsf{string} \times \mathsf{nat}).\mathcal{M}[\mathsf{s}[\mathit{Agent}_1]!\mathit{report}(\text{"first NZ-MG"}, 1).P]\\ \mid\ \mathit{Agent}_1?\mathit{report}(\mathsf{string} \times \mathsf{nat}).\mathcal{M}'[\mathsf{s}[\mathit{Travel\ Agency}]?\mathit{report}(x,y).Q] \mid \ldots\end{array}$$

will incur a soft writing violation since the information sent in the report has level priv_i (remember that this information will also be sent to client_i). Therefore the next rule to be applied is OutLoc, yielding:

$$\mathcal{M}[P] \mid \widehat{\mathcal{M}}[Q'\{\mathsf{s}[\mathit{Travel\ Agency}]/y\}] \mid \ldots$$

where $\widehat{\mathcal{M}} = <\mathcal{M}', \emptyset\!\!\rhd \backslash ?(\mathit{Agent}_1, \mathit{report})$ and $\mathsf{T} \propto \widehat{\mathcal{M}}$ and $(Q', \mathsf{T}) \in \mathcal{P}$.

Having presented and illustrated our framework, we now move on to establish its properties.

5. Well-Typed Networks: Main Properties

In this section, we extend typability from processes to networks and present the technical results of the paper. First, we prove subject reduction and progress theorems for networks (Theorems 5.20 and 5.21). In the proofs we closely follow the strategy of [CDCV15]. The main novelties of our calculus—the local and global mechanisms for adaptation—are the key technical difficulties in proving that well-typed, monitored processes always behave in a type-safe way. Then, we show that reduction of well-typed networks always respects reading/writing permissions and boundaries (Theorem 5.22). Thus, besides respecting the choreographies described by global types, well-typed networks do not contain insecure accesses to data nor insecure information flows. The combined effect of communication and security guarantees for networks formally realises the integration of concerns that was one of the main motivations of our work.

The first step in our development consists in extending types to deal with networks. Next we introduce typing rules for networks, which associate types with session channels.

5.1. Preliminaries

The main issue in typing networks is relating the type of a monitored process with the type of its message queue. Given a session s, both these types are defined relatively to each session channel $\mathsf{s}[\mathsf{p}]$; to this purpose, we introduce *generalised types* for session channels. On the one hand, to abstractly describe the message queue of s we associate with each channel $\mathsf{s}[\mathsf{p}]$ a *queue type*: this is built by recording, for each message $(\mathsf{p}, \mathsf{q}, \lambda(u))$ with sender p in the queue, the message type $\mathsf{q}!\lambda(S)$, where S is the sort of u when u is a value, and an arbitrary sort when u is a nonce. On the other hand, the type of a monitored process $\mathcal{M}[P]$ simply associates the monitor $\mathcal{M}$ with the session channel $\mathsf{s}[\mathsf{p}]$ owned by P. To sum up, message types specify the destination and the sort of a message, queue types are sequences of message types, and generalised types are either a queue type (for queues), a monitor (for monitored processes), or a pair made of a queue type and a monitor. This pair is used to define *session typings*, in which the association between the queue types and the monitors of all participants of the same session is finally realised via their *composition*.

We now proceed to formalise these intuitions.

Definition 5.1 (Message Types, Queue Types and Generalised Types). *Message types*, *queue types* and *generalised types* are defined by:

$$\epsilon \upharpoonright \mathtt{q} = \epsilon \qquad \langle \mathcal{H}, \mathcal{M} \rangle \upharpoonright \mathtt{q} = \mathcal{H} \upharpoonright \mathtt{q} \,.\, \mathcal{M} \upharpoonright \mathtt{q} \qquad (\mathtt{p}!\lambda(S)\,;\,\mathcal{H}) \upharpoonright \mathtt{q} = \begin{cases} !\lambda(S).(\mathcal{H} \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p} \\ \mathcal{H} \upharpoonright \mathtt{q} & \text{otherwise} \end{cases}$$

$$\mathtt{p}?\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I} \upharpoonright \mathtt{q} = \begin{cases} ?\{\lambda_i(S_i).(\mathcal{M}_i \upharpoonright \mathtt{q})\}_{i\in I} & \text{if } \mathtt{q} = \mathtt{p} \\ \mathcal{M}_{i_0} \upharpoonright \mathtt{q} & \text{where } i_0 \in I, \text{ if } \mathtt{q} \neq \mathtt{p} \text{ and } \mathcal{M}_i \upharpoonright \mathtt{q} = \mathcal{M}_j \upharpoonright \mathtt{q} \ \forall i, j \in I \end{cases}$$

$$\mathtt{p}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I} \upharpoonright \mathtt{q} = \begin{cases} !\{\lambda_i(S_i).(\mathcal{M}_i \upharpoonright \mathtt{q})\}_{i\in I} & \text{if } \mathtt{q} = \mathtt{p} \\ \mathcal{M}_{i_0} \upharpoonright \mathtt{q} & \text{where } i_0 \in I, \text{ if } \mathtt{q} \neq \mathtt{p} \text{ and } \mathcal{M}_i \upharpoonright \mathtt{q} = \mathcal{M}_j \upharpoonright \mathtt{q} \ \forall i, j \in I \end{cases}$$

$$\mathsf{end} \upharpoonright \mathtt{q} = \epsilon \qquad \mathbf{t} \upharpoonright \mathtt{q} = \mathbf{t} \qquad (\mu\mathbf{t}.\tau) \upharpoonright \mathtt{q} = \begin{cases} \mu\mathbf{t}.\tau \upharpoonright \mathtt{q} & \text{if } \mathtt{q} \text{ occurs in } \tau \\ \epsilon & \text{otherwise} \end{cases}$$

Table 7. Projection of generalised types onto participants.

Message Types	m	::=	$\mathtt{q}!\lambda(S)$
Queue Types	$\mathcal{H}$	::=	$\epsilon \mid \mathcal{H}\,;\,\mathsf{m}$
Generalised Types	τ	::=	$\mathcal{M} \mid \mathcal{H} \mid \langle \mathcal{H}, \mathcal{M} \rangle$

where in queue types, *concatenation* " ;" is associative and ϵ is the type of the empty sequence of messages, such that $\epsilon\,;\,\mathcal{H} = \mathcal{H}\,;\,\epsilon = \mathcal{H}$.

Example 5.2. An example of a generalised type is

$$\tau = \langle \mathtt{p}!\lambda_1(\mathsf{bool}); \mathtt{q}!\lambda_2(\mathsf{nat})\,,\ \mu\mathbf{t}.\mathtt{q}?\{\lambda_3(\mathsf{bool}).\mathbf{t}, \lambda_4(\mathsf{bool}).\mathsf{end}\}\rangle$$

This type says that a participant has sent a boolean to $\mathtt{p}$ with label λ_1 and a natural to $\mathtt{q}$ with label λ_2. This participant will receive from $\mathtt{q}$ a boolean with either label λ_3 or label λ_4. In the first case he will continue receiving booleans from $\mathtt{q}$, in the second case he will stop.

The typing judgements for networks are of the shape

$$\vdash_\Sigma N \rhd \Delta$$

where Σ is a set of session names (the buffer names which occur free in the network) and Δ is a *session typing*, defined next.

Definition 5.3 (Session Typings). *Session typings* Δ are maps from session channels to generalised types:

$$\Delta ::= \emptyset \mid \Delta, \mathsf{s}[\mathtt{p}] : \tau$$

Session typings are subject to the same conventions as environments Γ. In particular, a session typing Δ_1, Δ_2 is defined only if the domains of Δ_1 and Δ_2 are disjoint.

To ensure type safety it is essential that the communications are performed in a consistent way, i.e., that appropriately typed values are exchanged in the prescribed order. The formal definition of consistent session typings (Definition 5.5) relies on the *projection of generalised types* and on *duality*, given in Tables 7 and 8, respectively.

Given a session channel $\mathsf{s}[\mathtt{p}]$, the projection of its generalised type τ onto participant $\mathtt{q}$ represents the sequence of communications offered by $\mathtt{p}$ to $\mathtt{q}$. It uses the projection of queue types and of monitors, denoted $\mathcal{H} \upharpoonright \mathtt{q}$ and $\mathcal{M} \upharpoonright \mathtt{q}$, respectively. The projection of a generalised type $\tau = \langle \mathcal{H}, \mathcal{M} \rangle$ onto $\mathtt{q}$ (denoted $\tau \upharpoonright \mathtt{q}$) is the concatenation of the projections of $\mathcal{H}$ and $\mathcal{M}$. Here, $\mathcal{H} \upharpoonright \mathtt{q}$ represents the sequence of messages already sent by $\mathtt{p}$ to $\mathtt{q}$ and $\mathcal{M} \upharpoonright \mathtt{q}$ the further communications between $\mathtt{p}$ and $\mathtt{q}$. Formally, the projections of generalised types, ranged over by $\Theta, \Theta', \ldots$, are given by the syntax:

$$\Theta ::= \Phi \mid \Psi \mid \Phi.\Theta \qquad \text{generalised type projections}$$

$$\Phi ::= \epsilon \mid !\lambda(S).\Phi \qquad \text{queue type projections}$$

$$\Psi ::= \epsilon \mid ?\{\lambda_i(S_i).\Psi_i\}_{i\in I} \mid !\{\lambda_i(S_i).\Psi_i\}_{i\in I} \mid \mu\mathbf{t}.\Psi \mid \mathbf{t} \qquad \text{monitor projections}$$

We assume $\epsilon.\Theta = \Theta.\epsilon = \Theta$, since ϵ represents no communication.

$$\Phi \bowtie \Psi_j \ \&\ j \in I \text{ imply } !\lambda_j(S_j).\Phi \bowtie ?\{\lambda_i(S_i).\Psi_i\}_{i\in I}$$
$$\forall i \in I\ \Psi_i \bowtie \Psi_i' \text{ imply } !\{\lambda_i(S_i).\Psi_i\}_{i\in I} \bowtie ?\{\lambda_i(S_i).\Psi_i'\}_{i\in I}$$
$$\Phi_1 \bowtie \Phi_2 \text{ and } \Psi_1 \bowtie \Psi_2 \text{ imply } \Phi_1.\Psi_1 \bowtie \Phi_2.\Psi_2$$
$$\Phi_1 \bowtie \Phi_2 \text{ and } \Phi_1' \bowtie \Phi_2' \text{ imply } \Phi_1.\Phi_1' \bowtie \Phi_2.\Phi_2'$$
$$\epsilon \bowtie \epsilon \qquad \mathbf{t} \bowtie \mathbf{t} \qquad \Psi \bowtie \Psi' \text{ implies } \mu\mathbf{t}.\Psi \bowtie \mu\mathbf{t}.\Psi'$$

Table 8. Duality between projections of generalised types onto participants.

