Information Flow Safety in Multiparty Sessions

Sara Capecchi
Dipartimento di Informatica
Università di Torino, corso Svizzera 185, 10149 Torino, Italy*
capecchi@di.unito.it

Ilaria Castellani
INRIA
2004 route des Lucioles, 06902 Sophia Antipolis, France
ilaria.castellani@inria.fr

Mariangiola Dezani-Ciancaglini
Dipartimento di Informatica
Università di Torino, corso Svizzera 185, 10149 Torino, Italy
dezani@di.unito.it

Abstract 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 our monitored semantics with various examples, and show that the induced safety property implies a noninterference property studied previously.

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

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 [15], on a variant of the π-calculus [11] including a construct for session creation and two n-ary operators of labelled 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 delegation of tasks among participants and multiparty interaction within a single session, and equipped with increasingly sophisticated session types, ensuring additional properties like progress.

In previous work [5], we addressed the question of incorporating security requirements into session types. To this end, we considered a calculus for multiparty sessions with delegation, enriched with security levels for both session participants and data. We proposed a session type system for this calculus, adding access control and secure information flow requirements in the typing rules in order to guarantee the preservation of data *confidentiality* during session execution.

*Work partially funded by the ANR-08-EMER-010 grant PARTOUT, and by the MIUR Projects DISCO and IPODS.

B. Luttik and F. D. Valencia (Eds.): 18th International Workshop on
Expressiveness in Concurrency (EXPRESS 2011)
EPTCS 64, 2011, pp. 16–30, doi:10.4204/EPTCS.64.2

In this paper, we move one step further by equipping 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 or testing 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 a passive attacker, which may only change secret information at each step).

Expectedly, this monitored semantics is closely related to the security type system presented in [5]. Indeed, some of the constraints imposed by the monitored operational rules are simply lifted from the typing rules. However, there are two respects in which the constraints of the monitoring semantics are simpler. First, they refer to individual computations. In other words, they are *local* whereas type constraints are both local and *global*. Second, one of these constraints (the lower bound for the level of communications in a given session) may be dynamically computed during execution, and hence services do not need to be statically annotated with levels, as was required by the type system of [5]. This means that the language itself may be simplified when the concern is on safety rather than typability.

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: for instance, in our setting, a high conditional with a low branch will be ruled out by the type system but will be considered safe if the low branch is never taken. Safety also offers more *flexibility* than types in the context of web programming, where security policies may change dynamically.

Compared to security, safety has again the advantage of locality versus globality. 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* leak detection, whereas security only allows *late* detection.

Finally, safety seems more appealing than security when the dangerous behaviour comes from an accidental rather than an intentional transgression of the security policy. Indeed, in this case a monitored semantics could offer useful feedback to the programmer, in the form of informative error messages. Although this possibility is not explored in the present 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 the induced *information flow safety* property strictly implies the *information flow security* property of [5]. While the issue of safety has recently received much attention in the security community (see Section 7), 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 recall the definition of security from [5] and illustrate it with examples. Section 5 presents our monitored semantics and Section 6 introduces our notion of safety and establishes its relation to security. Finally, Section 7 concludes with a discussion on related and future work.

2 Motivating example

Let us illustrate our approach with an introductory example, inspired by [2]. Suppose we want to model the interaction between an online health service `S` and a user `U`. Each time the user wishes to consult the service, she opens a connection with the server and sends him her username (here by convention we shall use “she” for the user and “he” for the server), assuming she has previously registered with the service. She may then choose between two kinds of service:

$$
\begin{array}{lcl}
\mathtt{I} & = & \bar{a}[2] \\
\mathtt{U} & = & a[1](\alpha_1).\alpha_1!\langle 2,\mathsf{un}^{\perp}\rangle. \quad \text{if } \mathit{simple}^{\perp} \text{ then } \alpha_1 \oplus^{\perp} \langle 2,\mathbf{sv1}\rangle.\alpha_1!\langle 2,\mathsf{que}^{\perp}\rangle.\alpha_1?(1,\mathit{ans}^{\perp}).\ \mathbf{0} \\
& & \qquad\qquad \text{else } \alpha_1 \oplus^{\perp} \langle 2,\mathbf{sv2}\rangle.\alpha_1!\langle 2,\mathsf{pwd}^{\top}\rangle.\alpha_1?(2,\mathit{form}^{\top}). \\
& & \qquad\qquad\quad \text{if } \mathit{gooduse}(\mathit{form}^{\top}) \text{ then } \alpha_1!\langle 2,\mathsf{que}^{\top}\rangle.\alpha_1?(2,\mathit{ans}^{\top}).\mathbf{0} \\
& & \qquad\qquad\qquad\qquad \text{else } \alpha_1!\langle 2,\mathsf{que}^{\perp}\rangle.\alpha_1?(2,\mathit{ans}^{\perp}).\mathbf{0} \\
\mathtt{S} & = & a[2](\alpha_2). \quad \alpha_2?(1,\mathit{un}^{\perp}). \\
& & \qquad \alpha_2\ \&^{\perp}(1,\{\mathbf{sv1} : \alpha_2?(1,\mathit{que}^{\perp}).\alpha_2!\langle 1,\mathsf{ans}^{\perp}\rangle.\mathbf{0}, \\
& & \qquad\qquad \mathbf{sv2} : \alpha_2?(1,\mathit{pwd}^{\top}).\alpha_2!\langle 1,\mathsf{form}^{\top}\rangle.\alpha_2?(1,\mathit{que}^{\top}).\alpha_2!\langle 1,\mathsf{ans}^{\top}\rangle.\mathbf{0}\}
\end{array}
$$

Figure 1: The online medical service example.

