Monitors and Blame Assignment for Higher-Order Session Types

Limin Jia Hannah Gommerstadt Frank Pfenning
Carnegie Mellon University
liminjia@cmu.edu hgommers@cs.cmu.edu fp@cs.cmu.edu

Abstract

Session types provide a means to prescribe the communication behavior between concurrent message-passing processes. However, in a distributed setting, some processes may be written in languages that do not support static typing of sessions or may be compromised by a malicious intruder, violating invariants of the session types. In such a setting, dynamically monitoring communication between processes becomes a necessity for identifying undesirable actions. In this paper, we show how to dynamically monitor communication to enforce adherence to session types in a higher-order setting. We present a system of blame assignment in the case when the monitor detects an undesirable action and an alarm is raised. We prove that dynamic monitoring does not change system behavior for well-typed processes, and that one of an indicated set of possible culprits must have been compromised in case of an alarm.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; F.3.2 [*Semantics of Programming Languages*]: Process models

Keywords Session types, contracts, monitors, blame assignment

1. Introduction

Session types (Honda 1993; Honda et al. 1998) provide a means to prescribe the communication behavior between concurrent message-passing processes. A variant of previously defined session type systems has been discovered to be in a Curry-Howard correspondence with linear logic (Caires and Pfenning 2010; Wadler 2012; Caires et al. 2013), where session types correspond to linear propositions, process expressions to sequent proofs, and communication to cut reduction. This observation has formed the basis for SILL (Toninho et al. 2013), a programming language integrating ordinary functional with message-passing concurrent computation. Separately, it has been shown that the logical foundation supports not only the synchronous communication model of Caires and Pfenning (2010) but also asynchronous communication with message queues (DeYoung et al. 2012). Recently, these two styles of communication have been unified using the logical concept of *polarization* (Pfenning and Griffith 2015). In this formulation, communication is a priori asynchronous, but synchronization can be specified via so-called *shift operators* in the type.

All of the above mentioned systems and languages satisfy *session fidelity* (a generalization of type preservation) and *global progress* (a generalization of ordinary progress). These properties are inherited directly from cut elimination in linear logic, since the operational semantics is based on cut reduction. They continue to hold in the presence of recursive session types when other properties such as productivity and termination fail. Despite exhibiting significant concurrency in the operational semantics, the languages also have strong confluence properties (Pérez et al. 2014), again inherited from the confluence of cut reduction in linear logic. They therefore occupy a kind of middle ground between functional languages and foundational calculi such as the π-calculus with much inherent nondeterminism.

At this point one might consider the basic foundational questions regarding logic-based session-typed languages to be solved. Usually, static typing together with a type preservation theorem is seen as license to erase types at runtime. In the setting of message-passing concurrency, however, there are two important reasons to also consider *dynamic monitoring* of communication. The first is that when spawning a new process, part of the execution of a program now escapes immediate control of the original process. If the new process is compromised by a malicious intruder, then incorrect yet unchecked messages can wreak havoc on the original process, inducing undefined behavior such as the infamous buffer overruns. A second reason is that session types are explicitly designed to abstract away from local computation. This means we can use them to safely connect communicating processes written in a variety of different and even incompatible languages, as long as they (dynamically!) adhere to the session protocol and basic data formats. But there is often no reason to trust remote processes, or believe in the correctness of the code even for trusted processes, so dynamically monitoring communication becomes a necessity.

However, designing monitoring infrastructure that allows precise blame assignment in the presence of *higher-order* processes is challenging. In such settings, channels of arbitrary type can be passed along other channels. When this occurs, the runtime monitor cannot immediately determine if the process communicating over the channel being passed along satisfies session fidelity, but must monitor further communication over this channel.

This paper makes the following main contributions:

- We define a powerful but intuitive adversary model for session-typed communicating processes.
- We show how to *dynamically monitor* communication to enforce adherence to session types in a higher-order setting.
- We present a precise system of *blame assignment* in the case an alarm is raised.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837662

- We prove that dynamic monitoring does not change system behavior for well-typed processes, and that one of an indicated set of possible culprits must have been compromised in case of an alarm.

For simplicity, the formal development omits affine and shared channels present in Pfenning and Griffith (2015), but our techniques are general enough to also encompass them. Of course, intruders could fly under the rader of the dynamic monitor and send incorrect values of the correct types. To capture such violations, additional measures such as *contracts* or *dependent types* (Swamy et al. 2011; Pfenning et al. 2011; Caires et al. 2012; Griffith and Gunter 2013; Scholliers et al. 2015) need to be considered. We conjecture that the framework presented in this paper is sufficient to permit generalizations to these more precise properties.

Due to space constraints, we only present the key definitions and theorems. Additional definitions and detailed proofs can be found in our companion technical report (Jia et al. 2015).

2. Polarized Session Typed Processes

The first fundamental idea behind our system of session types is that every *linear channel* has two endpoints: a *provider* and a *client*. A *process* always provides along a single channel, but it may be the client of multiple channels. This identification is quite strong: each channel is provided by a unique process, and each process provides along a unique channel which never changes. Our basic typing judgment is therefore

$$x_1{:}A_1, \ldots x_n{:}A_n \vdash P :: (x : A)$$

where P is a process providing $x : A$ and using $x_i{:}A_i$. The *session types* A and A_i prescribe the communication behavior along channels x and x_i, respectively.

We first explain the session types A and their intuitive interpretation in terms of process behavior, followed by the process expression and the typing rules.

2.1 Polarized Session Types

The distinction between provider and client is orthogonal to the *direction* of communication: a process can both send and receive along the channel it provides and any channel that it uses. Of course, the session type of the channel will tell us which.

We explicitly *polarize* the types: any change in the direction of communication must be explicitly denoted by a *shift* operator (Laurent 1999). The polarized formulation of session types supports synchronous and asynchronous communication in the same language, while at the same time providing a uniform integration of linear, affine, and shared channels (Pfenning and Griffith 2015). It also simplifies monitoring since message queues always have a definitive direction. From the perspective of monitoring and blame

τ	$::= \mathsf{int} \mid \mathsf{bool} \mid \cdots$	
A^+, B^+	$::= \mathbf{1}$	send end and terminate
	$\mid\ A^+ \otimes B^+$	send channel $a{:}A^+$, cont. as B^+
	$\mid\ \oplus\{lab_i : B_i^+\}_i$	send some label lab_j, cont. as B_j^+
	$\mid\ \tau \wedge B^+$	send a data value $v{:}\tau$, cont. as B^+
	$\mid\ {\downarrow}A^-$	send shift, then receive
A^-, B^-	$::= A^+ \multimap B^-$	receive a channel $a{:}A^+$, cont. as B^-
	$\mid\ \&\{lab_i : B_i^-\}$	receive a label lab_j, cont. as B_j^-
	$\mid\ \tau \rightarrow B^-$	receive a data value $v{:}\tau$, cont. as B^-
	$\mid\ {\uparrow}A^+$	receive shift, then send

Figure 1: Polarized linear session types

assignment, under our assumptions, there is no significant difference between linear and affine channels, so for simplicity we treat only linear ones. Shared channels can be handled straightforwardly with our techniques, so we omit them from the development in order to streamline the presentation.

From the perspective of the provider, *positive types* correspond to sending a message and *negative types* correspond to receiving a message. The kinds of messages we consider here are *basic data values* v (like integers or booleans), labels lab that indicate an internal or external choice between alternative ways to continue a session, channels a, and special tokens 'end', to indicate the end of a session and 'shift', to indicate a change in direction of communications.