The relation of duality expresses the fact that two participants $\mathtt{p}$ and $\mathtt{q}$ have matching communication behaviours. We say that two projections Θ and Θ' are *dual* whenever $\Theta \bowtie \Theta'$ holds, where $\bowtie$ is the symmetric closure of the partial operator in Table 8, defined only on projections of generalised types. The duality of projections exploits the duality between messages in queues and inputs in monitors (Line 1) and the duality between outputs and inputs in monitors (Line 2). The relation $\Theta \bowtie \Theta'$ makes sense when Θ and Θ' are mutual projections, namely one of them is the projection of the generalised type of $\mathtt{s}[\mathtt{p}]$ on $\mathtt{q}$ and the other is the projection of the generalised type of $\mathtt{s}[\mathtt{q}]$ on $\mathtt{p}$.

Example 5.4. The projection of the generalised type τ of Example 5.2 on participant $\mathtt{q}$ is

$$!\lambda_2(\mathsf{nat}).\mu\mathbf{t}.?\{\lambda_3(\mathsf{bool}).\mathbf{t}, \lambda_4(\mathsf{bool})\}$$

This projection is dual to the generalised type projection $?\lambda_2(\mathsf{nat}).\mu\mathbf{t}.!\{\lambda_3(\mathsf{bool}).\mathbf{t}, \lambda_4(\mathsf{bool})\}$.

We now define consistency of session typings:

Definition 5.5 (Consistent Typing). A session typing Δ is *consistent for the session* $\mathtt{s}$, notation $\mathsf{cons}(\Delta, \mathtt{s})$, if

$$\mathtt{s}[\mathtt{p}] : \tau \in \Delta \text{ and } \mathtt{s}[\mathtt{q}] : \tau' \in \Delta \text{ with } \mathtt{p} \neq \mathtt{q} \text{ imply } \tau \upharpoonright \mathtt{q} \bowtie \tau' \upharpoonright \mathtt{p}\,.$$

A session typing is *consistent* if it is consistent for all sessions which occur in it.

Example 5.6. The session typing

$$\Delta = \{\mathtt{s}[\mathtt{p}] : \mathtt{q}?\lambda(\mathsf{bool}).\mathtt{q}?\lambda'(\mathsf{nat}).\mathsf{end}, \mathtt{s}[\mathtt{q}] : \langle \mathtt{p}!\lambda(\mathsf{bool}), \mathtt{p}!\lambda'(\mathsf{nat}).\mathsf{end}\rangle\}$$

is consistent, while the session typings

$$\Delta' = \{\mathtt{s}[\mathtt{p}] : \mathtt{q}?\lambda(\mathsf{bool}).\mathtt{q}?\lambda'(\mathsf{bool}).\mathsf{end}, \mathtt{s}[\mathtt{q}] : \langle \mathtt{p}!\lambda(\mathsf{bool}), \mathtt{p}!\lambda'(\mathsf{nat}).\mathsf{end}\rangle\}$$

and

$$\Delta'' = \{\mathtt{s}[\mathtt{p}] : \mathtt{q}?\lambda(\mathsf{bool}).\mathtt{q}?\lambda'(\mathsf{nat}).\mathsf{end}, \mathtt{s}[\mathtt{q}] : \mathtt{p}!\lambda(\mathsf{bool})\}$$

are not. Example 5.17 gives networks typed by Δ and Δ''.

It is easy to check that projections of the same global type onto two different participants are always dual.

Proposition 5.7. Let G be a global type and $\mathtt{p} \neq \mathtt{q}$. Then $(\mathsf{G}\upharpoonright \mathtt{p})\upharpoonright \mathtt{q} \bowtie (\mathsf{G}\upharpoonright \mathtt{q})\upharpoonright \mathtt{p}$.

This proposition assures that session typings obtained by projecting global types are consistent.

We now move on to define the typing rules for networks. To this end, we first define partial operators for concatenation and composition of session typings. The concatenation of session typings is naturally inherited from the concatenation of queue types (Definition 5.1). The composition of two generalised types is defined only when one of them is a queue type and the other is a monitor. Its extension to session typings is as expected. While concatenation is required for typing named buffers, composition is used for typing parallel composition within networks.

Definition 5.8 (Partial Operations on Typings). Let Δ and Δ' be session typings (cf. Definition 5.3).

- The extension of concatenation ; to session typings is defined by:

$$\begin{aligned}\Delta\,;\,\Delta' \;=\; &\{\mathtt{s}[\mathtt{p}] : \mathcal{H}\,;\,\mathcal{H}' \mid \mathtt{s}[\mathtt{p}] : \mathcal{H} \in \Delta \ \&\ \mathtt{s}[\mathtt{p}] : \mathcal{H}' \in \Delta'\} \cup \\ &\{\mathtt{s}[\mathtt{p}] : \mathcal{H} \mid \mathtt{s}[\mathtt{p}] : \mathcal{H} \in \Delta \cup \Delta' \ \&\ \mathtt{s}[\mathtt{p}] \notin dom(\Delta) \cap dom(\Delta')\}\end{aligned}$$

$$\frac{}{\vdash_\emptyset \mathsf{new}(\mathsf{G},\mathsf{L}) \rhd \emptyset}\ \textsc{New} \qquad \frac{}{\vdash_\emptyset \mathsf{end}[P] \rhd \emptyset}\ \mathsf{endP} \qquad \frac{\mathcal{M} \neq \mathsf{end} \quad \mathsf{T} \propto \mathcal{M} \quad \vdash P \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}}{\vdash_\emptyset \mathcal{M}[P] \rhd \{\mathsf{s}[\mathsf{p}] : \mathcal{M}\}}\ \mathrm{MP}$$

$$\frac{}{\vdash_{\{\mathsf{s}\}} \mathsf{s} : \langle \emptyset; \sigma\rangle \rhd \emptyset}\ \textsc{QInit} \qquad \frac{\vdash_{\{\mathsf{s}\}} \mathsf{s} : \langle h; \sigma\rangle \rhd \Delta \qquad \vdash u : S}{\vdash_{\{\mathsf{s}\}} \mathsf{s} : \langle h \cdot (\mathsf{p},\mathsf{q},\lambda(u)); \sigma\rangle \rhd \Delta\,;\, \{\mathsf{s}[\mathsf{p}] : \mathsf{q}!\lambda(S)\}}\ \textsc{QSendV}$$

$$\frac{\vdash_{\Sigma_1} N_1 \rhd \Delta_1 \quad \vdash_{\Sigma_2} N_2 \rhd \Delta_2 \quad \Sigma_1 \cap \Sigma_2 = \emptyset}{\vdash_{\Sigma_1 \cup \Sigma_2} N_1 \mid N_2 \rhd \Delta_1 * \Delta_2}\ \textsc{NPar}$$

$$\frac{\vdash_\Sigma N \rhd \Delta \qquad \Delta \approx \Delta'}{\vdash_\Sigma N \rhd \Delta'}\ \textsc{SEquiv} \qquad \frac{\vdash_\Sigma N \rhd \Delta \qquad \mathsf{cons}(\Delta, \mathsf{s})}{\vdash_{\Sigma\setminus\{\mathsf{s}\}} (\nu\,\mathsf{s})N \rhd \Delta \setminus \mathsf{s}}\ \textsc{Res}$$

Table 9. Typing rules for networks.

- The *composition* $*$ of queue types and monitors is defined by:
$$\mathcal{H} * \mathcal{M} = \mathcal{M} * \mathcal{H} = \langle \mathcal{H}, \mathcal{M}\rangle$$
- The extension of composition $*$ to session typings is defined by:
$$\begin{array}{rcl} \Delta * \Delta' & = & \{\mathsf{s}[\mathsf{p}] : \tau * \tau' \mid \mathsf{s}[\mathsf{p}] : \tau \in \Delta \ \&\ \mathsf{s}[\mathsf{p}] : \tau' \in \Delta'\} \cup \\ & & \{\mathsf{s}[\mathsf{p}] : \tau \mid \mathsf{s}[\mathsf{p}] : \tau \in \Delta \cup \Delta' \ \&\ \mathsf{s}[\mathsf{p}] \notin dom(\Delta) \cap dom(\Delta')\} \end{array}$$

Notice that both ; and $*$ are partial operators on session typings, since they can be undefined when applied to arbitrary generalised types.

Example 5.9. Let Δ, Δ' and Δ'' be as in Example 5.6. Composing Δ'' with $\{\mathsf{s}[\mathsf{q}] : \mathsf{p}!\lambda'(\mathsf{nat}).\mathsf{end}\}$ we get Δ, while there is no way of obtaining a consistent typing from Δ' by means of composition.

The last ingredient needed for typing networks is a notion of equivalence on queue types. Intuitively, the equivalence captures possible reorderings of messages associated with different pairs of participants. Like concatenation and composition, this equivalence extends to session typings.