1. *simple consultation*: the user asks questions from the medical staff. Questions and answers are public, for instance they can be published in a forum visible to every user. The staff has no privacy constraint.
2. *medical consultation*: the user sends questions together with private data (e.g., results of medical exams) to the medical staff, in order to receive a diagnosis or advice about medicines or further exams. To access these features she must enter a password, and wait for a secure form on which to send her data. Here questions and answers are secret (modelling the fact that the they are sent in a secure mode and that the staff is compelled to maintain privacy).

More precisely, this interaction may be described by the following protocol, in which we add the possibility that the user accidentally reveals her private information:

1. U opens a connection with S and sends her username to S;
2. U chooses between Service 1 and Service 2;

3.a Service 1: U sends a question to S and waits for an answer;

3.b Service 2: U sends her password to S and waits for a secure form. She then sends her question and data on the form and waits for an answer from S. A reliable user will use the form correctly and send data in a secure mode. Instead, an unreliable user will forget to use the form, or use it wrongly, thus leaking some of her private data. This may result in private information being sent to a public forum or to medical staff which is not compelled to maintain privacy.

In our calculus, this scenario may be described as the parallel composition of the processes in Figure 1.

A session is an activation of a service, involving a number of participants with predefined roles. Here processes U and S communicate by opening a session on service a. The initiator $\bar{a}[2]$ specifies that the number of participants is 2. Participants are denoted by integers: here U=1, S=2. 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. Dually, in S, $a[2](\alpha_2)$ means that S will act as participant 2 in service a, communicating via channel α_2.

Security levels appear as superscripts on both data and some operators (here $\perp$ means "public" and $\top$ means "secret"): the user name un and the message contents in Service 1 can be public; the password pwd and the information exchanged in Service 2 should be secret. Levels on the operators are needed to track indirect flows, as will be explained in Section 6. They may be ignored for the time being.

When the session is established, via a synchronisation between the initiator and the prefixes $a[i](\alpha_i)$, U sends to S her username $\mathsf{un}^{\perp}$. Then, according to whether she wishes a simple consultation or not, she chooses between the two services $\mathbf{sv1}^{\perp}$ and $\mathbf{sv2}^{\perp}$. This choice is expressed by the internal choice construct $\oplus$: if $\mathit{simple}^{\perp}$ then $\alpha_1 \oplus^{\perp} \langle 2,\mathbf{sv1}\rangle\ldots$ else $\alpha_1 \oplus^{\perp} \langle 2,\mathbf{sv2}\rangle\ldots$ describes a process sending on α_1 to participant 2 either label **sv1** or **sv2**, depending on the value of $\mathit{simple}^{\perp}$. If U chooses **sv1**, then

she sends to S a question (construct $\alpha_1!\langle 2, \mathsf{que}^{\perp}\rangle$), receives the answer (construct $\alpha_1?(1, ans^{\perp})$). If U chooses **sv2**, then she sends her password to S and then waits to get back a secure form. At this point, according to her reliability (if $gooduse(form^{\top})$) she either sends her question and data in the secure form, or wrongly sends them in clear. The difference between the secure and insecure exchanges is modelled by the different security levels tagging values and variables in the prefixes $\alpha_1!\langle 2, \mathsf{que}^{\top/\perp}\rangle$ and $\alpha_1?(2, ans^{\top/\perp})$.

Dually, process S receives the username from U and then waits for her choice of either label **sv1** or label **sv2**. This is described by the external choice operator &: $\&^{\perp}(1, \{\mathbf{sv1} : \ldots, \mathbf{sv2} : \ldots)$ expresses the reception of the label **sv1** or of the label **sv2** from participant 1. In the first case, S will then receive a question and send the answer. This whole interaction is public. In the second case, S receives a password and sends a form, and then receives a question and sends the answer. In this case the interaction is secret.

Note that the execution of process I | S | U may be insecure if U is unreliable. Indeed, in U's code, the test on $gooduse(form^{\top})$ uses the secret value $form^{\top}$. Now, for security to be granted in the subsequent execution, all communications depending on $form^{\top}$ should be secret. However, this will not be the case if the second branch of the conditional is taken, since in this case U sends a public question. On the other hand, the execution is secure when the first service is used, or when the second service is used properly.

This process is rejected by the type system of [5], which must statically ensure the correction of all possible executions. Similarly, the security property of [5] fails to hold for this process, since two different public behaviours may be exhibited after testing the secret value $gooduse(form^{\top})$: in one case the empty behaviour, in the other case the emission of $\mathsf{que}^{\perp}$. Moreover, the bisimulation used to check security will fail only once $\mathsf{que}^{\perp}$ 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 $\mathsf{que}^{\perp}$ in the queue.