As an example for the use of session types, we consider the specification of the behavior of a priority queue. A priority queue offers the client a choice between inserting an element into the queue, or deleting the element of the currently highest priority. The elements themselves are channels of some arbitrary positive type A^+. This is represented as an *external choice*:

$$\begin{aligned} \mathsf{pq}^-\ A^+ = \&\{\ & \mathsf{ins} : \ldots, \\ & \mathsf{del} : \ldots\ \} \end{aligned}$$

The type $\mathsf{pq}\,A^+$ is negative, since the process providing a priority queue has to start by receiving, either the label ins or the label del.

When the client elects to insert, it first has to send the priority as an integer, and then the channel. After that, the provider has to once again behave as a priority queue which means the type is recursive.

$$\begin{aligned} \mathsf{pq}^-\ A^+ = \&\{\ & \mathsf{ins} : \mathsf{int} \rightarrow A^+ \multimap \mathsf{pq}^-\ A^+, \\ & \mathsf{del} : \ldots\ \} \end{aligned}$$

When the client elects to delete the element of highest priority, the provider sends back either the label none (indicating that the priority queue is empty) and terminates, or the label some, followed by the priority and then element itself. This is an example of an *internal choice*, since the provider has to send the label in this case.

$$\begin{aligned} \mathsf{pq}^-\ A^+ = \&\{\ & \mathsf{ins} : \mathsf{int} \rightarrow A^+ \multimap \mathsf{pq}^-\ A^+, \\ & \mathsf{del} : {\uparrow}{\oplus}\{\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{int} \wedge A^+ \otimes {\downarrow}\mathsf{pq}^-\ A^+\ \} \end{aligned}$$

Note that we needed to insert some shifts to indicate the change of direction in communication: after receiving label del, the provider will then have to send a shift followed by another label (either none or some). Before we recurse, we have to shift once again to be ready to receive the next insert or delete.

The type $\mathsf{pq}\,A$ is an example of a *recursive session type*. Recursive types are equal to their unfolding, i.e., *equirecursive*, so there are no special type constructors for them. However, we allow matching recursive process definitions. Similarly, we allow types and process definitions to be (implicitly) polymorphic. Type variables, however, must have a well-defined polarity and can only be instantiated with types of that polarity. For example, the definition of $\mathsf{pq}^-\ A^+$ is parameterized by a positive type A^+. If we want to instantiate it with a negative type B^-, we have to use ${\downarrow}B^-$.

2.2 Process Expressions

The syntactic constructs for writing processes are relatively straightforward. We indicate dependence on a bound variable x (for channels) or n (a value) using a subscript that we may substitute for.

Returning to our example, we now give some (recursive) process definitions. We write $c \leftarrow p\, n_1 \cdots n_k \leftarrow d_1, \ldots, d_m$ for a process that provides a session along channel c and uses channels $d_1, \ldots, d_m$. It is parameterized by data values $n_1, \ldots, n_k$ which constitute a form of local state. We write the type of such as function as $\tau_1 \rightarrow \cdots \rightarrow \tau_k \rightarrow \{A \leftarrow A_1, \ldots, A_m\}$ as in SILL (Toninho et al. 2013).

$$empty : \{\mathsf{pq}^-\ A^+\}$$

$$\frac{}{\Delta, y{:}A \vdash (x \leftarrow y) :: (x{:}A)}\ \mathsf{id} \qquad \frac{\Delta \vdash P_x :: (x{:}A) \quad \Delta', x{:}A \vdash Q_x :: (z{:}C)}{\Delta, \Delta' \vdash (x \leftarrow P_x \,;\, Q_x) :: (z{:}C)}\ \mathsf{cut}$$

$$\frac{\Delta \vdash P :: (x{:}A^+)}{\Delta \vdash (\mathsf{shift} \leftarrow \mathsf{recv}\ x \,;\, P) :: (x{:}{\uparrow}A^+)}\ {\uparrow}\mathsf{R} \qquad \frac{\Delta, x{:}A^+ \vdash Q :: (z{:}C)}{\Delta, x{:}{\uparrow}A^+ \vdash (\mathsf{send}\ x\ \mathsf{shift} \,;\, Q) :: (z{:}C)}\ {\uparrow}\mathsf{L}$$

$$\frac{\Delta \vdash Q :: (x{:}A^-)}{\Delta \vdash (\mathsf{send}\ x\ \mathsf{shift} \,;\, Q) :: (x{:}{\downarrow}A^-)}\ {\downarrow}\mathsf{R} \qquad \frac{\Delta, x{:}A^- \vdash P :: (z{:}C)}{\Delta, x{:}{\downarrow}A^- \vdash (\mathsf{shift} \leftarrow \mathsf{recv}\ x \,;\, P) :: (z{:}C)}\ {\downarrow}\mathsf{L}$$

$$\frac{}{\cdot \vdash (\mathsf{close}\ x) :: (x{:}\mathbf{1})}\ \mathbf{1}R \qquad \frac{\Delta \vdash Q :: (z{:}C)}{\Delta, x{:}\mathbf{1} \vdash (\mathsf{wait}\ x \,;\, Q) :: (z{:}C)}\ \mathbf{1}L$$

$$\frac{\Delta \vdash P_y :: (y{:}A^+) \quad \Delta' \vdash Q :: (x{:}B^+)}{\Delta, \Delta' \vdash (\mathsf{send}\ x\ (y \leftarrow P_y) \,;\, Q) :: (x{:}A^+ \otimes B^+)}\ \otimes R \qquad \frac{\Delta, y{:}A^+, x{:}B^+ \vdash R_y :: (z{:}C)}{\Delta, x{:}A^+ \otimes B^+ \vdash (y \leftarrow \mathsf{recv}\ x \,;\, R_y) :: (z{:}C)}\ \otimes L$$

$$\frac{\Delta, y{:}A^+ \vdash R_y :: (x{:}B^-)}{\Delta \vdash (y \leftarrow \mathsf{recv}\ x \,;\, R_y) :: (x{:}A^+ \multimap B^-)}\ \multimap R \qquad \frac{\Delta \vdash P_y :: (y{:}A^+) \quad \Delta', x{:}B^- \vdash Q :: (z{:}C)}{\Delta, \Delta', x{:}A^+ \multimap B^- \vdash (\mathsf{send}\ x\ (y \leftarrow P_y) \,;\, Q) :: (z{:}C)}\ \multimap L$$

$$\frac{(\Delta \vdash P_i :: (x{:}A_i^-))_i}{\Delta \vdash \mathsf{case}\ x\ \{lab_i \rightarrow P_i\}_i :: (x{:}\&\{lab_i : A_i^-\}_i)}\ \& R \qquad \frac{\Delta, x{:}A_j^- \vdash Q :: (z{:}C)}{\Delta, x{:}\,\&\,\{lab_i : A_i^-\}_i \vdash (x.lab_j \,;\, Q) :: (z{:}C)}\ \& L_j$$

$$\frac{\Delta \vdash Q :: (x{:}A_j^+)}{\Delta \vdash (x.lab_j \,;\, Q) :: (x{:}\oplus\{lab_i : A_i^+\}_i)}\ \oplus R_j \qquad \frac{(\Delta, x{:}A_i^+ \vdash P_i :: (z{:}C))_i}{\Delta, x{:}\oplus\{lab_i : A_i^+\}_i \vdash \mathsf{case}\ x\ \{lab_i \rightarrow P_i\}_i :: (z{:}C)}\ \oplus L$$

Figure 3: Typing process expressions

```
c ← empty =
  case c of
  | ins → p ← recv c ;          % receive priority p
          x ← recv c ;          % receive channel x to be stored
          e ← empty ;           % spawn new empty pri. queue
          c ← elem p ← x, e     % cont. as singleton pri. queue
                                % holding p and x
  | del → shift ← recv c ;      % shift direction to send
          c.none ;              % send lab none (queue is empty)
          close c               % close channel c and terminate
```

Next, the code which implements a non-empty priority queue. The process $c \leftarrow elem\ p \leftarrow x, d$ holds the element x with priority p, offering the priority queue interface along c. It also connects along d to all the elements of lower priority. In other words, we maintain the priority queue as a linked collection of processes with decreasing priority.