Definition 5.10 (Equivalence on Queue Types and Session Typings).
We write $\approx$ to denote the equivalence relation on queue types induced by the following rule:

$$\mathcal{H}; \mathsf{q}!\lambda(S); \mathsf{q}'!\lambda'(S'); \mathcal{H}' \approx \mathcal{H}; \mathsf{q}'!\lambda'(S'); \mathsf{q}!\lambda(S); \mathcal{H}' \qquad \text{if } \mathsf{q} \neq \mathsf{q}'$$

This equivalence relation on queue types extends to generalised types by:

$$\mathcal{M} \approx \mathcal{M} \qquad \mathcal{H} \approx \mathcal{H}' \text{ implies } \langle \mathcal{H}, \mathcal{M}\rangle \approx \langle \mathcal{H}', \mathcal{M}\rangle$$

Two session typings Δ and Δ' are said to be equivalent (notation $\Delta \approx \Delta'$) if $\mathsf{s}[\mathsf{p}] : \tau \in \Delta$ implies $\mathsf{s}[\mathsf{p}] : \tau' \in \Delta'$ with $\tau \approx \tau'$ and vice versa.

We are now ready to give the typing rules for networks, which are collected in Table 9. A session initiator is typed with the empty set of session names and with the empty session typing (Rule New). To type a monitored process, we distinguish two cases. If the monitor is end, then the session typing is empty for any P (Rule endP). Otherwise, the channel owned by the process is associated with the monitor, provided that the type of the process (Table 3) is adequate for the monitor, according to Definition 2.5 (Rule MP). Notice that this typing rule applies to closed processes. For example, we get

$$\vdash_\emptyset \mathsf{p}!\lambda(\mathsf{nat}).\mathsf{end}[\mathsf{s}[\mathsf{q}]!\lambda(5)] \rhd \{\mathsf{s}[\mathsf{q}] : \mathsf{p}!\lambda(\mathsf{nat}).\mathsf{end}\} \tag{1}$$

since $\vdash \mathsf{s}[\mathsf{q}]!\lambda(5) \rhd \mathsf{s}[\mathsf{q}] : !\lambda(\mathsf{nat}).\mathsf{end}$ and $!\lambda(\mathsf{nat}).\mathsf{end} \propto \mathsf{p}!\lambda(\mathsf{nat}).\mathsf{end}$. Instead the network $\mathsf{p}!\lambda(\mathsf{bool}).\mathsf{end}[\mathsf{s}[\mathsf{q}]!\lambda(5)]$ is not typable, since $!\lambda(\mathsf{nat}).\mathsf{end} \not\propto \mathsf{p}!\lambda(\mathsf{bool}).\mathsf{end}$.

The next two rules (QInit and QSendV) type named buffers. The store is transparent for the typing. In these rules the turnstile is decorated with the name of the buffer. An empty buffer $\langle \emptyset; \sigma\rangle$ is typed with

the empty session typing (Rule QInit) for all σ. Rule QSendV uses the extension of " ;" to session typings given in Definition 5.8. When a new message $(\mathsf{p},\mathsf{q},\lambda(u))$ is added to the queue $\mathsf{s} : \langle h;\sigma\rangle$, Rule QSendV appends the message type $\mathsf{q}!\lambda(S)$ to the queue type of $\mathsf{s} : \langle h;\sigma\rangle$, where S is the sort of u. For example:

$$\vdash_{\{\mathsf{s}\}} \mathsf{s} : \langle(\mathsf{q},\mathsf{p},\lambda(\mathsf{true}));\sigma\rangle \rhd \{\mathsf{s}[\mathsf{q}] : \mathsf{p}!\lambda(\mathsf{bool}).\mathsf{end}\}. \tag{2}$$

The typing rule for parallel composition of networks (Rule NPar) prescribes that no buffer name occurs more than once (condition $\Sigma_1 \cap \Sigma_2 = \emptyset$). The session typing for the resulting network is obtained by composing the typings of the components, as specified in Definition 5.8. Note that two monitored processes $\mathcal{M}[P]$ and $\mathcal{M}'[Q]$ can be composed in parallel if and only if their session typings $\Delta = \{\mathsf{s}[\mathsf{p}] : \mathcal{M}\}$ and $\Delta' = \{\mathsf{s}'[\mathsf{q}] : \mathcal{M}'\}$ are such that $\mathsf{s} \neq \mathsf{s}'$ or $\mathsf{p} \neq \mathsf{q}$; the composition $\Delta * \Delta'$ is undefined otherwise.

As a simple example, if $\vdash_\Sigma N \rhd \Delta$ then we get $\vdash_\Sigma \mathsf{end}[P] \mid N \rhd \emptyset * \Delta$ (by rules endP and NPar) and $\emptyset * \Delta = \Delta$; this fits with the structural equivalence $\mathsf{end}[P] \mid N \equiv N$.

As another example, consider the parallel composition of the above monitored process $\mathsf{p}!\lambda(\mathsf{nat}).\mathsf{end}[\mathsf{s}[\mathsf{q}]!\lambda(5)]$ with the named buffer $\langle(\mathsf{q},\mathsf{p},\lambda(\mathsf{true}));\sigma\rangle$. Applying Rule NPar to the premises (1) and (2), we get:

$$\vdash_{\{\mathsf{s}\}} \mathsf{p}!\lambda(\mathsf{nat}).\mathsf{end}[\mathsf{s}[\mathsf{q}]!\lambda(5)] \mid \mathsf{s} : \langle(\mathsf{q},\mathsf{p},\lambda(\mathsf{true}));\sigma\rangle \rhd \{\mathsf{s}[\mathsf{q}] : \langle \mathsf{p}!\lambda(\mathsf{bool}).\mathsf{end}, \mathsf{p}!\lambda(\mathsf{nat}).\mathsf{end}\rangle\}.$$

Finally, Rule SEquiv defines typing modulo the equivalence $\approx$, and Rule Res requires the session typing to be consistent for the session s in order to type the restriction on s.

5.2. Main Properties

We are now ready to state and prove the main properties of our typed framework. As usual we may establish an inversion lemma for networks, by induction on the derivations.

Lemma 5.11 (Inversion Lemma). Given the typing rules in Table 9, we have:

1. If $\vdash_\Sigma \mathsf{new}(\mathsf{G},\mathsf{L}) \rhd \Delta$, then $\Sigma = \Delta = \emptyset$.
2. If $\vdash_\Sigma \mathsf{end}[P] \rhd \Delta$, then $\Sigma = \Delta = \emptyset$.
3. If $\vdash_\Sigma \mathcal{M}[P] \rhd \Delta$ and $\mathcal{M} \neq \mathsf{end}$, then $\Sigma = \emptyset$ and $\Delta = \{\mathsf{s}[\mathsf{p}] : \mathcal{M}\}$ and $\vdash P \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}$ and $\mathsf{T} \propto \mathcal{M}$.
4. If $\vdash_\Sigma \mathsf{s} : \langle\emptyset;\sigma\rangle \rhd \Delta$, then $\Sigma = \{\mathsf{s}\}$ and $\Delta = \emptyset$.
5. If $\vdash_\Sigma \mathsf{s} : \langle h \cdot (\mathsf{p},\mathsf{q},\lambda(u));\sigma\rangle \rhd \Delta$, then $\Sigma = \{\mathsf{s}\}$ and $\Delta \approx \Delta'$; $\{\mathsf{s}[\mathsf{p}] : \mathsf{q}!\lambda(S)\}$ and $\vdash_{\{\mathsf{s}\}} \mathsf{s} : \langle h;\sigma\rangle \rhd \Delta'$ and $\vdash u : S$.
6. If $\vdash_\Sigma N_1 \mid N_2 \rhd \Delta$, then $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\Delta = \Delta_1 * \Delta_2$ and $\vdash_{\Sigma_1} N_1 \rhd \Delta_1$ and $\vdash_{\Sigma_2} N_2 \rhd \Delta_2$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$.
7. If $\vdash_\Sigma (\nu\,\mathsf{s})N \rhd \Delta$, then $\Sigma = \Sigma' \setminus \{\mathsf{s}\}$ and $\Delta = \Delta' \setminus \mathsf{s}$ and $\vdash_{\Sigma'} N \rhd \Delta'$ and $\mathsf{cons}(\Delta',\mathsf{s})$.

We also need to take into account how the typing depends on the first message in the queue. This is the task of the next lemma, whose proof follows immediately from the typing rules for queues.

Lemma 5.12. Suppose $\vdash_\Sigma \mathsf{s} : \langle(\mathsf{p},\mathsf{q},\lambda(u)) \cdot h;\sigma\rangle \rhd \Delta$. Then $\Sigma = \{\mathsf{s}\}$ and $\Delta \approx \{\mathsf{s}[\mathsf{p}] : \mathsf{q}!\lambda(S)\}\,;\Delta'$ and $\vdash_{\{\mathsf{s}\}} \mathsf{s} : \langle h;\sigma\rangle \rhd \Delta'$ and $\vdash u : S$.

The LTS for monitors is useful to reveal a monitor's shape, as detailed in the next lemma, whose proof follows by a straightforward case analysis.

Lemma 5.13. Let $\mathcal{M}$ be a monitor as in Definition 2.2.

1. If $\mathcal{M} \xrightarrow{\mathsf{p}?\lambda} \mathcal{M}'$, then $\mathcal{M} = \mathsf{p}?\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}$ and $\lambda = \lambda_j$ and $\mathcal{M}' = \mathcal{M}_j$ for some $j \in I$.
2. If $\mathcal{M} \xrightarrow{\mathsf{q}!\lambda} \mathcal{M}'$, then $\mathcal{M} = \mathsf{q}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}$ and $\lambda = \lambda_j$ and $\mathcal{M}' = \mathcal{M}_j$ for some $j \in I$.

The next lemma relates the actions of a process (as given by the LTS for processes) with its type.

Lemma 5.14. Let P be a process as in Definition 2.3.

