Information flow safety in multiparty sessions

Under consideration for publication in Math. Struct. in Comp. Science

Information Flow Safety in Multiparty Sessions[†]

Dedicated to the Memory of Kohei Honda

Sara Capecchi, Università di Torino, capecchi@di.unito.it

Ilaria Castellani, INRIA, ilaria.castellani@inria.fr

Mariangiola Dezani-Ciancaglini, Università di Torino, dezani@di.unito.it

Received 17 January 2014

We consider a calculus for multiparty sessions enriched with security levels for messages. We propose a monitored semantics for this calculus, which blocks the execution of processes as soon as they attempt to leak information. We illustrate the use of this semantics with various examples, and show that the induced safety property is compositional and that it is strictly included between a typability property and a security property proposed for an extended calculus in previous work.

Keywords: concurrency, session calculi, secure information flow, monitored semantics, safety, session types.

1. Introduction

With the advent of web technologies, we are faced today with a powerful computing environment which is inherently parallel, distributed and heavily relies on communication. Since computations take place concurrently on several heterogeneous devices, controlled by parties which possibly do not trust each other, security properties such as confidentiality and integrity of data become of crucial importance.

A *session* is an abstraction for various forms of "structured communication" that may occur in a parallel and distributed computing environment. Examples of sessions are a client-service negotiation, a financial transaction, or an interaction among different services within a web application. *Session types*, which specify the expected behaviour of participants in sessions, were originally introduced in (Honda, 1993; Takeuchi et al., 1994; Honda et al., 1998), on a variant of the π-calculus (Milner, 1999) including a construct for session creation and two n-ary operators of internal and external choice, called selection and branching. The basic properties ensured by session types are the absence of communication errors (communication safety) and the conformance to the session protocol (session fidelity). Since then, more powerful *session calculi* have been investigated, allowing multiparty interaction within a session (Honda et al., 2008),

[†] Work partially funded by the ANR-08-EMER-010 grant PARTOUT, by the EU Collaborative project ASCENS 257414, by the ICT COST Action IC1201 BETTY, by the MIUR PRIN Project CINA Prot. 2010LHT4KM and by the Torino University/Compagnia San Paolo Project SALT.

and equipped with increasingly sophisticated session types, ensuring additional properties like progress (Bettini et al., 2008; Coppo et al., 2014).

In previous work (Capecchi et al., 2010) and in its expanded version (Capecchi et al., 2013), we addressed the question of incorporating security requirements into session types. To this end, we considered a calculus for multiparty sessions enriched with security levels for both session participants and data, and we defined a notion of *security* for this calculus. We then proposed a session type system for this calculus, adding access control and secure information flow requirements in the typing rules, and we showed that this type system guarantees our security property (preservation of data *confidentiality*) during session execution.

In this paper, we move one step further by equipping (a slightly simplified version of) the above calculus with a *monitored semantics*, which blocks the execution of processes as soon as they attempt to leak information, raising an error. Typically, this happens when a process tries to participate in a public communication after receiving a secret value. This monitored semantics induces a natural notion of *safety* on processes: a process is safe if all its monitored computations are successful, in a dynamically evolving environment and in the presence of an attacker which may add or withdraw messages at each step. Expectedly, this monitored semantics is closely related to the security type system presented in (Capecchi et al., 2013). Indeed, some of the constraints imposed by the monitored operational rules are simply lifted from the typing rules. However, the constraints of the monitoring semantics are simpler than those of the typing rules, since they refer to individual computations. In other words, they are *local* whereas type constraints are both local and global. We also introduce a refinement of the security property of (Capecchi et al., 2013), which allows the exploration of the behaviour of waiting session participants "ahead of time". This look ahead capacity seems needed when the recourse to a static analysis (like a session type system) to check the conformance of potential session participants to the session protocol and security policy is either not desired or not possible.

Other advantages of safety over typability are not specific to session calculi. Like security, safety is a semantic notion. Hence it is *more permissive* than typability in that it ignores unreachable parts of processes. Safety also offers more *flexibility* than types in dynamic environments such as that of web programming, where security policies may evolve over time.

Compared to security, safety has again the advantage of locality versus globality: safety is a property of individual computations, while security is a property of the set of computations of a process. In session calculi, it also improves on security in another respect. Indeed, in these calculi processes communicate asynchronously and messages transit in queues before being consumed by their receivers. Then, while the monitored semantics blocks the very act of putting a public message in the queue after a secret message has been received, a violation of the security property can only be detected after the public message has been put in the queue, that is, after the confidentiality breach has occurred and possibly already caused damage. This means that safety allows *early detection* of leaks, whereas security only allows late leak detection.

Finally, safety seems more appealing than security when the dangerous behaviour comes from an accidental rather than a malicious transgression of the security policy. Indeed, in this case a monitored semantics could offer feedback to the programmer, in the form of informative error messages. Although this possibility is not explored in this paper, it is the object of ongoing work.

The main contribution of this work is a monitored semantics for a multiparty session calculus, and the proof that:

1 the induced *information flow safety* property strictly implies (a refined version of) our previously defined *information flow security* property;
2 a simplified version of our previously defined *type system*, adapted to the present calculus, strictly implies the induced information flow safety.

A more general contribution of our work is to raise the question of information flow safety in session calculi. Indeed, while the issue of safety has recently received much attention in the security community (see Section 10), it has not, to our knowledge, been addressed in the context of session calculi so far.

The rest of the paper is organised as follows. In Section 2 we motivate our approach with an example. Section 3 introduces the syntax and semantics of our calculus. In Section 4 we present our refined security property and illustrate it with examples. Section 5 presents the monitored semantics and Section 6 introduces the notion of safety and establishes its compositionality. The relation between safety and security is examined in Section 7. Section 8 presents our simplified type system and Section 9 shows that it implies safety. Finally, Section 10 concludes with a discussion on related and future work.

This work is an extended version of (Capecchi et al., 2011), with complete proofs. It introduces refined versions of the safety and security properties examined in that paper and provides two additional results: compositionality of (the refined) safety property, and the proof that this property is assured by a simplified version of the type system of (Capecchi et al., 2013).

2. Motivating example

Let us illustrate our approach with an introductory example, modelling an online medical system. Consider a ternary protocol involving a user `U`, a generic medical service `S-gen` and a specialised medical service `S-spe`. The interaction of the user with the generic service is supposed to be less confidential than her interaction with the specialised service. For simplicity we assume the former to have level $\bot$ and the latter to have level $\bot$ or $\top$ according to the privacy of the exchanged data (here $\bot$ means "public" and $\top$ means "secret" or "private") [†]. The user addresses the generic service first, describing her symptoms and getting back a prescription for a specialised visit. All this interaction is public, so it takes place at level $\bot$. Then the user sends her symptoms to the specialised service and waits for a diagnosis. The symptoms are again sent publicly, but the diagnosis needs to be kept confidential, so it should be communicated at level $\top$. Now, according to whether the diagnosis is critical or not, the user asks either to be admitted to hospital or to be prescribed some drug treatment. The specialised service answers by sending the name of a hospital or a drug prescription, according to the user's request. For the protocol to be secure, these last interactions should take place at level $\top$, since they depend on the private diagnosis.

More precisely, the whole interaction may be described by the following protocol:

1 a connection is established among `U`, `S-gen` and `S-spe`;
2 `U` sends her symptoms to `S-gen`, who sends back a visit prescription;

[†] In a real-life protocol, the interaction of the user with the generic service would have some level ℓ higher than $\bot$, but for simplicity we consider only two levels here. Also, a practical medical system should include more than one specialised service, namely one for each main category of diseases.

$$
\begin{array}{lcl}
\texttt{I} & = & \bar{a}[3] \\
\texttt{U} & = & a[1](\alpha_1).\ \alpha_1!\langle 2, \mathsf{symptom}^{\perp}\rangle.\alpha_1?(2, \mathit{visit\text{-}prescr}^{\perp}). \\
 & & \quad \alpha_1!\langle 3, \mathsf{symptom}^{\perp}\rangle.\alpha_1?(3, \mathit{diagn}^{\top}).\ \mathsf{if}\ \mathit{critical}(\mathit{diagn}^{\top}) \\
 & & \qquad\qquad \mathsf{then}\ \alpha_1!\langle 3, \mathsf{hosp\text{-}admiss}^{\ell}\rangle.\alpha_1?(3, \mathit{hosp\text{-}name}^{\top}).\mathbf{0} \\
 & & \qquad\qquad \mathsf{else}\ \ \alpha_1!\langle 3, \mathsf{treatment}^{\top}\rangle.\alpha_1?(3, \mathit{drug\text{-}prescr}^{\top}).\mathbf{0} \\
\texttt{S-gen} & = & a[2](\alpha_2).\ \alpha_2?(1, \mathit{symptom}^{\perp}).\alpha_2!\langle 1, \mathsf{visit\text{-}prescr}^{\perp}\rangle.\mathbf{0} \\
\texttt{S-spe} & = & a[3](\alpha_3).\alpha_3?(1, \mathit{symptom}^{\perp}).\alpha_3!\langle 1, \mathsf{diagn}^{\top}\rangle.\alpha_3?(1, \mathit{option}^{\top}).\ \mathsf{if}\ \mathit{hosp}(\mathit{option}^{\top}) \\
 & & \qquad\qquad \mathsf{then}\ \alpha_3!\langle 1, \mathsf{hosp\text{-}name}^{\top}\rangle.\mathbf{0} \\
 & & \qquad\qquad \mathsf{else}\ \ \alpha_3!\langle 1, \mathsf{drug\text{-}prescr}^{\top}\rangle.\mathbf{0}
\end{array}
$$

Fig. 1. The medical service protocol.

3 U sends her symptoms to S-spe, who sends back a diagnosis;
4 U has two possible options according to the seriousness of the diagnosis:
  4.1 U asks S-spe for hospital admission and waits for a hospital name from S-spe;
  4.2 U requests a treatment to S-spe, who answers with a drug prescription.

In our calculus, this scenario may be described as the parallel composition of the processes I, U, S-gen and S-spe in Figure 1, where the level ℓ of $\mathsf{hosp\text{-}admiss}^{\ell}$ in U is deliberately left unspecified, as we wish to discuss two variants of the protocol, a secure one and an insecure one.

A *session* is an activation of a service, involving a number of participants with predefined roles. Here processes U, S-gen and S-spe communicate by opening a session on service a. The initiator $\bar{a}[3]$ specifies the number of participants of a and triggers a session as soon as there is a participant for each role. Participants are denoted by integers: here U=1, S-gen=2, S-spe=3. In process U, the prefix $a[1](\alpha_1)$ means that U wants to act as participant 1 in service a, using channel α_1 to communicate. The meaning of $a[2](\alpha_2)$ in S-gen and $a[3](\alpha_3)$ in S-spe is similar.

Security levels appear as superscripts on data and variables[‡]. When the session is established, via a synchronisation between the initiator and the prefixes $a[i](\alpha_i)$, U sends a public message to S-gen (prefix $\alpha_1!\langle 2, \mathsf{symptom}^{\perp}\rangle$) and waits for a public answer (prefix $\alpha_1?(2, \mathit{visit\text{-}prescr}^{\perp})$). Then U sends the same message to S-spe and waits for a secret answer (prefix $\alpha_1?(3, \mathit{diagn}^{\top})$). Depending on whether the diagnosis is critical or not, U issues two different requests to S-spe: either a request of level ℓ for hospital admission (prefix $\alpha_1!\langle 3, \mathsf{hosp\text{-}admiss}^{\ell}\rangle$), or a secret request for a drug treatment (prefix $\alpha_1!\langle 3, \mathsf{treatment}^{\top}\rangle$). We let the reader figure out the rest of the interaction between U and S-spe, to focus now on the issue of safety and security.

Corresponding to the two possibilities for the level ℓ in $\alpha_1!\langle 3, \mathsf{hosp\text{-}admiss}^{\ell}\rangle$, we obtain two different versions of the protocol, one of which is safe and secure, while the other is neither safe nor secure. In case $\ell = \top$, the protocol is secure since a public observer will see no difference between the two branches of the conditional (in both cases he will observe the empty behaviour). It is also safe since no computation exhibits a "level drop" from an input or test of level $\top$ to a following action of level $\perp$. Indeed, the monitored semantics records the level of inputs and tests, and checks them again subsequent actions to prevent level drops. Instead, if $\ell = \perp$, the

[‡] Levels on operators, which are needed to track *indirect flows* (as explained in Section 5), are omitted in this example.

first branch of the conditional has a level drop, and the monitored semantics will block before executing the request that would violate safety. This version of the protocol is also insecure, since two different behaviours can be observed by a public observer after testing the secret diagnosis: in one case the emission of $\mathsf{hosp\text{-}admiss}^{\perp}$, in the other case the empty behaviour.

To sum up, for $\ell = \top$ the protocol is safe and secure, while for $\ell = \perp$ it is both unsafe and insecure. This is what we expect, since by asking *publicly* for hospital admission, U accidentally reveals some information about her diagnosis, namely that it is critical.

Let us note that the insecure version of the protocol is also rejected by the type system of (Capecchi et al., 2013), which statically ensures the correction of all possible executions. Moreover, this example illustrates the fact that safety ensures early leak detection, while security only allows late leak detection. Indeed, the bisimulation used to check security will fail only once $\mathsf{hosp\text{-}admiss}^{\perp}$ has been put in the queue, and thus possibly exploited by an attacker. By contrast, the monitored semantics will block the very act of putting $\mathsf{hosp\text{-}admiss}^{\perp}$ in the queue.

For the sake of conciseness, we deliberately simplified the scenario in the above example. A more realistic example would involve persistent services and allow several users to interact with them. Our simple example is mainly meant to highlight the novel issue of monitored execution.

3. Syntax and Standard Semantics

Our calculus is a variant of that studied in (Capecchi et al., 2013): it focuses on the constructs which are meaningful for safety. For the sake of simplicity, we do not consider here access control and declassification, although their addition would not pose any problem. We also restrict the range of values that processes may exchange, omitting the send/receive of service and channel names, because our monitored semantics treats them in the same way as other values. Since service names cannot be exchanged, we do not consider the possibility of restricting them via name hiding.

Let $(\mathscr{S}, \leq)$ be a finite lattice of *security levels*, ranged over by ℓ, ℓ'. We denote by $\sqcup$ and $\sqcap$ the join and meet operations on the lattice, and by $\perp$ and $\top$ its minimal and maximal elements. We assume the following sets: *values* (booleans, integers), ranged over by $v, v' \ldots$, *value variables*, ranged over by $x, y \ldots$, *service names*, ranged over by $a, b, \ldots$, each of which has an *arity* $n \geq 2$ (its number of participants) and a security level ℓ (Example 5.2 justifies the necessity of this level), *channel variables*, ranged over by $\alpha, \beta, \ldots$, and *labels*, ranged over by $\lambda, \lambda', \ldots$ (acting like labels in labelled records). *Sessions*, the central abstraction of our calculus, are denoted with $s, s' \ldots$. A session represents a particular instance or activation of a service. Hence sessions only appear at runtime. We use $\mathtt{p}$, $\mathtt{q}$,... to denote the *participants* of a session. In an n-ary session (a session corresponding to an n-ary service) $\mathtt{p}$, $\mathtt{q}$ are assumed to range over the natural numbers $1, \ldots, n$. We denote by Π a non empty set of participants. Each session s has an associated set of *channels with role* $s[\mathtt{p}]$, one for each participant. Channel $s[\mathtt{p}]$ is the private channel through which participant $\mathtt{p}$ communicates with the other participants in the session s. A new session s on an n-ary service a^{ℓ} is opened when the *initiator* $\bar{a}^{\ell}[n]$ of the service synchronises with n processes of the form $a^{\ell}[1](\alpha_1).P_1, \ldots, a^{\ell}[n](\alpha_n).P_n$, whose channels $\alpha_{\mathtt{p}}$ then get replaced by $s[\mathtt{p}]$ in the body of $P_{\mathtt{p}}$. The use of an initiator to start a new session was first proposed in our previous work (Capecchi et al. 2010). The idea is that, while binary sessions may be viewed as an interaction between a client and a server, which is naturally started by the client, in a multiparty session there

c	::=	$\alpha \mid s[\mathtt{p}]$	Channel	P	::=	$\tilde{a}^{\ell}[n]$	Session init.
v	::=	$\mathsf{true} \mid \mathsf{false} \mid \ldots$	Value		$\mid$	$a^{\ell}[\mathtt{p}](\alpha).P$	p-th session part.
e	::=	$x^{\ell} \mid v^{\ell} \mid \mathsf{not}\ e$			$\mid$	$c!\langle \Pi, e\rangle.P$	Output
	$\mid$	$e\ \mathsf{and}\ e' \mid \ldots$	Expression		$\mid$	$c?(\mathtt{p}, x^{\ell}).P$	Input
D	::=	$X(x,\alpha) = P$	Declaration		$\mid$	$c \oplus^{\ell} \langle \Pi, \lambda\rangle.P$	Selection
Π	::=	$\{\mathtt{p}\} \mid \Pi \cup \{\mathtt{p}\}$	Set of participants		$\mid$	$c\&^{\ell}(\mathtt{p}, \{\lambda_i : P_i\}_{i\in I})$	Branching
					$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	Conditional
ϑ	::=	$v^{\ell} \mid \lambda^{\ell}$	Message content		$\mid$	$P \mid Q$	Parallel
					$\mid$	$\mathbf{0}$	Inaction
m	::=	$(\mathtt{p}, \Pi, \vartheta)$	Message		$\mid$	$\mathsf{def}\ D\ \mathsf{in}\ P$	Recursion
h	::=	$m \cdot h \mid \varepsilon$	Queue		$\mid$	$X\langle e, c\rangle$	Process call
H	::=	$H \cup \{s : h\} \mid \emptyset$	**Q**-set	C	::=	$< P\ ,\ H >$	Basic config.
					$\mid$	$C \| C$	Config. parallel
					$\mid$	$(\nu s)C$	Session name hiding

Table 1. *Syntax of processes, expressions and queues.*

is no such natural choice of a starting participant. The participants are on an equal footing, and it seems preferable to add a separate initiator than to arbitrarily select one of the peer participants as the starting one (as it is done in other work on multiparty sessions). Moreover, although this is not exploited here, the presence of the initiator somehow indicates the availability of the service, and in more complex calculi one could envisage more refined forms for the initiator (for instance, in case of unavailability of the service, the initiator could act as a forwarder to a substitute service). We use c to range over channel variables and channels with roles. Finally, we assume a set of *process variables* $X, Y, \ldots$, in order to define recursive behaviours.

As in (Honda et al., 2008), in order to model TCP-like asynchronous communications (with non-blocking send but message order preservation between a given pair of participants), we use *queues of messages*, denoted by h; an element of h may be a value message $(\mathtt{p}, \Pi, v^{\ell})$, indicating that the value v is sent by participant $\mathtt{p}$ to all participants in Π, or a label message $(\mathtt{p}, \Pi, \lambda^{\ell})$, indicating that $\mathtt{p}$ selects the process with label λ among those offered by the set of participants Π. The empty queue is denoted by ε, and the concatenation of a message m to a queue h by $h \cdot m$. Conversely, $m \cdot h$ means that m is the head of the queue. Since there may be interleaved, nested and parallel sessions, we distinguish their queues with names. We denote by $s : h$ the *named queue* h associated with session s. We use H, K to range over sets of named queues with different session names, also called **Q**-sets.

Table 1 summarises the syntax of *expressions*, ranged over by $e, e', \ldots$, and of *processes*, ranged over by $P, Q \ldots$, as well as the *runtime syntax* of the calculus (session names, channels with role, messages, queues). A *user process* is a process generated without using runtime syntax.

Let us briefly comment on the primitives of the language. We already described session initiation. Communications within a session are performed on channels with role using the next two pairs of primitives: the send and receive of a value and the selection and branching operators (where one participant chooses one of the branches offered by another participant). Choice primitives are decorated with security levels, whose use will be justified in Example 5.3. The variables in the process declarations do not have levels since they are not necessary, contrary to what happens for the variables in inputs (see Example 4.11). When there is no risk of confusion we will omit the set delimiters $\{,\}$, particularly around singletons.

$$P \mid \mathbf{0} \equiv P \quad P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \quad \mathsf{def}\ D\ \mathsf{in}\ \mathbf{0} \equiv \mathbf{0}$$
$$(\mathsf{def}\ D\ \mathsf{in}\ P) \mid Q \equiv \mathsf{def}\ D\ \mathsf{in}\ (P \mid Q) \quad \mathsf{def}\ D\ \mathsf{in}\ (\mathsf{def}\ D'\ \mathsf{in}\ P) \equiv \mathsf{def}\ D'\ \mathsf{in}\ (\mathsf{def}\ D\ \mathsf{in}\ P)$$

$$h \cdot (\mathtt{p}, \Pi, \vartheta) \cdot (\mathtt{p}', \Pi', \vartheta') \cdot h' \equiv h \cdot (\mathtt{p}', \Pi', \vartheta') \cdot (\mathtt{p}, \Pi, \vartheta) \cdot h' \quad \text{if } \Pi \cap \Pi' = \emptyset \text{ or } \mathtt{p} \neq \mathtt{p}'$$
$$h \cdot (\mathtt{p}, \Pi, \vartheta) \cdot h' \equiv h \cdot (\mathtt{p}, \Pi', \vartheta) \cdot (\mathtt{p}, \Pi'', \vartheta) \cdot h' \quad \text{if } \Pi = \Pi' \cup \Pi'' \text{ and } \Pi' \cap \Pi'' = \emptyset$$
$$H \equiv H' \text{ and } h \equiv h' \quad \Rightarrow \quad H \cup \{s : h\} \equiv H' \cup \{s : h'\}$$

$$P \equiv Q \text{ and } H \equiv K \quad \Rightarrow \quad < P\ ,\ H > \equiv < Q\ ,\ K >$$
$$C \equiv C' \quad \Rightarrow \quad (\nu s)C \equiv (\nu s)C' \quad (\nu ss')C \equiv (\nu s's)C$$
$$(\nu \tilde{s}) < P\ ,\ H > \| (\nu \tilde{s}') < Q\ ,\ K > \equiv (\nu \tilde{s}\tilde{s}') < P \mid Q\ ,\ H \cup K > \quad \text{if } \mathrm{qn}(H) \cap \mathrm{qn}(K) = \emptyset$$

Table 2. *Structural equivalence.*

$$a^{\ell}[1](\alpha_1).P_1 \mid \ldots \mid a^{\ell}[n](\alpha_n).P_n \mid \bar{a}^{\ell}[n] \longrightarrow (\nu s) < P_1\{s[1]/\alpha_1\} \mid \ldots \mid P_n\{s[n]/\alpha_n\}\ ,\ s : \varepsilon > \quad \text{[Link]}$$

$$< s[\mathtt{p}]!\langle \Pi, e \rangle.P\ ,\ s : h > \longrightarrow < P\ ,\ s : h \cdot (\mathtt{p}, \Pi, v^{\ell}) > \quad \text{where } e{\downarrow}v^{\ell} \quad \text{[Send]}$$

$$< s[\mathtt{q}]?(\mathtt{p}, x^{\ell}).P\ ,\ s : (\mathtt{p}, \mathtt{q}, v^{\ell}) \cdot h > \longrightarrow < P\{v/x\}\ ,\ s : h > \quad \text{[Rec]}$$

$$< s[\mathtt{p}] \oplus^{\ell} \langle \Pi, \lambda \rangle.P\ ,\ s : h > \longrightarrow < P\ ,\ s : h \cdot (\mathtt{p}, \Pi, \lambda^{\ell}) > \quad \text{[Label]}$$

$$< s[\mathtt{q}] \&^{\ell} (\mathtt{p}, \{\lambda_i : P_i\}_{i \in I})\ ,\ s : (\mathtt{p}, \mathtt{q}, \lambda_{i_0}^{\ell}) \cdot h > \longrightarrow < P_{i_0}\ ,\ s : h > \quad \text{where } i_0 \in I \quad \text{[Branch]}$$

$$\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow P \quad \text{where } e \downarrow \mathsf{true}^{\ell} \qquad \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow Q \quad \text{where } e \downarrow \mathsf{false}^{\ell} \quad \text{[If-T, If-F]}$$

$$\mathsf{def}\ X(x, \alpha) = P\ \mathsf{in}\ (X\langle e, s[\mathtt{p}] \rangle \mid Q) \longrightarrow \mathsf{def}\ X(x, \alpha) = P\ \mathsf{in}\ (P\{v^{\ell}/x\}\{s[\mathtt{p}]/\alpha\} \mid Q) \quad \text{where } e \downarrow v^{\ell} \quad \text{[Def]}$$

$$< P\ ,\ H > \longrightarrow (\nu \tilde{s}) < P'\ ,\ H' > \quad \Rightarrow \quad < \mathscr{E}[P]\ ,\ H > \longrightarrow (\nu \tilde{s}) < \mathscr{E}[P']\ ,\ H' > \quad \text{[Cont]}$$

$$C \longrightarrow (\nu \tilde{s})C' \quad \Rightarrow \quad (\nu \tilde{s}')(C \| C'') \longrightarrow (\nu \tilde{s}')(\nu \tilde{s})(C' \| C'') \quad \text{[Scop]}$$

$$C \equiv C' \text{ and } C' \longrightarrow C'' \text{ and } C'' \equiv C''' \quad \Rightarrow \quad C \longrightarrow C''' \quad \text{[Struct]}$$

Table 3. *Standard reduction rules.*

The operational semantics is defined on configurations $< P\ ,\ H >$, which are pairs of a process P and a $\mathbf{Q}$-set H. Indeed, in our calculus queues need to be isolated from processes (unlike in other session calculi, where queues are handled by running them in parallel with processes), since they will constitute the observable part of processes in our security and safety notions.