$elem : \mathsf{int} \rightarrow \{\mathsf{pq}^-\ A^+ \leftarrow A^+, \mathsf{pq}^-\ A^+\}$

```
c ← elem p ← x, d =
  case c of
  | ins → q ← recv c ;
          y ← recv c ;
          case (p > q) of
          | true →  d.ins ;             % pass y with priority q to d
                    send d q ;
                    send d y ;
                    c ← elem p ← x, d
          | false → e ← elem p ← x, d
                    c ← elem q ← y, e
  | del → shift ← recv c ;  % shift c to send
          c.some ;          % send label some
          send c p ;        % send priority p
          send c x ;        % send highest priority channel x
          send c shift ;    % shift c to recv
          c ← d             % forward d as c
```

```
P, Q, R ::=
    close c                 send end and terminate
  | wait c ; Q              recv end, cont. with Q
  | send c (x ← Px) ; Q     send new a along c,
                            spawn Pa and cont. as Q
  | x ← recv c ; Qx         receive a along c, cont. as Qa
  | c.labj ; Q              send labj along c, cont. as Q
  | case c of {labi ⇒ Qi}i  recv labj along c, cont. as Qj
  | send c v ; Q            send value v along c, cont. as Q
  | n ← recv c ; Qn         recv value v along c, cont. as Qv
  | send c shift ; Q        send shift along c, cont. as Q
  | shift ← recv c ; Q      receive shift along c, cont. as Q

  | x ← Px ; Qx             create new a, spawn Pa, cont. as Qa
  | c ← d                   connect c with d and terminate
```

Figure 2: Linear process expressions

The typing rules for process expressions, defining the judgment $\Delta \vdash P :: (x : A)$ are given in Figure 3, omitting only the simple cases of sending and receiving basic values (types $\tau \wedge B^+$, $\tau \rightarrow B^-$). Here, Δ is a sequence of channel declarations $x_i{:}A_i$ for pairwise distinct x_i whose order is irrelevant.

2.3 Operational Semantics

The computational meaning of process expressions is given in the form of a *substructural operational semantics* (Pfenning 2004; Simmons 2012) which relies on *multiset rewriting* (Cervesato and Scedrov 2009). The rules can be found in Figure 4, again omitting the sending and receiving of basic values.

$$
\begin{array}{lll}
\mathsf{id}^{+} & : & \mathsf{queue}(c', \overleftarrow{p}, c) \otimes \mathsf{proc}(c, c \leftarrow d') \otimes \mathsf{queue}(d', \overleftarrow{q}, d) \multimap \{\mathsf{queue}(c', \overleftarrow{p \cdot q}, d)\} \\
\mathsf{id}^{-} & : & \mathsf{queue}(c', \overrightarrow{p}, c) \otimes \mathsf{proc}(c, c \leftarrow d') \otimes \mathsf{queue}(d', \overrightarrow{q}, d) \multimap \{\mathsf{queue}(c', \overrightarrow{p \cdot q}, d)\} \\
 & & \\
\mathsf{cut}^{+} & : & \mathsf{proc}(c, x{:}A^{+} \leftarrow P_x \;;\; Q_x) \\
 & & \quad \multimap \{\exists a.\ \exists a'.\ \mathsf{proc}(c, Q_{a'}) \otimes \mathsf{queue}(a', \overleftarrow{\cdot}, a) \otimes \mathsf{proc}(a, P_a)\} \\
\mathsf{cut}^{-} & : & \mathsf{proc}(c, x{:}A^{-} \leftarrow P_x \;;\; Q_x) \\
 & & \quad \multimap \{\exists a.\ \exists a'.\ \mathsf{proc}(c, Q_{a'}) \otimes \mathsf{queue}(a', \overrightarrow{\cdot}, a) \otimes \mathsf{proc}(a, P_a)\} \\
 & & \\
\mathsf{up_s} & : & \mathsf{proc}(a, \mathsf{send}\ c'\ \mathsf{shift} \;;\; Q) \otimes \mathsf{queue}(c', \overrightarrow{q}, c) \\
 & & \quad \multimap \{\mathsf{proc}(a, Q) \otimes \mathsf{queue}(c', \overrightarrow{\mathsf{shift} \cdot q}, c)\} \\
\mathsf{up_r} & : & \mathsf{queue}(c', \overrightarrow{\mathsf{shift}}, c) \otimes \mathsf{proc}(c, \mathsf{shift} \leftarrow \mathsf{recv}\ c \;;\; P) \\
 & & \quad \multimap \{\mathsf{queue}(c', \overleftarrow{\cdot}, c) \otimes \mathsf{proc}(c, P)\} \\
 & & \\
\mathsf{down_s} & : & \mathsf{queue}(c', \overleftarrow{p}, c) \otimes \mathsf{proc}(c, \mathsf{send}\ c\ \mathsf{shift} \;;\; Q) \\
 & & \quad \multimap \{\mathsf{queue}(c', \overleftarrow{p \cdot \mathsf{shift}}, c) \otimes \mathsf{proc}(c, Q)\} \\
\mathsf{down_r} & : & \mathsf{proc}(a, \mathsf{shift} \leftarrow \mathsf{recv}\ c' \;;\; P) \otimes \mathsf{queue}(c', \overleftarrow{\mathsf{shift}}, c) \\
 & & \quad \multimap \{\mathsf{proc}(a, P) \otimes \mathsf{queue}(c', \overrightarrow{\cdot}, c)\} \\
\mathsf{one_s} & : & \mathsf{queue}(a', \overleftarrow{p}, a) \otimes \mathsf{proc}(a, \mathsf{close}\ a) \multimap \{\mathsf{queue}(a', \overleftarrow{p \cdot \mathsf{end}}, _)\} \\
\mathsf{one_r} & : & \mathsf{proc}(c, \mathsf{wait}\ a' \;;\; Q) \otimes \mathsf{queue}(a', \overleftarrow{\mathsf{end}}, _) \multimap \{\mathsf{proc}(c, Q)\} \\
 & & \\
\mathsf{tensor_s} & : & \mathsf{queue}(a', \overleftarrow{p}, a) \otimes \mathsf{proc}(a, \mathsf{send}\ a\ (y \leftarrow P_y) \;;\; Q) \\
 & & \quad \multimap \{\exists b.\ \exists b'.\ \mathsf{queue}(a', \overleftarrow{p \cdot b'}, a) \otimes \mathsf{proc}(a, Q) \otimes \mathsf{queue}(b', \overleftarrow{\cdot}, b) \otimes \mathsf{proc}(b, P_b)\} \\
\mathsf{tensor_r} & : & \mathsf{proc}(c, y \leftarrow \mathsf{recv}\ a' \;;\; R_y) \otimes \mathsf{queue}(a', \overleftarrow{b' \cdot q}, a) \\
 & & \quad \multimap \{\mathsf{proc}(c, R_{b'}) \otimes \mathsf{queue}(a', \overleftarrow{q}, a)\} \\
 & & \\
\mathsf{lolli_s} & : & \mathsf{proc}(c, \mathsf{send}\ a'\ (y \leftarrow P_y) \;;\; Q) \otimes \mathsf{queue}(a', \overrightarrow{q}, a) \\
 & & \quad \multimap \{\exists b.\ \exists b'.\ \mathsf{proc}(c, Q) \otimes \mathsf{queue}(a', \overrightarrow{b' \cdot q}, a) \otimes \mathsf{queue}(b', \overrightarrow{\cdot}, b) \otimes \mathsf{proc}(b, P_b)\} \\
\mathsf{lolli_r} & : & \mathsf{queue}(a', \overrightarrow{p \cdot b'}, a) \otimes \mathsf{proc}(a, y \leftarrow \mathsf{recv}\ a \;;\; R_y) \\
 & & \quad \multimap \{\mathsf{queue}(a', \overrightarrow{p}, a) \otimes \mathsf{proc}(a, R_{b'})\} \\
 & & \\
\mathsf{with_s} & : & \mathsf{proc}(c, a'.lab_j \;;\; Q) \otimes \mathsf{queue}(a', \overrightarrow{q}, a) \\
 & & \quad \multimap \{\mathsf{proc}(c, Q) \otimes \mathsf{queue}(a', \overrightarrow{lab_j \cdot q}, a)\} \\
\mathsf{with_r} & : & \mathsf{queue}(a', \overrightarrow{p \cdot lab_j}, a) \otimes \mathsf{proc}(a, \mathsf{case}\ a\ \{lab_i \to P_i\}_i) \\
 & & \quad \multimap \{\mathsf{queue}(a', p, a) \otimes \mathsf{proc}(a, P_j)\} \\
 & & \\
\mathsf{plus_s} & : & \mathsf{queue}(a', \overleftarrow{p}, a) \otimes \mathsf{proc}(a, a.lab_j \;;\; Q) \\
 & & \quad \multimap \{\mathsf{queue}(a', \overleftarrow{p \cdot lab_j}, a) \otimes \mathsf{proc}(a, Q)\} \\
\mathsf{plus_r} & : & \mathsf{proc}(c, \mathsf{case}\ a'\ \{lab_i \to P_i\}_i) \otimes \mathsf{queue}(a', \overleftarrow{lab_j \cdot q}, a) \\
 & & \quad \multimap \{\mathsf{proc}(c, P_j) \otimes \mathsf{queue}(a', \overleftarrow{q}, a)\}
\end{array}
$$