1. If $P \xrightarrow{\mathsf{s}[\mathsf{p}]?\lambda(u)} P'$ and $\vdash P \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}$, then either $\mathsf{T} = ?\lambda(S).\mathsf{T}'$ or $\mathsf{T} = ?\lambda(S).\mathsf{T}' \wedge \mathsf{T}''$, and $\vdash P' \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}'$ and $\vdash u : S$.

$$\emptyset \implies \{\mathsf{s}[\mathsf{p}] : \mathsf{end}\} \qquad \emptyset \implies \{\mathsf{s}[\mathsf{p}] : \epsilon\}$$
$$\{\mathsf{s}[\mathsf{p}] : \langle \mathcal{H}, \mathsf{end}\rangle\} \implies \{\mathsf{s}[\mathsf{p}] : \mathcal{H}\} \qquad \{\mathsf{s}[\mathsf{p}] : \epsilon\} \implies \emptyset$$
$$\{\mathsf{s}[\mathsf{p}] : \langle \mathcal{H}, \mathsf{q}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}\rangle\} \implies \{\mathsf{s}[\mathsf{p}] : \langle \mathcal{H}, \mathcal{M}_j\rangle\}\ j \in I \qquad \{\mathsf{s}[\mathsf{p}] : \langle \mathcal{H}, \mathcal{M}\rangle\} \implies \{\mathsf{s}[\mathsf{p}] : \langle \mathcal{H}, <\mathcal{M}, \Lambda> \backslash ?(\mathsf{q}, \lambda)\rangle\}$$
$$\{\mathsf{s}[\mathsf{p}] : \langle \mathcal{H}, \mathsf{q}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}\rangle\} \implies \{\mathsf{s}[\mathsf{p}] : \langle \mathcal{H}\,;\, \mathsf{q}!\lambda_j(S_j), \mathcal{M}_j\rangle\}\ j \in I$$
$$\{\mathsf{s}[\mathsf{p}] : \mathsf{q}!\lambda_j(S_j)\,;\, \mathcal{H},\ \mathsf{s}[\mathsf{q}] : \langle \mathcal{H}', \mathsf{p}?\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}\rangle\} \implies \{\mathsf{s}[\mathsf{p}] : \mathcal{H},\ \mathsf{s}[\mathsf{q}] : \langle \mathcal{H}', \mathcal{M}_j\rangle\}\ j \in I$$
$$\Delta_1\,;\,\Delta \implies \Delta_2\,;\,\Delta \text{ if } \Delta_1 \implies \Delta_2 \qquad \Delta * \Delta_1 \implies \Delta * \Delta_2 \text{ if } \Delta_1 \implies \Delta_2$$

Table 10. Reduction of session typings.

2. If $P \xrightarrow{\mathsf{s}[\mathsf{p}]!\lambda(u)} P'$ and $\vdash P \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}$, then either $\mathsf{T} = !\lambda(S).\mathsf{T}'$ or $\mathsf{T} = !\lambda(S).\mathsf{T}' \vee \mathsf{T}''$, and $\vdash P' \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}'$ and $\vdash u : S$.

Proof. The proof is by induction on the LTS of processes (Table 5). We show only (1), as the proof for (2) is similar. If $P \xrightarrow{\mathsf{s}[\mathsf{p}]?\lambda(u)} P'$, then either $P = \mathsf{s}[\mathsf{p}]?\lambda(x).P_0$ and $P' = P_0\{u/x\}$ or $P = P_1 + P_2$ and $P_i \xrightarrow{\mathsf{s}[\mathsf{p}]?\lambda(u)} P'$ for $i = 1$ or $i = 2$. In the first case P must be typed by Rule RCV and the thesis follows immediately. In the second case P must be typed by Rule CHOICE. Then $\mathsf{T} = \mathsf{T}_1 \wedge \mathsf{T}_2$ and $\vdash P_i \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}_i$ for $i = 1, 2$. By induction either $\mathsf{T}_i = ?\lambda(S).\mathsf{T}'$ or $\mathsf{T}_i = ?\lambda(S).\mathsf{T}' \wedge \mathsf{T}'_i$ for $i = 1$ or $i = 2$, and $\vdash P' \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}'$ and $\vdash u : S$. This concludes the proof. □

Our final lemma relates monitors with processes whose types are adequate for the monitors after a communication action.

Lemma 5.15. Let $\mathcal{M}$ be a monitor.

1. If $\mathcal{M} \xrightarrow{\mathsf{q}?\lambda} \mathcal{M}'$ and $P \xrightarrow{\mathsf{s}[\mathsf{p}]?\lambda(u)} P'$ and $\vdash P \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}$ and $\mathsf{T} \propto \mathcal{M}$, then $\vdash P' \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}'$ and $\mathsf{T}' \propto \mathcal{M}'$.
2. If $\mathcal{M} \xrightarrow{\mathsf{q}!\lambda} \mathcal{M}'$ and $P \xrightarrow{\mathsf{s}[\mathsf{p}]!\lambda(u)} P'$ and $\vdash P \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}$ and $\mathsf{T} \propto \mathcal{M}$, then $\vdash P' \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}'$ and $\mathsf{T}' \propto \mathcal{M}'$.

Proof. Easy consequence of Lemmas 5.13 and 5.14. □

Session types are not preserved under network reduction: this is expected, for they evolve according to the actions performed by the corresponding participants. This is formalised by the reduction rules given in Table 10, where queue types are considered modulo the equivalence $\approx$ given in Definition 5.10. The rules in the first line allow us to create monitors offering no communications and empty queue types. The rules in the second line allow us to discard types carrying no information. The rules in the third line deal with the local adaptation of senders and receivers in Rules InLoc and OutLoc (cf. Table 6). The rule in the fourth line represents the addition of a message to a queue, whereas the rule in the fifth line stands for the reading of a message from a queue. In both cases the content of the message can be a nonce, so these rules deal with Rules In, Out, InGlob and OutGlob. The last line closes reduction up to concatenation and composition. No rule is needed for reduction via Rule Reconf, since both session typings of the left- and right-hand sides are the empty set.

Notice that a generalised type of the form $\langle \mathcal{H}, \mathsf{end}\rangle$ will never be derived for a channel, but for example by reducing $\{\mathsf{s}[\mathsf{p}] : \langle \mathcal{H}, \mathsf{q}!\lambda(S).\mathsf{end}\rangle\}$ we get $\{\mathsf{s}[\mathsf{p}] : \langle \mathcal{H}\,;\, \mathsf{q}!\lambda(S), \mathsf{end}\rangle\}$ and then $\{\mathsf{s}[\mathsf{p}] : \mathcal{H}\,;\, \mathsf{q}!\lambda(S)\}$. This also shows that a single step in the reduction of processes may correspond to more than one step in the reduction of session typings.

The next lemma shows that typings for networks are invariant under structural equivalence, as expected.

Lemma 5.16. If $\vdash_\Sigma N \rhd \Delta$ and $N \equiv N'$, then $\vdash_\Sigma N' \rhd \Delta$.

Proof. The proof is by induction on the definition of structural equivalence, observing that $\vdash_\emptyset \mathsf{end}[P] \rhd \emptyset$ and using typing rule SEquiv. □

A crucial observation is that not all the left-hand sides of the reduction rules for networks are typed by consistent session typings.

Example 5.17. Consider the following typing judgment:

$$\vdash_{\{\mathsf{s}\}} \mathcal{M}[\mathsf{s}[\mathsf{p}]?\lambda(x).\mathsf{s}[\mathsf{p}]?\lambda'(y).\mathbf{0}] \mid \mathsf{s} : (\mathsf{q}, \mathsf{p}, \lambda(\mathsf{true})) \rhd \Delta''$$

where $\mathcal{M} = \mathsf{q}?\lambda(\mathsf{bool}).\mathsf{q}?\lambda'(\mathsf{nat}).\mathsf{end}$ and Δ'' is as defined in Example 5.6. Observe that

$$\mathcal{M}[\mathsf{s}[\mathsf{p}]?\lambda(x).\mathsf{s}[\mathsf{p}]?\lambda'(y).\mathbf{0}] \mid \mathsf{s} : (\mathsf{q}, \mathsf{p}, \lambda(\mathsf{true}))$$

matches the left-hand side of the Rule In and Δ'' is not consistent. The network obtained by putting this network in parallel with the monitored process $\mathsf{p}!\lambda'(\mathsf{nat}).\mathsf{end}[\mathsf{s}[\mathsf{q}]!\lambda'(7).\mathbf{0}]$ has the consistent session typing Δ defined in Example 5.6.

It is then easy to show that if the left-hand side of a reduction rule is typed by a session typing, which is consistent when composed with some session typing, then the same property holds for the right-hand side too. This is formalised in Lemma 5.19, which uses Lemma 5.18 that connects projections and skipping of inputs.

We extend the function # (cf. Definition 3.1) to queue types in a natural way. Given a queue type $\mathcal{H}$ and a participant $\mathtt{p}$, we define $\#(\mathcal{H}, \mathtt{p})$ by induction on $\mathcal{H}$:

$$\#(\epsilon, \mathtt{p}) = \emptyset$$

$$\#(\mathtt{q}!\lambda(S); \mathcal{H}', \mathtt{p}) = \begin{cases} \lambda \cdot \#(\mathcal{H}', \mathtt{p}) & \text{if } \mathtt{p} = \mathtt{q} \\ \#(\mathcal{H}', \mathtt{p}) & \text{otherwise.} \end{cases}$$

Lemma 5.18. If $\langle \mathcal{H}, \mathtt{q}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}\rangle \upharpoonright \mathtt{q} \bowtie \mathcal{M} \upharpoonright \mathtt{p}$ then $\langle \mathcal{H}, \mathcal{M}_i\rangle \upharpoonright \mathtt{q} \bowtie (\lessdot \mathcal{M}, \#(\mathcal{H}, \mathtt{q})\gtrdot \setminus ?(\mathtt{p}, \lambda_i)) \upharpoonright \mathtt{p}$.