For the sake of conciseness, we deliberately simplified the scenario in the above example, by using finite services and a binary session between a server and a user. Note that several such sessions could run in parallel, each corresponding to a different impersonation of the user. A more realistic example would involve persistent services and allow several users to interact within the same session, and the server to delegate the question handling to the medical staff. This would bring into the scene other important features of our calculus, namely multiparty interaction and the mechanism of delegation. Our simple example is mainly meant to highlight the novel issue of monitored execution.

3 Syntax and Standard Semantics

Our calculus is essentially the same as that studied in [5]. For the sake of simplicity, we do not consider here access control and declassification, although their addition would not pose any problem.

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), *service name variables*, ranged over by $\zeta, \zeta', \ldots$, *identifiers*, i.e., service names and value variables, ranged over by $u, w, \ldots$, *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 p, q,... to denote the *participants* of a session. In an n-ary session (a session corresponding to an n-ary service) p, 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 p communicates with the other participants in the session s. A new session s on an n-ary service a is opened when the

r	::=	$a \mid s$	Service/Session Name	P	::=	$\bar{u}[n]$	n-ary session initiator
c	::=	$\alpha \mid s[\mathtt{p}]$	Channel			$u[\mathtt{p}](\alpha).P$	p-th session participant
u	::=	$\zeta \mid a$	Identifier			$c!\langle \Pi, e\rangle.P$	Value send
v	::=	$\mathsf{true} \mid \mathsf{false} \mid \ldots$	Value			$c?(\mathtt{p}, x^\ell).P$	Value receive
e	::=	$x^\ell \mid v^\ell \mid \mathsf{not}\ e$				$c!^\ell\langle \Pi, u\rangle.P$	Service name send
		$e\ \mathsf{and}\ e' \mid \ldots$	Expression			$c?^\ell(\mathtt{p}, \zeta).P$	Service name receive
D	::=	$X(x,\alpha) = P$	Declaration			$c!^\ell\langle\langle \mathtt{q}, c'\rangle\rangle.P$	Channel send
Π	::=	$\{\mathtt{p}\} \mid \Pi \cup \{\mathtt{p}\}$	Set of participants			$c?^\ell((\mathtt{p}, \alpha)).P$	Channel receive
						$c \oplus^\ell \langle \Pi, \lambda\rangle.P$	Selection
ϑ	::=	$v^\ell \mid s[\mathtt{p}]^\ell \mid \lambda^\ell$	Message content			$c\&^\ell(\mathtt{p}, \{\lambda_i : P_i\}_{i\in I})$	Branching
						$\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	Conditional
m	::=	$(\mathtt{p}, \Pi, \vartheta)$	Message in transit			$P \mid Q$	Parallel
h	::=	$m \cdot h \mid \varepsilon$	Queue			$\mathbf{0}$	Inaction
H	::=	$H \cup \{s : h\} \mid \emptyset$	**Q**-set			$(\nu a)P$	Name hiding
						$\mathsf{def}\ D\ \mathsf{in}\ P$	Recursion
						$X\langle e, c\rangle$	Process call

Table 1: Syntax of processes, expressions and queues.

initiator $\bar{a}[n]$ of the service synchronises with n processes of the form $a[1](\alpha_1).P_1, \ldots, a[n](\alpha_n).P_n$, whose channels $\alpha_\mathtt{p}$ then get replaced by $s[\mathtt{p}]$ in the body of $P_\mathtt{p}$. While binary sessions may often be viewed as an interaction between a user and a server, multiparty sessions do not exhibit the same asymmetry. This is why we use of an initiator to start the session once all the required "peer" participants are present. 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 [9], 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 one of the following: a value message $(\mathtt{p}, \Pi, v^\ell)$, indicating that the value v^ℓ is sent by participant $\mathtt{p}$ to all participants in Π; a service name message $(\mathtt{p}, \Pi, a^\ell)$, with a similar meaning; a channel message $(\mathtt{p}, \mathtt{q}, s[\mathtt{p}']^\ell)$, indicating that $\mathtt{p}$ delegates to $\mathtt{q}$ the role of $\mathtt{p}'$ with level ℓ in session s; and 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 (sessions, channels with role, messages, queues).

Let us briefly comment on the primitives of the language. We already described session initiation. Communications within a session are performed on a channel using the next four pairs of primitives: the send and receive of a value; the send and receive of a service name; the send and receive of a channel (where one participant transmits to another the capability of participating in another session with a given role) and the selection and branching operators (where one participant chooses one of the branches offered by another participant). Apart from the value send and receive constructs, all the send/receive and choice primitives are decorated with security levels, whose use will be justified later. When there is no risk of confusion we will omit the set delimiters $\{,\}$, particularly around singletons.