Figure 4: Operational Semantics

Because communication is *asynchronous*, a configuration of multiple interacting processes is described by a multiset of propositions of the form $\mathsf{queue}(c', q, c)$ and $\mathsf{proc}(c, P)$. Here, $\mathsf{proc}(c, P)$ means that a process providing a service along c is executing process expression P. We do not explicitly record the channels that are *used* by P, which are simply its free names $\mathsf{fn}(P) - \{c\}$. The predicate $\mathsf{queue}(c', q, c)$ describes the state of a message queue q connecting a providing process along c with its client along c'. Message queues have a definitive direction which is needed so processes don't accidentally receive their own messages. The direction of a queue is indicated as $\overleftarrow{p}$ (the provider sends) or $\overrightarrow{p}$ (the client sends). Receipt of a shift will change the direction of the queue. We often elide this direction information when it can be inferred or is insignificant. This bare form of the propositions does not track the types which are necessary for monitoring. Therefore, the queue propositions in the monitored semantics in Figure 6 contain some additional information.

We use Ω to refer to a multiset of proc and queue propositions, which describe the state of a concurrent computation. Each rule of the operational semantics has the form

$$F_1 \otimes \ldots \otimes F_m \multimap \{\exists x_1.\ \ldots\ \exists x_k.\, G_1 \otimes \ldots \otimes G_n\}$$

which represents a transition from one multiset to another:

$$\Omega, F_1, \ldots, F_m \longrightarrow \Omega, [a_1/x_1, \ldots, a_k/x_k](G_1, \ldots G_n)$$

were $a_1, \ldots a_k$ are freshly chosen names. Persistent propositions which can not be rewritten are preceded by an exponential modality $!F$. On the left-hand side, such propositions can represent a condition to be checked.

3. Monitoring Untrusted Processes

In a distributed setting, processes can be compromised by an attacker, may be untrusted, or may be written in a language that can not statically enforce session types. They may therefore deviate from their prescribed session types. For simplicity, we sometimes refer to all these situations as an "attack" compromising a process. We use runtime monitors to detect such deviations and attribute blame to rogue processes. In this section, we discuss the adversary and trust model and explain the monitor design. We then formally define the operational semantics for the monitoring mechanism and for the blame assignment.

3.1 System Assumptions

Adversary and trust models We assume that processes are distributed across the network and communicate with each other using the message queues. We assume that there is a secure (trusted) network layer which maintains the message queues. In other words, we do not consider network attackers that eavesdrop or intercept messages, or tamper with the message queue. We say that *all message queues are trusted*. In contrast, *all processes are untrusted*; any process could be compromised by an attacker. We consider this a conservative assumption – we could easily refine our system to distinguish between trusted and untrusted processes which would lead to more precise blame assignments.

Monitor capabilities We assume that the monitor can inspect communications between processes to check session fidelity, but it cannot observe internal operations of the executing processes. Only send, receive, spawn (cut), and forward (identity) requests can be seen by the monitor. This design decision is important because it allows our monitoring techniques to be applied in the situation where we make no assumptions about the internal structure of the communicating processes. The monitor is also trusted.

Monitors can raise alarms and assign blame when messages sent to queues are of the *wrong type*, which we explain in detail in Section 3.2. In this paper we do not consider error recovery or network failure, which are important, but belong to a different level of abstraction. If a protocol violation is detected and alarm is raised, we assume the whole distributed computation will be aborted.

We have a one-to-one correspondence between channels and processes, since every channel is provided by exactly one process, and each process provides exactly one channel. When a process wants to spawn a new process, it makes a corresponding request to the network layer, which also creates a new message queue of specified type. This type must be taken as prescriptive; violating it will raise an alarm. If we had access to the source of the code executed in the new process, we could type-check it against the given type, and absolve the spawning process. However, this may be difficult to do when executing a new binary, or cloning the current process, so we do not assume we can type-check processes before they start executing.

Adversary capabilities Our adversary model has to describe adversary capabilities: what an attacker might do with an untrusted process. The first possibility would be to simply replace the executing code with arbitrary other code. However, this model is too strong: under this attacker model, the adversary can gain send and receive access to any channel in the current system of processes. Under those circumstances, any hope at precise blame assignment has to be abandoned. Instead, we assume that channels are *private* in that only the processes at the two endpoints of a channel can send to or receive from it. Further, channel names are capabilities that are hard to forge. An attacker only knows the channel names that are given to it by the trusted runtime (e.g., through spawning a new process).

We define the following transition rule (named havoc) to represent an attacker's action of taking control of a process. The attacker replaces the original process with one of the attacker's choice. However, the attacker cannot forge channel names, and therefore, the set of channel names in Q is a subset of that in P.

$$\mathsf{havoc} : \mathsf{proc}(c, P) \otimes !(\mathsf{fn}(P) \supseteq \mathsf{fn}(Q)) \multimap \{\mathsf{proc}(c, Q)\}$$

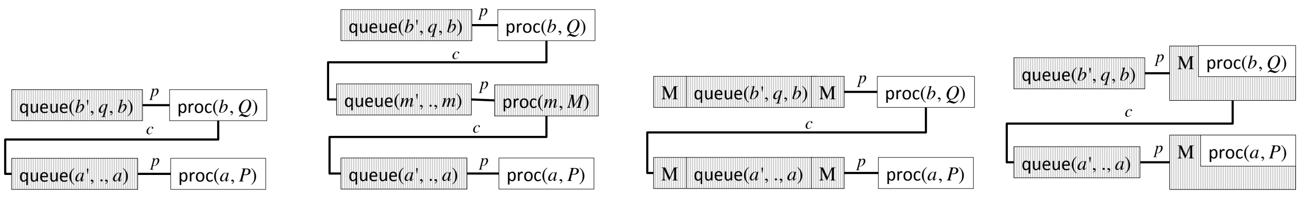
Finally, because processes are untrusted, they cannot raise an alarm.