Proof. The proof is by induction on $\mathcal{H}$ and $\mathcal{M}$ and by cases on their shapes. Notice that by definition

$$\langle \mathcal{H}, \mathtt{q}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}\rangle \upharpoonright \mathtt{q} = \lessdot \mathcal{H} \upharpoonright \mathtt{q}, !\{\lambda_i(S_i).(\mathcal{M}_i \upharpoonright \mathtt{q})\}_{i\in I} \gtrdot.$$

We analyse the different cases:

- If $\#(\mathcal{H}, \mathtt{q}) = \emptyset$, then $\mathcal{M} \upharpoonright \mathtt{p} = ?\{\lambda_i(S_i).\Psi_i\}_{i\in I}$ and $\mathcal{M}_i \upharpoonright \mathtt{q} \bowtie \Psi_i$. Moreover, by Definition 3.2 we have $(\lessdot \mathcal{M}, \emptyset \gtrdot \setminus ?(\mathtt{p}, \lambda_i)) \upharpoonright \mathtt{p} = \Psi_i$.
- If $\mathcal{H} = \mathtt{k}!\lambda(S).\mathcal{H}'$ and $\mathtt{k} \neq \mathtt{q}$, then $\mathcal{H} \upharpoonright \mathtt{q} = \mathcal{H}' \upharpoonright \mathtt{q}$ and we are done by induction.
- If $\mathcal{M} = \mathtt{k}!\{\lambda'_j(S'_j).\mathcal{M}'_j\}_{j\in J}$ or $\mathcal{M} = \mathtt{k}?\{\lambda'_j(S'_j).\mathcal{M}'_j\}_{j\in J}$ and $\mathtt{k} \neq \mathtt{q}$, then $\mathcal{M} \upharpoonright \mathtt{q} = \mathcal{M}'_j \upharpoonright \mathtt{q}$ for all $j \in J$ and we are done by induction.
- If $\mathcal{H} = \mathtt{q}!\lambda(S).\mathcal{H}'$ and $\mathcal{M} = \mathtt{p}?\{\lambda'_j(S'_j).\mathcal{M}'_j\}_{j\in J}$, then by duality $\lambda = \lambda'_{j_0}$ for some $j_0 \in J$ and

  $$\langle \mathcal{H}', \mathtt{q}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}\rangle \upharpoonright \mathtt{q} \bowtie \mathcal{M}'_{j_0} \upharpoonright \mathtt{p}.$$

  By induction, we have $\langle \mathcal{H}', \mathcal{M}_i\rangle \upharpoonright \mathtt{q} \bowtie (\lessdot \mathcal{M}'_{j_0}, \#(\mathcal{H}', \mathtt{q})\gtrdot \setminus ?(\mathtt{p}, \lambda_i)) \upharpoonright \mathtt{p}$. While by Definition 3.1 we have $\#(\mathcal{H}, \mathtt{q}) = \lambda \cdot \#(\mathcal{H}', \mathtt{q})$, Definition 3.2 gives

  $$\lessdot \mathcal{M}, \#(\mathcal{H}, \mathtt{q})\gtrdot \setminus ?(\mathtt{p}, \lambda_i) = \mathtt{p}?\lambda(S'_{j_0}).\ \lessdot \mathcal{M}'_{j_0}, \#(\mathcal{H}', \mathtt{q})\gtrdot \setminus ?(\mathtt{p}, \lambda_i)$$

  which in turn implies

  $$\langle \mathcal{H}, \mathcal{M}_i\rangle \upharpoonright \mathtt{q} \bowtie (\lessdot \mathcal{M}, \#(\mathcal{H}, \mathtt{q})\gtrdot \setminus ?(\mathtt{p}, \lambda_i)) \upharpoonright \mathtt{p}$$

  and this concludes the proof.

□

Lemma 5.19. Let $\vdash_\Sigma N \rhd \Delta$, and $N \longrightarrow_{\mathcal{P}} N'$, and $\Delta * \Delta_0$ be consistent for some Δ_0. Then there is Δ' such that $\vdash_\Sigma N' \rhd \Delta'$ and $\Delta \Longrightarrow^* \Delta'$ and $\Delta' * \Delta_0$ is consistent.

Proof. The proof is by cases on network reduction rules. It is sufficient to consider the reduction rules which do not contain network reductions as premises, i.e., which are the leaves in the reduction trees. We only consider two cases, representative of our model (Rules OutLoc and Reconf).

- Rule OutLoc:

$$\frac{\begin{array}{c}\mathcal{M}_{\mathsf{p}} \xrightarrow{\mathsf{q}!\lambda} \widehat{\mathcal{M}}_{\mathsf{p}} \quad P \xrightarrow{\mathsf{s}[\mathsf{p}]!\lambda(v)} P' \quad \mathsf{w}_p \not\sqsubseteq lev(v) \quad \mathsf{w}_b \sqsubseteq lev(v) \\ \widehat{\mathcal{M}}_{\mathsf{q}} = \lessdot \mathcal{M}_{\mathsf{q}}, \#(h, \mathsf{p}, \mathsf{q})\gtrdot \setminus ?(\mathsf{p}, \lambda) \quad \mathsf{T} \propto \widehat{\mathcal{M}}_{\mathsf{q}} \quad (Q', \mathsf{T}) \in \mathcal{P}\end{array}}{\mathcal{M}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathcal{M}_{\mathsf{q}}^{\mathsf{r}',\mathsf{w}'}[Q] \mid \mathsf{s} : \langle h; \sigma\rangle \longrightarrow_{\mathcal{P}} \widehat{\mathcal{M}}_{\mathsf{p}}^{\mathsf{r},\mathsf{w}}[P'] \mid \widehat{\mathcal{M}}_{\mathsf{q}}^{\mathsf{r}',\mathsf{w}'}[Q'\{\mathsf{s}[\mathsf{q}]/y\}] \mid \mathsf{s} : \langle h; \sigma\rangle}$$

  Here the Inversion Lemma (Lemma 5.11) applied to

$$\vdash_\Sigma \mathcal{M}_{\mathtt{p}}[P] \mid \mathcal{M}_{\mathtt{q}}[Q] \mid \mathtt{s} : \langle h; \sigma\rangle \rhd \Delta$$

gives $\Sigma = \{\mathtt{s}\}$ and $\Delta = \{\mathtt{s}[\mathtt{p}] : \langle \mathcal{H}_{\mathtt{p}}, \mathcal{M}_{\mathtt{p}}\rangle, \mathtt{s}[\mathtt{q}] : \langle \mathcal{H}_{\mathtt{q}}, \mathcal{M}_{\mathtt{q}}\rangle\} \cup \Delta_1$ and $\vdash_{\{\mathtt{s}\}} \mathtt{s} : \langle h; \sigma\rangle \rhd \{\mathtt{s}[\mathtt{p}] : \mathcal{H}_{\mathtt{p}}, \mathtt{s}[\mathtt{q}] : \mathcal{H}_{\mathtt{q}}\} \cup \Delta_1$. Consistency of $\Delta * \Delta_0$ and the definition of composition imply $\langle \mathcal{H}_{\mathtt{p}}, \mathcal{M}_{\mathtt{p}}\rangle \upharpoonright \mathtt{q} \bowtie \langle \mathcal{H}_{\mathtt{q}}, \mathcal{M}_{\mathtt{q}}\rangle \upharpoonright \mathtt{p}$. Since $\langle \mathcal{H}_{\mathtt{p}}, \mathcal{M}_{\mathtt{p}}\rangle \upharpoonright \mathtt{q}$ starts with an output, we get $\langle \mathcal{H}_{\mathtt{p}}, \mathcal{M}_{\mathtt{p}}\rangle \upharpoonright \mathtt{q} \bowtie \mathcal{M}_{\mathtt{q}} \upharpoonright \mathtt{p}$. Lemma 5.13(2) and $\mathcal{M}_{\mathtt{p}} \xrightarrow{\mathtt{q}!\lambda} \widehat{\mathcal{M}}_{\mathtt{p}}$ give $\mathcal{M}_{\mathtt{p}} = \mathtt{q}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}$ and $\lambda = \lambda_j$ and $\widehat{\mathcal{M}}_{\mathtt{p}} = \mathcal{M}_j$ for some $j \in I$. It is easy to verify that $\#(h, \mathtt{p}, \mathtt{q}) = \#(\mathcal{H}_{\mathtt{p}}, \mathtt{q})$: we put $\Lambda = \#(\mathcal{H}_{\mathtt{p}}, \mathtt{q})$. By Lemma 5.18, $\langle \mathcal{H}_{\mathtt{p}}, \mathcal{M}_j\rangle \upharpoonright \mathtt{q} \bowtie (<\mathcal{M}_{\mathtt{q}}, \Lambda> \backslash ?(\mathtt{p}, \lambda)) \upharpoonright \mathtt{p}$. Notice that $\mathcal{M}_{\mathtt{p}}$ differs from $\mathcal{M}_j$ only for an output to $\mathtt{q}$ with label λ and $\mathcal{M}_{\mathtt{q}}$ differs from $<\mathcal{M}_{\mathtt{q}}, \Lambda> \backslash ?(\mathtt{p}, \lambda)$ only for branchings with receiver $\mathtt{p}$ without corresponding selections in $\mathcal{H}_{\mathtt{p}}$ and for an input from $\mathtt{p}$ with label λ. Let $\widehat{\mathcal{M}}_{\mathtt{q}} = <\mathcal{M}_{\mathtt{q}}, \Lambda> \backslash ?(\mathtt{p}, \lambda)$ and

$$\Delta' = \{\mathtt{s}[\mathtt{p}] : \langle \mathcal{H}_{\mathtt{p}}, \widehat{\mathcal{M}}_{\mathtt{p}}\rangle, \mathtt{s}[\mathtt{q}] : \langle \mathcal{H}_q, \widehat{\mathcal{M}}_{\mathtt{q}}\rangle\} \cup \Delta_1.$$

By Lemma 5.15(2), we have $\vdash P' \rhd \mathtt{s}[\mathtt{p}] : \mathsf{T}'$ and $\mathsf{T}' \propto \widehat{\mathcal{M}}_{\mathtt{p}}$. By assumption $\vdash Q' \rhd y : \mathsf{T}$ and $\mathsf{T} \propto \widehat{\mathcal{M}}_{\mathtt{q}}$. So using Rules MP, QInit, QSendV, and NPar we derive

$$\vdash_{\{\mathtt{s}\}} \widehat{\mathcal{M}}_{\mathtt{p}}[P'] \mid \widehat{\mathcal{M}}_{\mathtt{q}}[Q'\{\mathtt{s}[\mathtt{q}]/y\}] \mid \mathtt{s} : \langle h; \sigma\rangle \rhd \Delta'.$$

Consistency of $\Delta * \Delta_0$ implies consistency of $\Delta' * \Delta_0$. Finally, notice that $\Delta \implies^* \Delta'$ by the fifth and sixth rules in Table 10.