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

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

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

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

$$< s[\mathtt{p}]!^\ell\langle \Pi, a\rangle.P \ , \ s : h > \longrightarrow < P \ , \ s : h \cdot (\mathtt{p}, \Pi, a^\ell) >$$
[SendS]

$$< s[\mathtt{q}]?^\ell(\mathtt{p}, \zeta).P \ , \ s : (\mathtt{p}, \mathtt{q}, a^\ell) \cdot h > \longrightarrow < P\{a/\zeta\} \ , \ s : h >$$
[RecS]

$$< s[\mathtt{p}]!^\ell\langle\langle \mathtt{q}, s'[\mathtt{p}']\rangle\rangle.P \ , \ s : h > \longrightarrow < P \ , \ s : h \cdot (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']^\ell) >$$
[SendC]

$$< s[\mathtt{q}]?^\ell((\mathtt{p}, \alpha)).P \ , \ s : (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']^\ell) \cdot h > \longrightarrow < P\{s'[\mathtt{p}']/\alpha\} \ , \ s : h >$$
[RecC]

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

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

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

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

$$\frac{< P \ , \ H > \longrightarrow (\nu \tilde{s}) < P' \ , \ H' >}{< \mathsf{def}\ D\ \mathsf{in}\ (P \mid Q) \ , \ H > \longrightarrow (\nu \tilde{s}) < \mathsf{def}\ D\ \mathsf{in}\ (P' \mid Q) \ , \ H' >}$$
[Defin]

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

Table 2: Standard reduction rules.

A *configuration* is a pair $C = < P \ , \ H >$ of a process P and a $\mathbf{Q}$-set H, possibly restricted with respect to service and 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 and of s in H are bound. By abuse of notation we often write P instead of $< P \ , \ \emptyset >$.

As usual, the operational semantics is defined modulo a structural equivalence $\equiv$. The structural rules for processes are standard [11]. 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 via the rule:

$$(\nu \tilde{r}) < P \ , \ H > \| \, (\nu \tilde{r}') < Q \ , \ K > \equiv \ (\nu \tilde{r}\tilde{r}') < P \mid Q \ , \ H \cup K >$$

where by hypothesis the session names in the $\mathbf{Q}$-sets H, K are disjoint, by Barendregt convention $\tilde{r}$ and $\tilde{r}'$ have empty intersection and there is no capture of free names, and $(\nu \tilde{r})C$ stands for $(\nu r_1) \cdots (\nu r_k)C$, if $\tilde{r} = r_1 \cdots r_k$. Note that, modulo $\equiv$, each configuration has the form $(\nu \tilde{r}) < P \ , \ H >$.

The transitions for configurations have the form $C \longrightarrow C'$. They are derived using the reduction rules

in Table 2, where we write P as short for $< P , \emptyset >$.

Rule [Link] describes the initiation of a new session among n processes, corresponding to an activation of the service a of arity n. After the connection, the participants share a private session name s and the corresponding queue, initialised to $s : \varepsilon$. In each participant $P_{\mathtt{p}}$, the channel variable $\alpha_{\mathtt{p}}$ is replaced by the channel with role $s[\mathtt{p}]$. This 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 [SendV], [SendS], [SendC] and [Label] push values, service names, channels and labels, respectively, into the queue $s : h$. In rule [SendV], $e \downarrow v^{\ell}$ 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 [RecV], [RecS], [RecC] and [Branch] perform the complementary operations. Rules [If-T], [If-F], [Def] and [Defin] are standard. The contextual rule [Scop] is also standard. In this rule, Barendregt convention ensures that the names in $\tilde{s}$ are disjoint from those in $\tilde{r}$ and do not appear in C''. As usual, we use $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

We assume that communication safety and session fidelity are assured by a standard session type system [9]

4 Security

As in [5], 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, ultimately, to discover the $\overline{\mathscr{L}}$-messages), by injecting himself different $\overline{\mathscr{L}}$-messages at each step and observing their effect 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. Based on $=_{\mathscr{L}}$, a notion of $\mathscr{L}$-bisimulation $\simeq_{\mathscr{L}}$ formalises 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 $\mathscr{L}$. 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: $lev(v^{\ell}) = lev(a^{\ell}) = lev(s[\mathtt{p}]^{\ell}) = lev(\lambda^{\ell}) = \ell$.

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

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

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

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 $H \Downarrow \mathscr{L} = K \Downarrow \mathscr{L}$.*

The idea is to test processes by running them in conjunction with $\mathscr{L}$-equal queues. However, we cannot allow arbitrary combinations of processes with queues, since this would lead us to reject intuitively secure processes as simple as $s[2]?(1,x^{\bot}).\mathbf{0}$ and $s[1]!\langle 2,\mathsf{true}^{\bot}\rangle.\mathbf{0}$. As argued in [5], we may get around this problem by imposing two simple conditions, one on **Q**-sets (*monotonicity*) and the other on configurations (*saturation*). These conditions are justified by the fact that they are always satisfied in initial computations generated by typable processes (in the sense of [5]).

The first condition requires that in a **Q**-set, the security levels of messages with the same sender and common receivers should never decrease along a sequence.

Definition 4.3 (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* $\Pi_1 \cap \Pi_2 \neq \emptyset$.