3.2 Monitoring and Blame Assignment

Placement of the monitor First, we examine several possible design choices for the monitor and explain our chosen design. Figure 5a shows two processes that offer along channels a and b respectively. Trusted components have a grey background. We further assume that process b uses the client channel a'. (Since processes are uniquely identified by the channel they provide, we use the channel name as a process id.) We link a process to its providing channel at the right end of the queue with a line labeled p. We link a process to the client channels that it uses with lines labeled c. These client channels are the left ends of the queues. Let us assume that process a is a newly spawned process offering a service of type A. Processes a and b can communicate by sending and receiving messages through the queue $\mathsf{queue}(a', \cdot, a)$. Because both a and b can be compromised by an attacker, the task of the monitor is to mediate their communications through $\mathsf{queue}(a', \cdot, a)$: initially, a is supposed to offer a service of type A and b is supposed to use a service of type A through a'.

Figures 5b to 5d show three different monitor designs for mediating communication between processes a and b. Figure 5b illustrates the design where a partial identity process acts as the monitor and mediates communications between a and b. Here, the monitor process m relays messages between a and b and makes sure that the types of the messages entering the queue $\mathsf{queue}(m', \cdot, m)$ and the messages entering the queue $\mathsf{queue}(a', \cdot, a)$ are consistent. Initially, m and a' are of type A, which is the type of the service offered by process a. The process $\mathsf{proc}(m, M)$ is a partial identity process from A to A. In this design, every spawned process (by means of the cut, tensor, or lolli rules), has an accompanying monitoring process generated at the time when it is spawned. The advantage is that the monitor is a process defined within the same language as the rest of the system, so implementing the monitors is straightforward. The drawback is that blame assignment becomes difficult. When a process c sends a message of the wrong type to a client channel m', the monitor can only raise an alarm after the message reaches the other end of the queue and is observed by the monitor. At this point, the monitor does not know where the message came from, because several processes could legitimately have access to m'. To move forward with this approach, additional bookkeeping would be needed to mark the provenance of messages.

A second approach is to place the monitors directly at the ends of the message queues (Figure 5c). The monitor then keeps track of the types of the ends of the queues and checks the message pattern



(a) No monitors (b) Partial identity processes (c) Placed at the ends of queues (d) Local to processes

Figure 5: Monitoring Architectures

before a message is placed into the queue. The monitor does not need to check the messages leaving the queue, because they will have already been checked. The advantage of this approach is that when a process attempts to send messages of the wrong type, the monitor will raise an alarm before the message is added to the queue. This leads to more precise blame attribution than the first approach.

Finally, in a mutually untrusting environment, it may make sense to place the monitor locally at each process node (Figure 5d). When a process contacts the queue to receive a message, the monitor is contacted instead, and the monitor checks the message before handing it to the process. This is necessary because these messages are sent by other processes. The monitor trusts its local process; when a process sends messages to queues, its local monitor does not check the messages. This approach fits the distributed model more naturally, as the monitor is local to each process. However, this approach suffers from a similar problem as the first approach: the blame assignment is difficult. Moreover, since all processes are a priori untrusted, a separate mechanism would be needed to somehow verify alarms and blame processes.

To help solve the provenance issue mentioned earlier, one can imagine requiring processes to always send a message together with its signature which is generated by the private key of the process. Unfortunately, the signature alone is not enough; the system still suffers from replay attacks. Consider the following scenario: a process b receives a message of type int from process c via channel a', with c's signature, then later, b sends this message back to a', but now a' expects a channel name. When the monitor eventually raises an alarm, the signature from c is valid, but c is not to be blamed. We either need to record which processes have access to which channels, or need to associate each message with more detailed provenance information. In either case, this design is significantly more complicated than the approach of placing the monitors at the ends of the queues (shown in Figure 5c).

After considering these different approaches, we chose the approach that places monitors at the ends of the queues because it is simple and provides relatively precise blame assignment. Next, we define the formal monitor semantics.

Monitor Semantics We augment the operational semantics in Figure 4 to include monitor actions. To assign blame, the monitor maintains a graph data structure that records process spawns throughout the execution of the entire system. We write G to denote the graph (defined below). The nodes in the graph, denoted N, are provider channel names. After a process offering along channel a spawns a process offering along channel b, an edge $a \to_{sp} b$ is added to the graph. G is a set of trees.

Process graph	G	$::=$	(N, E)
Edges	E	$::=$	$\cdot \mid E, a \to_{sp} b$
messages	m	$::=$	$n \mid \mathsf{end} \mid \mathsf{ch}(a) \mid \mathsf{lab}(l) \mid \mathsf{shift}$

We write m to denote message patterns. We define $m \rhd A$ to mean that a message pattern m is compatible with type A. It describes the pattern of the message that a channel of type A expects to receive (defined below). The monitor uses message patterns to decide whether a message is allowed to be enqueued; the monitor raises an alarm if the message is not compatible with the type expected by the queue.

$$
\begin{array}{ll}
\mathsf{ch}(a) \rhd \mathsf{A}^+ \otimes \mathsf{B}^+ & \mathsf{ch}(a) \rhd \mathsf{A}^+ \multimap \mathsf{B}^- \\
\mathsf{lab}(lab_j) \rhd \oplus\{lab_i : A_i^+\}_i & \mathsf{lab}(lab_j) \rhd \&\{lab_i : A_i^-\}_i \\
\mathsf{shift} \rhd \downarrow A^- & \mathsf{shift} \rhd \uparrow A^+ \\
\mathsf{end} \rhd \mathbf{1} & \mathsf{n} \rhd \mathsf{int} \\
s \rhd \diamond_K \tau \;\; \mathit{if} \;\; \mathsf{verify}(s, \tau, \mathsf{pub\_Key}(K)) &
\end{array}
$$

The last pattern matches a signed value s against the type $\diamond_K \tau$, which requires the monitor to use cryptographic primitives to check whether s is a validly signed value of type τ using K's public key. This signature is only valid if it is generated using K's private key. This type is explained in detail in (Pfenning et al. 2011). Our monitoring framework is extensible, so the details of typechecking whether the first-order value (e.g.,$\diamond_K \tau$) are a matter of additional theory.

We show the key semantic rules in Figure 6. Most of the rules in Figure 6 involve actions by the monitor. We write $\mathsf{monitor}(t)$ to denote a check on condition t by the monitor. The conditions highlighted in gray are only used to prove properties of the monitor; they are not needed to understand the semantics of the monitor nor could they actually be checked by a monitor. For example, an application of the havoc rule is by design an internal transition and not observable. For the rest of this section, conditions highlighted in gray can be safety ignored. We will come back to these rules and explain these conditions in detail in Section 4.

The first five rules define the semantics for a forwarding process where the first two rules apply to both positive and negative types. The only differences between these two rules are the highlighted gray boxes which we will explain in detail in Section 4.2. A forwarding process is only allowed to concatenate the two message queues if their corresponding types match. Rule $\mathsf{id\_a}$ raises an alarm when a process tries to connect two queues that are expecting messages of different types. The argument of the alarm is the provider channel of the process that forwards. The equality check of the types also ensures that the directions of the two queues are the same, which is an invariant maintained by the queue.

Rule id_{ap} raises an alarm because the forwarding process tries to connect two wrong ends of the queues which would incorrectly identify two providers. Rule id'_{ap} raises an alarm because the process c tries to forward to a queue with the wrong provider channel. Well-typed processes will not trigger the last three rules, as these processes cannot be typed using rules in Figure 3.

The rules for cut are augmented, so that new edges are added persistently to the graph G. The monitor also records the types of the ends of the newly generated message queue. The monitor will check session fidelity based on these types. These two rules only differ for proof purposes. We omit the cut rules for negative types.