Formally, a *configuration* is a pair $C = < P\ ,\ H >$, possibly restricted with respect to session names, or a parallel composition $(C \| C')$ of two configurations whose $\mathbf{Q}$-sets have disjoint session names. In a configuration $(\nu s) < P\ ,\ H >$, all occurrences of $s[\mathtt{p}]$ in P and H are bound, as well as those of s in H. By abuse of notation we often write P instead of $< P\ ,\ \emptyset >$.

As usual, the reduction relation is defined modulo a structural equivalence $\equiv$. The rules for $\equiv$ are given in Table 2. We use $\mathrm{qn}(H)$ for the queue names of H. Assuming Barendregt convention, no bound name can occur free or in two different bindings. The structural rules for processes are standard (Milner, 1999). Among the rules for queues, we have one for commuting independent messages and another one for splitting a message for multiple recipients. The structural equivalence of configurations allows the parallel composition $\|$ to be eliminated so that, modulo

$\equiv$, each configuration has the form $(\nu\tilde{s}) < P\ ,\ H >$, where $(\nu\tilde{s})C$ stands for $(\nu s_1)\cdots(\nu s_k)C$ if $\tilde{s} = s_1 \cdots s_k$ and for C if $\tilde{s}$ is empty.

The transitions for configurations have the form $(\nu\tilde{s}) < P\ ,\ H > \longrightarrow (\nu\tilde{s'}) < P'\ ,\ H' >$. They are derived using the reduction rules in Table 3. Let us comment on the most interesting rules.

Rule [Link] describes the initiation of a new session among n processes, corresponding to an activation of the service a of arity n and security level ℓ. After the connection, the participants share a private session name s and the corresponding queue, initialised to $s : \varepsilon$. In each participant P_{p}, the channel variable α_{p} is replaced by the channel with role $s[\mathrm{p}]$. Session initiation is the only synchronous interaction of the calculus. All the other communications, which take place within an established session, are performed asynchronously in two steps, via push and pop operations on the queue associated with the session.

The output rules [Send] and [Label] push values and labels, respectively, into the queue $s : h$. In rule [Send], $e \downarrow v^{\ell}$ denotes the evaluation of the expression e to the value v^{ℓ}, where ℓ is the join of the security levels of the variables and values occurring in e. The input rules [Rec] and [Branch] perform the complementary operations. Rules [If-T], [If-F], [Def] and [Cont] are standard.

The evaluation contexts $\mathscr{E}$ used in Rule [Cont] are defined by:

$$\mathscr{E} ::= [\,] \mid \mathsf{def}\ D\ \mathsf{in}\ \mathscr{E} \mid \mathscr{E} \mid P \mid P \mid \mathscr{E}$$

The contextual rule [Scop] for configurations is also standard. In this rule, Barendregt convention ensures that the names in $\tilde{s}$ are disjoint from those in $\tilde{s'}$ and do not appear in C''. As usual, we use $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

4. Security

We introduce now our property of *information flow security* for processes (referred to simply as security in the sequel), which is a refined version of that proposed in (Capecchi et al., 2011) for an extension of the present calculus, discussed at the beginning of Section 3. As will be argued later in some depth, this refinement is interesting in that it leads to a more discriminating security property, which is closer to that of typability. Notably our main result on security, which states that safety implies security (Theorem 7.8), also holds for the original definition of security.

Our security property is essentially *noninterference*, adapted to our session calculus. We introduce it by an informal discussion, which is largely recalled from (Capecchi et al., 2013).

We are looking for a security property which is *persistent* in the sense of (Bossi et al., 2004), namely which holds in any reachable state of a process, assuming the process may be restarted with fresh $\mathbf{Q}$-sets in any state. This means that we view processes as evolving in a dynamic and potentially hostile environment, where at each step an attacker/observer may inject new messages or consume existing messages, in order to discover the secret data used by the processes.

In our calculus, an insecure information flow – or *information leak* – occurs when an action of level ℓ' depends on the occurrence or the value of an input action of level $\ell \not\leq \ell'$.

A typical information leak occurs when a low action is guarded by a high input, as in the following process P, where s is some previously opened session with two participants:

$$(*) \qquad P = s[1]?(2, x^{\top}).s[1]!\langle 2, e\rangle.\mathbf{0} \qquad \text{where } e{\downarrow}v^{\perp}$$

In process P, the first participant of session s waits for a message from the second participant, and then she replies by sending to the second participant the value of a low expression e.

This process is insecure because the occurrence of the high input depends on whether or not a matching message is offered by the high environment, which is something we do not control. Since input is a blocking operator, the low output will be able to occur in the first case but not in the latter, and the observer will conclude that the low output depends on the high input. Note that in process P the value $v^{\bot}$ of the expression e cannot depend on the value received for $x^{\top}$, since in this case e should contain $x^{\top}$ and therefore its value could not have level $\bot$[§]. Hence the leak in P is only due to the *presence* or *absence* of a $\top$-message in the environment, not on its value.

Another typical leak occurs when a high input guards a high conditional which tests the received value in order to choose between two different low behaviours in its branches. Consider the following process:

$$(**) \qquad Q = s[1]!\langle 2, \mathsf{true}^{\top}\rangle.\mathbf{0} \mid s[2]?(1, x^{\top}).\mathsf{if}\ x^{\top}\ \mathsf{then}\ s[2]!\langle 3, \mathsf{true}^{\bot}\rangle.\mathbf{0}\ \mathsf{else}\ s[2]!\langle 3, \mathsf{false}^{\bot}\rangle.\mathbf{0}$$

Here session s has three participants. The second participant may input the message sent by the first participant, and depending on its value (which can be changed by an attacker between the asynchronous emission and reception) she will send two different low values to the third participant. This process is insecure because the value of the low message sent by participant 2 to participant 3 depends on the value of the high message received by participant 2 from participant 1. Here the leak is caused by the *value* of a high message rather than by its presence or absence.

As a matter of fact, the Example $(**)$ above is not completely correct, since in our persistent notion of security we assume that the attacker can add or withdraw high messages at each step. Hence, after participant 1 has sent her message to participant 2, this message could again disappear from the queue, and participant 2 would be blocked, just as participant 1 in Example $(*)$. However, Example $(**)$ can be easily fixed by rendering both components of process Q recursive, so that the high message for participant 2 is continuously offered by participant 1.

This is the reason why most examples in this section will be composed of recursive processes. We will also make sure that all our examples are well-behaved with respect to session protocols, in the sense that they are typable by means of classical session types (this means for instance that if a participant is recursive, then the matching participants must be recursive too). Indeed, interestingly enough, session type systems already rule out a number of insecure processes. This is why we shall focus on the insecurities that can appear in session-typable processes.

As in (Capecchi et al., 2013), we assume that the observer can see the messages in session queues. As usual for security, observation is relative to a given downward-closed set of levels $\mathscr{L} \subseteq \mathscr{S}$, the intuition being that an observer who can see messages of level ℓ can also see all messages of level ℓ' lower than ℓ. In the following, we shall always use $\mathscr{L}$ to denote a downward-closed subset of levels. For any such $\mathscr{L}$, an $\mathscr{L}$-observer will only be able to see messages whose levels belong to $\mathscr{L}$, what we may call $\mathscr{L}$*-messages*. Hence two queues that agree on $\mathscr{L}$-messages will be indistinguishable for an $\mathscr{L}$-observer. Let now $\overline{\mathscr{L}}$*-messages* be the complementary messages, those the $\mathscr{L}$-observer cannot see. Then, an $\mathscr{L}$-observer may also be viewed as an attacker who tries to reconstruct the dependency between $\overline{\mathscr{L}}$-messages and $\mathscr{L}$-messages (and hence, ulti-

[§] Recall that the level of an expression is the join of the levels of the variables and values occurring in it.

mately, to discover the $\overline{\mathscr{L}}$-messages), by injecting or withdrawing $\overline{\mathscr{L}}$-messages at each step and observing the effect thus produced on $\mathscr{L}$-messages.

To formalise this intuition, a notion of $\mathscr{L}$-equality $=_{\mathscr{L}}$ on **Q**-sets is introduced, representing indistinguishability of **Q**-sets by an $\mathscr{L}$-observer. Using $=_{\mathscr{L}}$, we then define a notion of $\mathscr{L}$-bisimulation $\simeq_{\mathscr{L}}$ representing indistinguishability of processes by an $\mathscr{L}$-observer. Formally, a queue $s : h$ is $\mathscr{L}$-observable if it contains some message with level in $\mathscr{L}$. Note that this means that empty queues are never observable, even when they are associated with a service of level in $\mathscr{L}$. Then two **Q**-sets are *$\mathscr{L}$-equal* if their $\mathscr{L}$-observable queues have the same names and contain the same messages with level in $\mathscr{L}$. This equality is based on an $\mathscr{L}$-projection operation on **Q**-sets, which discards all messages whose level is not in $\mathscr{L}$.

Let the function *lev* be given by: $lev(v^{\ell}) = lev(\lambda^{\ell}) = \ell$.

Definition 4.1 ($\mathscr{L}$-Projection). The projection operation $\Downarrow \mathscr{L}$ is defined inductively on messages, queues and **Q**-sets as follows:

$$(\mathrm{p},\Pi,\vartheta) \Downarrow \mathscr{L} = \begin{cases} (\mathrm{p},\Pi,\vartheta) & \text{if } lev(\vartheta) \in \mathscr{L}, \\ \varepsilon & \text{otherwise} \end{cases} \qquad \begin{array}{l} \varepsilon \Downarrow \mathscr{L} = \varepsilon \\ (m \cdot h) \Downarrow \mathscr{L} = m \Downarrow \mathscr{L} \cdot h \Downarrow \mathscr{L} \end{array}$$

$$\emptyset \Downarrow \mathscr{L} = \emptyset \qquad (H \cup \{s : h\}) \Downarrow \mathscr{L} = \begin{cases} H \Downarrow \mathscr{L} \cup \{s : h \Downarrow \mathscr{L}\} & \text{if } h \Downarrow \mathscr{L} \neq \varepsilon, \\ H \Downarrow \mathscr{L} & \text{otherwise} \end{cases}$$

Definition 4.2 ($\mathscr{L}$-Equality of Q-sets). Two **Q**-sets H and K are *$\mathscr{L}$-equal*, written $H =_{\mathscr{L}} K$, if $H \Downarrow \mathscr{L} = K \Downarrow \mathscr{L}$.

Let us move now to the notion $\mathscr{L}$-bisimulation. The idea is to test processes by running them in conjunction with $\mathscr{L}$-equal queues. However, as argued in (Capecchi et al., 2013), we cannot allow arbitrary combinations of processes with queues, since this would lead us to reject intuitively secure processes as simple as $s[1]!\langle 2, \mathsf{true}^{\perp}\rangle.\mathbf{0}$ and $s[2]?(1, x^{\perp}).\mathbf{0}$, as explained next.

For example, the process $P = s[1]!\langle 2, \mathsf{true}^{\perp}\rangle.\mathbf{0}$ does not react in the same way when combined with the two **Q**-sets $H_1 = \{s : \varepsilon\} =_{\perp} \emptyset = H_2$. Indeed, while P reduces when combined with $\{s : \varepsilon\}$, it is stuck when combined with $\emptyset$. Formally:

$$< P \ , \ \{s : \varepsilon\} > \longrightarrow < \mathbf{0} \ , \ \{s : (1, 2, \mathsf{true}^{\perp})\} > \qquad < P \ , \ \emptyset > \not\longrightarrow$$

Therefore an $\mathscr{L}$-observer with $\mathscr{L} = \{\perp\}$ would detect an insecurity by looking at the resulting **Q**-sets $H_1' = \{s : (1, 2, \mathsf{true}^{\perp})\} \neq_{\perp} \emptyset = H_2'$.

To get around this problem, a natural requirement is that the **Q**-sets always contain enough queues to enable all outputs of the process to reduce. For this it suffices that in a configuration $< P \ , \ H >$, every session name which occurs free in P has a corresponding queue in H. We use $\mathrm{sn}(P)$ (resp. $\mathrm{fsn}(P)$) to denote the set of session names (resp. free session names) in P.

Definition 4.3. A configuration $\langle P, H\rangle$ is *saturated* if all session names in P are unrestricted and have corresponding queues in H, i.e. $s \in \mathrm{sn}(P)$ implies $s \in \mathrm{fsn}(P)$ and $s \in \mathrm{qn}(H)$.

Since the only rule that creates a new session is [Link], which simultaneously creates the corresponding queue, it is easy to check that starting from a closed user process and the empty **Q**-set, we always obtain configurations $(\nu \tilde{s})\langle P, H\rangle$ such that $\langle P, H\rangle$ is saturated.

A dual phenomenon occurs with inputs. Consider the process Q with $\mathbf{Q}$-sets $H_1 =_\perp H_2$:

$$Q = s[2]?(1,x^\perp).\mathbf{0} \qquad H_1 = \{s:(1,2,\mathsf{true}^\perp)\} \qquad H_2 = \{s:(1,2,\mathsf{true}^\top)\cdot(1,2,\mathsf{true}^\perp)\}$$

Note that $< Q\ ,\ H_1 > \longrightarrow < \mathbf{0}\ ,\ \{s:\varepsilon\} >$, while $< Q\ ,\ H_2 > \not\longrightarrow$, and $\{s:\varepsilon\} \neq_\perp H_2$. What happens here is that the top level message in H_2 is transparent for the $\perp$-equality, but it prevents the process from reading the subsequent bottom level message. Note that the problem comes from the fact that both messages have the same receiver and the same sender, since if we took $H_3 = \{s:(3,2,\mathsf{true}^\top)\cdot(1,2,\mathsf{true}^\perp)\}$, then we would have $< Q\ ,\ H_3 > \longrightarrow < \mathbf{0}\ ,\ \{s:(3,2,\mathsf{true}^\top)\} >$, thanks to the structural equivalence on queues given in Table 2.

We may get around this problem by imposing one simple condition on $\mathbf{Q}$-sets (*monotonicity*). Monotonicity states that in every queue of a $\mathbf{Q}$-set, the security levels of messages with the same sender and common receivers can only stay equal or increase along a sequence.

Definition 4.4 (Monotonicity). A queue is *monotone* if $lev(\vartheta_1) \leq lev(\vartheta_2)$ whenever the message $(\mathtt{p},\Pi_1,\vartheta_1)$ precedes the message $(\mathtt{p},\Pi_2,\vartheta_2)$ in the queue and $\Pi_1 \cap \Pi_2 \neq \emptyset$. A $\mathbf{Q}$-set H is monotone if every queue in H is monotone.

Note that saturation and monotonicity are constraints on the $\mathbf{Q}$-sets, which the observer must respect when experimenting on a process. Intuitively, they require that the $\mathbf{Q}$-sets chosen by the observer are "compatible" with the process, in the sense that they do not artificially block its execution. In other words, the observer is only allowed to detect existing insecurities, not to introduce new insecurities in the process.

To define our bisimulation, we need one further ingredient ¶. Note that, while the asynchronous communications among participants can be observed by looking at the changes they induce on the $\mathbf{Q}$-sets, there is no means to observe the behaviour of a potential session participant, as long as it is not triggered by the initiator together with a complete set of matching participants. This means that an "incomplete process" like $P = a^\perp[1](\alpha).\alpha!\langle 2,\mathsf{true}^\perp\rangle.\mathbf{0}$ would be equivalent to $\mathbf{0}$. However, it is clear that it behaves differently when plugged into a complying context. For instance, if we put P in parallel with the initiator and the missing partner we get:

$$< P \mid a^\perp[2](\beta).\beta?(1,x^\perp).\mathbf{0} \mid \bar{a}^\perp[2]\ ,\ \emptyset > \longrightarrow^* (\nu s)(< s[2]?(1,x^\perp).\mathbf{0}\ ,\ \{s:(1,2,\mathsf{true}^\perp)\} >)$$

while if we put $\mathbf{0}$ in the same context we get $< \mathbf{0} \mid a^\perp[2](\beta).\beta?(1,x^\perp).\mathbf{0} \mid \bar{a}^\perp[2]\ ,\ \emptyset > \not\longrightarrow$.

Since session initiation is a synchronous interaction, which is blocking for all session participants, it would not be possible to use the $\mathbf{Q}$-sets to keep track of it. We therefore introduce a notion of *tester*, a sort of behaviourless process which triggers session participants in order to reveal their potential behaviour. Our testers bear a strong analogy with the notion of test introduced by De Nicola and Hennessy in the 80's (De Nicola and Hennessy, 1983), although they are much simpler in our case, since they are only used to explore potential participant behaviours and not to test active processes. Moreover, they are not aimed at establishing a testing equivalence, but merely at rendering the bisimulation more contextual.

Formally, a *tester* is a parallel composition of session initiators and degenerate session participants, which cannot reduce by itself.

¶ Strictly speaking, this ingredient is not essential to define the bisimulation, but it leads to a more discriminating security notion than that of (Capecchi et al., 2011), closer to that of typability in the type system of Section 8.

Definition 4.5. The syntax of *pre-testers* is

$$Z ::= \bar{a}^{\ell}[n] \mid a^{\ell}[\mathrm{p}](\alpha).\mathbf{0} \mid Z \mid Z$$

A *tester* is an irreducible pre-tester.

$$(a^{\ell}[\mathrm{p}_1](\alpha_{\mathrm{p}_1}).P_{\mathrm{p}_1} \mid ... \mid a^{\ell}[\mathrm{p}_k](\alpha_{\mathrm{p}_k}).P_{\mathrm{p}_k} \mid \bar{a}^{\ell}[n]) \ \curvearrowleft \ (a^{\ell}[\mathrm{p}_{k+1}](\alpha_{\mathrm{p}_{k+1}}).\mathbf{0} \mid ... \mid a^{\ell}[\mathrm{p}_n](\alpha_{\mathrm{p}_n}).\mathbf{0}) \rightsquigarrow$$
$$(\nu s) < P_{\mathrm{p}_1}\{s[\mathrm{p}_1]/\alpha_{\mathrm{p}_1}\} \mid ... \mid P_{\mathrm{p}_k}\{s[\mathrm{p}_k]/\alpha_{\mathrm{p}_k}\} \ , \ s : \varepsilon > \qquad \text{where } k < n$$
[ExtLink1]

$$(a^{\ell}[\mathrm{p}_1](\alpha_{\mathrm{p}_1}).P_{\mathrm{p}_1} \mid ... \mid a^{\ell}[\mathrm{p}_k](\alpha_{\mathrm{p}_k}).P_{\mathrm{p}_k}) \ \curvearrowleft \ (a^{\ell}[\mathrm{p}_{k+1}](\alpha_{\mathrm{p}_{k+1}}).\mathbf{0} \mid ... \mid a^{\ell}[\mathrm{p}_n](\alpha_{\mathrm{p}_n}).\mathbf{0} \mid \bar{a}^{\ell}[n]) \rightsquigarrow$$
$$(\nu s) < P_{\mathrm{p}_1}\{s[\mathrm{p}_1]/\alpha_{\mathrm{p}_1}\} \mid ... \mid P_{\mathrm{p}_k}\{s[\mathrm{p}_k]/\alpha_{\mathrm{p}_k}\} \ , \ s : \varepsilon >$$
[ExtLink2]

$$< P \curvearrowleft Z \ , \ H > \rightsquigarrow (\nu \tilde{s}) < P' \ , \ H' > \quad \Rightarrow \quad < \mathscr{E}[P] \curvearrowleft Z \ , \ H > \rightsquigarrow (\nu \tilde{s}) < \mathscr{E}[P'] \ , \ H' > \qquad \text{[ContTest]}$$

$$< P \ , \ H > \longrightarrow (\nu \tilde{s}) < P' \ , \ H' > \quad \Rightarrow \quad < P \curvearrowleft Z \ , \ H > \rightsquigarrow (\nu \tilde{s}) < P' \ , \ H' > \qquad \text{[DropTest]}$$

Table 4. *Reduction rules for testers.*

To run processes together with testers, we introduce an asymmetric (non commutative) *triggering* operator $P \curvearrowleft Z$, whose behaviour is given by the transition relation $\rightsquigarrow$ defined by the rules in Table 4. The two "external link" rules [ExtLink1] and [ExtLink2] only differ for the presence or not of the initiator inside the process. Note the condition $k < n$ in rule [ExtLink1], which is needed since $\mathbf{0}$ is not a tester (as we will discuss further below). Rule [ContTest] is a standard context rule. Rule [DropTest] allows the process to move alone in case the tester is not needed. Let us stress that in all cases the tester Z disappears after the reduction. This is because testers are not interesting in themselves, but only insofar as they may reveal the ability of processes to contribute to a session initiation. A term $P \curvearrowleft Z$ is read "P triggered by Z".