The second condition requires that in a configuration, the **Q**-set should always contain enough queues to enable all outputs of the process to reduce.

Definition 4.4 (Saturation) *A configuration* $\langle P,H\rangle$ *is* saturated *if each session name s occurring in P has a corresponding queue* $s : h$ *in H*.

We are now ready for defining our $\mathscr{L}$-bisimulation, expressing indistinguishability by an $\mathscr{L}$-observer. Unlike early definitions of $\mathscr{L}$-bisimulation, which only allowed the "high state" to be changed at the start of computation, our definition allows it to be changed at each step, to account for dynamic contexts [7].

Definition 4.5 ($\mathscr{L}$-Bisimulation)
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 pair of monotone* **Q***-sets* H_1 *and* H_2 *such that* $H_1 =_{\mathscr{L}} H_2$ *and each* $< P_i , H_i >$ *is saturated:*

$$\begin{array}{l} \textit{If } \ < P_1 , H_1 > \longrightarrow (\nu \tilde{r}) < P_1' , H_1' >, \textit{ then there exist } P_2', H_2' \textit{ such that} \\ \quad < P_2 , H_2 > \longrightarrow^{*} \equiv (\nu \tilde{r}) < P_2' , H_2' >, \textit{ where } H_1' =_{\mathscr{L}} H_2' \textit{ and } P_1' \mathscr{R} P_2'. \end{array}$$

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 $\tilde{r}$ may either be the empty string or a single name, since it appears after a one-step transition. If it is a name, it may either be a service name a (communication of a private service) or a fresh session name s (opening of a new session). In the latter case, s cannot occur in P_2 and H_2 by Barendregt convention.

Intuitively, a transition that adds or removes an $\mathscr{L}$-message must be simulated in one or more steps, producing the same effect on the **Q**-set, whereas a transition that does not affect $\mathscr{L}$-messages may be simulated by inaction. In such case, the structural equivalence $\equiv$ may be needed in case the first process has created a restriction. The notions of $\mathscr{L}$-security and security are now defined in the standard way:

Definition 4.6 (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* $\mathscr{L}$.

The need for considering all downward-closed sets $\mathscr{L}$ is justified by the following example.

Example 4.7 *Let* $\mathscr{S} = \{\bot,\ell,\top\}$ *where* $\bot \leq \ell \leq \top$ *and*

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

The process P is $\{\bot\}$*-secure and* $\mathscr{S}$*-secure, but it is not* $\{\bot,\ell\}$*-secure, since there is a flow from level* $\top$ *to level* ℓ *in* P_1*, which is detectable by a* $\{\bot,\ell\}$*-observer but not by a* $\{\bot\}$*-observer. We let the reader verify this fact formally, possibly after looking at the next example.*

We show next that an input of level ℓ should not be followed by an action of level $\ell' \not\geq \ell$:

Example 4.8 (Insecurity of high input followed by low action)
Consider the process P and the $\mathbf{Q}$-sets H_1 and H_2, where $H_1 =_{\{\bot\}} H_2$:

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

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

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

The need for security levels on value variables are justified by the following example.

Example 4.9 (Need for levels on value variables)
Suppose we had no levels on value variables. Consider the process, which should be secure:

$$P \ = \ s[1]?(2,x).s[1]?(2,y).\mathbf{0} \ \mid \ s[2]!\langle 1,\mathsf{true}^{\bot}\rangle.s[2]!\langle 1,\mathsf{true}^{\bot}\rangle.\mathbf{0}$$

Let $H_1 = \{s:(2,1,\mathsf{true}^{\top})\} =_{\mathscr{L}} \{s:\varepsilon\} = H_2$. Then the transition:

$$< P\ ,\ H_1 > \longrightarrow < s[1]?(2,y).\mathbf{0} \ \mid \ s[2]!\langle 1,\mathsf{true}^{\bot}\rangle.s[2]!\langle 1,\mathsf{true}^{\bot}\rangle.\mathbf{0}\ ,\ \{s:\varepsilon\} > = < P_1\ ,\ H_1' >$$

could not be matched by $< P\ ,\ H_2 >$. In fact, the first component of P cannot move in H_2, and each computation of the second component yields an $\mathscr{L}$-observable H_2' such that $H_1' \neq_{\{\bot\}} H_2'$. Moreover, P cannot stay idle in H_2, since P is not $\mathscr{L}$-bisimilar to P_1 (as it is easy to see by a similar reasoning). By adding the level $\bot$ to the variables x and y, we force the second component to move first in both $< P\ ,\ H_1 >$ and $< P\ ,\ H_2 >$.

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 is not 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 6.