The rules for waiting for a channel to terminate only differ for proof purposes and are the same as rule $\mathsf{one\_r}$. If a process waits on the wrong channel, the process gets stuck and the monitor does

$$\begin{array}{lll}
\mathsf{id} & : & \mathsf{queue}(c'{:}A_1, p, c{:}A_2) \otimes \mathsf{proc}(c, c \leftarrow d') \otimes \mathsf{queue}(d'{:}B_1, q, d{:}B_2) \otimes !(\mathsf{monitor}(A_2 = B_1)) \otimes \;!(c \notin H) \\
 & & \quad \multimap \{\mathsf{queue}(c'{:}A_1, p \cdot q, d{:}B_2)\} \\
\mathsf{id}_h & : & \mathsf{queue}(c'{:}A_1, p, c{:}A_2) \otimes \mathsf{proc}(c, c \leftarrow d') \otimes \mathsf{queue}(d'{:}B_1, q, d{:}B_2) \otimes !(\mathsf{monitor}(A_2 = B_1)) \otimes \;!(c \in H) \\
 & & \quad \multimap \{\mathsf{queue}(c'{:}A_1, p \cdot q, d{:}B_2) \otimes \;!(\mathsf{dangling}(d'))\;\} \\
\mathsf{id}_a & : & \mathsf{queue}(c'{:}A_1, p, c{:}A_2) \otimes \mathsf{proc}(c, c \leftarrow d') \otimes \mathsf{queue}(d'{:}B_1, q, d{:}B_2) \otimes !(\mathsf{monitor}(A_2 \neq B_1)) \\
 & & \quad \multimap \{\mathsf{alarm}(c)\} \\
\mathsf{id}_{ap} & : & \mathsf{queue}(c'{:}A_1, p, c{:}A_2) \otimes \mathsf{proc}(c, c \leftarrow d) \otimes \mathsf{queue}(d'{:}B_1, q, d{:}B_2) \multimap \{\mathsf{alarm}(c)\} \\
\mathsf{id}'_{ap} & : & \mathsf{queue}(c'{:}A_1, p, c{:}A_2) \otimes \mathsf{proc}(c, a \leftarrow d') \otimes !(\mathsf{monitor}(a \neq c)) \multimap \{\mathsf{alarm}(c)\} \\
 & & \\
\mathsf{cut}^+ & : & \mathsf{proc}(c, x{:}A^+ \leftarrow P_x \;;\; Q_x) \otimes \;!(c \notin H) \\
 & & \quad \multimap \{\exists a.\ \exists a'.\ \mathsf{proc}(c, Q_{a'}) \otimes \mathsf{queue}(a'{:}A^+, \overleftarrow{\cdot}, a{:}A^+) \otimes \mathsf{proc}(a, P_a) \otimes !(G(c \rightarrow_{sp} a))\} \\
\mathsf{cut}^+_h & : & \mathsf{proc}(c, x{:}A^+ \leftarrow P_x \;;\; Q_x) \otimes \;!(c \in H) \\
 & & \quad \multimap \{\exists a.\ \exists a'.\ \mathsf{proc}(a, Q_{a'}) \otimes \mathsf{queue}(a'{:}A^+, \overleftarrow{\cdot}, a{:}A^+) \otimes \mathsf{proc}(a, P_a) \otimes !(G(c \rightarrow_{sp} a)) \otimes \;!(H(c \rightarrow_{sp} a))\;\} \\
\mathsf{one\_r} & : & \mathsf{proc}(c, \mathsf{wait}\ a' \;;\; Q) \otimes \mathsf{queue}(a'{:}\mathbf{1}, \overleftarrow{\mathsf{end}}, \_) \otimes \;!(c \notin H)\; \multimap \{\mathsf{proc}(c, Q)\} \\
\mathsf{one\_r}_h & : & \mathsf{proc}(c, \mathsf{wait}\ a' \;;\; Q) \otimes \mathsf{queue}(a'{:}\mathbf{1}, \overleftarrow{\mathsf{end}}, \_) \otimes \;!(c \in H)\; \multimap \{\mathsf{proc}(c, Q) \otimes \;!(\mathsf{dangling}(a'))\;\} \\
\mathsf{one\_s}_a & : & \mathsf{queue}(a'{:}B, q, a{:}A) \otimes \mathsf{proc}(a, \mathsf{close}\ a) \otimes !(\mathsf{monitor}(\neg(\mathsf{end} \rhd A))) \multimap \{\mathsf{alarm}(a)\} \\
\mathsf{one\_s}_{ap} & : & \mathsf{queue}(a'{:}B, q, a{:}A) \otimes \mathsf{proc}(a, \mathsf{close}\ b) \otimes !(\mathsf{monitor}(a \neq b)) \multimap \{\mathsf{alarm}(a)\} \\
 & & \\
\mathsf{alarm_s} & : & \mathsf{proc}(c, \mathsf{send}\ a'\ m \;;\; R) \otimes \mathsf{queue}(a'{:}A, q, a{:}B) \otimes !(\mathsf{monitor}(\neg(m \rhd A))) \multimap \{\mathsf{alarm}(c)\} \\
\mathsf{alarm_p} & : & \mathsf{queue}(b'{:}A, q, b{:}B) \otimes \mathsf{proc}(a, \mathsf{send}\ b\ \_ \;;\; R) \otimes !(\mathsf{monitor}(a \neq b)) \multimap \{\mathsf{alarm}(a)\} \\
 & & \\
\mathsf{havoc} & : & \mathsf{proc}(c, P) \otimes !(\mathsf{fn}(P) \supseteq \mathsf{fn}(Q)) \multimap \{\mathsf{proc}(c, Q) \otimes \;!(\mathsf{havoc}(c))\;\}
\end{array}$$

Figure 6: Selected Monitor Rules. We write $\mathsf{monitor}(t)$ to denote a check on condition t by the monitor. The conditions highlighted in gray are only used to prove properties of the monitor. Predicate $\mathsf{dangling}(a)$ means that the channel name a is a *dangling* channel name that has no queue associated with it, either because it was closed or forwarded, but may still appear in some processes.

not take any action. This is because the monitor does not inspect any internal actions of the process. Rule one_sa raises an alarm if a process tries to close a channel that either is not the process' provider channel or is not of type $\mathbf{1}$.

The two alarm rules in Figure 6 summarize common cases where the monitor raises an alarm. Rule alarm_s raises an alarm when the message sent to the queue is not compatible with the type expected by the queue. This rule also covers the case when a process sends a message to a queue that has the wrong direction. Rule alarm_p raises an alarm when a process a uses a provider channel that is not its own provider channel.

We omit additional rules as they are similar to the ones in Figure 6. For the send and receive cases where the monitor's checks pass, the monitor additionally is in charge of changing the types of the channels recorded at the end of the message queues.

Blame assignment When an alarm ($\mathsf{alarm}(a)$) is raised, the monitor assigns blame to all the direct ancestors of a in the graph G. That is, we find the tree in G that contains a, and let $c_1 \rightarrow_{sp} c_2 \cdots c_n \rightarrow_{sp} a$ be the path in that tree from the root to a. Then, the processes in the set $\{c_1, \cdots, c_n, a\}$ are jointly blamed. Informally, at least one of the processes in that set must have "havoced"; otherwise, type preservation will ensure that no alarm is raised. We formally show the correctness of the blame assignment in Section 4.

3.3 Motivating Examples

In this example, we illustrate monitoring with a mobile photosharing application, `Snapchat`, that takes and shares a user's photos and sends them to some remote entity. To take photos, `Snapchat` needs to operate the camera. To prevent the `Snapchat` application from continuously taking and sharing the user's photos, the camera requires that the user grant `Snapchat` permission every time `Snapchat` wants to take and share a photo.