- Rule Reconf:

$$\frac{\mathcal{A}(\{\mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{p} \in \Pi\}, \sigma, \mathit{nonce}_i) = \Pi' \quad F(\{\mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{p} \in \Pi'\}, \sigma) = (\mathsf{G}, \mathsf{L})}{(\nu\, \mathtt{s})\ (\prod_{\mathtt{p}\in\Pi} \mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{s} : \langle h; \sigma\rangle) \longrightarrow_{\mathcal{P}} (\nu\, \mathtt{s})\ (\prod_{\mathtt{p}\in\Pi\setminus\Pi'} \mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{s} : \langle h \setminus \Pi'; \sigma \setminus \mathit{nonce}_i\rangle) \mid \mathsf{new}(\mathsf{G}, \mathsf{L})}$$

In this case, the Inversion Lemma (Lemma 5.11) applied to

$$\vdash_\Sigma (\nu\, \mathtt{s})\ (\textstyle\prod_{\mathtt{p}\in\Pi} \mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{s} : \langle h; \sigma\rangle) \rhd \Delta$$

gives $\Sigma = \Delta = \emptyset$ and $\vdash_{\{\mathtt{s}\}} \prod_{\mathtt{p}\in\Pi} \mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{s} : \langle h; \sigma\rangle \rhd \{\mathtt{s}[\mathtt{p}] : \tau_{\mathtt{p}} \mid \mathtt{p} \in \Pi\}$. Lemma 5.11(7) implies $\tau_{\mathtt{p}} \upharpoonright \mathtt{q} \bowtie \tau_{\mathtt{q}} \upharpoonright \mathtt{p}$ for all $\mathtt{p}, \mathtt{q} \in \Pi$. Definition 3.4 gives $\mathcal{M}_{\mathtt{p}} \upharpoonright \mathtt{q} = \epsilon$ whenever $\mathtt{p} \in \Pi'$ and $\mathtt{q} \in \Pi \setminus \Pi'$ or vice versa. By Definition 3.3 all receivers of the messages in the queue $h \setminus \Pi'$ belong to $\Pi \setminus \Pi'$. Therefore $\{\mathtt{s}[\mathtt{p}] : \tau_{\mathtt{p}} \mid \mathtt{p} \in \Pi \setminus \Pi'\}$ is consistent and we can derive $\vdash_{\{\mathtt{s}\}} \prod_{\mathtt{p}\in\Pi\setminus\Pi'} \mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{s} : \langle h \setminus \Pi'; \sigma\rangle \rhd \{\mathtt{s}[\mathtt{p}] : \tau_{\mathtt{p}} \mid \mathtt{p} \in \Pi \setminus \Pi'\}$. We can then conclude

$$\vdash_\emptyset (\nu\, \mathtt{s})\ (\textstyle\prod_{\mathtt{p}\in\Pi\setminus\Pi'} \mathcal{M}_{\mathtt{p}}[P_{\mathtt{p}}] \mid \mathtt{s} : \langle h \setminus \Pi'; \sigma \setminus \mathit{nonce}_i\rangle) \mid \mathsf{new}(\mathsf{G}, \mathsf{L}) \rhd \emptyset$$

by Rules Res, New and NPar.

□

Our first main result is:

Theorem 5.20 (Subject Reduction). If $\vdash_\Sigma N \rhd \Delta$ with Δ consistent and $N \longrightarrow_{\mathcal{P}}^* N'$, then $\vdash_\Sigma N' \rhd \Delta'$ for some consistent Δ' such that $\Delta \implies^* \Delta'$.

Proof. It is enough to show the statement for the case $N \equiv \mathcal{E}[N_0]$ and $N' \equiv \mathcal{E}[N_0']$, where $N_0 \longrightarrow_{\mathcal{P}} N_0'$ by one of the rules considered in Lemma 5.19. By the structural equivalence on networks we can assume $\mathcal{E} = (\overrightarrow{\nu \mathtt{s}})([\,] \mid N_1)$ without loss of generality. Lemma 5.16 and 5.11(7) and (6) applied to $\vdash_\Sigma N \rhd \Delta$ give $\vdash_{\Sigma_0} N_0 \rhd \Delta_0$ and $\vdash_{\Sigma_1} N_1 \rhd \Delta_1$, where $\Sigma = (\Sigma_0 \cup \Sigma_1) \setminus \overrightarrow{\mathtt{s}}$ and $\Delta = (\Delta_0 * \Delta_1) \setminus \overrightarrow{\mathtt{s}}$. The consistency of Δ implies the consistency of $\Delta_0 * \Delta_1$ by Lemma 5.11(7). By Lemma 5.19 there is Δ_0' such that $\vdash_{\Sigma_0} N_0' \rhd \Delta_0'$ and $\Delta_0 \implies^* \Delta_0'$ and $\Delta_0' * \Delta_1$ is consistent. Therefore, we derive $\vdash_\Sigma N' \rhd \Delta'$, where $\Delta' = (\Delta_0 * \Delta_1') \setminus \overrightarrow{\mathtt{s}}$ by applying typing rules NPar and Res. Observe that $\Delta \implies^* \Delta'$ and Δ' is consistent. □

We say that a network is *initial* when it is a parallel composition of session initiators, which is always typable. The type system can guarantee progress provided that the collection of processes and types contains at least one process for each monitor which is created at runtime in the adaptations. This can be statically checked (cf. Definition 3.7).

Theorem 5.21 (Progress). Let $\mathcal{G}$ be a set of global types and $\mathcal{P}$ be a complete collection for it. Suppose

that the co-domain of the adaptation function F (cf. Rule RECONF) is contained in $\mathcal{G}$. If N is an initial network with global types in $\mathcal{G}$, and $N \longrightarrow_{\mathcal{P}}^{*} N'$, then N' has progress, i.e.,

1. Every input monitored process will eventually receive a message, and
2. Every message in a queue will eventually be received by an input monitored process.

Proof. As proved in [CDCYP16], a single multiparty session in a standard calculus with global and session types, like the calculus in [HYC08], always enjoys progress whenever it is well typed. In fact, by the Subject Reduction Theorem (Theorem 5.20), reduction preserves well-typedness of sessions and the consistency of session typings. Moreover, all required session participants are present. Thus, all communications among participants in a unique session will take place, in the order prescribed by the global type.

It is easy to see that our calculus could be mapped to the calculus of [HYC08] while maintaining typability, the main difference being in the reduction rules. In our networks, several sessions may run in parallel, since an initial configuration can have many initiators. Moreover, Rule RECONF splits one session into two, provided the required processes can be found, which is ensured by the completeness of $\mathcal{P}$. However, there is no interleaving between sessions, since each monitored process implements a participant in a single session. Hence, the only rules that could jeopardise progress are OUTLOC and OUTGLOB. This does not happen, since the required process is in $\mathcal{P}$ by completeness, and the reductions only modify:

- the monitors of the two participants, preserving the consistency of session typings, and
- the processes of the two participants, ensuring agreement of their types with the new monitors.

Therefore, our calculus has progress under the required conditions. □

As regards security, the main properties of our calculus are given by the following theorem, which ensures that both reading and writing respect the security permissions and the boundaries of session participants.

Theorem 5.22 (Access Control and Information Flow). Let $\mathcal{M}_{\mathtt{p}}^{\mathsf{r},\mathsf{w}}[P]$ be a monitored process.

1. If $\mathcal{M}_{\mathtt{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathtt{s} : \langle (\mathtt{q},\mathtt{p},\lambda(v)) \cdot h ; \sigma\rangle \longrightarrow_{\mathcal{P}} \mathcal{M}^{\mathsf{r},\mathsf{w}}[P'] \mid \mathtt{s} : \langle h ; \sigma'\rangle$, then either $lev(v) \sqsubseteq \mathsf{r}_p$ or P' is not obtained by consuming the message $(\mathtt{q},\mathtt{p},\lambda(v))$. Moreover P' may contain a fresh nonce only if $lev(v) \not\sqsubseteq \mathsf{r}_b$.
2. If $\mathcal{M}_{\mathtt{p}}^{\mathsf{r},\mathsf{w}}[P] \mid \mathtt{s} : \langle h ; \sigma\rangle \longrightarrow_{\mathcal{P}} \mathcal{M}^{\mathsf{r},\mathsf{w}}[P'] \mid \mathtt{s} : \langle h \cdot (\mathtt{p},\mathtt{q},\lambda(v)) ; \sigma\rangle$, then $\mathsf{w}_p \sqsubseteq lev(v)$.
3. If $\mathcal{M}_{\mathtt{p}}^{\mathsf{r},\mathsf{w}}[P] \mid N \mid \mathtt{s} : \langle h ; \sigma\rangle \longrightarrow_{\mathcal{P}} \mathcal{M}_{\mathtt{p}}^{\mathsf{r}',\mathsf{w}}[P'] \mid N' \mid \mathtt{s} : \langle h \cdot (\mathtt{p},\mathtt{q},\mathit{nonce}_i) ; \sigma, (\mathtt{p},\mathit{nonce}_i)\rangle$, then $P \xrightarrow{\mathtt{s}[\mathtt{p}]!\lambda(v)} P'$ and $\mathsf{w}_b \not\sqsubseteq lev(v)$.

Proof. The proof follows easily by inspecting the reduction rules of Table 6. □

Theorem 5.22(1) says that the reading permission of a monitor is either (i) respected and the reading from the queue takes place, or (ii) not respected and the operational semantics ensures that the disallowed value is never read from the queue—by virtue of the runtime mechanisms implemented by Rules INLOC and INGLOB. Analogously, Theorem 5.22(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 OUTLOC and OUTGLOB can always be triggered to handle the situations in which the given value does not respect the lower bound defined by the writing permission —that is, the situations in which the value should not be admitted in the session queue. Moreover, Theorem 5.22(1) and (3) assure that the creation of nonces is always triggered by exchanges of values that do not only violate permissions, but also do not respect the boundaries.