We are now ready for defining our bisimulation. A relation $\mathscr{R}$ on processes is a $\mathscr{L}$*-bisimulation* if, whenever two related processes are put in parallel with the same tester and then coupled with $\mathscr{L}$-equal monotone $\mathbf{Q}$-sets yielding saturated configurations, then the reduction relation preserves both the relation $\mathscr{R}$ on processes and the $\mathscr{L}$-equality of $\mathbf{Q}$-sets:

Definition 4.6 ($\mathscr{L}$-Bisimulation on processes). A symmetric relation $\mathscr{R} \subseteq (\mathscr{P}r \times \mathscr{P}r)$ is a $\mathscr{L}$-*bisimulation* if $P_1 \mathscr{R} P_2$ implies, for any tester Z and any pair of monotone $\mathbf{Q}$-sets H_1 and H_2 such that $H_1 =_{\mathscr{L}} H_2$ and each $< P_i \ , \ H_i >$ is saturated:

If $< P_1 \curvearrowleft Z \ , \ H_1 > \rightsquigarrow (\nu \tilde{s}) < P_1' \ , \ H_1' >$, then either $H_1' =_{\mathscr{L}} H_2$ and $P_1' \mathscr{R} P_2$, or there exist P_2', H_2' such that $< P_2 \curvearrowleft Z \ , \ H_2 > \rightsquigarrow \cdot \longrightarrow^* (\nu \tilde{s}) < P_2' \ , \ H_2' >$, where $H_1' =_{\mathscr{L}} H_2'$ and $P_1' \mathscr{R} P_2'$.

Processes P_1, P_2 are $\mathscr{L}$*-bisimilar*, $P_1 \simeq_{\mathscr{L}} P_2$, if $P_1 \mathscr{R} P_2$ for some $\mathscr{L}$-bisimulation $\mathscr{R}$.

Note that $\tilde{s}$ is either the empty string or a fresh session name s (when the applied rule opens a new session, in which case by Barendregt convention the name s cannot occur in P_2 and H_2).

Intuitively, a transition that adds or removes a $\mathscr{L}$-message must be simulated in one or more steps, yielding a $\mathscr{L}$-equal $\mathbf{Q}$-set, whereas a transition that does not affect $\mathscr{L}$-messages (but possibly opens a new session, using or not the tester) may be simulated by inaction.

Let us illustrate the use of testers with some examples. Note that although "complete processes" may reduce autonomously without the help of any tester, we do not allow the inaction process as a tester. Indeed, its use may be simulated by any tester whose names are disjoint from those of the tested process, what we call a *fresh tester*. In fact, the quantification over Z in Definition 4.6 may be restricted in practice to those testers that make the process react, by triggering some of its waiting participants, together with one fresh tester that forces the process to progress by itself, if possible, and get stuck otherwise.

We will use $< P\ ,\ H > \longrightarrow < P'\ ,\ H' >$ as a shorthand for $< P \ \triangleleft\ Z\ ,\ H > \rightsquigarrow < P'\ ,\ H' >$, when the reduction of $P \ \triangleleft\ Z$ does not use the rules [ExtLink1] and [ExtLink2] in Table 4.

Example 4.7. Consider the three processes:

$$\begin{array}{lcl} P_1 & = & \mathbf{0} \\ P_2 & = & a^{\perp}[1](\alpha).\alpha!\langle 2, \mathsf{true}^{\perp}\rangle.\mathbf{0} \\ P_3 & = & \bar{a}^{\perp}[2] \mid a^{\perp}[1](\alpha).\alpha!\langle 2, \mathsf{true}^{\perp}\rangle.\mathbf{0} \mid a^{\perp}[2](\beta).\beta?(1, x^{\perp}).\mathbf{0} \end{array}$$

It is easy to see that no pair of P_i is in the $\perp$-bisimilarity relation. Assume in all cases the starting $\mathbf{Q}$-sets to be $\emptyset$. To distinguish P_2 from P_3, which is a complete process that can progress by itself, we may use the fresh tester $\bar{b}^{\perp}[2]$. Then this tester will just be dropped when put in parallel with P_3, and P_3 will move by itself to a state P_3', opening a new session s and generating the $\mathbf{Q}$-set $\{s : \varepsilon\}$. Since P_2 cannot move alone and the tester $\bar{b}^{\perp}[2]$ cannot make it react, P_2 will respond by staying put, thus keeping unchanged the $\mathbf{Q}$-set $\emptyset =_{\perp} \{s : \varepsilon\}$. But now, if P_2 and P_3' are tested again with $\bar{b}^{\perp}[2]$ and coupled with the $\mathbf{Q}$-set $\{s : \varepsilon\}$, then P_3' will produce the $\mathbf{Q}$-set $\{s : (1, 2, \mathsf{true}^{\perp})\}$, while P_2 will remain stuck on the $\mathbf{Q}$-set $\{s : \varepsilon\} \neq_{\perp} \{s : (1, 2, \mathsf{true}^{\perp})\}$.

To distinguish P_1 from P_2, we may use the tester $\bar{a}^{\perp}[2] \mid a^{\perp}[2](\beta).\mathbf{0}$, which produces no effect on P_1 but activates P_2, allowing it to move to a state P_2', from which a low output is possible. To distinguish P_1 from P_3 we use the same fresh tester $\bar{b}^{\perp}[2]$ and a similar reasoning.

Testers may also be used to explore some deadlocked processes, when the deadlock is caused by inverted "calls" to two different services in dual components, as shown by the next example. However, as expected, the bisimulation will not equate such a deadlocked process with a correct process where the two calls occur in the same order in the two components.

Example 4.8. Consider the following processes, where s is some previously opened session:

$$\begin{array}{lcl} P_1 & = & a^{\perp}[1](\alpha_1).b^{\perp}[1](\beta_1).s[1]!\langle 2, \mathsf{true}^{\perp}\rangle.\mathbf{0} \mid b^{\perp}[2](\beta_2).a^{\perp}[2](\alpha_2).\mathbf{0} \mid \bar{a}^{\perp}[2] \mid \bar{b}^{\perp}[2] \\ P_2 & = & a^{\perp}[1](\alpha_1).b^{\perp}[1](\beta_1).s[1]!\langle 2, \mathsf{true}^{\perp}\rangle.\mathbf{0} \mid a^{\perp}[2](\alpha_2).b^{\perp}[2](\beta_2).\mathbf{0} \mid \bar{a}^{\perp}[2] \mid \bar{b}^{\perp}[2] \end{array}$$

It is easy to see that processes P_1 and P_2 are not $\perp$-bisimilar. Indeed, P_1 is deadlocked since the service calls to a and b occur in reverse order in its two components. Instead, the calls occur in the same order in the components of P_2. Hence P_2 may progress by itself and thus it is sufficient to test it with a fresh tester, as in Example 4.7, to distinguish it from the blocked process P_1.

Note that while being deadlocked P_1 is not equated with $\mathbf{0}$, since the latter is inert while P_1

can be explored with testers that allow the sessions on services $a^{\perp}$ and $b^{\perp}$ to be opened and the following low output to be uncovered.

The notions of $\mathscr{L}$-security and security are now defined in the standard way:

Definition 4.9 (Security).

1 A process is *$\mathscr{L}$-secure* if it is $\mathscr{L}$-bisimilar with itself.
2 A process is *secure* if it is $\mathscr{L}$-secure for every $\mathscr{L}$.

Let us illustrate our security property with some examples. The simplest insecure process in our calculus is an input of level ℓ followed by an action of level $\ell' \not\geq \ell$:

Example 4.10. *(Insecurity of high input followed by low action)*
Consider the process P and the $\mathbf{Q}$-sets H_1 and H_2, where $H_1 =_{\perp} H_2$:

$$P = s[1]?(2,x^{\top}).s[1]!\langle 2,\mathsf{true}^{\perp}\rangle.\mathbf{0}, \qquad H_1 = \{s : (2,1,\mathsf{true}^{\top})\}, \qquad H_2 = \{s : \varepsilon\}$$

Here we have $< P\ ,\ H_1 > \longrightarrow < s[1]!\langle 2,\mathsf{true}^{\perp}\rangle.\mathbf{0}\ ,\ \{s : \varepsilon\} > = < P_1\ ,\ H_1' >$, while $< P\ ,\ H_2 > \not\longrightarrow$. Since $H_1' = \{s : \varepsilon\} = H_2$, we can proceed with $P_1 = s[1]!\langle 2,\mathsf{true}^{\perp}\rangle.\mathbf{0}$ and $P_2 = P$. Take now $K_1 = K_2 = \{s : \varepsilon\}$. Then $< P_1\ ,\ K_1 > \longrightarrow < \mathbf{0}\ ,\ \{s : (1,2,\mathsf{true}^{\perp})\} >$, while $< P_2\ ,\ K_2 > \not\longrightarrow$. Since $\{s : (1,2,\mathsf{true}^{\perp})\} \neq_{\perp} \{s : \varepsilon\} = K_2$, P is not $\perp$-secure.

With a similar argument we may show that $Q = s[1]?(2,x^{\top}).s[1]?(2,y^{\perp}).\mathbf{0}$ is not $\perp$-secure.

We may now justify the use of security levels on value variables.

Example 4.11. *(Need for levels on value variables)*
Suppose we had no levels on value variables. Then any variable could be replaced by values of any level. In this case, any process that performs an input and then outputs a value of level $\ell \neq \top$ would turn out to be insecure. For instance, consider the process $\widehat{P}$, which differs from P in Example 4.10 only because it has no level on the variable x, together with the $\mathbf{Q}$-sets H_1, H_2:

$$\widehat{P} = s[1]?(2,x).s[1]!\langle 2,\mathsf{true}^{\perp}\rangle.\mathbf{0}, \qquad H_1 = \{s : (2,1,\mathsf{true}^{\top})\}, \qquad H_2 = \{s : \varepsilon\}$$

Then $\widehat{P}$ would be insecure when run with H_1 and H_2, for the same reason why P was insecure in Example 4.10. This means that we would have no way to describe a secure process that receives a value of level $\perp$ and then outputs a value of level $\ell \neq \top$, for instance.

By annotating the variable x with the level $\perp$, we make sure that $\widehat{P}$ cannot reduce with H_1 either, and thus the insecurity disappears.

Interestingly, an insecure component may be "sanitised" by its context, so that the insecurity is not detectable in the overall process. Clearly, in case of a deadlocking context, the insecurity is masked simply because the dangerous part cannot be executed. However, the curing context could also be a partner of the insecure component, as shown by the next example. This example is significant because it constitutes a non trivial case of a process that is secure but *not safe*, as will be further discussed in Section 7.

Example 4.12. *(Insecurity sanitised by recursion and parallel context)*

Consider the recursive version R of process P in Example 4.10, in parallel with a dual process $\overline{R}$:

$$\begin{array}{rcl} R & = & \mathsf{def}\ X(x,\alpha) = Q\ \mathsf{in}\ X\langle \mathsf{true}^{\perp}, s[1]\rangle \\ \overline{R} & = & \mathsf{def}\ Y(z,\beta) = \overline{Q}\ \mathsf{in}\ Y\langle \mathsf{true}^{\top}, s[2]\rangle \end{array}$$

where $Q = \alpha?(2,y_1^{\top}).\alpha!\langle 2,x\rangle.X(x,\alpha)$ and $\overline{Q} = \beta!\langle 1,z\rangle.\beta?(1,y_2^{\perp}).Y(z,\beta)$.

We show that $R \mid \overline{R}$ is secure (while R is not). Take again $H_1 = \{s : (1,2,\mathsf{true}^{\top})\} =_{\perp} \{s : \varepsilon\} = H_2$. Then the moves

$$< R \mid \overline{R}\ ,\ H_1 > \longrightarrow^* < R' \mid \overline{R}\ ,\ \{s : \varepsilon\} >$$

where $R' = \mathsf{def}\ X(x,\alpha) = Q\ \mathsf{in}\ s[1]!\langle 2,\mathsf{true}^{\perp}\rangle.X\langle \mathsf{true}^{\perp}, s[1]\rangle$, can be simulated by:

$$< R \mid \overline{R}\ ,\ H_2 > \longrightarrow^* < R \mid \overline{R}'\ ,\ \{s : (2,1,\mathsf{true}^{\top})\} > \longrightarrow^* < R' \mid \overline{R}'\ ,\ \{s : \varepsilon\} >$$

where $\overline{R}' = \mathsf{def}\ Y(z,\beta) = \overline{Q}\ \mathsf{in}\ s[2]?(1,y_2^{\perp}).Y\langle \mathsf{true}^{\top}, s[2]\rangle$. Let us now compare $R_1 = R' \mid \overline{R}$ and $R_2 = R' \mid \overline{R}'$. Let K_1, K_2 be monotone $\mathbf{Q}$-sets such that $K_1 =_{\perp} K_2$, and both containing a queue $s : h$. Now, if $< R_1\ ,\ K_1 >$ moves first, either it does the high output of $\overline{R}$, in which case $< R_2\ ,\ K_2 >$ replies by staying idle, since the resulting processes will be equal and the resulting queues K_1', K_2' will be such that $K_1' =_{\perp} K_2'$, or it executes its first component, in which case $< R_2\ ,\ K_2 >$ does exactly the same, clearly preserving the $\perp$-equality of $\mathbf{Q}$-sets, and it remains to prove that $R \mid \overline{R}$ is $\perp$-bisimilar to $R \mid \overline{R}'$. But this is easy to see since if the first process moves its right component, then the second process may stay idle, while if the second process moves its right component, then the first process may simulate the move in two steps.

Conversely, if $< R_2\ ,\ K_2 >$ moves first, then either it executes its right component (if the queue allows it), in which case $< R_1\ ,\ K_1 >$ simulates the move in two steps, or it executes its left component, in which case $< R_1\ ,\ K_1 >$ simulates it by moving its left component in the same way, and then we are reduced once again to prove that $R \mid \overline{R}$ is $\perp$-bisimilar to $R \mid \overline{R}'$.

We conclude this section by showing that, notwithstanding the use of testers to mimic the presence of session participants in the environment, our security property fails to be compositional.

Let us start with an example that suggests how testers could help ensuring compositionality.

Example 4.13. Without testers, the following process:

$$s[1]?(2,x^{\top}).a^{\top}[1](\alpha).s[1]!\langle 2,\mathsf{true}^{\perp}\rangle.\mathbf{0}$$

would be secure, because it can only read a top message on a queue. However, when composed in parallel with the process $\bar{a}^{\top}[2] \mid a^{\top}[2](\beta).\mathbf{0}$, it gives rise to an insecure process, as discussed in Example 4.10.

Example 4.14. Consider the following process:

$$\begin{array}{rcl} P & = & P_1 \mid P_2 \mid b^{\ell}[1](\beta_1).\beta_1!\langle 2,\mathsf{true}^{\perp}\rangle.\mathbf{0} \mid b^{\ell}[2](\beta_2).\beta_2?(1,y^{\perp}).\mathbf{0} \mid \bar{b}^{\ell}[3] \\ P_1 & = & \mathsf{def}\ X(x,\alpha) = \alpha!\langle 2,x\rangle.X(x,\alpha)\ \mathsf{in}\ X\langle \mathsf{true}^{\top}, s[1]\rangle \\ P_2 & = & \mathsf{def}\ Y(y,\beta) = \beta?(1,z^{\top}).\mathsf{if}\ z^{\top}\ \mathsf{then}\ b^{\ell}[3](\beta_3).Y(y,\beta)\ \mathsf{else}\ a^{\ell}[1](\alpha_1).Y(y,\beta) \\ & & \qquad\qquad \mathsf{in}\ Y\langle \mathsf{false}^{\top}, s[2]\rangle \end{array}$$

This process is insecure. Since the input of P_2 is continuously enabled (because P_1 continuously emits a matching message), the insecurity stems from the fact that, according to the value received by P_2 for $z^{\top}$, the residual process with a fresh tester will either start a new session on

service b^ℓ, where a $\bot$-value is exchanged, or get stuck. However, note that all components are secure, both with $\ell = \bot$ and with $\ell = \top$. Indeed, testers are of no help for compositionality here.

We could try to get around this example by adding levels to queues, thus considering queues of the form $s^\ell : h$, and adapting our notion of projection for **Q**-sets so as to observe low empty queues. This will render insecure the first component with $\ell = \bot$, but not with $\ell = \top$. For $\ell = \top$, all components would still be secure, while the global process is insecure. Hence, levels on queues would not be sufficient by themselves to ensure security compositionality. We could try to go one step further by blocking the execution of components of the form $b^\ell[1](\beta_1).\beta_1!\langle 2, \mathsf{true}^{\ell'}\rangle.\mathbf{0}$ whenever $\ell \not\leq \ell'$, producing a failure signal to stop the bisimulation game, but this would not be satisfactory as it would go against the extensional character of the security property.

5. Monitored Semantics

In this section we introduce the monitored semantics for our calculus. This semantics is defined on *monitored processes* M, M', whose syntax is the following, assuming $\mu \in \mathscr{S}$:

$$M ::= P^{\rceil\mu} \mid M \mid M \mid \mid \mathsf{def}\ D\ \mathsf{in}\ M$$

In a monitored process $P^{\rceil\mu}$, the level μ that tags P is called the *monitoring level* for P. It controls the execution of P by blocking any communication of level $\ell \not\geq \mu$. Intuitively, $P^{\rceil\mu}$ represents a partially executed process, and μ is the join of the levels of received objects (values or labels) up to this point in execution.

The monitored semantics is defined on monitored configurations $C =< M\ ,\ H >$. By abuse of notation, we use the same symbol C for standard and monitored configurations.

The semantic rules define simultaneously a reduction relation $C \multimap\!\!\rightarrow C'$ and an error predicate $C\dagger$ on monitored configurations. As usual, the semantic rules are applied modulo a *structural equivalence* $\equiv$ (see Table 5). The new specific structural rules for monitored processes are:

$$(P \mid Q)^{\rceil\mu} \ \equiv\ P^{\rceil\mu} \mid Q^{\rceil\mu} \qquad\qquad (\mathsf{def}\ D\ \mathsf{in}\ P)^{\rceil\mu} \ \equiv\ \mathsf{def}\ D\ \mathsf{in}\ P^{\rceil\mu}$$

The reduction rules of the monitored semantics are given in Tables 6 and 7. Intuitively, the monitoring level is initially $\bot$ and gets increased each time an input of higher level is crossed. The reason why the monitoring level should take into account the level of inputs is that, as argued in Section 4, the process $s[1]?(2,x^\top).s[1]!\langle 2, \mathsf{true}^\bot\rangle.\mathbf{0}$ is not secure. Hence it should not be safe either. Moreover, if $< P^{\rceil\mu}\ ,\ H >$ attempts to perform a communication action of level $\ell \not\geq \mu$, then $< P^{\rceil\mu}\ ,\ H > \dagger$. We say in this case that the reduction produces error.

In the conditional (rules [MIf-T, MIf-F]) we can ignore the level ℓ of the tested expression and keep the monitoring level μ unchanged. This is because in the process $\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$, the expression e can only be evaluated if all its variables have been instantiated by previous inputs, thus the monitor is already greater than or equal to the level of all variables originally present in e. On the other hand, if e originally contained *only* constants, then its level does not really matter. In this way the process $\mathsf{if}\ \mathsf{true}^\top\ \mathsf{then}\ P\ \mathsf{else}\ Q$ turns out to be safe even if P performs $\bot$-actions, provided P itself is safe.

The evaluation contexts $\mathscr{E}$ for monitored processes in rule [MCont] are defined as expected.

One may wonder whether monitored processes of the form $P_1^{\rceil\mu_1} \mid P_2^{\rceil\mu_2}$, where $\mu_1 \neq \mu_2$, are

really needed. The following example shows that, in the presence of concurrency, a single monitoring level (as used for instance in (Boudol, 2009)) would not be enough.

$$P \equiv Q \;\Rightarrow\; P^{\rceil\mu} \equiv Q^{\rceil\mu} \qquad (P \mid Q)^{\rceil\mu} \equiv P^{\rceil\mu} \mid Q^{\rceil\mu}$$
$$M \mid \mathbf{0}^{\rceil\mu} \equiv M \qquad M \mid M' \equiv M' \mid M \qquad (M \mid M') \mid M'' \equiv M \mid (M' \mid M'')$$
$$(\mathsf{def}\ D\ \mathsf{in}\ P)^{\rceil\mu} \equiv \mathsf{def}\ D\ \mathsf{in}\ P^{\rceil\mu}$$
$$(\mathsf{def}\ D\ \mathsf{in}\ M) \mid M' \equiv \mathsf{def}\ D\ \mathsf{in}\ (M \mid M') \qquad \mathsf{def}\ D\ \mathsf{in}\ (\mathsf{def}\ D'\ \mathsf{in}\ M) \equiv \mathsf{def}\ D'\ \mathsf{in}\ (\mathsf{def}\ D\ \mathsf{in}\ M)$$
$$M \equiv M' \text{ and } H \equiv K \quad\Rightarrow\quad <M\ ,\ H> \equiv <M'\ ,\ K>$$

$$C \equiv C' \;\Rightarrow\; (\nu s)C \equiv (\nu s)C' \qquad (\nu ss')C \equiv (\nu s's)C$$
$$(\nu \tilde{s}) <M\ ,\ H> \parallel (\nu \tilde{s'}) <M'\ ,\ K> \equiv\ (\nu \tilde{ss'}) <M \mid M'\ ,\ H \cup K> \qquad \text{if } \mathrm{qn}(H) \cap \mathrm{qn}(K) = \emptyset$$

Table 5. *Structural equivalence of monitored processes and configurations.*

Example 5.1. *(Need for multiple monitoring levels)*
Suppose we could only use a single monitoring level for the parallel process P below, which should intuitively be safe. Then a computation of $P^{\rceil\perp}$ would be successful or not depending on the order of execution of its parallel components:

$$P = a^{\perp}[1](\alpha_1).P_1 \mid a^{\perp}[2](\alpha_2).P_2 \mid a^{\perp}[3](\alpha_3).P_3 \mid a^{\perp}[4](\alpha_4).P_4 \mid \bar{a}^{\perp}[4]$$
$$P_1 = \alpha_1!\langle 2, \mathsf{true}^{\perp}\rangle.\mathbf{0} \qquad P_2 = \alpha_2?(1, x^{\perp}).\mathbf{0}$$
$$P_3 = \alpha_3!\langle 4, \mathsf{true}^{\top}\rangle.\mathbf{0} \qquad P_4 = \alpha_4?(3, y^{\top}).\mathbf{0}$$

Here, if P_1 and P_2 communicate first, we would have the successful computation:

$$P^{\rceil\perp} \multimap^{*} (\nu s) <(P_3\{s[3]/\alpha_3\} \mid P_4\{s[4]/\alpha_4\})^{\rceil\perp}\ ,\ s:\varepsilon> \multimap (\nu s) <\mathbf{0}^{\rceil\top}\ ,\ s:\varepsilon>$$

Instead, if P_3 and P_4 communicate first, then we would run into error:

$$P^{\rceil\perp} \multimap^{*} (\nu s) <(P_1\{s[1]/\alpha_1\} \mid P_2\{s[2]/\alpha_2\})^{\rceil\top}\ ,\ s:\varepsilon> \dagger$$

Intuitively, the monitoring level resulting from the communication of P_3 and P_4 should not constrain the communication of P_1 and P_2, since there is no causal dependency between them. Allowing different monitoring levels for different parallel components, when P_3 and P_4 communicate first we get:

$$P^{\rceil\perp} \multimap^{*} (\nu s) <\mathbf{0}^{\rceil\top} \mid (P_1\{s[1]/\alpha_1\} \mid P_2\{s[2]/\alpha_2\})^{\rceil\perp}\ ,\ s:\varepsilon> \multimap^{*} (\nu s) <\mathbf{0}^{\rceil\top} \mid \mathbf{0}^{\rceil\perp}\ ,\ s:\varepsilon>$$

The following example justifies the levels of service names and the condition in rule [MLink]. Session initiation is the only synchronisation operation of our calculus. Since this synchronisation involves an initiator as well as a set of participants, the monitoring level of each of them must be lower than or equal to the monitoring level of the starting session.