This example contains three main processes: the `Snapchat` application process, the camera process, and the user process. The monitor checks messages when they are enqueued by a sender. If the `Snapchat` process deviates from its prescribed session types, for instance, if it tries to sends an invalid permission to the camera process, the monitor should raise an alarm.

3.3.1 Types and Encoding

We encode the expected behavior of each process as a session type declaration below.

$$\begin{array}{l}
\mathsf{stype\ Cam}^- = \&\{\mathsf{take} : \mathsf{photoPerm}^+ \\
\qquad\qquad\qquad \multimap\uparrow(\mathsf{picHandle}^+ \otimes \downarrow\mathsf{Cam}^-)\} \\
\mathsf{stype\ User}^- = \&\{\mathsf{picPerm} : \uparrow\oplus\{\mathsf{fail} : \downarrow\mathsf{User}^-; \\
\qquad\qquad\qquad\qquad\qquad \mathsf{succ} : \mathsf{photoPerm}^+ \\
\qquad\qquad\qquad\qquad\qquad\quad \otimes\downarrow\mathsf{User}^-\}\} \\
\mathsf{stype\ photoPerm}^+ = \oplus\{\mathsf{once} : \diamond_U \mathsf{ok} \wedge \mathbf{1}\} \\
\mathsf{stype\ Snap}^+ \quad\;\; = \oplus\{\mathsf{share} : \mathsf{pic} \wedge \downarrow\uparrow\mathsf{Snap}^+\}
\end{array}$$

Camera The camera process offers a service of type Cam, which is an external choice with a single option: take. After the client selects take, the camera process requires the client to send the channel of a photo permission process of type photoPerm before sending a handle to a picture to the client. Once the permission is received and checked and picture handle is sent, the camera process continues to offer a service of type Cam.

The camera application runs the following *CameraFun* process. We assume that the *takePic* function returns a picture handle. The code also details the communication pattern between the camera process and the photo permission process (lines 5–7). Upon receiving a channel pm, the camera process receives a signature from the channel pm. In this example, the camera process does not validate the signature of the permission by itself, but instead relies on the monitor to check the signature, which we will explain later when we discuss the monitoring scenarios.

```
1   CameraFun : {Cam⁻} =
2   c ← CameraFun =
3   case c of
4   | take → pm ← recv c ;
5              case pm of
6              | once → x ← recv pm ;
7                       wait pm ;
8                       shift ← recv c ;
9                       picH ← takePic ;
10                      send c picH ;
11                      send c shift ;
12                      c ← CameraFun
```

User The user process offers a service of type User, which is an external choice with a single option picPerm. When a process needs permission to access the camera, it communicates with the user process and selects picPerm. The user process then waits to receive a shift before it can send an internal choice label of either fail or succ. After the user sends the fail label, the user process continues to offer a service of type User, without granting its client permission to use the camera. After the user process sends a succ label, it then spawns a new process that provides a service of type photoPerm and sends the new process' channel to its client. The type photoPerm is an internal choice, labeled once. The newly spawned process first sends the label once, then sends a digital signature of a token ok (of type ok) using the camera's private key, before it terminates. The digital signature serves as an unforgeable authentication token for a permission to access the camera once.

The code snippet that corresponds to the permission process spawned by the user is given below.

```
1 send pm once ;
2 send (sign K_priv(U) ok) ;
3 close pm ;
```

The function *sign* will use the user's private key to sign the abstract type ok. We assume that this permission process has access to the user's private key as it is spawned by the user process.

Snapchat Finally, the `Snapchat` application offers a service of type Snap, which states that the application offers a single internal choice with label share that first sends a picture, then continues behaving as Snap. The double shifts forces a synchronization between `Snapchat` and its client; `Snapchat` will not send a second picture before the client has received the first picture in the message queue. The `Snapchat` application uses services offered by the camera process and the user process. These communications are not specified in the Snap type, but in the Cam and User type.

Next, we define the process for the `Snapchat` application. The *ToSnap* process uses c to communicate with the camera process c and u to communicate with the user process and offers the picture sharing service along channel s. We note that `convert` is a function that converts a picture handle to a string of type pic.

```
1   ToSnap : {Snap⁺ ← User⁻, Cam⁻}
2   s ← ToSnap ← u, c =
3   c.take ;
4   u.picPerm ;
5   send u shift ;
6   case u of
7   | fail → shift ← recv c ;
8            shift ← recv u ;
9            s ← ToSnap ← u, c
10  | succ → s.share ;
11            perm ← recv u ;
12            send c (y ← (y ← perm)) ;
13            send c shift ;
14            picH ← recv c ;
15            send s convert(picH) ;
16            send s shift ;
17            shift ← recv c ;
18            shift ← recv u ;
19            shift ← recv s ;
20            s ← ToSnap ← u, c
```

The `Snapchat` application first instructs the camera via channel c to take a picture. The *ToSnap* process then asks the user process for permission and cases on the response from the user. If the user does not grant the permission (line 7), then no picture is sent and the `Snapchat` process continues to try and send a picture. If the user grants the permission (line 10), the `Snapchat` application sends its client the label share, receives a channel connecting to a permission process from the user, then forwards the channel to the camera (line 12). It then receives a picture handle from the camera, which is converted to a picture and sent to `Snapchat`'s client.

3.3.2 Monitoring Scenarios

We show three monitoring scenarios to demonstrate how our monitor can detect violations of invariants specified by the session types. In the scenarios, an attacker tries to take pictures without being granted permissions required by the camera.

Scenario 1 The `Snapchat` process is compromised by an attacker. The havoced process does not ask for permission from the user and instead of sending a permission to the camera, sends an integer value (i.e. replacing lines 6-20 of the *ToSnap* process with (send $c\ n$). Right before this send, the queue associated with the camera process is:

$$\mathsf{queue}(cm'{:}\mathsf{photoPerm}^{+} \multimap {\uparrow}(\mathsf{picHandle}^{+} \otimes {\downarrow}\mathsf{Cam}^{-}), \overrightarrow{\cdot}, cm{:}\mathsf{photoPerm}^{+} \multimap {\uparrow}(\mathsf{picHandle}^{+} \otimes {\downarrow}\mathsf{Cam}^{-})$$

The `Snapchat` process is $\mathsf{proc}(s, \mathsf{send}\ cm'\ n\ ;\ \cdots)$. Here, the program variable c is substituted by the concrete channel cm'. The queue is expecting a value of type PhotoPerm which should be a channel. The monitor checks $n \rhd \mathsf{photoPerm}^{+} \multimap {\uparrow}(\mathsf{picHandle}^{+} \otimes {\downarrow}\mathsf{Cam}^{-})$. The check fails, causing the monitor to raise an alarm ($\mathsf{alarm}(s)$). In this case, s is a node in the monitor's graph G with no edges attached. Here, blame is assigned to one process, `Snapchat` (offering along channel s).

Scenario 2 `Snapchat` is again compromised. Instead of asking for permission, it tries to spawn a permission process using a fake signature. That is, replacing lines 6-20 of the *ToSnap* process with

```
send c (y ← y.once ;
            send y (sign K_priv(A) ok) ;
            close y)
```

The above send will succeed and spawn a new process:

$$\mathsf{proc}(d, \mathsf{send}\ d\ \mathsf{once}\ ;\ \mathsf{send}\ d\ (\mathit{sign}\ \mathsf{ok})\ ;\ \mathsf{close}\ d)$$

At this point, the graph G is augmented with $s \rightarrow_{sp} d$. After the label once is sent, the queue associated with process d is:

$$\mathsf{queue}(d' : \mathsf{photoPerm}^{+}, \overleftarrow{\mathsf{once}}, d : \diamond_{U}\mathsf{ok} \wedge \mathbf{1})$$

When the process d tries to send a signature, it does not have the user's key, it cannot generate a value v such that $v \rhd \diamond_{U}\mathsf{ok}$. When the process d sends $(sign\ \mathsf{ok})$ to d, the monitor's check fails and raises an alarm ($\mathsf{alarm}(d)$). The blame is assigned to the set $\{s, d\}$ as G includes $s \rightarrow_{sp} d$, and s is the root.