Example 4.10 (Insecurity sanitised by parallel context)
Let R be obtained by composing the process P of Example 4.8 in parallel with a dual process $\overline{P}$, and consider again the $\mathbf{Q}$-sets H_1 and H_2, where $H_1 =_{\{\bot\}} H_2$:

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

Then the move $< P \mid \overline{P}\ ,\ H_1 > \longrightarrow < s[2]!\langle 1,\mathsf{true}^{\bot}\rangle.\mathbf{0} \mid \overline{P}\ ,\ \{s:\varepsilon\} >$ can be simulated by the sequence of two moves

$$< P \mid \overline{P}\ ,\ H_2 > \longrightarrow < P \ \mid \ s[1]?(2,y^{\bot}).\mathbf{0}\ ,\ \{s:(1,2,\mathsf{true}^{\top})\} >$$
$$\longrightarrow < s[2]!\langle 1,\mathsf{true}^{\bot}\rangle.\mathbf{0} \mid s[1]?(2,y^{\bot}).\mathbf{0}\ ,\ \{s:\varepsilon\} >,$$

where $H_1' = H_2' = \{s:\varepsilon\}$.
Let us now compare the processes $R_1 = s[2]!\langle 1,\mathsf{true}^{\bot}\rangle.\mathbf{0} \mid \overline{P}$ and $R_2 = s[2]!\langle 1,\mathsf{true}^{\bot}\rangle.\mathbf{0} \mid s[1]?(2,y^{\bot}).\mathbf{0}$. Let K_1, K_2 be monotone $\mathbf{Q}$-sets containing a queue $s:h$ and such that $K_1 =_{\{\bot\}} K_2$. Now, if $< R_1\ ,\ K_1 >$ moves first, either it does the high output of $\overline{P}$, 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 $K_1' =_{\{\bot\}} K_2'$, or it executes its first component, in which case $< R_2\ ,\ K_2 >$ does exactly the same, clearly preserving the $\{\bot\}$-equality of $\mathbf{Q}$-sets, and it remains to prove that $\overline{P} = s[1]!\langle 2,\mathsf{true}^{\top}\rangle.s[1]?(2,y^{\bot}).\mathbf{0}$ is $\bot$-bisimilar to $s[1]?(2,y^{\bot}).\mathbf{0}$. But this is easy to see since if the first process moves, the second may stay idle, while if the second moves, the first may simulate it in two steps.

Conversely, if $< R_2 , K_2 >$ moves first, either it executes its second component (if the queue allows it), in which case $< R_1 , K_1 >$ simulates it in two steps, or it executes its first component, in which case we are reduced once again to prove that $\overline{P} = s[1]!\langle 2, \mathsf{true}^\top\rangle.s[1]?(2,y^\bot).\mathbf{0}$ is $\bot$-bisimilar to $s[1]?(2,y^\bot).\mathbf{0}$.

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 $\mu \in \mathscr{S}$:

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

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, labels or channels) and of tested conditions 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* $\equiv$. The new specific structural rules are:

$$(P_1 \mid P_2)^{\rceil\mu} \equiv P_1^{\rceil\mu} \mid P_2^{\rceil\mu} \qquad\qquad C\dagger \ \wedge\ C \equiv C' \implies C'\dagger$$

The reduction rules of the monitored semantics are given in Table 3. Intuitively, the monitoring level is initially $\bot$ and gets increased each time a test of higher or incomparable level or an input of higher level is crossed. 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.

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.

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 [4]) would not be enough.

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 $P^{\rceil\bot}$ would be successful or not depending on the order of execution of its parallel components:

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

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

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

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

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

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:

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

$$a[1](\alpha_1).P_1^{\rceil\mu_1} \mid \ldots \mid a[n](\alpha_n).P_n^{\rceil\mu_n} \mid \bar{a}[n]^{\rceil\mu_{n+1}} \multimap\!\!\rightarrow (\nu s) < P_1\{s[1]/\alpha_1\}^{\rceil\mu} \mid \ldots \mid P_n\{s[n]/\alpha_n\}^{\rceil\mu} , s : \varepsilon > \quad \text{where } \mu = \bigsqcup_{i\in\{1\ldots n+1\}} \mu_i \qquad \text{[MLink]}$$

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

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

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

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

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

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

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

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

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

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

$$< M , H > \multimap\!\!\rightarrow (\nu\tilde{s}) < M' , H' > \ \Rightarrow\ < \mathsf{def}\ D \ \mathsf{in}\ (M \mid M'') , H > \multimap\!\!\rightarrow (\nu\tilde{s}) < \mathsf{def}\ D \ \mathsf{in}\ (M' \mid M'') , H' > \qquad \text{[MDefin]}$$

$$C \multimap\!\!\rightarrow (\nu\tilde{s})C' \text{ and } \neg\, C''\dagger \ \Rightarrow\ (\nu\tilde{r})(C \,\|\, C'') \multimap\!\!\rightarrow (\nu\tilde{r})(\nu\tilde{s})(C' \,\|\, C'') \qquad C\dagger \ \Rightarrow\ (\nu\tilde{r})(C \,\|\, C')\dagger \qquad \text{[MScopC]}$$

Table 3: Monitored reduction rules.

The following example justifies the use of the join in rule [MLink]. Session initiation is the only synchronisation operation of our calculus. Since this synchronisation requires the initiator $\bar{a}[n]$ as well as a complete set of "service callers" $a[\mathtt{p}](\alpha_1).P_\mathtt{p}, 1 \leq \mathtt{p} \leq n$, the monitoring level of each of them contributes to the monitoring level of the session. Note that the fact that this monitoring level may be computed dynamically as the join of the monitoring levels of the participants exempts us from statically annotating services with levels, as it was necessary to do in [5] in order to type the various participants consistently.