Example 5.2. (*Need for levels on service names)*
Consider the process:

$$s[2]?(1, x^{\top}).\mathsf{if}\ x^{\top}\ \mathsf{then}\ \bar{b}^{\ell}[2]\ \mathsf{else}\ \mathbf{0} \mid b^{\ell}[1](\beta_1).\beta_1!\langle 2, \mathsf{true}^{\perp}\rangle.\mathbf{0} \mid b^{\ell}[2](\beta_2).\beta_2?(1, y^{\perp}).\mathbf{0}$$

This process is $\perp$-insecure, since if there is no $\top$-message in the $\mathbf{Q}$-set the process is blocked,

$$\text{if } \bigsqcup_{i\in\{1\ldots n\}} \mu_i \sqcup \mu \leq \ell \quad \text{then } a^\ell[1](\alpha_1).P_1^{\rceil\mu_1} \mid \ldots \mid a^\ell[n](\alpha_n).P_n^{\rceil\mu_n} \mid \tilde{a}^\ell[n]^{\rceil\mu} \multimap\!\!\rightarrow (\nu s) < P_1\{s[1]/\alpha_1\}^{\rceil\ell} \mid \ldots \mid P_n\{s[n]/\alpha_n\}^{\rceil\ell} \,,\, s : \varepsilon > \text{ else } a^\ell[1](\alpha_1).P_1^{\rceil\mu_1} \mid \ldots \mid a^\ell[n](\alpha_n).P_n^{\rceil\mu_n} \mid \tilde{a}^\ell[n]^{\rceil\mu}\dagger \quad \text{[MLink]}$$

$$\text{if } \mu \leq \ell \quad \text{then } < s[\mathtt{p}]!\langle \Pi, e\rangle.P^{\rceil\mu} \,,\, s : h > \multimap\!\!\rightarrow < P^{\rceil\mu} \,,\, s : h\cdot(\mathtt{p},\Pi,v^\ell) > \text{ else } < s[\mathtt{p}]!\langle \Pi, e\rangle.P^{\rceil\mu} \,,\, s : h > \dagger \quad \text{where } e \downarrow v^\ell \quad \text{[MSend]}$$

$$\text{if } \mu \leq \ell \quad \text{then } < s[\mathtt{q}]?(\mathtt{p}, x^\ell).P^{\rceil\mu} \,,\, s : (\mathtt{p},\mathtt{q},v^\ell)\cdot h > \multimap\!\!\rightarrow < P\{v/x\}^{\rceil\ell} \,,\, s : h > \text{ else } < s[\mathtt{q}]?(\mathtt{p}, x^\ell).P^{\rceil\mu} \,,\, s : (\mathtt{p},\mathtt{q},v^\ell)\cdot h > \dagger \quad \text{[MRec]}$$

$$\text{if } \mu \leq \ell \quad \text{then } < s[\mathtt{p}] \oplus^\ell \langle \Pi, \lambda\rangle.P^{\rceil\mu} \,,\, s : h > \multimap\!\!\rightarrow < P^{\rceil\mu} \,,\, s : h\cdot(\mathtt{p},\Pi,\lambda^\ell) > \text{ else } < s[\mathtt{p}] \oplus^\ell \langle \Pi, \lambda\rangle.P^{\rceil\mu} \,,\, s : h > \dagger \quad \text{[MLabel]}$$

$$\text{if } \mu \leq \ell \quad \text{then } < s[\mathtt{q}] \&^\ell (\mathtt{p}, \{\lambda_i : P_i\}_{i\in I})^{\rceil\mu} \,,\, s : (\mathtt{p},\mathtt{q},\lambda_{i_0}^\ell)\cdot h > \multimap\!\!\rightarrow < P_{i_0}^{\rceil\ell} \,,\, s : h > \text{ else } < s[\mathtt{q}] \&^\ell (\mathtt{p}, \{\lambda_i : P_i\}_{i\in I})^{\rceil\mu} \,,\, s : (\mathtt{p},\mathtt{q},\lambda_{i_0}^\ell)\cdot h > \dagger \quad \text{where } i_0 \in I \quad \text{[MBranch]}$$

$$\text{if } e \text{ then } P \text{ else } Q^{\rceil\mu} \multimap\!\!\rightarrow P^{\rceil\mu} \quad \text{if } e \downarrow \mathsf{true}^\ell \qquad \text{if } e \text{ then } P \text{ else } Q^{\rceil\mu} \multimap\!\!\rightarrow Q^{\rceil\mu} \quad \text{if } e \downarrow \mathsf{false}^\ell \quad \text{[MIf-T, MIf-F]}$$

$$\text{def } X(x,\alpha) = P \text{ in } (X\langle e, s[\mathtt{p}]\rangle^{\rceil\mu} \mid M) \multimap\!\!\rightarrow \text{def } X(x,\alpha) = P \text{ in } ((P\{v^\ell/x\}\{s[\mathtt{p}]/\alpha\})^{\rceil\mu} \mid M) \quad \text{where } e \downarrow v^\ell \quad \text{[MDef]}$$

$$< M \,,\, H > \multimap\!\!\rightarrow (\nu\tilde{s}) < M' \,,\, H' > \quad \Rightarrow \quad < \mathscr{E}[M] \,,\, H > \multimap\!\!\rightarrow (\nu\tilde{s}) < \mathscr{E}[M'] \,,\, H' > \quad \text{[MCont]}$$

$$C \multimap\!\!\rightarrow (\nu\tilde{s})C' \text{ and } \neg\, C''\dagger \quad \Rightarrow \quad (\nu\tilde{s'})(C \parallel C'') \multimap\!\!\rightarrow (\nu\tilde{s'})(\nu\tilde{s})(C' \parallel C'')$$
$$C\dagger \quad \Rightarrow \quad (\nu\tilde{s})(C \parallel C')\dagger \quad \text{[MScop]}$$

$$C \equiv C' \text{ and } C' \multimap\!\!\rightarrow C'' \text{ and } C'' \equiv C''' \quad \Rightarrow \quad C \multimap\!\!\rightarrow C'''$$
$$C\dagger \text{ and } C \equiv C' \quad \Longrightarrow \quad C'\dagger \quad \text{[MStruct]}$$

Table 6. *Monitored reduction rules for processes.*

hence it has a null $\bot$-behaviour, while if there is a $\top$-message in the **Q**-set, there is a possibility that the process subsequently exhibits a $\bot$-action. Hence this process should be unsafe too. Indeed, the monitoring level of the first component becomes $\top$ after the input and thus, assuming the `then` branch of the conditional is taken, this branch must have monitoring level $\top$ as well. Then, in order to start the session on service b^ℓ, Rule [MLink] requires $\top \leq \ell$, i.e. $\ell = \top$, and sets the monitoring level of the continuation to ℓ. This will block the output of the $\bot$-value, raising an error, and thus the process will turn out to be unsafe, as expected.

A similar example shows why we cannot avoid levels on choice operators.

$$\begin{array}{l}\textsf{if } \bigsqcup_{j\in\{1\ldots k\}} \mu_{\mathrm{p}_j} \sqcup \mu \leq \ell \\ \quad \textsf{then } (a^{\ell}[\mathrm{p}_1](\alpha_{\mathrm{p}_1}).P_{\mathrm{p}_1}^{\rceil\mu_{\mathrm{p}_1}} \mid \ldots \mid a^{\ell}[\mathrm{p}_k](\alpha_{\mathrm{p}_k}).P_{\mathrm{p}_k}^{\rceil\mu_{\mathrm{p}_k}} \mid \bar{a}^{\ell}[n]^{\rceil\mu}) \ \triangleleft\ (a^{\ell}[\mathrm{p}_{k+1}](\alpha_{\mathrm{p}_{k+1}}).\mathbf{0} \mid \ldots \mid a^{\ell}[\mathrm{p}_n](\alpha_{\mathrm{p}_n}).\mathbf{0}) \\ \qquad \multimap\!\!\rightarrow (\nu s) < P_{\mathrm{p}_1}\{s[\mathrm{p}_1]/\alpha_{\mathrm{p}_1}\}^{\rceil\ell} \mid \ldots \mid P_{\mathrm{p}_k}\{s[\mathrm{p}_k]/\alpha_{\mathrm{p}_k}\}^{\rceil\ell}, s:\varepsilon > \\ \quad \textsf{else } a^{\ell}[\mathrm{p}_1](\alpha_{\mathrm{p}_1}).P_{\mathrm{p}_1}^{\rceil\mu_{\mathrm{p}_1}} \mid \ldots \mid a^{\ell}[\mathrm{p}_k](\alpha_{\mathrm{p}_k}).P_{\mathrm{p}_k}^{\rceil\mu_{\mathrm{p}_k}} \mid \bar{a}^{\ell}[n]^{\rceil\mu} \dagger \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \mathrm{k} < \mathrm{n} \qquad \text{[MExtLink1]}\end{array}$$

$$\begin{array}{l}\textsf{if } \bigsqcup_{j\in\{1\ldots k\}} \mu_{\mathrm{p}_j} \leq \ell \\ \quad \textsf{then } (a^{\ell}[\mathrm{p}_1](\alpha_{\mathrm{p}_1}).P_{\mathrm{p}_1}^{\rceil\mu_{\mathrm{p}_1}} \mid \ldots \mid a^{\ell}[\mathrm{p}_k](\alpha_{\mathrm{p}_k}).P_{\mathrm{p}_k}^{\rceil\mu_{\mathrm{p}_k}}) \ \triangleleft\ (a^{\ell}[\mathrm{p}_{k+1}](\alpha_{\mathrm{p}_{k+1}}).\mathbf{0} \mid \ldots \mid a^{\ell}[\mathrm{p}_n](\alpha_{\mathrm{p}_n}).\mathbf{0} \mid \bar{a}^{\ell}[n]) \\ \qquad \multimap\!\!\rightarrow (\nu s) < P_{\mathrm{p}_1}\{s[\mathrm{p}_1]/\alpha_{\mathrm{p}_1}\}^{\rceil\ell} \mid \ldots \mid P_{\mathrm{p}_k}\{s[\mathrm{p}_k]/\alpha_{\mathrm{p}_k}\}^{\rceil\ell}, s:\varepsilon > \\ \quad \textsf{else } a^{\ell}[\mathrm{p}_1](\alpha_{\mathrm{p}_1}).P_{\mathrm{p}_1}^{\rceil\mu_{\mathrm{p}_1}} \mid \ldots \mid a^{\ell}[\mathrm{p}_k](\alpha_{\mathrm{p}_k}).P_{\mathrm{p}_k}^{\rceil\mu_{\mathrm{p}_k}} \dagger \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{[MExtLink2]}\end{array}$$

$$< M \ \triangleleft\ Z, H > \multimap\!\!\rightarrow (\nu\tilde{s}) < M', H' > \quad \Rightarrow \quad < \mathscr{E}[M] \ \triangleleft\ Z, H > \multimap\!\!\rightarrow (\nu\tilde{s}) < \mathscr{E}[M'], H' > \qquad \text{[MContTest]}$$

$$< M, H > \multimap\!\!\rightarrow (\nu\tilde{s}) < M', H' > \quad \Rightarrow \quad < M \ \triangleleft\ Z, H > \multimap\!\!\rightarrow (\nu\tilde{s}) < M', H' > \qquad \text{[MDropTest]}$$

Table 7. *Monitored reduction rules for testers.*

Example 5.3. *(Need for levels on selection and branching)*
Consider the process:

$$s[2]?(1,x^{\top}).s[2] \oplus^{\ell} \langle 3,\lambda\rangle.\mathbf{0} \mid \ s[3]\&^{\ell}(2,\{\lambda : s[3]!\langle 1,\mathsf{true}^{\bot}\rangle.\mathbf{0}\})$$

This process is $\bot$-insecure because if there is no $\top$-message in the **Q**-set the process is blocked, while if there is a $\top$-message in the **Q**-set, the process will proceed with the selection/branching and perform a $\bot$-output. Thus this process should be unsafe too. By adding a level ℓ in the selection/branching constructs and checking this level in Rules [MLabel] and [MBranch], we allow the monitored semantics to block the computation before the $\bot$-output. Indeed, if $\ell < \top$, then Rule [MLabel] will raise an error, while if $\ell = \top$, then both Rules [MLabel] and [MBranch] will go through successfully and then [MSend] will raise an error.

6. Safety

In this section we introduce a property of *safety* for monitored processes, from which we derive a property of safety also for processes. We then establish the two first results about this property, namely that (1) safety implies the absence of run-time errors, and (2) safety is compositional for both processes and monitored processes.

The remaining results, relating safety to security and typability, will be proven in Section 7 and Section 9, respectively.

A monitored process may be "relaxed" to a simple process by removing its monitoring levels.

Definition 6.1 (Demonitoring). If M is a monitored process, its *demonitoring* $\lfloor M \rfloor$ is defined

by:

$$\lfloor P^{\rceil\mu} \rfloor = P \qquad \lfloor M_1 \mid M_2 \rfloor = \lfloor M_1 \rfloor \mid \lfloor M_2 \rfloor \qquad \lfloor \mathsf{def}\ D\ \mathsf{in}\ M \rfloor = \mathsf{def}\ D\ \mathsf{in}\ \lfloor M \rfloor$$

We may now define our safety property. Intuitively, a monitored process M is safe if it can mimic at each step the transitions of the underlying process $\lfloor M \rfloor$.

Definition 6.2 (Monitored process safety). The safety predicate on monitored processes is coinductively defined by:
M is safe if for any tester Z and monotone $\mathbf{Q}$-set H such that $< \lfloor M \rfloor , H >$ is saturated:

If $< \lfloor M \rfloor \ \triangleleft\ Z , H > \rightsquigarrow (\nu \tilde{s}) < P , H' >$
then $< M \ \triangleleft\ Z , H > \multimap\!\!\rightarrow (\nu \tilde{s}) < M' , H' >$, where $\lfloor M' \rfloor = P$ and M' is safe.

Definition 6.3 (Process safety). A process P is safe if $P^{\rceil\perp}$ is safe.

We show now that if a process is safe, then none of its monitored computations starting with $H = \emptyset$ and monitor $\perp$ gives rise to error. This result rests on the observation that $< M , H > \multimap\!\!\rightarrow$ if and only if $< \lfloor M \rfloor , H > \longrightarrow$ and $\neg < M , H > \dagger$, and that if M is safe, then whenever a standard communication rule is applicable to $\lfloor M \rfloor$, the corresponding monitored communication rule is applicable to M.

Proposition 6.4 (Safety implies absence of run-time errors). If P is safe, then every monitored computation:

$$< P^{\rceil\perp} , \emptyset > = < M_0 , H_0 > \multimap\!\!\rightarrow (\nu \tilde{s_1}) < M_1 , H_1 > \multimap\!\!\rightarrow \cdots (\nu \tilde{s_k}) < M_k , H_k >$$

is such that $\neg < M_k , H_k > \dagger$.

Proof. Note that $< M , H > \multimap\!\!\rightarrow (\nu \tilde{s}) < M' , H' >$ implies $< \lfloor M \rfloor , H > \longrightarrow (\nu \tilde{s}) < \lfloor M' \rfloor , H' >$ and $< M , H > \dagger$ implies $< \lfloor M \rfloor , H > \longrightarrow (\nu \tilde{s}) < Q , H' >$ for some $\tilde{s}, Q, H'$. In other words, both $< M , H > \multimap\!\!\rightarrow (\nu \tilde{s}) < M' , H' >$ and $< M , H > \dagger$ imply that $< \lfloor M \rfloor , H >$ can move. Assume ad absurdum that $< M_k , H_k > \dagger$. Then, corresponding to the monitored computation $< P^{\rceil\perp} , \emptyset > \multimap\!\!\rightarrow^* (\nu \tilde{s_k}) < M_k , H_k > \dagger$, we would get

$$< P , \emptyset > \longrightarrow^* (\nu \tilde{s_k}) < \lfloor M_k \rfloor , H_k > \longrightarrow (\nu \widetilde{s_k s_{k+1}}) < Q , H_{k+1} >$$

for some $\widetilde{s_k s_{k+1}}$, Q, H_{k+1}. By definition this implies that $P^{\rceil\perp}$ is not safe, therefore P is not safe either, contradicting our assumption. □

Note that the converse of Proposition 6.4 does not hold, as shown by the next example. This means that we could not use absence of run-time errors as a definition of safety, since that would not be strong enough to guarantee our security property, which allows the pair of $\mathscr{L}$-equal $\mathbf{Q}$-sets to be refreshed at each step (while maintaining $\mathscr{L}$-equality).

Example 6.5. Consider the process P:

$$\begin{array}{lcl} P & = & a^{\perp}[1](\alpha_1).P_1 \mid a^{\perp}[2](\alpha_2).P_2 \mid \bar{a}^{\perp}[2] \\ P_1 & = & \alpha_1!\langle 2, \mathsf{true}^{\top}\rangle.\alpha_1?(2, x^{\top}).\mathbf{0} \\ P_2 & = & \alpha_2?(1, z^{\top}).\mathsf{if}\ z^{\top}\ \mathsf{then}\ \alpha_2!\langle 1, \mathsf{false}^{\top}\rangle.\mathbf{0}\ \mathsf{else}\ \alpha_2!\langle 1, \mathsf{true}^{\perp}\rangle.\mathbf{0} \end{array}$$

Note first that this process is not $\perp$-secure because after P_1 has put the value $\mathsf{true}^{\top}$ in the $\mathbf{Q}$-set,

this value may be changed to $\mathsf{false}^{\top}$ while preserving $\mathscr{L}$-equality of $\mathbf{Q}$-sets, thus allowing the `else` branch of P_2 to be explored by the bisimulation. This process is not safe either, because our definition of safety mimics $\mathscr{L}$-bisimulation by refreshing the $\mathbf{Q}$-set at each step. By contrast, the monitored execution of $< P^{\rceil\perp} , \emptyset >$ uses at each step the $\mathbf{Q}$-set produced at the previous step. Therefore the monitored execution will never take the `else` branch and will always succeed. Hence the simple absence of run-time errors is not sufficient by itself to enforce safety.

Intuitively, this discrepancy between safety and error-freedom is due to the fact that in Proposition 6.4 a computation of $< \lfloor M \rfloor , H >$ is assumed to proceed in isolation, in a static and protected environment, while our notions of safety and security assume that processes evolve in a dynamic and potentially hostile environment, where the $\mathbf{Q}$-sets may vary at each step.

Let us turn now to the main result of this section, namely the compositionality of the safety property. We start by proving that safety is compositional for monitored processes. The compositionality for processes will then follow as an easy corollary.

Theorem 6.6. If M_1 and M_2 are safe, then $M_1 \mid M_2$ is safe.

Proof. Let *SMP* be the set of safe monitored processes. Since *SMP* is defined coinductively, it is enough to show that the set of parallel compositions of safe monitored processes:

$$ParSMP = \{M = M_1 \mid M_2 \mid M_i \text{ is safe, for } i = 1,2\}$$

is closed with respect to the property of Definition 6.2 to obtain $ParSMP \subseteq SMP$. Namely, we have to prove that, if $M \in ParSMP$, then for any tester Z and for any monotone $\mathbf{Q}$-set H such that $< \lfloor M \rfloor , H >$ is saturated, the following holds:

If $< \lfloor M \rfloor \upharpoonleft Z , H > \rightsquigarrow (\nu \tilde{s}) < P , H' >$
then $< M \upharpoonleft Z , H > \multimap\!\!\rightarrow (\nu \tilde{s}) < M' , H' >$, where $\lfloor M' \rfloor = P$ and $M' \in ParSMP$.

Note that there is a 1-1 correspondence between the rules of the standard semantics and those of the monitored semantics. For any rule [Rule] in the former, there is a corresponding rule [MRule] in the latter. We proceed by case analysis on the rule [Rule] applied to infer the transition $< \lfloor M \rfloor , H > \rightsquigarrow (\nu \tilde{s}) < P , H' >$.

The interesting rules [Rule] are those for which [MRule] imposes some monitoring constraint, namely the communication rules and the three Link rules. The case of a communication rule, say [Send], is simple since the action comes from one of the two components $\lfloor M_i \rfloor$ (i=1 or i=2) and the tester is not used. Suppose the action comes from $\lfloor M_1 \rfloor$. In this case the transition has the form $< \lfloor M_1 \mid M_2 \rfloor \upharpoonleft Z , H > \rightsquigarrow < P , H' >$ and it is deduced by Rule [DropTest] from $< \lfloor M_1 \mid M_2 \rfloor , H > \rightsquigarrow < P , H' >$, where $P = P_1 \mid \lfloor M_2 \rfloor$ for some P_1 such that $< \lfloor M_1 \rfloor , H > \longrightarrow < P_1 , H' >$. Since M_1 is safe, we have then $< M_1 , H > \multimap\!\!\rightarrow < M_1' , H' >$ by Rule [MSend], where $\lfloor M_1' \rfloor = P_1$ and M_1' is safe. This implies $< M_1 \mid M_2 , H > \multimap\!\!\rightarrow < M_1' \mid M_2 , H' >$, whence by Rule [MDropTest] we deduce $< M_1 \mid M_2 \upharpoonleft Z , H > \multimap\!\!\rightarrow < M_1' \mid M_2 , H' >$, which is the required move since $\lfloor M_1' \mid M_2 \rfloor = P_1 \mid \lfloor M_2 \rfloor$ and $M_1' \mid M_2 \in ParSMP$.

The case of the Link rules is more interesting. There are essentially two subcases, according to whether the session participants and the initiator are all in the same M_i and possibly in the tester or they come from both M_i and possibly from the tester. We prove the result for Rule [ExtLink1], the proofs for Rules [Link] and [ExtLink2] being similar and simpler. We distinguish two cases:

1 The initiator and a subset of the participants come from the same M_i and the remaining participants come from the tester. Then the reasoning is similar to that for the communication case above, using the safety of M_i and the Rules [ExtLink1] and [MExtLink1].