6. Related Work

To the best of our knowledge, the framework presented here integrates for the first time two distinct concerns within formal models of structured communications, namely (self-)adaptation and security—here understood as the interplay of policies for access control and secure information flow. As a consequence, our framework can be related to several previous works on (typed) process formalisms tailored to either concern. In particular, as already mentioned, our approach owes much to [CDCV15], where a model of self-adaptation for multiparty communication is put forward based on networks of monitored systems. However, the framework in [CDCV15] does not address issues of access control and secure information flow. Our work goes well beyond [CDCV15]

in that it proposes to use adaptation/reconfiguration mechanisms to mitigate the effect of security violations both in reading and in writing (see below for more detailed comparisons). Next we review some related works, distinguishing between the adaptation and security dimensions.

Adaptation. There have been several studies on adaptive systems in various application contexts, approaching the concept of adaptation from different perspectives. The paper [BCG$^+$12] provides a thorough discussion on this issue and an interesting classification of various approaches. A recent survey on service choreography adaptation is presented in [LON$^+$13]. A process calculus of *adaptable processes* is proposed in [BDPZ12]; it includes so-called located processes that can be modified by "update patterns". Based on a fragment of CCS, the calculus in [BDPZ12] is not equipped with types for structured communications. A variant of the adaptable processes of [BDPZ12], recast in a session typed setting, is developed in [DP13, DP15]; it combines the constructors for adaptable processes of [BDPZ12] with the session type system proposed in [GCDC06] for the Boxed Ambient calculus [BCC04]. The type discipline in [DP13, DP15] ensures a basic consistency between adaptation actions and session communications, but it does not account for multiparty protocols.

Works addressing adaptation for multiparty communications include [BCH$^+$14, DGL$^+$14] and [CDCV15]. The work [BCH$^+$14] enhances a choreographic language with constructs defining adaptation scopes and dynamic code update; an associated endpoint language for local descriptions, and a projection mechanism for obtaining (low-level) endpoint specifications from (high-level) choreographies are also described. A typing discipline for these languages is left for future work. The paper [DGL$^+$14] proposes a choreographic language for distributed applications. Adaptation follows a rule-based approach, in which all interactions, under all possible changes produced by the adaptation rules, proceed as prescribed by an abstract model. In particular, the system is deadlock-free by construction. The adaptive system is composed by interacting participants deployed on different locations, each executing its own code.

As already mentioned, our work builds on [CDCV15], 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 [CDCV15] relied on a global state, and the global types described 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 [CDCV15] involved all session participants. In contrast, in our calculus adaptation is triggered by security violations, and an adaptation may be either local or global. In conclusion, we consider our adaptation mechanism to be more flexible than that of [CDCV15] in two respects. First, adaptation is dynamically triggered in reaction to security violations (whose occurrence is hard to predict) rather than at fixed points of computation. Second, adaptation may be restricted to a subset of participants (those directly involved in the security violation), thus resulting in a lighter procedure, which does not affect well-behaved participants.

Security. The integration of security requirements into process calculi with structured communication (such as session calculi) is still at an early stage. Enforcement of *integrity* properties in multiparty sessions, using session types, has been studied in [BCD$^+$09, PCF09]. These papers propose a compiler which, given a multiparty session description, implements cryptographic protocols that guarantee session execution integrity.

In the present work, we have been concerned with *confidentiality* rather than integrity. To this end, we targeted two security properties: *access control* (AC) and *secure information flow* (SIF) [SM03]. It is well known that AC and SIF are complementary properties, both of which are necessary to ensure end-to-end confidentiality. In our setting, the classical argumentation may be rephrased as follows:

- With AC but without SIF, a participant could *test* any high information she is entitled to receive, and then *leak* it to a participant with a lower reading permission by sending her different low values depending on the result of the high test.
- With SIF but without AC, a participant could *receive* and *forward* (retransmit without testing it) information of any level to any participant. This would not per se constitute a security violation, since in any case only observers of the appropriate level would be able to read the information, but it would lead to uncontrolled dissemination of sensitive information, making the system as a whole more vulnerable (more fragility points for an attacker to break through). Although in most research on SIF such dissemination is not considered harmful, in a distributed setting the wide-spreading of sensitive information could constitute a serious security weakness.

Our treatment of access control (AC) is similar to that in [CCDC14], since in both cases participants are assigned reading permissions that limit their capacity to receive data from the other participants. There are some important differences, though. In [CCDC14], reading permissions are fixed once and for all and they appear in the types; then, the targeted result is that well-typed processes satisfy the AC policy. In our case, instead, the initial reading permissions of participants may decrease during the computation by effect of their insecure behaviour, and conformance with the AC policy is checked dynamically. In this sense, our approach is closer to that of the subsequent paper [CCDC15], where the calculus of [CCDC14] is equipped with a monitored semantics, which blocks the execution of processes as soon as they attempt to perform an insecure communication.

As regards secure information flow (SIF), it is worth noticing that the SIF property we enforce here is incomparable with that of [CCDC14], as already illustrated by the example at the end of § 3. Indeed, in one sense our SIF property is stronger than the one of [CCDC14], because it is a safety property, ensuring the absence of a "level drop" between a test and the following communications in every possible computation. For instance, we rule out a conditional statement which tests a high expression and performs two equal low outputs in its branches. In another sense, our SIF property is weaker than that of [CCDC14], since it allows a high input to be followed by a low output (as shown by the Example at the end of Section 3), a situation that was considered insecure in [CCDC14] on the grounds that the high input was not guaranteed to occur. In fact, based on this very observation, it could be shown that our SIF property is strictly weaker than the safety property of [CCDC15].

Notice that our choice of using a monitored semantics to enforce both AC and SIF, through the use of similar $(permission, boundary)$ pairs, enables a fully symmetric treatment of these properties. Indeed, a reading permission violation is an AC violation, while a writing permission violation is a SIF violation. The adaptation mechanisms also deal with reading and writing violations in a symmetric way, except for the case of a hard writing violation, where the transgressing sender is treated as a culprit and penalised by a decrease of her reading permission.

Finally, one novelty of our work with respect to both [CCDC14] and [CCDC15] is that possible infringements to the security requirements are envisaged and not considered catastrophic: indeed, adaptation and reconfiguration mechanisms are provided to fix breaches or mitigate their effects (in different ways according to their degree of gravity), so that the interaction is not blocked and may proceed according to the protocol, although in a possibly degraded mode. Indeed, the present work focuses on ensuring that the overall protocol can carry on notwithstanding the possibility of security breaches, rather than on giving absolute confidentiality guarantees to individual participants.

In recent years there have been a number of studies on security monitoring (often in combination with types), mainly aimed at dynamically enforcing the SIF property of *noninterference* in sequential languages, and particularly in JavaScript [GBJS06, Bou09, SR10, RSC09, DP10, AF12]. For instance, [GBJS06] proposes an automaton-based monitoring mechanism, which combines static and dynamic analyses, for a sequential imperative while-language with outputs. The work [Bou09] defines a monitored semantics for an ML-like language that ensures noninterference, while [AS09] presents a combination of monitoring and static analysis for a sequential language with dynamic code evaluation, enforcing *information-release policies*, which are relaxations of noninterference. In a slightly different direction, the earlier works [ML00, ZM07] considered *dynamic security policies* and proposed means for expressing them via security labels.

Dynamic analysis techniques induce a safety property which is an approximation of the targeted security property. Therefore, they are bound to produce *false alarms*, leading to rejection of secure programs. Hence, an important issue is that of the precision of the dynamic analysis. While some monitored semantics are "fail-stop", namely they block the execution of a program when a potential leak is detected, others are more permissive, allowing a program to proceed beyond an offending statement, but instrumenting the program in such a way that the leak cannot be actually realised. We shall give a brief account of the latter semantics, which are closer to our approach.

The work [DP10] proposes a technique called *secure multi-execution* for a deterministic language with I/O operations. This technique is shown to enforce a time-sensitive noninterference property with a good precision. The idea is to execute a program multiple times, one time for each security level, using special rules for I/O operations: in the execution associated with a given security level, the inputs of higher or incomparable level are replaced by default values, and only the outputs of the given security level are delivered. This way, if the outputs produced at a given level are independent of the inputs of higher or incomparable level, they will be observed with their correct value. Otherwise, they will be observed with a meaningless value (the default value), but in any case no information leak will occur. The subsequent work [AF12] improves on [DP10] by

introducing *faceted values*: with this mechanism, closely related to the work of [PS03], it is possible to use a unique process to simultaneously and efficiently simulate multiple executions for different security levels. The faceted-value technique is shown to achieve termination-insensitive noninterference.

Monitored semantics that do not necessarily block but possibly adapt the behaviour of unsafe programs are referred to as *edit automata* [LBW05]. Such automata, an evolution of Schneider's *security automata* [Sch00], allow sequences of program actions that deviate from a security policy (in a given class of policies) to be transformed by means of operations of truncation and action insertion/deletion. Note that unlike the above-described approaches, our framework allows the distinction between two kinds of leaks, soft and hard ones, and provides different responses for each of them. In the same spirit, although the technical treatment is quite different from ours, the work [BM11] advocates a flexible response to security violations by distinguishing two classes of non-critical security errors: *venial errors*, which can be tolerated up to a certain number of occurrences without transforming the computation, and *amendable errors*, which may be neutralised by transforming the computation by means of edit automata. In this approach there still remains a category of non-amendable errors, which cannot be fixed and thus require the computation to be aborted.

To conclude, let us mention that various approaches for enforcing security into calculi and languages for structured communications have been recently surveyed in the state-of-the art report produced by the Security Working Group of the COST Action BETTY, entitled "Combining Behavioural Types with Security Analysis" [BCD$^+$15].