Consider the process:

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

Here the monitoring level of the conditional becomes $\top$ after the test, and thus, assuming the `if` branch is taken, rule [MLink] will set the monitoring level of the session to $\top$. This will block the exchange of the $\perp$-value between the last two components.

Example 5.2 (Need for security levels on transmitted service names)
This example shows the need for security levels on service names in rules [MSendS] and [MRecS].

$$\begin{array}{l} s[2]?(1,x^\top).\mathsf{if}\ x^\top\ \mathsf{then}\ s[2]!^\ell\langle 3,a\rangle.\mathbf{0}\ \mathsf{else}\ s[2]!^\ell\langle 3,b\rangle.\mathbf{0} \\ \mid\ s[3]?^\ell(2,\zeta).\bar{\zeta}[2] \\ \mid\ a[1](\alpha_1).\alpha_1!\langle 2,\mathsf{true}^\perp\rangle.\mathbf{0} \mid\ a[2](\alpha_2).\alpha_2?(1,y^\perp).\mathbf{0} \\ \mid\ b[1](\beta_1).\beta_1!\langle 2,\mathsf{false}^\perp\rangle.\mathbf{0} \mid\ b[2](\beta_2).\beta_2?(1,y^\perp).\mathbf{0} \end{array}$$

This process is insecure because, depending on the high value received for $x^\top$, it will initiate a session on service a or on service b, which both perform a low value exchange. If $\ell \neq \top$, the monitored semantics will yield error in the outputs of the first line, otherwise it yields error in the outputs of the last two lines.

Similar examples show the need for security levels on transmitted channels and labels.

6 Safety

We define now the property of *safety* for monitored processes, from which we derive also a property of safety for processes. We then prove that if a process is safe, it is also secure.

A monitored process may be "relaxed" to an ordinary process by removing all its monitoring levels.

Definition 6.1 (Demonitoring) *If M is a monitored process, its* demonitoring $|M|$ *is defined by:*

$$\begin{array}{lcllcl} |P^{\rceil\mu}| & = & P & |M_1 \mid M_2| & = & |M_1| \mid |M_2| \\ |(\nu a)M| & = & (\nu a)|M| & |\mathsf{def}\ D\ \mathsf{in}\ M| & = & \mathsf{def}\ D\ \mathsf{in}\ |M| \end{array}$$

Intuitively, a monitored process M is safe if it can mimic at each step the transitions of the process $|M|$.

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

If $< |M| , H > \longrightarrow (\nu\tilde{r}) < P , H' >$
then $< M , H > \multimap\!\!\rightarrow (\nu\tilde{r}) < M' , H' >$, where $|M'| = P$ 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 monitor $\perp$ gives rise to error. This result rests on the observation that $< M , H > \multimap\!\!\rightarrow$ if and only if $< |M| , H > \longrightarrow$ and $\neg < M , H > \dagger$, and that if M is safe, then if a standard communication rule is applicable to $|M|$, 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:

$$< P^{\rceil\perp} , \emptyset >=< M_0 , H_0 > \multimap\!\!\rightarrow (\nu \tilde{r_1}) < M_1 , H_1 > \multimap\!\!\rightarrow \cdots (\nu \tilde{r_k}) < M_k , H_k >$$

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

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

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

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. On the other hand, a simple monitored execution of $< P^{\rceil\perp} , \emptyset >$, which uses at each step the $\mathbf{Q}$-set produced at the previous step, would never take the* `else` *branch and would therefore always succeed. Hence the simple absence of run-time errors would not be sufficient to enforce security.*

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

Lemma 6.6 (Monotonicity of monitoring) *Monitoring levels may only increase along execution: If $\langle P^{\rceil\mu}, H\rangle \multimap\!\!\rightarrow (\nu\tilde{r})\langle P'^{\rceil\mu'} \mid M, H'\rangle$, then $\mu \leq \mu'$.*

As usual, $\mathscr{L}$-high processes modify $\mathbf{Q}$-sets in a way which is transparent for $\mathscr{L}$-observers.

Definition 6.7 ($\mathscr{L}$-highness of processes) *A process P is* $\mathscr{L}$-high *if for any monotone $\mathbf{Q}$-set H such that $< P , H >$ is saturated, it satisfies the property:*

$$\textit{If} < P , H > \longrightarrow (\nu\tilde{r}) < P' , H' >, \textit{then } H =_{\mathscr{L}} H' \textit{ and } P' \textit{ is } \mathscr{L}\textit{-high.}$$

Lemma 6.8 *If $P^{\rceil\mu}$ is safe and $\mu \notin \mathscr{L}$, then P is $\mathscr{L}$-high.*

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.