2 The participants and the initiator are scattered among the tester and both M_i. This is the most interesting case, since we need to use a different tester T_i for each component M_i, whose role is to simulate the other component.
Suppose $M_1 = \prod_{j\in J_1} a^\ell[\mathtt{p}_j](\alpha_{\mathtt{p}_j}).P_{\mathtt{p}_j}^{(1)\rceil\mu_j} \mid \bar{a}^\ell[n]^\rceil\mu$ and $M_2 = \prod_{j\in J_2} a^\ell[\mathtt{p}_j](\alpha_{\mathtt{p}_j}).P_{\mathtt{p}_j}^{(2)\rceil\mu_j}$, where $\{\mathtt{p}_j \mid j \in J_1 \cup J_2\} \subseteq \{1,\ldots,n\}$ and $\{\mathtt{p}_j \mid j \in J_1\} \cap \{\mathtt{p}_j \mid j \in J_2\} = \emptyset$. (Note that since all participant roles are different, we cannot assume without loss of generality that the first $\#(J_1)$[‖] participants come from M_1 and the following $\#(J_2)$ from M_2.) Then the transition inferred by means of Rule [ExtLink1] has the form $\lfloor M_1 \mid M_2 \rfloor \upharpoonleft Z \rightsquigarrow (\nu s) < P_1 \mid P_2\ ,\ s : \varepsilon >$, where $P_1 = \prod_{j\in J_1} P_{\mathtt{p}_j}^{(1)}\{s[\mathtt{p}_j]/\alpha_{\mathtt{p}_j}\}$ and $P_2 = \prod_{j\in J_2} P_{\mathtt{p}_j}^{(2)}\{s[\mathtt{p}_j]/\alpha_{\mathtt{p}_j}\}$. Define now the two testers $Z_1 = \prod_{j\in J_2} a^\ell[\mathtt{p}_j](\alpha_{\mathtt{p}_j}).\mathbf{0} \mid Z$ and $Z_2 = \prod_{j\in J_1} a^\ell[\mathtt{p}_j](\alpha_{\mathtt{p}_j}).\mathbf{0} \mid \bar{a}^\ell[n] \mid Z$. Then by construction $\lfloor M_i \rfloor \upharpoonleft Z_i \rightsquigarrow (\nu s) < P_i\ ,\ s : \varepsilon >$. Since both M_i are safe, by Rule [MExtLink1] we get for each of them a monitored transition $M_i \upharpoonleft Z_i \multimap\!\!\rightarrow (\nu s) < P_i^{\rceil\ell}\ ,\ s : \varepsilon >$, where both $P_i^{\rceil\ell}$ are safe and $\bigsqcup_{k=1}^{k_1+k_2} \mu_{j_k} \sqcup \mu \leq \ell$. This implies $(M_1 \mid M_2) \upharpoonleft Z \multimap\!\!\rightarrow (\nu s) < (P_1 \mid P_2)^{\rceil\ell}\ ,\ s : \varepsilon >$, which is the required move since $\lfloor (P_1 \mid P_2)^{\rceil\ell} \rfloor = P_1 \mid P_2$ and $P_1^{\rceil\ell} \mid P_2^{\rceil\ell} \in ParSMP$.

□

Corollary 6.7. If P_1 and P_2 are safe, then $P_1 \mid P_2$ is safe.

Proof. By definition of safety, $P_1^{\rceil\perp}$ and $P_2^{\rceil\perp}$ are safe. Then, by Theorem 6.6, also $P_1^{\rceil\perp} \mid P_2^{\rceil\perp} \equiv (P_1 \mid P_2)^{\rceil\perp}$ is safe. □

7. Relating Safety and Security

In this section we prove that safety implies security. As already discussed in the previous sections, both these properties are *persistent*, namely they are required to hold in any reachable state of the process, to take into account the possibility of intrusive observations (or "attacks") in the course of execution. However, note that safety is a property of individual computations, while security is a property of the whole set of computations of a process. Hence proving that safety implies security amounts to prove that a global property of the set of computations is implied by a local property of each computation in the set.

In order to prove that safety implies security, we need some preliminary results.

Lemma 7.1 (Monotonicity of monitoring). Monitoring levels may only stay equal or increase along execution: If $\langle P^{\rceil\mu} \upharpoonleft Z, H\rangle \multimap\!\!\rightarrow (\nu\tilde{s})\langle P'^{\rceil\mu'}, H'\rangle$, then $\mu \leq \mu'$.

We define now the class of $\mathscr{L}$*-high processes*, namely processes which always modify the **Q**-sets in a way that is transparent for $\mathscr{L}$-observers.

[‖] We use $\#(J)$ to denote the cardinality of the set J.

Definition 7.2 ($\mathscr{L}$-highness of processes). A process P is *$\mathscr{L}$-high* if for any monotone $\mathbf{Q}$-set H such that $< P\ ,\ H >$ is saturated and any tester Z it satisfies the property:

$$\text{If } < P \upharpoonright Z\ ,\ H > \rightsquigarrow (\nu \tilde{s}) < P'\ ,\ H' >, \text{ then } H =_{\mathscr{L}} H' \text{ and } P' \text{ is } \mathscr{L}\text{-high.}$$

Lemma 7.3. If $P^{\rceil\mu}$ is safe and $\mu \notin \mathscr{L}$, then P is $\mathscr{L}$-high.

Proof. By induction on P. We only consider some interesting cases.

If $P = s[\mathtt{p}]!\langle \Pi, e\rangle.P'$, then $< P\ ,\ s : h > \longrightarrow < P'\ ,\ s : h \cdot (\mathtt{p}, \Pi, v^{\ell}) >$, where $e \downarrow v^{\ell}$. From the safety of $P^{\rceil\mu}$ we get that $< P^{\rceil\mu}\ ,\ s : h > \multimap\!\!\rightarrow < P'^{\rceil\mu}\ ,\ s : h \cdot (\mathtt{p}, \Pi, v^{\ell}) >$ and $P'^{\rceil\mu}$ is safe. This means that $\mu \leq \ell$. Then, since $\mu \notin \mathscr{L}$, also $\ell \notin \mathscr{L}$ and thus $s : h =_{\mathscr{L}} s : h \cdot (\mathtt{p}, \Pi, v^{\ell})$. Finally, P' is $\mathscr{L}$-high by induction.

If $P = s[\mathtt{q}]?(\mathtt{p}, x^{\ell}).P'$, then $< P\ ,\ s : (\mathtt{p}, \mathtt{q}, v^{\ell}) \cdot h > \longrightarrow < P'\{v/x\}\ ,\ s : h >$, which implies $< P^{\rceil\mu}\ ,\ s : (\mathtt{p}, \mathtt{q}, v^{\ell}) \cdot h > \multimap\!\!\rightarrow < P'\{v/x\}^{\rceil\ell}\ ,\ s : h >$ and $\mu \leq \ell$ and $P'\{v/x\}^{\rceil\ell}$ is safe. Since $\mu \notin \mathscr{L}$ implies $\ell \notin \mathscr{L}$ we get $s : (\mathtt{p}, \mathtt{q}, v^{\ell}) \cdot h =_{\mathscr{L}} s : h$. Again, $P'\{v/x\}$ is $\mathscr{L}$-high by induction. □

Lemma 7.4. If $P = s[\mathtt{q}]?(\mathtt{p}, x^{\ell}).P'$ and P is safe, then $P^{\rceil\ell}$ is safe.

Proof. For any v of level ℓ, P has a unique transition. Correspondingly, $P^{\rceil\bot}$ has a monitored transition $< P^{\rceil\bot}\ ,\ \{s : (\mathtt{p}, \mathtt{q}, v^{\ell}) \cdot h\} > \multimap\!\!\rightarrow < P'\{v/x\}^{\rceil\ell}\ ,\ s : h >$. Since P is safe, by definition $P^{\rceil\bot}$ is safe, therefore its residual $P'\{v/x\}^{\rceil\ell}$ is safe. Then, since also $< P^{\rceil\ell}\ ,\ \{s : (\mathtt{p}, \mathtt{q}, v^{\ell}) \cdot h\} > \multimap\!\!\rightarrow < P'\{v/x\}^{\rceil\ell}\ ,\ s : h >$, we may conclude that $P^{\rceil\ell}$ is safe. □

We next define the bisimulation relation that will be used in the proof of soundness. Roughly, all monitored processes with a high monitoring level are related, while the other processes are related if they are congruent.

To define the bisimulation it is handy to consider monitored processes where monitors are pushed as deeply as possible into terms, so that a single monitor never controls two processes in parallel. This is formalised by the following mapping df, which essentially transforms a monitored parallel composition of processes into a parallel composition of monitored processes.

Definition 7.5 (Distributed Form of monitored processes). The *distributed form* of a monitored process M (notation $\mathsf{df}(M)$) is defined by:

$$\mathsf{df}(M) = \begin{cases} \mathsf{df}(M_1) \mid \mathsf{df}(M_2) & \text{if } M = M_1 \mid M_2, \\ \mathsf{df}(P_1^{\rceil\mu}) \mid \mathsf{df}(P_2^{\rceil\mu}) & \text{if } M = (P_1 \mid P_2)^{\rceil\mu} \text{ and } P_i \not\equiv \mathbf{0},\ i = 1,2 \\ \mathsf{df}(P_i^{\rceil\mu}) & \text{if } M = (P_1 \mid P_2)^{\rceil\mu} \text{ and } P_j \equiv \mathbf{0},\ i \neq j \\ \mathsf{def}\ D\ \mathsf{in}\ \mathsf{df}(N) & \text{if } M = \mathsf{def}\ D\ \mathsf{in}\ N, \\ \mathsf{def}\ D\ \mathsf{in}\ \mathsf{df}(P^{\rceil\mu}) & \text{if } M = (\mathsf{def}\ D\ \mathsf{in}\ P)^{\rceil\mu}, \\ M & \text{otherwise.} \end{cases}$$

It is easy to verify that $\mathsf{df}(M) \equiv M$.

Definition 7.6 (Bisimulation for soundness proof: monitored processes). Given a downward-closed set of security levels $\mathscr{L} \subseteq \mathscr{S}$, the relation $\mathscr{R}_{\circ}^{\mathscr{L}}$ on monitored processes in distributed form is defined inductively as follows:

$M_1 \mathscr{R}_{\circ}^{\mathscr{L}} M_2$ if M_1 and M_2 are safe and one of the following holds:

1 $M_1 = P_1^{\rceil\mu_1}$, $M_2 = P_2^{\rceil\mu_2}$, and $\mu_1, \mu_2 \notin \mathscr{L}$;
2 $M_1 = M_2 = P^{\rceil\mu}$, and $\mu \in \mathscr{L}$;
3 $M_i = \prod_{j=1}^{m} N_j^{(i)}$, where $\forall j \in \{1,\ldots,m\}$, $N_j^{(1)} \mathscr{R}_\circ^{\mathscr{L}} N_j^{(2)}$ follows from (1) or (2);
4 $M_i = \mathsf{def}\ D\ \mathsf{in}\ N_i$, where $N_1 \mathscr{R}_\circ^{\mathscr{L}} N_2$.

Definition 7.7 (Bisimulation for soundness proof: processes). Given a downward-closed set of security levels $\mathscr{L} \subseteq \mathscr{S}$, the relation $\mathscr{R}^{\mathscr{L}}$ on processes is defined by:

$$P_1 \mathscr{R}^{\mathscr{L}} P_2 \text{ if there are monitored processes in distributed form } M_1, M_2 \text{ such that } P_i \equiv \lfloor M_i \rfloor \text{ for } i = 1,2 \text{ and } M_1 \mathscr{R}_\circ^{\mathscr{L}} M_2.$$

We may now show our main result, namely that safety implies security. The proof consists in showing that safety implies $\mathscr{L}$–security, for any $\mathscr{L}$. The informal argument goes as follows. Let "low" mean "in $\mathscr{L}$" and "high" mean "not in $\mathscr{L}$". If P is not $\mathscr{L}$–secure, this means that there are two different observable low behaviours after a high input. This implies that in at least one of the two computations there is some observable low action after the high input. But in this case the monitored semantics will yield error, since it does so as soon as it meets an action of level $\ell \not\geq \mu$, where μ is the monitoring level of the executing component (which will have been set to high after crossing the high input).

Theorem 7.8 (Safety implies security). If P is safe, P is also secure.

Proof. We prove that P is $\mathscr{L}$–secure for any $\mathscr{L}$. The proof consists in showing that the relation $\mathscr{R}^{\mathscr{L}}$ defined in Definition 7.7 is an $\mathscr{L}$-bisimulation containing the pair (P,P) for any safe process P. Note first that if P is safe, then $P \mathscr{R}^{\mathscr{L}} P$ because $P^{\rceil\perp} \mathscr{R}_\circ^{\mathscr{L}} P^{\rceil\perp}$ by Clause (2). Suppose now that $P_1 \mathscr{R}^{\mathscr{L}} P_2$. This means that there exist monitored processes in distributed form M_1, M_2 such that $P_i \equiv \lfloor M_i \rfloor$ for $i = 1,2$ and $M_1 \mathscr{R}_\circ^{\mathscr{L}} M_2$. Since processes and configurations are considered modulo $\equiv$ we can assume without loss of generality that $P_i = \lfloor M_i \rfloor$ for $i = 1,2$. Let H_1, H_2 be two monotone $\mathbf{Q}$-sets such that $H_1 =_{\mathscr{L}} H_2$ and $< P_i , H_i >$ for $i = 1,2$ are saturated.
We want to show that for each tester Z and each reduction $< P_1 \triangleleft Z , H_1 > \rightsquigarrow (\nu\tilde{s}) < P_1' , H_1' >$, either we have $H_1' =_{\mathscr{L}} H_2$ and $P_1' \mathscr{R} P_2$, or $< P_2 \triangleleft Z , H_2 > \rightsquigarrow \cdot \longrightarrow^* (\nu\tilde{s}) < P_2' , H_2' >$, where $H_1' =_{\mathscr{L}} H_2'$ and $P_1' \mathscr{R}_\circ^{\mathscr{L}} P_2'$.

We proceed by induction on the definition of $\mathscr{R}_\circ^{\mathscr{L}}$ (Definition 7.6).

— **Clause (1).** In this case $M_1 = P_1^{\rceil\mu_1}$, $M_2 = P_2^{\rceil\mu_2}$ and $\mu_1, \mu_2 \notin \mathscr{L}$. Suppose there is a transition $< P_1 \triangleleft Z , H_1 > \rightsquigarrow (\nu\tilde{s}) < P_1' , H_1' >$. By Lemma 7.3, P_1 is $\mathscr{L}$-high, thus $H_1' =_{\mathscr{L}} H_1$. Since P_1 is safe, there exists μ_1' such that $< P_1^{\rceil\mu_1} \triangleleft Z , H_1 > \multimap\!\!\rightarrow (\nu\tilde{s}) < P_1'^{\rceil\mu_1'} , H_1' >$, where $P_1'^{\rceil\mu_1'}$ is safe (the fact that M_1 is in distributed form ensures that the residual is of the form $P_1'^{\rceil\mu_1'}$). By Lemma 7.1, we have $\mu_1 \leq \mu_1'$, whence $\mu_1' \notin \mathscr{L}$. Then $< P_2 , H_2 >$ may reply by the empty move, since $H_1' =_{\mathscr{L}} H_2$ and $P_1'^{\rceil\mu_1'} \mathscr{R}_\circ^{\mathscr{L}} P_2^{\rceil\mu_2}$ by Clause (1) again, whence $P_1' \mathscr{R}^{\mathscr{L}} P_2$ by Definition 7.7.

— **Clause (2).** In this case $M_1 = M_2 = P^{\rceil\mu}$, and $\mu \in \mathscr{L}$, and the proof proceeds by case analysis on the form of P. We examine some interesting cases. We will consider testers only when they are necessary.

– Let $M_i = a^\ell[\mathrm{p}](\alpha_\mathrm{p}).Q_\mathrm{p}^{\rceil\mu}$, then $P = a^\ell[\mathrm{p}](\alpha_\mathrm{p}).Q_\mathrm{p}$. Take $Z = \prod_{1 \leq \mathrm{q} \leq n, \mathrm{q} \neq \mathrm{p}} a^\ell[\mathrm{q}](\alpha_\mathrm{q}).\mathbf{0} \mid \bar{a}^\ell[n]$, where n is the arity of a. We can only reduce by applying rule [ExtLink2]. So we get

$< P \triangleright Z \ , \ H_1 > \rightsquigarrow \ (\nu s) < Q_{\mathtt{p}}\{s[\mathtt{p}]/\alpha_{\mathtt{p}}\} \ , \ H_1 \cup \{s : \varepsilon\} >$, where s is fresh. Now, this can be matched by $< P \triangleright Z \ , \ H_2 > \rightsquigarrow \ (\nu s) < Q_{\mathtt{p}}\{s[\mathtt{p}]/\alpha_{\mathtt{p}}\} \ , \ H_2 \cup \{s : \varepsilon\} >$. From $H_1 =_{\mathscr{L}} H_2$ we get $H_1 \cup \{s : \varepsilon\} =_{\mathscr{L}} H_2 \cup \{s : \varepsilon\}$. Since $P^{\rceil\mu}$ is safe, there is a transition $< P^{\rceil\mu} \triangleright Z \ , \ H_1 > \multimap\!\!\rightarrow < Q_{\mathtt{p}}\{s[\mathtt{p}]/\alpha_{\mathtt{p}}\}^{\rceil\ell} \ , \ H_1 \cup \{s : \varepsilon\} >$ and $Q_{\mathtt{p}}\{s[\mathtt{p}]/\alpha_{\mathtt{p}}\}^{\rceil\ell}$ is safe. Hence $Q_{\mathtt{p}}\{s[\mathtt{p}]/\alpha_{\mathtt{p}}\}^{\rceil\ell} \mathscr{R}_{\circ}^{\mathscr{L}} Q_{\mathtt{p}}\{s[\mathtt{p}]/\alpha_{\mathtt{p}}\}^{\rceil\ell}$ by Clause (1) if $\ell \notin \mathscr{L}$ and by Clause (2) if $\ell \in \mathscr{L}$. In both cases $Q_{\mathtt{p}}\{s[\mathtt{p}]/\alpha_{\mathtt{p}}\} \mathscr{R}^{\mathscr{L}} Q_{\mathtt{p}}\{s[\mathtt{p}]/\alpha_{\mathtt{p}}\}$.

- Let $M_i = s[\mathtt{p}]!\langle \Pi, e\rangle.P'^{\rceil\mu}$ and $\exists v, \exists \ell \geq \mu$ such that $e \downarrow v^{\ell}$. In this case $P = s[\mathtt{p}]!\langle \Pi, e\rangle.P'$ and the reduction $< P \ , \ H_1 > \longrightarrow < P' \ , \ H_1' >$ is obtained by rule [Send]. Applicability of rule [Send] assures that there is a queue $s : h_1$ in H_1. Since $\langle P, H_2\rangle$ is saturated, there will be a queue $s : h_2$ in H_2. If $H_i = K_i \cup s : h_i$, we have $H_1' = K_1 \cup s : h_1 \cdot (\mathtt{p}, \Pi, v^{\ell})$, and the matching move will be $< P \ , \ H_2 > \longrightarrow < P' \ , \ H_2' >$, where $H_2' = K_2 \cup s : h_2 \cdot (\mathtt{p}, \Pi, v^{\ell})$. Indeed, if $\ell \notin \mathscr{L}$ then $H_1' =_{\mathscr{L}} H_1 =_{\mathscr{L}} H_2 =_{\mathscr{L}} H_2'$. If $\ell \in \mathscr{L}$ then $H_1' =_{\mathscr{L}} H_2'$ follows from $K_1 =_{\mathscr{L}} K_2$ and $s : h_1 =_{\mathscr{L}} s : h_2$. Since $P^{\rceil\mu}$ is safe, we know that there is a monitored transition $< P^{\rceil\mu} \ , \ H_1 > \multimap\!\!\rightarrow < P'^{\rceil\mu} \ , \ H_1' >$ and that $P'^{\rceil\mu}$ is safe. Hence $P'^{\rceil\mu} \mathscr{R}_{\circ}^{\mathscr{L}} P'^{\rceil\mu}$ by Clause (2) again, which implies $P' \mathscr{R}^{\mathscr{L}} P'$.
- Let $M_i = s[\mathtt{q}]?(\mathtt{p}, x^{\ell}).P'^{\rceil\mu}$, where $\mu \leq \ell$. In this case $P = s[\mathtt{q}]?(\mathtt{p}, x^{\ell}).P'$ and the move of $< P \ , \ H_1 >$ is obtained by rule [Rec] and has the form $< P \ , \ H_1 > \longrightarrow < P'\{v/x\} \ , \ H_1' >$ for some v such that the message $(\mathtt{p}, \mathtt{q}, v^{\ell})$ occurs at the head of queue s in H_1. As above, since $P^{\rceil\mu}$ is safe, we know that $< P^{\rceil\mu} \ , \ H_1 > \multimap\!\!\rightarrow < P'\{v/x\}^{\rceil\ell} \ , \ H_1' >$ and that $P'\{v/x\}^{\rceil\ell}$ is safe. Now, if $\ell \notin \mathscr{L}$ we get $H_1' =_{\mathscr{L}} H_2$ and $P^{\rceil\ell}$ is safe by Lemma 7.4. Then $< P \ , \ H_2 >$ may reply by the empty move, since $P'\{v/x\}^{\rceil\ell} \mathscr{R}_{\circ}^{\mathscr{L}} P^{\rceil\ell}$ by Clause (1) and thus $P'\{v/x\} \mathscr{R}^{\mathscr{L}} P$. If $\ell \in \mathscr{L}$, since $H_1 =_{\mathscr{L}} H_2$, the message $(\mathtt{p}, \mathtt{q}, v^{\ell})$ must occur at the head of queue s also in H_2. Then, assuming $H_i = K_i \cup s : (\mathtt{p}, \mathtt{q}, v^{\ell}) \cdot h_i$, we have:
  $< P \ , \ K_1 \cup s : (\mathtt{p}, \mathtt{q}, v^{\ell}) \cdot h_1 > \longrightarrow < P'\{v/x\} \ , \ K_1 \cup s : h_1 >$
  $< P \ , \ K_2 \cup s : (\mathtt{p}, \mathtt{q}, v^{\ell}) \cdot h_2 > \longrightarrow < P'\{v/x\} \ , \ K_2 \cup s : h_2 >$
  Here $P'\{v/x\}^{\rceil\ell} \mathscr{R}_{\circ}^{\mathscr{L}} P'\{v/x\}^{\rceil\ell}$ by Clause (2), which implies $P'\{v/x\} \mathscr{R}^{\mathscr{L}} P'\{v/x\}$. Lastly $H_1 =_{\mathscr{L}} H_2$ implies $K_1 \cup s : h_1 =_{\mathscr{L}} K_2 \cup s : h_2$.

— **Clause (3).** Let $M_i = \prod_{j=1}^{m} N_j^{(i)}$, where $\forall j \ (1 \leq j \leq m)$ we have $N_j^{(1)} \mathscr{R}_{\circ}^{\mathscr{L}} N_j^{(2)}$ either by Clause (1) or by Clause (2). This means that for each j, there exists μ_j such that $N_j^{(i)} = Q_j^{(i)\rceil\mu_j}$. Then $P_i = \prod_{j=1}^{m} Q_j^{(i)}$ and the move of $< P_1 \ , \ H_1 >$ comes either from a single component $Q_j^{(1)}$, or from a group of $k \leq m$ components that synchronise to open an n-ary session on service a^{ℓ}, possibly with the help of a tester. Using structural equivalence, we may assume that in the first case the moving component is $Q_1^{(1)}$, and in the second case the group of moving components is $\prod_{j=1}^{k} Q_j^{(1)}$, for some $k \leq m$. We examine the two cases in turn.

(a) If $Q_1^{(1)}$ moves alone, then the move:

$$(\triangleleft) \qquad < P_1 \ , \ H_1 > \rightsquigarrow (\nu \tilde{s}) < Q_1'^{(1)} \mid \prod_{j=2}^{m} Q_j^{(1)} \ , \ H_1' >$$

is deduced from $< Q_1^{(1)} \ , \ H_1 > \rightsquigarrow (\nu \tilde{r}) < Q_1'^{(1)} \ , \ H_1' >$.
Since $N_1^{(1)} \mathscr{R}_{\circ}^{\mathscr{L}} N_1^{(2)}$, by induction we have either i) $H_1' =_{\mathscr{L}} H_2$ and $Q_1'^{(1)} \mathscr{R}^{\mathscr{L}} Q_1^{(2)}$, or ii) $< Q_1^{(2)} \ , \ H_2 > \rightsquigarrow \cdot \longrightarrow^* \ (\nu \tilde{s}) < Q_1'^{(2)} \ , \ H_2' >$ with $Q_1'^{(1)} \mathscr{R}^{\mathscr{L}} Q_1'^{(2)}$ and $H_1' =_{\mathscr{L}} H_2'$.