7. Concluding Remarks

Framed in the setting of formal models and analysis techniques for communication-centric systems [HLV$^+$16], we have introduced a framework for multiparty protocols (choreographies) in which the analysis of communication correctness is coupled with the run-time enforcement of self-adaptation and secure information flow policies. One leading motivation for our development is the observation that as communication-centric systems operate in dynamic and heterogeneous environments, correctness analysis for the underlying structured protocols must account for a range of different issues that influence the interactive behaviour of protocol participants. We have shown how two such issues, self-adaptation and security, exhibit an appealing complementarity and admit a unified treatment based on global types, monitors (local types with security levels), monitored processes, and a suitably instrumented operational semantics.

Based on the framework defined in [CDCV15], our approach relies on elementary assumptions on the nature of typed processes. Processes in our model are well-typed with respect to a simple discipline (based on intersection and union types) which does not mention security policies nor adaptation requirements. The relationship between choreographic specifications (global types) and the processes implementing such specifications is then indirect, but made formal by the notion of adequacy, which relates process types (which describe "pure" communication behaviours) and monitors (which maintain run-time information on security permissions/boundaries). This degree of independence between a global specification and its process implementations distinguishes our approach from the original multiparty session types of [HYC08], in which participants' types correspond directly to projections of global types. Since in our framework security and adaptation policies appear only at the level of global types and monitors, in principle one may, e.g., modify or upgrade such policies without altering associated implementations, or consider legacy implementations not directly related to any policy—in both cases, the types of the implementations would need to be adequate to the run-time monitors. Investigating how this kind of modular modifications to security and adaptation policies can be precisely expressed in extensions of our framework is an interesting direction for future work.

The semantics associated with the reconfiguration mechanism is admittedly simple, as it is nondeterministic and abstracts from the function that determines the next choreography to be installed. We prefer leaving these choices unspecified, for the sake of generality, as in our opinion different application scenarios may call for different adaptation functions and alternative strategies to cope with system degradation due to nonce generation. For instance, reputation mechanisms [BCCD12] could be added in order to refine/guide the reaction to major security violations based on a nonce record for each participant. It would be also interesting to define adaptation functions which depend on the current (type)state of the monitors; this mechanism has been developed in [DP16] for structured protocols using type-directed checks and event-based adaptation. In general, it would be interesting to consider application-driven instances of our global adaptation mechanisms in which concrete adaptation functions are used.

Acknowledgments. We are grateful to the anonymous reviewers of BEAT'14 and of the present paper for their useful suggestions, which led to substantial improvements. This work was supported by COST Action IC1201 BETTY (Behavioural Types for Reliable Large-Scale Software Systems) via a Short-Term Scientific Mission grant (funding a visit of Pérez to Torino). Dezani was also partially supported by EU H2020-644235 Rephrase project, EU H2020-644298 HyVar project, ICT COST Actions IC1402 ARVI and Ateneo/CSP project RunVar. Pérez is also affiliated to the NOVA Laboratory for Computer Science and Informatics (NOVA LINCS, Ref. UID/CEC/04516/2013), Universidade Nova de Lisboa, Portugal.

References

[AF12] Thomas H. Austin and Cormac Flanagan. Multiple Facets for Dynamic Information Flow. In *POPL 2012*, pages 165–178. ACM Press, 2012.

[AS09] Aslan Askarov and Andrei Sabelfeld. Tight Enforcement of Information-Release Policies for Dynamic Languages. In *CSF 2009*, pages 43–59. IEEE Computer Society, 2009.

[BCC04] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Access Control for Mobile Agents: The Calculus of Boxed Ambients. *ACM Transactions on Programming Languages and Systems*, 26(1):57–124, 2004.

[BCCD12] Viviana Bono, Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. A Reputation System for Multirole Sessions. In *TGC 2011*, volume 7173 of *LNCS*, pages 1–24. Springer, 2012.

[BCD+09] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic Protocol Synthesis and Verification for Multiparty Sessions. In *CSF 2009*, pages 124–140. IEEE Computer Society, 2009.

[BCD+13] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring Networks through Multiparty Session Types. In *FMOODS/FORTE 2013*, volume 7892 of *LNCS*, pages 50–65. Springer, 2013.

[BCD+15] Massimo Bartoletti, Ilaria Castellani, Pierre-Malo Deniélou, Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Jovanka Pantovic, Jorge A. Pérez, Peter Thiemann, Bernardo Toninho, and Hugo Torres Vieira. Combining Behavioural Types with Security Analysis. *Journal of Logical and Algebraic Methods in Programming*, 84(6):763 – 780, 2015. Special Issue on Open Problems in Concurrency Theory.

[BCG+12] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. A Conceptual Framework for Adaptation. In *FASE 2012*, volume 7212 of *LNCS*, pages 240–254. Springer, 2012.

[BCH+14] Mario Bravetti, Marco Carbone, Thomas T. Hildebrandt, Ivan Lanese, Jacopo Mauro, Jorge A. Pérez, and Gianluigi Zavattaro. Towards Global and Local Types for Adaptation. In *SEFM 2013*, volume 8368 of *LNCS*, pages 3–14. Springer, 2014.

[BDPZ12] Mario Bravetti, Cinzia Di Giusto, Jorge A. Pérez, and Gianluigi Zavattaro. Adaptable Processes. *Logical Methods in Computer Science*, 8(4), 2012.

[BM11] Nataliia Bielova and Fabio Massacci. Computer-Aided Generation of Enforcement Mechanisms for Error-Tolerant Policies. In *POLICY 2011*, pages 89–96. IEEE Computer Society Press, 2011.

[Bou09] Gerard Boudol. Secure Information Flow as a Safety Property. In *FAST 2008*, volume 5491 of *LNCS*, pages 20–34. Springer, 2009.

[BYY14] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed Multiparty Session Types. In *CONCUR 2014*, volume 8704 of *LNCS*, pages 419–434. Springer, 2014.

[CCDC14] Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. Typing Access Control and Secure Information Flow in Sessions. *Information and Computation*, 238:68–105, 2014.

[CCDC15] Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. Information Flow Safety in Multiparty Sessions. *Mathematical Structures in Computer Science*, FirstView:1–43, 2015. Available on CJO2015. doi:10.1017/S0960129514000619.

[CDCV15] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Self-Adaptive Multiparty Sessions. *Service Oriented Computing and Applications*, 9(3-4):249–268, 2015.

[CDCYP16] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016.

[CDP14] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Jorge A. Pérez. Self-Adaptation and Secure Information Flow in Multiparty Structured Communications: A Unified Perspective. In *BEAT 2014*, volume 162 of *EPTCS*, pages 9–18. Open Publishing Association, 2014.

[CHY12] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centered Programming for Web Services. *ACM Transactions on Programming Languages and Systems*, 34(2):8:1–8:78, June 2012.

[Den76] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of ACM*, 19(5):236–243, 1976.

[DGL+14] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. AIOCJ: A Choreographic Framework for Safe Adaptive Distributed Applications. In *SLE 2014*, volume 8706 of *LNCS*, pages 161–170. Springer, 2014.

[DP10] Dominique Devriese and Frank Piessens. Noninterference through Secure Multi-execution. In *Security and Privacy 2010*, pages 109–124. IEEE Computer Society, 2010.

[DP13] Cinzia Di Giusto and Jorge A. Pérez. Disciplined Structured Communications with Consistent Runtime Adaptation. In *SAC 2013*, pages 1913–1918. ACM Press, 2013.

[DP15] Cinzia Di Giusto and Jorge A. Pérez. Disciplined Structured Communications with Disciplined Runtime Adaptation. *Science of Computer Programming*, 97:235–265, 2015.

[DP16] Cinzia Di Giusto and Jorge A. Perez. An Event-Based Approach to Runtime Adaptation in Communication-Centric Systems. In *Web Services, Formal Methods, and Behavioral Types*, volume 9421 of *LNCS*, pages 67–85. Springer, 2016. Extended version to appear in Formal Aspects of Computing.

[GBJS06] Gurvan Le Guernic, Anindya Banerjee, Thomas P. Jensen, and David A. Schmidt. Automata-Based Confidentiality Monitoring. In Springer, editor, *ASIAN 2006*, volume 4435 of *LNCS*, pages 75–89, 2006.

[GCDC06] Pablo Garralda, Adriana B. Compagnoni, and Mariangiola Dezani-Ciancaglini. BASS: Boxed Ambients with Safe Sessions. In *PPDP 2006*, pages 61–72. ACM Press, 2006.

[GH05] Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[HLV+16] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Computing Surveys*, 49(1):3:1–3:36, April 2016.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP 1998*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL 2008*, pages 273–284. ACM Press, 2008.

[HYC16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of the ACM*, 63(1):9, 2016.

[LBW05] Jay Ligatti, Lujo Bauer, and David Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[LON+13] Leonardo A. F. Leite, Gustavo Ansaldi Oliva, Guilherme M. Nogueira, Marco Aurélio Gerosa, Fabio Kon, and Dejan S. Milojicic. A Systematic Literature Review of Service Choreography Adaptation. *Service Oriented Computing and Applications*, 7(3):199–216, 2013.

[ML00] Andrew C. Myers and Barbara Liskov. Protecting Privacy using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology*, 9:410–442, 2000.

[Pad11] Luca Padovani. Session Types = Intersection Types + Union Types. In *ITRS 2010*, volume 45 of *EPTCS*, pages 71–89. Open Publishing Association, 2011.

[PCF09] Jérémy Planul, Ricardo Corin, and Cédric Fournet. Secure Enforcement for Global Process Specifications. In *CONCUR 2009*, volume 5710 of *LNCS*, pages 511–526. Springer, 2009.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PS03] François Pottier and Vincent Simonet. Information Flow Inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.

[RSC09] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking Information Flow in Dynamic Tree Structures. In *ESORICS 2009*, volume 5789 of *LNCS*, pages 86–103. Springer, 2009.

[Sch00] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[SM03] Andrew Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[SR10] A. Sabelfeld and A. Russo. From Dynamic to Static and Back: Riding the Roller Coaster of Information-flow Control Research. In *PSI 2009*, volume 5947 of *LNCS*, pages 352–365. Springer, 2010.

[ZM07] Lantian Zheng and Andrew C. Myers. Dynamic Security Labels and Static Information Flow Control. *International Journal of Information Security*, 6:67–84, 2007.