Definition 6.9 (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 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. *$M_i = (\nu a)N_i$, where $N_1 \mathscr{R}_\circ^{\mathscr{L}} N_2$;*
5. *$M_i = \mathsf{def}\ D\ \mathsf{in}\ N_i$, where $N_1 \mathscr{R}_\circ^{\mathscr{L}} N_2$.*

Definition 6.10 (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:

$P_1 \mathscr{R}^{\mathscr{L}} P_2$ if there are M_1, M_2 such that $P_i \equiv |M_i|$ for $i = 1,2$ and $M_1 \mathscr{R}_\circ^{\mathscr{L}} M_2$.

We may now state 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 or in the two branches of a high conditional. This implies that there is some observable low action after the high input, or in at least one of the branches of the high conditional. 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 or the high condition).

Theorem 6.11 (Safety implies security) *If P is safe, P is also secure.*

The converse of Theorem 6.11 does not hold, as shown by the process R of Example 4.10. A more classical example is $s[1]?(2,x^{\top}).\mathsf{if}\ x^{\top}\ \mathsf{then}\ s[1]!\langle 2,\mathsf{true}^{\perp}\rangle.\mathbf{0}\ \mathsf{else}\ s[1]!\langle 2,\mathsf{true}^{\perp}\rangle.\mathbf{0}$.

7 Conclusion

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 [8, 4, 14]. More specifically, [8] considers an automaton-based monitoring mechanism for information flow, combining static and dynamic analyses, for a sequential imperative while-language with outputs. The paper [4], which provided the initial inspiration for our work, deals with an ML-like language and uses a single monitoring level to control sequential executions. The work [1] 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 [12, 16].

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[1] (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 [6] for a survey on the session type literature. However, the integration of security requirements into session calculi is still at an early stage. A type system assuring that services comply with a given data exchange security policy is presented in [10]. Enforcement of integrity properties in multiparty sessions, using session types, has been studied in [3, 13]. These papers propose a compiler which, given a multiparty session description, implements cryptographic protocols that guarantee session execution integrity.

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

[1] 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.

special circumstances, we could wish to allow the user to send her message 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 $\mathit{gooduse}(\mathit{form}^{\top})$ by a test on $\mathit{no\text{-}urgency}^{\top} \wedge \mathit{gooduse}(\mathit{form}^{\top})$, then in case of urgency we would have a security violation, which however should not be considered incorrect, given that it is expected by the programmer. A labelled transition monitored semantics, whose labels would represent 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).

Acknowledgments
We would like to thank Kohei Honda, Nobuko Yoshida and the anonymous referees for helpful feedback.

References

[1] A. Askarov & A. Sabelfeld (2009): *Tight Enforcement of Information-Release Policies for Dynamic Languages*. In: *Proc. CSF'09*, IEEE Computer Society, pp. 43–59.

[2] Adam Barth, John Mitchell, Anupam Datta & Sharada Sundaram (2007): *Privacy and Utility in Business Processes*. In: *Proc. CSF'07*, IEEE Computer Society, pp. 279–294.

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

[4] G. Boudol (2009): *Secure Information Flow as a Safety Property*. In: *Proc. FAST'08*, *LNCS* 5491, Springer, pp. 20–34.

[5] S. Capecchi, I. Castellani, M. Dezani Ciancaglini & T. Rezk (2010): *Session Types for Access and Information Flow Control*. In: *Proc. CONCUR'10*, *LNCS* 6269, Springer, pp. 237–252.

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

[7] R. Focardi & S. Rossi (2002): *Information Flow Security in Dynamic Contexts*. In: *Proc. CSFW'02)*, IEEE Computer Society Press, pp. 307–319.

[8] G. Le Guernic, A. Banerjee, T. Jensen & D. A. Schmidt (2007): *Automata-based Confidentiality Monitoring*. In: *Proc. ASIAN'06*, *LNCS* 4435, Springer, pp. 75–89.

[9] K. Honda, N. Yoshida & M. Carbone (2008): *Multiparty Asynchronous Session Types*. In: *Proc. POPL'08*, ACM Press, pp. 273–284, doi:10.1145/1328438.1328472.

[10] A. Lapadula, R. Pugliese & F. Tiezzi (2007): *Regulating Data Exchange in Service Oriented Applications*. In: *Proc. FSEN'07*, *LNCS* 4767, Springer, pp. 223–239.

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

[12] A. C. Myers & B. Liskov (2000): *Protecting Privacy using the Decentralized Label Model*. *ACM Transactions on Software Engineering and Methodology* 9, pp. 410–442, doi:10.1145/363516.363526.

[13] J. Planul, R. Corin & C. Fournet (2009): *Secure Enforcement for Global Process Specifications*. In: *Proc. CONCUR'09*, *LNCS* 5710, Springer, pp. 511–526.

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

[15] K. Takeuchi, K. Honda & M. Kubo (1994): *An Interaction-based Language and its Typing System*. In: *Proc. PARLE'94*, *LNCS* 817, Springer, pp. 398–413.

[16] L. Zheng & A. C. Myers (2007): *Dynamic Security Labels and Static Information Flow Control*. *International Journal of Information Security* 6, pp. 67–84, doi:10.1007/s10207-007-0019-9.