We consider the case ii), since the case i) is simpler. From ii) we infer:

$$(\lhd\lhd) \qquad < P_2 \, , \, H_2 > \rightsquigarrow \cdot \longrightarrow^* \, (\nu \tilde{s}) < Q_1^{\prime(2)} \mid \prod_{j=2}^{m} Q_j^{(2)} \, , \, H_2^{\prime} >$$

Since the M_i are safe, corresponding to $(\lhd)$ and $(\lhd\lhd)$ we have:

$$< M_1 \, , \, H_1 > \multimap\!\!\rightarrow (\nu \tilde{s}) < N_1^{\prime(1)} \mid \prod_{j=2}^{m} N_j^{(1)} \, , \, H_1^{\prime} >$$

$$< M_2 \, , \, H_2 > \multimap\!\!\rightarrow^* (\nu \tilde{s}) < N_1^{\prime(2)} \mid \prod_{j=2}^{m} N_j^{(2)} \, , \, H_2^{\prime} >$$

where the $M_i^{\prime} = N_1^{\prime(i)} \mid \prod_{j=2}^{m} N_j^{(i)}$ are again safe, $\lfloor N_j^{\prime(i)} \rfloor = Q_j^{\prime(i)}$ and $N_1^{\prime(1)} \, \mathscr{R}_{\circ}^{\mathscr{L}} \, N_1^{\prime(2)}$. Therefore $N_1^{\prime(1)} \mid \prod_{j=2}^{m} N_j^{(1)} \, \mathscr{R}_{\circ}^{\mathscr{L}} \, N_1^{\prime(2)} \mid \prod_{j=2}^{m} N_j^{(2)}$ by Clause (3) again, from which we derive $Q_1^{\prime(1)} \mid \prod_{j=2}^{m} Q_j^{(1)} \, \mathscr{R}^{\mathscr{L}} \, Q_1^{\prime(2)} \mid \prod_{j=2}^{m} Q_j^{(2)}$. We conclude that $(\lhd\lhd)$ is the required matching move.

(b) Here we assume that $P_1 = \prod_{j=1}^{k} Q_j^{(1)} \mid \prod_{j=k+1}^{m} Q_j^{(1)}$ and that all the processes in $\prod_{j=1}^{k} Q_j^{(1)}$ synchronise, possibly with the help of the tester. Let us consider the case where the applied rule is [ExtLink1] (the proofs in the other cases are similar). Let $k < n$, $Q_j^{(1)} = a^{\ell}[\mathrm{p}_j](\alpha_{\mathrm{p}_j}).R_{\mathrm{p}_j}^{(1)}$ for $1 \leq j \leq k-1$ and $Q_k^{(1)} = \bar{a}^{\ell}[n]$ and Z be the tester.
Suppose that the move of $< P_1 \upharpoonleft Z \, , \, H_1 >$ is

$$(\rhd) \qquad < P_1 \upharpoonleft Z \, , \, H_1 > \rightsquigarrow < P_1^{\prime} \, , \, H_1 \cup \{s : \varepsilon\} >$$

where $P_1^{\prime} = (\nu s) < \prod_{j=1}^{k-1} R_{\mathrm{p}_j}^{(1)}\{s[\mathrm{p}_j]/\alpha_{\mathrm{p}_j}\} \mid \prod_{j=k+1}^{n} Q_j^{(1)} \, , \, H_1 \cup \{s : \varepsilon\} >$. Since M_1 is safe, $< M_1 \upharpoonleft Z \, , \, H_1 >$ has a corresponding monitored transition deduced by Rule [MExtLink1]:
$< M_1 \upharpoonleft Z \, , \, H_1 > \multimap\!\!\rightarrow (\nu s) < \prod_{j=1}^{k-1} R_{\mathrm{p}_j}^{(1)}\{s[\mathrm{p}_j]/\alpha_{\mathrm{p}_j}\}^{\rceil \ell} \mid \prod_{j=k+1}^{m} N_j^{(1)} \, , \, H_1 \cup \{s : \varepsilon\} >$

where $\sqcup_{j=1}^{k} \mu_j \leq \ell$ and the safety of M_1 implies the safety of each $R_{\mathrm{p}_j}^{(1)}\{s[\mathrm{p}_j]/\alpha_{\mathrm{p}_j}\}^{\rceil \ell}$.

We distinguish now two cases, according to whether $\ell \in \mathscr{L}$ or $\ell \notin \mathscr{L}$:

i) If $\ell \in \mathscr{L}$, then $\mu_j \in \mathscr{L}$ for each j such that $1 \leq j \leq k$. In this case we know that $N_j^{(1)} \, \mathscr{R}_{\circ}^{\mathscr{L}} \, N_j^{(2)}$ follows from Clause (2), and hence $N_j^{(2)} = a^{\ell}[\mathrm{p}_j](\alpha_{\mathrm{p}_j}).R_{\mathrm{p}_j}^{(1)\rceil \mu_j}$ for each j such that $1 \leq j \leq k-1$ and $N_k^{(2)} = \bar{a}^{\ell}[n]^{\rceil \mu_k}$. This means that
$P_2 = \prod_{j=1}^{k-1} a^{\ell}[\mathrm{p}_j](\alpha_{\mathrm{p}_j}).R_{\mathrm{p}_j}^{(1)} \mid \bar{a}^{\ell}[n]^{\rceil \mu_k} \mid \prod_{j=k+1}^{m} Q_j^{(2)}$. Then $< P_2 \, , \, H_2 >$ has the following move:
$(\rhd\rhd) \;\; < P_2 \upharpoonleft Z \, , \, H_2 > \rightsquigarrow \; (\nu s) < \prod_{j=1}^{k-1} R_{\mathrm{p}_j}^{(1)}\{s[\mathrm{p}_j]/\alpha_{\mathrm{p}_j}\} \mid \prod_{j=k+1}^{m} Q_j^{(2)} \, , \, H_2 \cup \{s : \varepsilon\} >$

Since M_2 is safe, $< M_2 \upharpoonleft Z \, , \, H_2 >$ has a corresponding monitored transition:
$< M_2 \upharpoonleft Z \, , \, H_2 > \multimap\!\!\rightarrow (\nu s) < \prod_{j=1}^{k-1} R_{\mathrm{p}_j}^{(1)}\{s[\mathrm{p}_j]/\alpha_{\mathrm{p}_j}\}^{\rceil \ell} \mid \prod_{j=k+1}^{m} N_j^{(2)} \, , \, H_1 \cup \{s : \varepsilon\} >$

Define now $M_i^{\prime} = \prod_{j=1}^{k-1} R_{\mathrm{p}_j}^{(1)}\{s[\mathrm{p}_j]/\alpha_{\mathrm{p}_j}\}^{\rceil \ell} \mid \prod_{j=k+1}^{m} N_j^{(i)}$. Then, by Clause (2) we have that $R_{\mathrm{p}_j}^{(1)}\{s[\mathrm{p}_j]/\alpha_{\mathrm{p}_j}\}^{\rceil \ell} \, \mathscr{R}_{\circ}^{\mathscr{L}} \, R_{\mathrm{p}_j}^{(1)}\{s[\mathrm{p}_j]/\alpha_{\mathrm{p}_j}\}^{\rceil \ell}$ for each j such that $1 \leq j \leq k-1$, and by hypothesis we have $N_j^{(1)} \, \mathscr{R}_{\circ}^{\mathscr{L}} \, N_j^{(2)}$ for each j such that $k+1 \leq j \leq m$, either by Clause (1) or by Clause (2). Then $M_1^{\prime} \, \mathscr{R}_{\circ}^{\mathscr{L}} \, M_2^{\prime}$ by Clause (3) and hence $(\rhd\rhd)$ is the required matching transition for $(\rhd)$.

ii) If $\ell \notin \mathscr{L}$, we may assume without loss of generality that there exists $k' \leq k-1$ such that $\mu_j \in \mathscr{L}$ for $1 \leq j \leq k'$ and $\mu_j \notin \mathscr{L}$ for $k'+1 \leq j \leq k-1$. For the initiator we could have $\mu_k \in \mathscr{L}$ or $\mu_k \notin \mathscr{L}$. Note that it could be that $\mu_j \in \mathscr{L}$ for each j such that $1 \leq j \leq k-1$, i.e. $k' = k-1$, and $\mu_k \in \mathscr{L}$. In this case we proceed as in case i) above, except that we use Clause (1) to deduce $R_{\mathtt{p}_j}\{s[\mathtt{p}_j]/\alpha_{\mathtt{p}_j}\}^{\rceil\ell}\,\mathscr{R}_\circ^{\mathscr{L}}\,R_{\mathtt{p}_j}\{s[\mathtt{p}_j]/\alpha_{\mathtt{p}_j}\}^{\rceil\ell}$ for each j such that $1 \leq j \leq k-1$.
So, let us assume that $k' < k-1$ and $\mu_k \notin \mathscr{L}$ (the case where $\mu_k \in \mathscr{L}$ is similar and simpler). Then $N_j^{(1)} = N_j^{(2)}$ for $1 \leq j \leq k'$, because $N_j^{(1)}\,\mathscr{R}_\circ^{\mathscr{L}}\,N_j^{(2)}$ follows necessarily from Clause (2). Instead, $N_j^{(1)}\,\mathscr{R}_\circ^{\mathscr{L}}\,N_j^{(2)}$ for $k'+1 \leq j \leq k$ follows necessarily from Clause (1). Define now:

$$M_1' = \prod_{j=1}^{k-1} R_{\mathtt{p}_j}\{s[\mathtt{p}_j]/\alpha_{\mathtt{p}_j}\}^{\rceil\ell} \mid \mathbf{0}^{\rceil\ell} \mid \prod_{j=k+1}^{m} N_j^{(1)}$$
$$M_2' = \prod_{j=1}^{k'} a[\mathtt{p}_j](\alpha_{\mathtt{p}_j}).{R_{\mathtt{p}_j}}^{\rceil\ell} \mid \prod_{j=k'+1}^{m} N_j^{(2)}$$

Note that the component $\mathbf{0}^{\rceil\ell}$ in M_1' is used to match $N_k^{(2)}$ in M_2'. The safety of M_1 implies the safety of all $R_{\mathtt{p}_j}\{s[\mathtt{p}_j]/\alpha_{\mathtt{p}_j}\}^{\rceil\ell}$, which in turn imply the safety of each $a^\ell[\mathtt{p}_j](\alpha_{\mathtt{p}_j}).{R_{\mathtt{p}_j}}^{\rceil\ell}$ for $1 \leq j \leq k'$.
Notice that $M_1'\,\mathscr{R}_\circ^{\mathscr{L}}\,M_2'$ by Clause (3), because the first k components are related by Clause (1), and the remaining ones are related by hypothesis either by Clause (1) or by Clause (2). Let $P_1'' = \lfloor M_1' \rfloor$. Since $P_1'' \equiv P_1'$ and $P_2 = \lfloor M_2' \rfloor$, by Definition 7.7 we have $P_1'\,\mathscr{R}^{\mathscr{L}}\,P_2$. Thus, since $H_1 \cup \{s:\varepsilon\} =_{\mathscr{L}} H_1 =_{\mathscr{L}} H_2$, we may use the empty move of $< P_2 \,,\, H_2 >$ to match the move $(\triangleright)$ of $< P_1 \,,\, H_1 >$.

— **Clause (4).** If Rule [Def] is used to infer the transition of $< P_1 \leftarrow Z \,,\, H_1 >$, then, by definition of distributed monitored processes, there is only one monitor controlling the process variable which is replaced. Therefore we can use Clause (1) or (2). Otherwise, Rule [Cont] is used and induction applies.

□

The converse of Theorem 7.8 does not hold, as shown by the process R of Example 4.12, or by the following more classical example:

Example 7.9. *(Secure unsafe process)* Consider the process $P = P_1 \mid P_2$, whose second participant contains a high conditional that emits two equal low messages in its branches.

$$\begin{array}{rcl} P_1 & = & \mathsf{def}\ X(x,\alpha) = Q_1\ \mathsf{in}\ X\langle \mathsf{true}^\top, s[1]\rangle \\ P_2 & = & \mathsf{def}\ Y(y,\beta) = Q_2\ \mathsf{in}\ Y\langle \mathsf{false}^\bot, s[2]\rangle \end{array}$$

where $Q_1 = \alpha!\langle 2,x\rangle.X(x,\alpha)$ and $Q_2 = \beta?(1,z^\top).\mathsf{if}\ z^\top\ \mathsf{then}\ \beta!\langle 3,y\rangle.Y(y,\beta)\ \mathsf{else}\ \beta!\langle 3,y\rangle.Y(y,\beta)$. Let us first argue that this process is secure. Notice that, since the first participant persistently emits the message $(1,2,\mathsf{true}^\top)$, the input of the second participant is always enabled: then, even considering the worst case where P is run with the $\bot$-equal $\mathbf{Q}$-sets $H_1 = \{s:(1,2,\mathsf{false}^\top)\}$ and $H_2 = \{s:\varepsilon\}$, the moves:

$$< P \,,\, H_1 > \longrightarrow^* < P_1 \mid \mathsf{def}\ Y(y,\beta) = Q_2\ \mathsf{in}\ \mathsf{if}\ \mathsf{false}^\top\ \mathsf{then}\ Q\ \mathsf{else}\ Q \,,\, \{s:\varepsilon\} >$$

where $Q = s[2]!\langle 3, \mathsf{false}^{\perp}\rangle.Y\langle \mathsf{false}^{\perp}, s[2]\rangle$, can be simulated by:

$$< P \,,\, H_2 > \longrightarrow^* < P \,,\, \{s : (1,2,\mathsf{true}^{\top})\} > \longrightarrow^*$$
$$< P_1 \mid \mathsf{def}\ Y(y,\beta) = Q_2\ \mathsf{in\ if}\ \mathsf{true}^{\top}\ \mathsf{then}\ Q\ \mathsf{else}\ Q \,,\, \{s : \varepsilon\} >$$

and clearly the two continuations are $\perp$-bisimilar, since without touching the **Q**-set they both evolve to

$$P_1 \mid \mathsf{def}\ Y(y,\beta) = Q_2\ \mathsf{in}\ Q$$

On the other hand, P is clearly not safe since in both branches of the conditional it tries to perform a $\perp$-output after having set the monitor to $\top$ when receiving the $\top$-input.

8. Type System

In this section we present our security type system, which will be shown to imply safety. This type system is a restriction to our calculus of the system given for a richer calculus in (Capecchi et al., 2013), to which we refer for motivations, explanations and the proof of subject reduction.

8.1. *Typing expressions and processes*

As standard for typing processes we need two kinds of types, *global types* and *session types*.

Global	G	::=	$\mathtt{p} \to \Pi : \langle S^{\ell}\rangle.G$	*communication*
		|	$\mathtt{p} \to \Pi : \{\lambda_i : G_i\}^{\ell}_{i\in I}$	*choice*
		|	$\mathbf{t}$	*variable*
		|	$\mu\mathbf{t}.G$	*recursive*
		|	end	*end*
Session	T	::=	$!\langle \Pi, S^{\ell}\rangle;T$	*send*
		|	$?(\mathtt{p}, S^{\ell});T$	*receive*
		|	$\oplus^{\ell}\langle \Pi, \{\lambda_i : T_i\}_{i\in I}\rangle$	*selection*
		|	$\&^{\ell}(\mathtt{p}, \{\lambda_i : T_i\}_{i\in I})$	*branching*
		|	$\mu\mathbf{t}.T$	*recursive*
		|	$\mathbf{t}$	*variable*
		|	end	*end*
Sorts	S	::=	$\mathsf{bool} \mid \ldots$	

Table 8. *Global and session types.*

The grammar of global and session types is given in Table 8. Global types represent the whole session protocol and session types correspond to the communication actions, representing each participant's contribution to the session. We comment on global types and session types in turn.

The *communication* type $\mathtt{p} \to \Pi : \langle S^{\ell}\rangle.G$ says that participant $\mathtt{p}$ multicasts a message of type S and level ℓ to all participants in Π and then the interactions described in G take place. The *choice* type $\mathtt{p} \to \Pi : \{\lambda_i : G_i\}^{\ell}_{i\in I}$ says that participant $\mathtt{p}$ multicasts one of the labels λ_i to the participants

in Π. If λ_j is sent, interactions described in G_j take place. Type $\mu\mathbf{t}.G$ is a recursive type, where the type variable $\mathbf{t}$ is guarded in the standard way. Type end represents the termination of a session.

The *send* type $!\langle\Pi, S^\ell\rangle;T$ expresses the sending to all participants in Π of a value of type S and of level ℓ, followed by the communications described in T. The *selection* type $\oplus^\ell\langle\Pi, \{\lambda_i : T_i\}_{i\in I}\rangle$ represents the transmission to all participants in Π of a label λ_j in $\{\lambda_i \mid i \in I\}$, followed by the communications described in T_j. The *receive* and *branching* types are dual to the *send* and *selection* ones. Recursive types $\mu\mathbf{t}.G$ and $\mu\mathbf{t}.T$ are considered modulo folding and unfolding.

The relation between global types and session types is formalised by the notion of projection (Honda et al., 2008). The *projection of G onto* $\mathtt{q}$, denoted $(G \upharpoonright \mathtt{q})$, gives participant $\mathtt{q}$'s view of the protocol described by G. It is defined by induction on global types in Table 9. For the choice global type, the condition $G_i \upharpoonright \mathtt{q} = G_j \upharpoonright \mathtt{q}$ for all $i, j \in I$ assures that the projections of all the participants not involved in the branching are identical session types.

$$(\mathtt{p} \to \Pi : \langle S^\ell\rangle.G') \upharpoonright \mathtt{q} = \begin{cases} !\langle\Pi, S^\ell\rangle;(G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}, \\ ?(\mathtt{p}, S^\ell);(G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} \in \Pi, \\ G' \upharpoonright \mathtt{q} & \text{otherwise.} \end{cases}$$

$$(\mathtt{p} \to \Pi : \{\lambda_i : G_i\}^\ell_{i\in I}) \upharpoonright \mathtt{q} = \begin{cases} \oplus^\ell(\Pi, \{\lambda_i : G_i \upharpoonright \mathtt{q}\}_{i\in I}) & \text{if } \mathtt{q} = \mathtt{p} \\ \&^\ell(\mathtt{p}, \{\lambda_i : G_i \upharpoonright \mathtt{q}\}_{i\in I}) & \text{if } \mathtt{q} \in \Pi \\ G_1 \upharpoonright \mathtt{q} & \text{if } \mathtt{q} \neq \mathtt{p}, \mathtt{q} \notin \Pi \text{ and } G_i \upharpoonright \mathtt{q} = G_j \upharpoonright \mathtt{q}\ \forall i, j \in I. \end{cases}$$

$$(\mu\mathbf{t}.G') \upharpoonright \mathtt{q} = \begin{cases} \mu\mathbf{t}.(G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} \text{ occurs in } G', \\ \mathsf{end} & \text{otherwise.} \end{cases} \qquad \mathbf{t} \upharpoonright \mathtt{q} = \mathbf{t} \qquad \mathsf{end} \upharpoonright \mathtt{q} = \mathsf{end}$$

Table 9. *Projection of global types on participants.*

$$\mathsf{Meet}(\,!\langle\Pi, S^\ell\rangle;T) = \ell \sqcap \mathsf{Meet}(T) \qquad \mathsf{Meet}(?(\mathtt{p}, S^\ell);T) = \ell \qquad \mathsf{Meet}(\oplus^\ell\langle\Pi, \{\lambda_i : T_i\}_{i\in I}\rangle) = \ell$$
$$\mathsf{Meet}(\&^\ell(\mathtt{p}, \{\lambda_i : T_i\}_{i\in I})) = \ell \qquad \mathsf{Meet}(\mu\mathbf{t}.T) = \mathsf{Meet}(T) \qquad \mathsf{Meet}(\mathbf{t}) = \mathsf{Meet}(\mathsf{end}) = \top$$

Table 10. *Meet of session types.*

The *typing judgments for expressions* are of the form:

$$\Gamma \vdash e : S^\ell$$

where Γ is the *standard environment* which maps variables with security levels to sort types with the same security levels, service names with security levels to global types with the same security levels, and process variables to pairs of sort types with their security levels and session types. Formally, we define:

$$\Gamma ::= \emptyset \mid \Gamma, x^\ell : S^\ell \mid \Gamma, a^\ell : G^\ell \mid \Gamma, X : S^\ell\ T$$

assuming that we can write $\Gamma, x^\ell : S^{\ell'}$ only if x does not occur in Γ, that is, standard environments should not contain the same variable twice. Similarly for $\Gamma, a^\ell : G^\ell$ and $\Gamma, X : S^\ell\ T$.

$$\frac{\Gamma \vdash_\ell P \triangleright \Delta \quad \ell' \leq \ell}{\Gamma \vdash_{\ell'} P \triangleright \Delta} \lfloor \textsc{SubS} \rfloor \qquad \frac{\Delta \text{ end only}}{\Gamma \vdash_\top \mathbf{0} \triangleright \Delta} \lfloor \textsc{Inact} \rfloor \qquad \frac{\Gamma \vdash_\ell P \triangleright \Delta \quad \Gamma \vdash_\ell Q \triangleright \Delta'}{\Gamma \vdash_\ell P \mid Q \triangleright \Delta, \Delta'} \lfloor \textsc{Conc} \rfloor$$

$$\frac{\Gamma \vdash e : \mathsf{bool}^{\ell'} \quad \Gamma \vdash_\ell P \triangleright \Delta \quad \Gamma \vdash_\ell Q \triangleright \Delta}{\Gamma \vdash_\ell \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright \Delta} \lfloor \textsc{If} \rfloor \qquad \frac{\Gamma \vdash e : S^\ell \quad \ell' = \mathsf{Meet}(T) \quad \Delta \text{ end only}}{\Gamma, X : S^\ell\ T \vdash_{\ell \sqcap \ell'} X\langle e, c \rangle \triangleright \Delta, c : T} \lfloor \textsc{Var} \rfloor$$

$$\frac{\Gamma, X : S^\ell\ T, x^\ell : S^\ell \vdash_{\ell \sqcap \ell'} P \triangleright \{\alpha : T\} \quad \ell' = \mathsf{Meet}(T) \qquad \Gamma, X : S^\ell\ T \vdash_{\ell''} Q \triangleright \Delta}{\Gamma \vdash_{\ell''} \mathsf{def}\ X(x, \alpha) = P\ \mathsf{in}\ Q \triangleright \Delta} \lfloor \textsc{Def} \rfloor$$

$$\frac{n \text{ is the arity of } a}{\Gamma, a^\ell : G^\ell \vdash_\ell \bar{a}^\ell[n] \triangleright \emptyset} \lfloor \textsc{MInit} \rfloor \qquad \frac{\Gamma, a^\ell : G^\ell \vdash_\ell P \triangleright \Delta, \alpha : G \upharpoonright \mathtt{p}}{\Gamma, a^\ell : G^\ell \vdash_\ell a^\ell[\mathtt{p}](\alpha).P \triangleright \Delta} \lfloor \textsc{MAcc} \rfloor$$

$$\frac{\Gamma \vdash e : S^\ell \quad \Gamma \vdash_{\ell'} P \triangleright \Delta, c : T \quad \ell' \leq \ell}{\Gamma \vdash_{\ell'} c!\langle \Pi, e \rangle.P \triangleright \Delta, c : !\langle \Pi, S^\ell \rangle; T} \lfloor \textsc{Send} \rfloor \qquad \frac{\Gamma, x^\ell : S^\ell \vdash_\ell P \triangleright \Delta, c : T}{\Gamma \vdash_\ell c?(\mathtt{p}, x^\ell).P \triangleright \Delta, c : ?(\mathtt{p}, S^\ell); T} \lfloor \textsc{Rcv} \rfloor$$

$$\frac{\Gamma \vdash_\ell P \triangleright \Delta, c : T_j \quad j \in I \quad \ell \leq \ell' \leq \bigsqcap_{i \in I} \mathsf{Meet}(T_i)}{\Gamma \vdash_\ell c \oplus^{\ell'} \langle \Pi, \lambda_j \rangle.P \triangleright \Delta, c : \oplus^{\ell'} \langle \Pi, \{\lambda_i : T_i\}_{i \in I} \rangle} \lfloor \textsc{Sel} \rfloor$$

$$\frac{\Gamma \vdash_\ell P_i \triangleright \Delta, c : T_i \quad \ell \leq \bigsqcap_{i \in I} \mathsf{Meet}(T_i)}{\Gamma \vdash_\ell c \&^\ell (\mathtt{p}, \{\lambda_i : P_i\}_{i \in I}) \triangleright \Delta, c : \&^\ell (\mathtt{p}, \{\lambda_i : T_i\}_{i \in I})} \lfloor \textsc{Branch} \rfloor$$

Table 11. *Typing rules for processes.*

We type values by decorating their types with their security levels, and variables/service names according to Γ:

$$\Gamma \vdash \mathsf{true}^\ell, \mathsf{false}^\ell : \mathsf{bool}^\ell \qquad \Gamma, x^\ell : S^\ell \vdash x^\ell : S^\ell \qquad \Gamma, a^\ell : S^\ell \vdash a^\ell : S^\ell$$

We type expressions by decorating their types with the join of the security levels of the variables and values they are built from. For example a typing rule for and is:

$$\frac{\Gamma \vdash e_1 : \mathsf{bool}^{\ell_1} \quad \Gamma \vdash e_2 : \mathsf{bool}^{\ell_2}}{\Gamma \vdash e_1 \text{ and } e_2 : \mathsf{bool}^{\ell_1 \sqcup \ell_2}}$$

The meet of session types (Table 10) takes into account the lowest level of exchanged values and choices. Notice that only for the send type we need to consider the "future" exchanges, since the typing rules for the other constructors already do this job (see Table 11). The meet is used in the type system rules $\lfloor \textsc{Sel} \rfloor$ and $\lfloor \textsc{Branch} \rfloor$ for information flow control (see Table 11).

The *typing judgments for processes* are of the form:

$$\Gamma \vdash_\ell P \triangleright \Delta$$

where Δ is a *process environment* which associates session types with channels:

$$\Delta ::= \emptyset \mid \Delta, c : T$$

The merge "Δ,Δ'" of process environments is defined only if $dom(\Delta) \cap dom(\Delta') = \emptyset$, where $dom(\ )$ is as expected.

We decorate the derivation symbol $\vdash$ with the security level ℓ inferred for the process: this level is a lower bound for the communications performed in the process. The set of typing rules for processes is given in Table 11.

$\lfloor$SUBS$\rfloor$ is the classical subtyping rule of security type systems. It is crucial for some rule premises to hold, since it allows the security level inferred for a process to be decreased.

$\lfloor$INACT$\rfloor$ gives security level $\top$ to $\mathbf{0}$ since a terminated process cannot leak any information; in the premise, "Δ end only" means that each channel in $dom(\Delta)$ has session type end.

$\lfloor$CONC$\rfloor$ allows the parallel composition of two processes P,Q to be typed if both processes are typable and their process environments have disjoint domains.

$\lfloor$IF$\rfloor$ requires that the two branches P,Q of a conditional be typed with the same process environment Δ, and with the same security level ℓ. This rule is consistent with the monitored semantics rules $\lfloor$MIF-T$\rfloor$ $\lfloor$MIF-F$\rfloor$ which do not raise the monitoring level when executing a conditional. Indeed, the classical requirement that the security level ℓ' of the tested expression be less than or equal to the security level ℓ of the branches is not needed here because the process if e then P else Q can only be executed if the expression e can be evaluated, and this is the case only if all the variables occurring in e have been instantiated by previous inputs. Since these inputs guard the conditional, the join of their levels will be less than or equal to ℓ by Rule $\lfloor$RCV$\rfloor$.

$\lfloor$VAR$\rfloor$ types $X\langle e,c\rangle$ with a level which is the meet of the security levels of expression e and the security levels of the communications performed on channel c. In this way we take into account the levels of all communications which can be performed by a process bound to X.

In $\lfloor$DEF$\rfloor$ the process P is typed with the meet of the security levels associated with x and α, in agreement with rule $\lfloor$VAR$\rfloor$. The levels of P and Q can be unrelated, since it is possible that X does not occur in Q. Clearly if X occurs in Q we get $\ell'' \leq \ell \sqcap \ell'$.

In rules $\lfloor$MINIT$\rfloor$ and $\lfloor$MACC$\rfloor$ the standard environment must associate with the identifier a a global type. In $\lfloor$MINIT$\rfloor$ the emptiness of the process environment in the conclusion specifies that there is no further communication behaviour after the initiator. In $\lfloor$MACC$\rfloor$ the premise guarantees that the type of the channel α in P is the p-th projection of the global type G of a. Concerning security levels, in $\lfloor$MACC$\rfloor$ we check that the continuation process P conforms to the security level ℓ associated with the service name a.

$\lfloor$SEND$\rfloor$ types the sending of a basic value followed by the process P. The premise makes sure that i) the expression e has type S^{ℓ} in Γ, where ℓ is the join of all variables and values in e, and ii) P is typable in Γ. The condition $\ell' \leq \ell$ is not a constraint on P, since whatever level ℓ'' is inferred for P, it can always be downgraded to such an ℓ'; this condition simply ensures that the output process cannot be typed with a level higher than or incomparable with ℓ, thus preserving the invariant that the level ℓ' of a process is a lower bound for all security levels of its communications.

$\lfloor$RCV$\rfloor$ is the dual of rule $\lfloor$SEND$\rfloor$ but it is more restrictive, in that it requires the continuation P to be typable with level ℓ. This means that P cannot have any actions of level lower than or incomparable with ℓ. This rule forbids a $\top$-input followed by a $\bot$-output, for instance.

$\lfloor$SEL$\rfloor$ types a selecting process with the level associated with the selection operator, which is less than or equal to the meets of the types in the different possible continuations. $\lfloor$BRANCH$\rfloor$ is the dual of the $\lfloor$SEL$\rfloor$ rule.

8.2. *Typing queues and* **Q***-sets*

We type queues by describing the messages they contain: *message types* represent the messages contained in the queues, see Table 12. The *message value send type* $!\langle \Pi, S^\ell \rangle$, expresses the communication to all $\mathtt{p} \in \Pi$ of a value of type S^ℓ. The *message selection type* $\oplus^\ell \langle \Pi, \lambda \rangle$ represents the communication to all $\mathtt{p} \in \Pi$ of the label λ with level ℓ and $\mathbf{T};\mathbf{T}'$ represents sequencing of message types (we assume associativity for ;).

In order to take into account the structural congruence on queues given in Table 2, we consider message types modulo the equivalence relation $\approx$ induced by the rules shown in Table 12 (with $\natural \in \{!, \oplus^\ell\}$ and $A \in \{S^\ell, \lambda\}$).

$$\begin{array}{llll} \text{Message} & \mathbf{T} ::= & !\langle \Pi, S^\ell \rangle & \textit{message send} \quad | \quad \oplus^\ell \langle \Pi, \lambda \rangle \quad \textit{message selection} \\ & \quad | & \mathbf{T};\mathbf{T}' & \textit{message sequence} \end{array}$$

$$\begin{array}{c} \mathbf{T};\, \natural\langle \Pi, A\rangle;\, \natural'\langle \Pi', A\rangle;\mathbf{T}' \approx \mathbf{T};\, \natural'\langle \Pi', A\rangle;\, \natural\langle \Pi, A\rangle;\mathbf{T}' \quad \text{if } \Pi \cap \Pi' = \emptyset \\ \mathbf{T};\, \natural\langle \Pi, A\rangle;\mathbf{T}' \approx \mathbf{T};\, \natural\langle \Pi', A\rangle;\, \natural\langle \Pi'', A\rangle;\mathbf{T}' \quad \text{if } \Pi = \Pi' \cup \Pi'', \Pi' \cap \Pi'' = \emptyset \end{array}$$

Table 12. *Message types and equivalence relation on message types.*

$$\frac{}{\Gamma \vdash s : \varepsilon \triangleright \emptyset}\ \lfloor \textsc{QInit} \rfloor \qquad \frac{\Gamma \vdash s : h \triangleright \Theta \qquad \Gamma \vdash v^\ell : S}{\Gamma \vdash s : h \cdot (\mathtt{p}, \Pi, v^\ell) \triangleright \Theta; \{s[\mathtt{p}] : !\langle \Pi, S\rangle\}}\ \lfloor \textsc{QSend} \rfloor$$

$$\frac{\Gamma \vdash s : h \triangleright \Theta}{\Gamma \vdash s : h \cdot (\mathtt{p}, \Pi, \lambda^\ell) \triangleright \Theta; \{s[\mathtt{p}] : \oplus^\ell \langle \Pi, \lambda\rangle\}}\ \lfloor \textsc{QSel} \rfloor \qquad \frac{\Gamma \vdash s : h \triangleright \Theta \qquad \Theta \approx \Theta'}{\Gamma \vdash s : h \triangleright \Theta'}\ \lfloor \textsc{QCong} \rfloor$$

$$\frac{}{\Gamma \vdash_\emptyset \emptyset \triangleright \emptyset}\ \lfloor \textsc{QSInit} \rfloor \qquad \frac{\Gamma \vdash_\Sigma H \triangleright \Theta \qquad \Gamma \vdash s : h \triangleright \Theta'}{\Gamma \vdash_{\Sigma, s} H \cup \{s : h\} \triangleright \Theta, \Theta'}\ \lfloor \textsc{QSUnion} \rfloor$$

Table 13. *Typing rules for queues and* **Q***-sets.*

Typing judgments for queues have the shape

$$\Gamma \vdash s : h \triangleright \Theta$$

where Θ is a *queue environment* associating message types with channels with role:

$$\Theta ::= \emptyset \mid \Theta, s[\mathtt{p}] : \mathbf{T}$$

Typing judgments for **Q***-sets* have the shape:

$$\Gamma \vdash_\Sigma H \triangleright \Theta$$

where Σ is the set of session names which occur in H.

The equivalence $\approx$ on message types can be trivially extended to queue environments:

$$\{s[\mathtt{p}_i] : \mathbf{T}_i \mid i \in I\} \approx \{s[\mathtt{p}_i] : \mathbf{T}'_i \mid i \in I\} \text{ if } \mathbf{T}_i \approx \mathbf{T}'_i \text{ for all } i \in I$$

Typing rules for queues and **Q**-sets are given in Table 13. The composition ";" of queue environments is defined by:

$$\begin{array}{rcl}\Theta;\Theta' & = & \{s[\mathtt{p}] : \mathbf{T};\mathbf{T}' \mid s[\mathtt{p}] : \mathbf{T} \in \Theta \ \textit{and} \ s[\mathtt{p}] : \mathbf{T}' \in \Theta'\} \cup \\ & & \{s[\mathtt{p}] : \mathbf{T} \mid s[\mathtt{p}] : \mathbf{T} \in \Theta \ \textit{and} \ s[\mathtt{p}] \notin \mathit{dom}(\Theta')\} \cup \\ & & \{s[\mathtt{p}] : \mathbf{T}' \mid s[\mathtt{p}] \notin \mathit{dom}(\Theta) \ \textit{and} \ s[\mathtt{p}] : \mathbf{T}' \in \Theta'\}\end{array}$$

In Table 13, the merge "Σ, Σ'" of name sets and the merge "Θ, Θ'" of queue environments are defined only if $\Sigma \cap \Sigma' = \emptyset$ and $\mathit{dom}(\Theta) \cap \mathit{dom}(\Theta') = \emptyset$, respectively.

8.3. *Typing configurations*

Typing judgments for runtime configurations C have the form:

$$\Gamma \vdash_\Sigma C \triangleright < \Delta \diamond \Theta >$$

They associate with a configuration the environments Δ and Θ mapping channels to session and message types, respectively. We call $< \Delta \diamond \Theta >$ a *configuration environment*.

A *configuration type* is a session type, or a message type, or a message type followed by a session type:

$$\begin{array}{lcllll}\text{Configuration} & \mathscr{T} & ::= & T & \textit{session} & \\ & & \mid & \mathbf{T};T & \textit{continuation} & \mid \ \mathbf{T} \quad \textit{message}\end{array}$$

Since channels with role occur both in processes and in queues, a configuration environment associates configuration types with these channels.

Definition 8.1. The configuration type of a channel $s[\mathtt{p}]$ in a configuration environment $< \Delta \diamond \Theta >$ (notation $< \Delta \diamond \Theta > (s[\mathtt{p}])$) is defined by:

$$< \Delta \diamond \Theta > (s[\mathtt{p}]) = \begin{cases} \mathbf{T};T & \text{if } s[\mathtt{p}] : \mathbf{T} \in \Theta \text{ and } s[\mathtt{p}] : T \in \Delta, \\ \mathbf{T} & \text{if } s[\mathtt{p}] : \mathbf{T} \in \Theta \text{ and } s[\mathtt{p}] \notin \mathit{dom}(\Delta), \\ T & \text{if } s[\mathtt{p}] \notin \mathit{dom}(\Theta) \text{ and } s[\mathtt{p}] : T \in \Delta, \\ \mathsf{end} & \text{otherwise.} \end{cases}$$

Configuration types can be projected on participants (notation $\mathscr{T} \upharpoonright \mathtt{q}$), see Table 14.

We also define a duality relation $\bowtie$ between projections of configuration types, which holds when opposite communications are offered (input/output, selection/branching).

Definition 8.2. The *duality relation* between projections of configuration types is the minimal symmetric relation which satisfies:

$$\begin{array}{c}\mathsf{end} \bowtie \mathsf{end} \qquad \mathbf{t} \bowtie \mathbf{t} \qquad T \bowtie T' \quad \Rightarrow \quad \mu\mathbf{t}.T \bowtie \mu\mathbf{t}.T' \\ \forall i \in I \ T_i \bowtie T_i' \quad \Rightarrow \quad \oplus^\ell\{\lambda_i : T_i\}_{i\in I} \bowtie \&^\ell\{\lambda_i : T_i'\}_{i\in I} \\ \mathscr{T} \bowtie T \quad \Rightarrow \quad !S^\ell;\mathscr{T} \bowtie ?S^\ell;T \\ \exists i \in I \ \lambda = \lambda_i \ \text{ and } \ \mathscr{T} \bowtie T_i \quad \Rightarrow \quad \oplus^\ell\lambda;\mathscr{T} \bowtie \&^\ell\{\lambda_i : T_i\}_{i\in I}\end{array}$$

The above definitions are needed to state coherence of configuration environments. Informally, this holds when the inputs and the branchings offered by the process agree both with the outputs and the selections offered by the process and with the messages in the queues. More formally:

$$(\,!\langle\Pi,S^\ell\rangle;\mathscr{T})\upharpoonright \mathtt{q}=\begin{cases}!S^\ell;\mathscr{T}\upharpoonright \mathtt{q} & \text{if } \mathtt{q}\in\Pi,\\ \mathscr{T}\upharpoonright \mathtt{q} & \text{otherwise.}\end{cases}\qquad (?(\mathtt{p},S^\ell);T)\upharpoonright \mathtt{q}=\begin{cases}?S^\ell;T\upharpoonright \mathtt{q} & \text{if } \mathtt{q}=\mathtt{p},\\ T\upharpoonright \mathtt{q} & \text{otherwise.}\end{cases}$$

$$(\oplus^\ell\langle\Pi,\lambda\rangle;\mathscr{T})\upharpoonright \mathtt{q}=\begin{cases}\oplus^\ell\lambda;\mathscr{T}\upharpoonright \mathtt{q} & \text{if } \mathtt{q}\in\Pi,\\ \mathscr{T}\upharpoonright \mathtt{q} & \text{otherwise.}\end{cases}$$

$$(\oplus^\ell\langle\Pi,\{\lambda_i:T_i\}_{i\in I}\rangle)\upharpoonright \mathtt{q}=\begin{cases}\oplus^\ell\{\lambda_i:T_i\upharpoonright \mathtt{q}\}_{i\in I} & \text{if } \mathtt{q}\in\Pi,\\ T_1\upharpoonright \mathtt{q} & \text{otherwise.}\end{cases}$$

$$(\&^\ell(\mathtt{p},\{\lambda_i:T_i\}_{i\in I}))\upharpoonright \mathtt{q}=\begin{cases}\&^\ell\{\lambda_i:T_i\upharpoonright \mathtt{q}\}_{i\in I} & \text{if } \mathtt{q}=\mathtt{p},\\ T_1\upharpoonright \mathtt{q} & \text{otherwise.}\end{cases}$$

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

Table 14. *Projection of configuration types on participants.*

Definition 8.3. A configuration environment $<\Delta\diamond\Theta>$ is *coherent* if $s[\mathtt{p}]\in dom(\Delta)\cup dom(\Theta)$ and $s[\mathtt{q}]\in dom(\Delta)\cup dom(\Theta)$ imply

$$<\Delta\diamond\Theta>(s[\mathtt{p}])\upharpoonright \mathtt{q}\bowtie <\Delta\diamond\Theta>(s[\mathtt{q}])\upharpoonright \mathtt{p}$$

It can be easily shown that typing rules assure that configurations are always typed with coherent environments.

$$\frac{\Gamma\vdash_\ell P\triangleright\Delta\quad \Gamma\vdash_\Sigma H\triangleright\Theta\quad <\Delta\diamond\Theta> \text{ is coherent}}{\Gamma\vdash_\Sigma <P\,,\,H>\triangleright <\Delta\diamond\Theta>}\ \lfloor\text{CINIT}\rfloor$$

$$\frac{\Gamma\vdash_{\Sigma_1} C_1\triangleright <\Delta_1\diamond\Theta_1>\quad \Gamma\vdash_{\Sigma_2} C_2\triangleright <\Delta_2\diamond\Theta_2>\quad <\Delta_1,\Delta_2\diamond\Theta_1,\Theta_2> \text{ is coherent}}{\Gamma\vdash_{\Sigma_1,\Sigma_2} C_1\|C_2\triangleright <\Delta_1,\Delta_2\diamond\Theta_1,\Theta_2>}\ \lfloor\text{CPAR}\rfloor$$

$$\frac{\Gamma\vdash_\Sigma C\triangleright <\Delta\diamond\Theta>}{\Gamma\vdash_{\Sigma\setminus s}(\nu s)C\triangleright <\Delta\setminus s\diamond\Theta\setminus s>}\ \lfloor\text{GSRES}\rfloor$$

Table 15. *Typing rules for configurations.*

The typing rules for configurations are given in Table 15, where $\Sigma\setminus s$, $\Delta\setminus s$ and $\Theta\setminus s$ are defined as expected:

$$\Sigma\setminus s=\{s'|\ s'\in\Sigma \text{ and } s'\neq s\}\qquad \Delta\setminus s=\{s'[\mathtt{p}]:T|\ s'[\mathtt{p}]:T\in\Delta \text{ and } s'\neq s\}$$
$$\Theta\setminus s=\{s'[\mathtt{p}]:\mathbf{T}|\ s'[\mathtt{p}]:\mathbf{T}\in\Theta \text{ and } s'\neq s\}.$$

8.4. *Subject reduction*

Since process and queue environments represent future or ongoing communications, by reducing processes we get different configuration environments. This is formalised by the notion of reduction of configuration environments, denoted by $<\Delta\diamond\Theta>\ \Rightarrow\ <\Delta'\diamond\Theta'>$.

Definition 8.4 (Reduction of configuration environments). Let $\Rightarrow$ be the reflexive and transitive relation on configuration environments generated by:

1 $< \{s[\mathtt{p}] : !\langle \Pi, S^\ell \rangle;\ T\} \diamond \Theta > \ \Rightarrow\ < \{s[\mathtt{p}] : T\} \diamond \Theta; \{s[\mathtt{p}] : !\langle \Pi, S^\ell \rangle\} >$
2 $< \{s[\mathtt{p}] : \oplus^\ell \langle \Pi, \{\lambda_i : T_i\}_{i \in I} \rangle\} \diamond \Theta > \ \Rightarrow\ < \{s[\mathtt{p}] : T_j\} \diamond \Theta; \{s[\mathtt{p}] : \oplus^\ell (\Pi, \lambda_j)\} > \quad (j \in I)$
3 $< \{s[\mathtt{q}] : ?(\mathtt{p}, S^\ell);\ T\} \diamond \{s[\mathtt{p}] : !\langle \mathtt{q}, S^\ell \rangle\}; \Theta > \ \Rightarrow\ < \{s[\mathtt{q}] : T\} \diamond \Theta >$
4 $< \{s[\mathtt{q}] : \&^\ell (\mathtt{p}, \{\lambda_i : T_i\}_{i \in I})\} \diamond \{s[\mathtt{p}] : \oplus^\ell (\mathtt{q}, \lambda_j)\}; \Theta > \ \Rightarrow\ < \{s[\mathtt{q}] : T_j\} \diamond \Theta > \quad (j \in I)$
5 $< \Delta, \Delta'' \diamond \Theta > \ \Rightarrow\ < \Delta', \Delta'' \diamond \Theta' >$ if $< \Delta \diamond \Theta > \ \Rightarrow\ < \Delta' \diamond \Theta' >$

where message types are considered modulo the equivalence relation of Table 12.

The first two rules correspond to participant p putting a value or a label in the queue. The following two rules correspond to participant q reading a value or a label from the queue. The last rule is contextual: notice that we add only statements to the process environments, since only one statement is considered in the first four reduction rules, while we do not need to add statements to the queue environments, which are always arbitrary.

We are now able to state the result of type preservation under reduction. For the proof we refer to (Capecchi et al., 2013), where a similar type system was presented for a richer calculus (including declassification, delegation, as well as exchange and restriction of service names).

Theorem 8.5 (Subject Reduction). Suppose $\Gamma \vdash_\Sigma C \triangleright < \Delta \diamond \Theta >$ and $C \longrightarrow^* C'$. Then $\Gamma \vdash_\Sigma C' \triangleright < \Delta' \diamond \Theta' >$ with $< \Delta \diamond \Theta > \ \Rightarrow\ < \Delta' \diamond \Theta' >$. Moreover if $C =< P\ ,\ H >$, $C' =< P'\ ,\ H' >$ and $\Gamma \vdash_\ell P \triangleright \Delta$, then $\Gamma \vdash_\ell P' \triangleright \Delta'$.

Our type system also preserves the fundamental properties ensured by classical session types, i.e. linearity of communications inside sessions and absence of communication mismatches. This is easy to see, once we observe that our type system projects down to the classical session type system of (Bettini et al., 2008) when we ignore security levels and associated type constraints. We shall therefore concentrate here on establishing the soundness of our type system for safety.

9. Relating Safety and Typability

In this section we complete the picture set up in Section 7 by showing that typability implies safety. As a result, the relation among our three approaches for information flow analysis in multiparty sessions can be summarised as follows: typability $\Rightarrow$ safety $\Rightarrow$ security.

We consider substitutions, ranged over by σ, which map value variables to values and channel variables to channels with role. We say that σ agrees with Γ if $\Gamma \vdash x^\ell : S^\ell$ implies $\Gamma \vdash \sigma(x) : S^\ell$. It is easy to verify that $\Gamma \vdash_\ell P \triangleright \Delta$ implies $\Gamma \vdash_\ell \sigma(P) \triangleright \sigma(\Delta)$ for all σ that agree with Γ.

We prove now the following theorem, from which our result will follow as an easy corollary.

Theorem 9.1. If $\Gamma \vdash_\ell P \triangleright \Delta$ for some Γ, Δ, then $\sigma(P)^{\rceil \ell}$ is safe for all σ which agree with Γ.

Proof. By induction on the derivation of $\Gamma \vdash_\ell P \triangleright \Delta$ and then by case analysis on the last rule applied to infer the transition $< \sigma(P) \upharpoonleft Z\ ,\ H > \rightsquigarrow (\nu \tilde{s}) < P'\ ,\ H' >$, where Z is any tester.

$\dfrac{\Gamma \vdash_\ell P \triangleright \Delta \quad \ell' \leq \ell}{\Gamma \vdash_{\ell'} P \triangleright \Delta}$ $\lfloor$SUBS$\rfloor$ Inspecting the rules of Table 7 it is easy to verify that if $\sigma(P)^{\rceil \ell}$ reduces, then $\sigma(P)^{\rceil \ell'}$ reduces too for all $\ell' \leq \ell$. Therefore by definition the safety of $\sigma(P)^{\rceil \ell}$ implies the safety of $\sigma(P)^{\rceil \ell'}$ for all $\ell' \leq \ell$.

$\dfrac{\Delta \text{ end only}}{\Gamma \vdash_\top \mathbf{0} \triangleright \Delta}$ $\lfloor$INACT$\rfloor$ Since $\mathbf{0}$ cannot reduce, by definition $\mathbf{0}^{\rceil \ell}$ is trivially safe for all ℓ.

$\dfrac{\Gamma \vdash_\ell P \triangleright \Delta \quad \Gamma \vdash_\ell Q \triangleright \Delta'}{\Gamma \vdash_\ell P \mid Q \triangleright \Delta, \Delta'}\ \lfloor \textsc{Conc} \rfloor$ By induction $\sigma(P)^{\rceil\ell}$ and $\sigma(Q)^{\rceil\ell}$ are safe. The safety of $\sigma(P)^{\rceil\ell}$ and $\sigma(Q)^{\rceil\ell}$ implies the safety of $\sigma(P \mid Q)^{\rceil\ell}$ by compositionality (Corollary 6.7).

$\dfrac{\Gamma \vdash e : \mathsf{bool}^{\ell'} \quad \Gamma \vdash_\ell P \triangleright \Delta \quad \Gamma \vdash_\ell Q \triangleright \Delta}{\Gamma \vdash_\ell \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright \Delta}\ \lfloor \textsc{If} \rfloor$ By induction $\sigma(P)^{\rceil\ell}$ and $\sigma(Q)^{\rceil\ell}$ are safe. The process if e then P else Q reduces only if either $\sigma(e) \downarrow \mathsf{true}^{\ell'}$ or $\sigma(e) \downarrow \mathsf{false}^{\ell'}$. Then, taking the case $\sigma(e) \downarrow \mathsf{true}^{\ell'}$, the reduction if $\sigma(e)$ then $\sigma(P)$ else $\sigma(Q) \longrightarrow \sigma(P)$ can be mimicked by if $\sigma(e)$ then $\sigma(P)$ else $\sigma(Q)^{\rceil\ell} \multimap \sigma(P)^{\rceil\ell}$. Similarly for the case $\sigma(e) \downarrow \mathsf{false}^{\ell'}$.

$\dfrac{\Gamma \vdash e : S^\ell \quad \ell' = \mathsf{Meet}(T) \quad \Delta\ \text{end only}}{\Gamma, X : S^\ell\ T \vdash_{\ell \sqcap \ell'} X\langle e, c\rangle \triangleright \Delta, c : T}\ \lfloor \textsc{Var} \rfloor$ The process $X\langle e,c\rangle$ cannot reduce, so $X\langle e,c\rangle^{\rceil\ell}$ is trivially safe for all ℓ.

$\dfrac{\Gamma, X : S^\ell\ T, x^\ell : S^\ell \vdash_{\ell \sqcap \ell'} P \triangleright \{\alpha : T\} \quad \ell' = \mathsf{Meet}(T) \qquad \Gamma, X : S^\ell\ T \vdash_{\ell''} Q \triangleright \Delta}{\Gamma \vdash_{\ell''} \mathsf{def}\ X(x,\alpha) = P\ \mathsf{in}\ Q \triangleright \Delta}\ \lfloor \textsc{Def} \rfloor$ The process def $X(x,\alpha) = \sigma(P)$ in $\sigma(Q)$ can reduce only if $\sigma(Q) \equiv X\langle e, s[\mathtt{p}]\rangle \mid Q'$. In this case it reduces to $\sigma(P)\{v/x\}\{s[\mathtt{p}]/\alpha\} \mid Q'$, where $e \downarrow v^\ell$ and $\Gamma \vdash v^\ell : S^\ell$. Notice that x and α are bound, so they cannot occur in the domain of σ, and the substitution $\sigma\{v/x\}\{s[\mathtt{p}]/\alpha\}$ agrees with $\Gamma, X : S^\ell\ T, x^\ell : S^\ell$ whenever σ agrees with Γ. So by induction $\sigma(P)\{v/x\}\{s[\mathtt{p}]/\alpha\}^{\rceil\ell\sqcap\ell'}$ and $\sigma(Q)^{\rceil\ell''}$ are safe. Since $X\langle e, s[\mathtt{p}]\rangle \mid Q'$ must be typed using rules $\lfloor \textsc{Var} \rfloor$ and $\lfloor \textsc{Conc} \rfloor$, we get $\ell'' \leq \ell \sqcap \ell'$, thus $\sigma(P)\{v/x\}\{s[\mathtt{p}]/\alpha\}^{\rceil\ell''}$ is safe. The safety of $\sigma(Q)^{\rceil\ell''}$ implies by definition the safety of $\sigma(Q')^{\rceil\ell''}$. The safety of $\sigma(P)\{v/x\}\{s[\mathtt{p}]/\alpha\}^{\rceil\ell''}$ and $Q^{\rceil\ell''}$ imply the safety of $\sigma(P)\{v/x\}\{s[\mathtt{p}]/\alpha\} \mid Q'^{\rceil\ell''}$ by compositionality (Corollary 6.7).
In the monitored semantics we have $\mathsf{def}\ X(x,\alpha) = P\ \mathsf{in}\ \sigma(Q)^{\rceil\ell''} \multimap \sigma(P)\{v/x\}\{s[\mathtt{p}]/\alpha\} \mid Q'^{\rceil\ell''}$. Therefore $\mathsf{def}\ X(x,\alpha) = P\ \mathsf{in}\ \sigma(Q)^{\rceil\ell''}$ is safe.

$\dfrac{n \text{ is the arity of } a}{\Gamma, a^\ell : G^\ell \vdash_\ell \bar{a}^\ell[n] \triangleright \emptyset}\ \lfloor \textsc{MInit} \rfloor$ The process $\bar{a}^\ell[n]$ reduces to $(\nu s) < \mathbf{0}\ ,\ s : \varepsilon >$ with the tester which contains all the required n participants. In the monitored semantics too, the process $\bar{a}^\ell[n]^{\rceil\ell}$ with the same tester reduces to $(\nu s) < \mathbf{0}\ ,\ s : \varepsilon >^{\rceil\ell}$, so $\bar{a}^\ell[n]^{\rceil\ell}$ is safe.

$\dfrac{\Gamma, a^\ell : G^\ell \vdash_\ell P \triangleright \Delta, \alpha : G \upharpoonright \mathtt{p}}{\Gamma, a^\ell : G^\ell \vdash_\ell a^\ell[\mathtt{p}](\alpha).P \triangleright \Delta}\ \lfloor \textsc{MAcc} \rfloor$ The process $a^\ell[\mathtt{p}](\alpha).\sigma(P)$ reduces to
$(\nu s) < \sigma(P)\{s[\mathtt{p}]/\alpha\}\ ,\ s : \varepsilon >$ with the tester which contains all the other participants and the initiator. By induction $\sigma(P)\{s[\mathtt{p}]/\alpha\}^{\rceil\ell}$ is safe. In the monitored semantics too, the process $a^\ell[\mathtt{p}](\alpha).\sigma(P)^{\rceil\ell}$ with the same tester reduces to $(\nu s) < \sigma(P)\{s[\mathtt{p}]/\alpha\}\ ,\ s : \varepsilon >^{\rceil\ell}$. Therefore $a^\ell[\mathtt{p}](\alpha).\sigma(P)^{\rceil\ell}$ is safe.

$\dfrac{\Gamma \vdash e : S^\ell \quad \Gamma \vdash_{\ell'} P \triangleright \Delta, c : T \quad \ell' \leq \ell}{\Gamma \vdash_{\ell'} c!\langle \Pi, e\rangle.P \triangleright \Delta, c : !\langle \Pi, S^\ell\rangle; T}\ \lfloor \textsc{Send} \rfloor$ To reduce, the process $s[\mathtt{p}]!\langle \Pi, \sigma(e)\rangle.\sigma(P)$ needs a queue $s : h$. If $\sigma(e) \downarrow v^\ell$ we have $< s[\mathtt{p}]!\langle \Pi, \sigma(e)\rangle.\sigma(P)\ ,\ s : h > \longrightarrow < \sigma(P)\ ,\ s : h \cdot (\mathtt{p}, \Pi, v^\ell) >$. By induction $\sigma(P)^{\rceil\ell'}$ is safe. Similarly, in the monitored semantics,
$< s[\mathtt{p}]!\langle \Pi, \sigma(e)\rangle.\sigma(P)^{\rceil\ell'}\ ,\ s : h >$ reduces to $< \sigma(P)^{\rceil\ell'}\ ,\ s : h \cdot (\mathtt{p}, \Pi, v^\ell) >$, since by hypothesis $\ell' \leq \ell$. Thus $s[\mathtt{p}]!\langle \Pi, \sigma(e)\rangle.\sigma(P)^{\rceil\ell'}$ is safe.

$\dfrac{\Gamma, x^\ell : S^\ell \vdash_\ell P \triangleright \Delta, c : T}{\Gamma \vdash_\ell c?(\mathtt{p}, x^\ell).P \triangleright \Delta, c : ?(\mathtt{p}, S^\ell); T}\ \lfloor \textsc{Rcv} \rfloor$ To reduce the process $c?(\mathtt{p}, x^\ell).\sigma(P)$ we need a queue

$s : (\mathtt{p},\mathtt{q},v^\ell)\cdot h$, i.e. we have $< s[\mathtt{q}]?(\mathtt{p},x^\ell).\sigma(P)\ ,\ s : (\mathtt{p},\mathtt{q},v^\ell)\cdot h > \longrightarrow < \sigma(P)\{v/x\}\ ,\ s : h >$. If the configuration $< s[\mathtt{q}]?(\mathtt{p},x^\ell).\sigma(P)\ ,\ s : (\mathtt{p},\mathtt{q},v^\ell)\cdot h >$ is well typed we have $\Gamma \vdash v^\ell : S^\ell$, and therefore the substitution $\sigma\{v/x\}$ agrees with $\Gamma, x^\ell : S^\ell$ whenever σ agrees with Γ. We get by induction that $\sigma(P)\{v/x\}^{\rceil\ell}$ is safe. In the monitored semantics also the configuration $< c?(\mathtt{p},x^\ell).\sigma(P)^{\rceil\ell}\ ,\ s : (\mathtt{p},\mathtt{q},v^\ell)\cdot h >$ reduces to $< \sigma(P)\{v/x\}^{\rceil\ell}\ ,\ s : h >$, so $c?(\mathtt{p},x^\ell).\sigma(P)^{\rceil\ell}$ is safe.

$$\frac{\Gamma \vdash_\ell P \triangleright \Delta, c : T_j \quad j \in I \quad \ell \le \ell' \le \prod_{i\in I} \mathsf{Meet}(T_i)}{\Gamma \vdash_\ell c \oplus^{\ell'} \langle \Pi, \lambda_j \rangle.P \triangleright \Delta, c : \oplus^{\ell'} \langle \Pi, \{\lambda_i : T_i\}_{i\in I}\rangle}\ \lfloor \textsc{Sel} \rfloor$$

Similar to the case of rule $\lfloor$SEND$\rfloor$.

$$\frac{\Gamma \vdash_\ell P_i \triangleright \Delta, c : T_i \quad \ell \le \prod_{i\in I} \mathsf{Meet}(T_i)}{\Gamma \vdash_\ell c \&^{\ell} (\mathtt{p}, \{\lambda_i : P_i\}_{i\in I}) \triangleright \Delta, c : \&^{\ell} (\mathtt{p}, \{\lambda_i : T_i\}_{i\in I})}\ \lfloor \textsc{Branch} \rfloor$$

Similar to the case of rule $\lfloor$RCV$\rfloor$.

□

Corollary 9.2. If $\Gamma \vdash_\ell P \triangleright \Delta$ for some Γ, Δ, ℓ, then $\sigma(P)$ is safe for all σ which agree with Γ.

Proof. Immediate consequence of Theorem 9.1, since the safety of $\sigma(P)^{\rceil\ell}$ implies the safety of $\sigma(P)^{\rceil\bot}$, and by definition process $\sigma(P)$ is safe if the monitored process $\sigma(P)^{\rceil\bot}$ is safe. □

Clearly completeness fails, since there are safe processes which cannot be typed, a simple example being if $\mathsf{true}^\top$ then $s[1]!\langle 2, \mathsf{true}^\top\rangle.\mathbf{0}$ else $s[1]!\langle 2, \mathsf{false}^\bot\rangle.\mathbf{0}$.
Notice that also if $\mathsf{true}^\top$ then $s[1]!\langle 2, \mathsf{true}^\bot\rangle.\mathbf{0}$ else $s[1]!\langle 2, \mathsf{false}^\top\rangle.\mathbf{0}$ is safe (in spite of the level $\bot$ in the chosen branch), because the conditional does not raise the monitor level. Indeed, it would be weird to view this program as unsafe, given that it always behaves like $s[1]!\langle 2, \mathsf{true}^\bot\rangle.\mathbf{0}$.

10. Conclusion

We have proposed a monitored semantics to prevent runtime information leaks in a multiparty session calculus. We have shown that the safety property induced by this semantics is strictly included between the typability property and a refinement of the security property considered for an extended calculus in (Capecchi et al., 2013) (actually this refinement was already introduced in the Appendix of that paper).

There is a wide literature on the use of monitors (frequently in combination with types) for assuring security, but most of this work has focussed so far on sequential computations, see for instance (Guernic et al., 2007; Boudol, 2009; Sabelfeld and Russo, 2010). More specifically, (Guernic et al., 2007) considers an automaton-based monitoring mechanism for information flow, combining static and dynamic analyses, for a sequential imperative while-language with outputs. The paper (Boudol, 2009), which provided one of the inspirations for our work, deals with an ML-like language and uses a single monitoring level to control sequential executions. The work (Askarov and Sabelfeld, 2009) shows how to enforce information-release policies, which may be viewed as relaxations of noninterference, by a combination of monitoring and static analysis, in a sequential language with dynamic code evaluation. Dynamic security policies and means for expressing them via security labels have been studied for instance in (Myers and Liskov, 2000; Zheng and Myers, 2007).

In session calculi, concurrency is present not only among participants in a given session, but

also among different sessions running in parallel and possibly involving some common partners. Hence, different monitoring levels are needed to control different parallel components, and these levels must be joined when the components convene to start a new session. As we use a general lattice of security levels (rather than a two level lattice as it is often done), it may happen that while all the participants monitors are "low", their join is "high", constraining all their exchanges in the session to be high too. Furthermore, we deal with structured memories (the $\mathbf{Q}$-sets). In this sense, our setting is slightly more complex than some of the previously studied ones. Moreover, a peculiarity of session calculi is that data with different security levels are transmitted on the same channel[††] (which is also the reason why security levels are assigned to data, and not to channels). Hence, although the intuition behind monitors is rather simple, its application to our calculus is not completely straightforward.

Session types have been proposed for a variety of calculi and languages. We refer to (Dezani-Ciancaglini and de' Liguoro, 2010) for a survey on the session type literature. However, the integration of security requirements into session calculi is still at an early stage. Enforcement of integrity properties in multiparty sessions, using session types, has been studied in (Bhargavan et al., 2009; Planul et al., 2009). These papers propose a compiler which, given a multiparty session description, implements cryptographic protocols that guarantee session execution integrity.

It is easy to build a version of the typing rules of Table 11 without subtyping (Rule $\lfloor$SUBS$\rfloor$), in which there is a one-to-one correspondence between process constructors and typing rules. This syntax-directed type system would support a type inference algorithm of linear complexity in the process structure. Security is clearly undecidable in general, since it requires checking bisimulation. Safety is undecidable too, as computations may be infinite, but it should be more tractable than security in a large number of cases, mainly thanks to compositionality.

We expect that a version of our monitored semantics, enriched with labelled transitions, could turn useful to the programmer, either to help her localise and repair program insecurities, or to deliberately program well-controlled security transgressions, according to some dynamically determined condition. To illustrate this point, let us look back at our medical service example of Figure 1 in Section 2. In some special circumstances, we could wish to allow the user to send her hospital admission request in clear, for instance in case of an urgency, when the user cannot afford to wait for data encryption and decryption. Here, if in the code of U we replaced the test on $\mathit{critical}(\mathit{diagn}^{\top})$ by a test on $\mathit{no\text{-}urgency}^{\top}\ \mathit{and}\ \mathit{critical}(\mathit{diagn}^{\top})$, then in case of urgency we would have a safety violation, which however should not be considered incorrect, given that it is expected by the programmer. A labelled transition monitored semantics, with labels representing security errors, would then allow the programmer to check that her code's insecurities are exactly the expected ones. This labelled semantics could also be used to control error propagation, thus avoiding to block the execution of the whole process in case of non-critical or limited errors. In this case, labels could be recorded in the history of the process and the execution would be allowed to go on, postponing error analysis to natural breaking points (like the end of a session).

The work presented in this paper can be seen from two different perspectives. Our first goal was to extend the analysis of information leak detection/prevention in multiparty sessions by introducing safety, a notion which is more local and flexible than typability and security. To this

[††] Each session channel is used "polymorphically" to send objects of different types and levels, since it is the only means for a participant to communicate with the others in a given session.

aim, we kept our model very simple: the execution of a process is blocked as soon as there is a safety violation. In a more applied perspective, the present proposal could be seen as a first step towards a more flexible model, better suited for implementation in real systems, where safety violations could simply be signalled to the programmer and the execution be blocked only in very dangerous cases. In this model, the security levels composing the lattice could be enriched to incorporate information about the criticality of the violation, in order to drive different effects on the execution (warnings, errors, blockings).

Acknowledgments

We would like to thank Bernard Serpette and Nobuko Yoshida for helpful feedback, and the anonymous referees of EXPRESS 2011 and of the present submission for useful comments.

References

Askarov, A. and Sabelfeld, A. (2009). Tight Enforcement of Information-Release Policies for Dynamic Languages. In *Proc. CSF'09*, pages 43–59. IEEE Computer Society.

Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., and Yoshida, N. (2008). Global Progress in Dynamically Interleaved Multiparty Sessions. In *Proc. CONCUR'08*, volume 5201 of *LNCS*, pages 418–433. Springer.

Bhargavan, K., Corin, R., Deniélou, P. M., Fournet, C., and Leifer, J. J. (2009). Cryptographic Protocol Synthesis and Verification for Multiparty Sessions. In *Proc. CSF'09*, pages 124–140. IEEE Computer Society.

Bossi, A., Focardi, R., Piazza, C., and Rossi, S. (2004). Verifying Persistent Security Properties. *Computer Languages, Systems & Structures*, 30(3-4):231 – 258.

Boudol, G. (2009). Secure Information Flow as a Safety Property. In *Proc. FAST'08*, volume 5491 of *LNCS*, pages 20–34. Springer.

Capecchi, S., Castellani, I., and Dezani-Ciancaglini, M. (2011). Information Flow Safety in Multiparty Sessions. In *Proc. EXPRESS'11*, volume 64 of *EPTCS*, pages 16–30.

Capecchi, S., Castellani, I., and Dezani-Ciancaglini, M. (2013). Typing Access Control and Secure Information Flow in Sessions. *Information and Computation*. To appear in the Special Issue on Security and Rewriting Techniques.

Capecchi, S., Castellani, I., Dezani Ciancaglini, M., and Rezk, T. (2010). Session Types for Access and Information Flow Control. In *Proc. CONCUR'10*, volume 6269 of *LNCS*, pages 237–252. Springer.

Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., and Padovani, L. (2014). Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science*. to appear.

De Nicola, R. and Hennessy, M. (1983). Testing Equivalence for Processes. In *Proc. ICALP'83*, volume 154 of *LNCS*, pages 548–560. Springer.

Dezani-Ciancaglini, M. and de' Liguoro, U. (2010). Sessions and Session Types: an Overview. In *Proc. WS-FM'09*, volume 6194 of *LNCS*, pages 1–28. Springer.

Guernic, G. L., Banerjee, A., Jensen, T., and Schmidt, D. A. (2007). Automata-based Confidentiality Monitoring. In *Proc. ASIAN'06*, volume 4435 of *LNCS*, pages 75–89. Springer.

Honda, K. (1993). Types for Dyadic Interaction. In *Proc. CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer.

Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language Primitives and Type Disciplines for Structured Communication-based Programming. In *Proc. ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer.

Honda, K., Yoshida, N., and Carbone, M. (2008). Multiparty Asynchronous Session Types. In *Proc. POPL'08*, pages 273–284. ACM Press.

Milner, R. (1999). *Communicating and Mobile Systems: the Pi-Calculus*. CUP.

Myers, A. C. and Liskov, B. (2000). Protecting Privacy using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology*, 9:410–442.

Planul, J., Corin, R., and Fournet, C. (2009). Secure Enforcement for Global Process Specifications. In *Proc. CONCUR'09*, volume 5710 of *LNCS*, pages 511–526. Springer.

Sabelfeld, A. and Russo, A. (2010). From Dynamic to Static and Back: Riding the Roller Coaster of Information-flow Control Research. In *Proc. PSI'06*, volume 5947 of *LNCS*, pages 352–365. Springer.

Takeuchi, K., Honda, K., and Kubo, M. (1994). An Interaction-based Language and its Typing System. In *Proc. PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer.

Zheng, L. and Myers, A. C. (2007). Dynamic Security Labels and Static Information Flow Control. *International Journal of Information Security*, 6:67–84.