Scenario 3 The `Snapchat` process is working appropriately and has gotten a legitimate photo permission from the user. When the Snapchat process spawns a process on line 12 to send the permission to the camera, the spawned process is taken over by an attacker. Instead of $\mathsf{proc}(d, d \leftarrow pm)$, where the process offering along pm is spawned by the user, the attacker changes it to $\mathsf{proc}(d, \mathsf{send}\ d\ n)$. The monitor will raise an alarm ($\mathsf{alarm}(d)$) when the above compromised process tries to send an integer value to d, because the monitor is expecting the label once. The graph G includes $s \rightarrow_{sp} d$, so the blame is assigned to $\{s, d\}$. This scenario has the same blame assignment as the previous scenario, even though two different processes are compromised. One interesting point is that in this scenario, `Snapchat` actually has the right permission. The attacker effectively launched a denial of service attack.

4. Metatheory

We first define several properties of the monitor, then we prove that our monitor satisfies those properties. The main challenge in constructing these proofs lies in identifying the invariants maintained at runtime in the presence of havoced processes. Formalizing these invariants and proving that the invariants hold at runtime is essential for proving the blame correctness theorem.

4.1 Properties of the Monitor

We identify four high-level properties that the monitor should satisfy: correctness of the blame assignment, minimality of the blame assignment, the fact that well-typed processes are not blamed, and transparency of the monitor.

The correctness of the blame assignment is defined as follows. Recall that Ω is the multiset of processes and queues describing the current state of computation. We say that it is correct to blame a set of processes if at least one of the processes in the set has made a havoc transition. As some of the examples below show, we cannot in general narrow it down to a single process because the monitors can only observe messages and not anything regarding the internal state of process. We write $\models \Omega : \mathsf{wf}$ to denote that the state Ω is well-typed, formally defined in Section 4.2. This well-typedness requires that all processes in Ω be typed using typing rules in Figure 3 and that the use of the channel names in Ω satisfy linearity constraints[1].

Definition 1 (Correctness of blame)**.** *A set of processes $\mathcal{N}$ is correct to be blamed w.r.t. the execution trace $\mathcal{T} = \Omega, G \longrightarrow^* \Omega', \mathsf{alarm}(a)$ with $\models \Omega : \mathsf{wf}$ if there is a $b \in \mathcal{N}$ such that b has made a havoc transition in $\mathcal{T}$.*

Second, if all processes are well-typed to begin with and no process is compromised at runtime, then the monitor should not raise an alarm. This property shows that a havoc transition is necessary for the monitor to halt the execution and assign blame.

Definition 2 (Well-typed configurations do not raise alarms)**.** *Given any $\mathcal{T} = \Omega, G \longrightarrow^* \Omega', G'$ such that $\models \Omega : \mathsf{wf}$ and $\mathcal{T}$ does not contain any havoc transitions, there does not exists an a such that $\mathsf{alarm}(a) \in \Omega'$.*

Third, the set of processes that the monitor assigns blame to should be as small as possible. An algorithm that always blames all processes in the case of an alarm is correct (according to our definition), but not minimal. Formalizing minimality requires a theory of observational equivalence with respect to a class of monitors and is beyond the scope of this paper. However, as some of our examples illustrate, including ancestors of a for $\mathsf{alarm}(a)$ in the blame set is required. Since we do not include any processes beyond these, we conjecture that our blame sets are indeed minimal.

Finally, the monitor should not change the behavior of well-typed processes. We write $\longrightarrow^-$ to denote the operational semantics without the monitor. If the initial configuration is well-typed and no process is compromised, then executing the configuration with and without the monitor should yield the same result.

Definition 3 (Monitor transparency)**.** *Given any $\mathcal{T} = \Omega, G \longrightarrow^* \Omega', G'$ such that $\models \Omega : \mathsf{wf}$ and $\mathcal{T}$ does not contain any havoc transitions. Then $\Omega(\longrightarrow^-)^*\Omega''$, where Ω'' is obtained from Ω' by removing typing information from queues.*

4.2 Validating Properties of our Monitor

We prove that our monitor assigns blame correctly, does not blame well-typed processes, and is transparent. We illustrate the minimality of the blame assignment of our monitor through examples. For the rest of this section, we call a process that has made a havoc transition, or is a descendant of a process that has made a havoc transition a *havoced* process.

Augmenting the configuration To formally define and analyze the properties of our monitor, we augment the operational semantics with more information about the actions of havoced processes.

We write Ω_Q to denote a list of queues, and Ω_P and Ω_H to denote a list of unhavoced and havoced processes respectively.

States	Ω	$::=$	Ω_P, Ω_Q
Queues	Ω_Q	$::=$	$\cdot \mid \mathsf{queue}(a{:}A, q, R), \Omega_Q$
Processes	Ω_P, Ω_H	$::=$	$\cdot \mid \mathsf{proc}(c, P), \Omega_P$

First, we use an additional graph H to keep track of havoced processes. Graph H should be a sub-graph of G and each tree in H is rooted at a provider channel name of a process that has made a havoc transition. When a process makes a havoc transition, a new node is added to H, denoted by $!(\mathsf{havoc}(a))$ in the operational semantic rule for havoc (highlighted in gray in Figure 6). When a process offering along a spawns a new process b and a is a node in H, then an edge from a to b is added to H, denoted by $!(H(a \rightarrow_{sp} b))$ (e.g., in cut_h^+ rule in Figure 6). When b is a node in H (denoted $b \in H$), b is a havoced process.

Second, because havoced processes may violate linearity constraints, we define Θ to denote the set of channel names that are used by havoced processes. Channels in Θ could potentially have aliases or be dangling references to closed or forwarded channels.

The small-step operational semantics rules are either of the form $\Omega, G, H, \Theta \longrightarrow \Omega', G', H', \Theta'$ or $\Omega, G, H, \Theta \longrightarrow \_, H, \mathsf{alarm}(a)$. Since we assume a global abort, we do not write out the full configuration in the second kind of transition. In these rules, H and Θ are present purely for the purpose of proving metatheorems. The set of channel names in Θ may increase because compromised processes generate dangling pointers via closing a channel or forwarding a channel. In Figure 6, rules id_h and $\mathsf{one\_r}_h$ augment Θ and generate two dangling processes names. In Figure 6, rule id is different from id_h in that the forwarding process is a havoced process in the latter case. Similarly, the two cut rules differ in whether the process that spawns a new process is a havoced process.

It is easy to show that $\Omega, G, H, \Theta \longrightarrow \Omega', G', H', \Theta'$ implies $\Omega, G \longrightarrow \Omega', G'$ and $\Omega, G, H, \Theta \longrightarrow \_, H', \mathsf{alarm}(a)$ implies $\Omega, G \longrightarrow \_, \mathsf{alarm}(a)$.

Typing the configurations The typing rules for states use several contexts. The context Ψ maps channel names to their types. We note that all channels in Ψ are linear. The queue channel typing context Γ contains type bindings for both ends of a queue. For

[1] We use two notions of well-typedness: the above mentioned wf judgment and a weaker, more general one that types compromised processes based on free channel names that appear in them. We will explain this in Section 4.2.

instance, $d'{:}A \Leftarrow d{:}B$ denotes that the provider channel of the queue has type B and the client channel of the queue has type A.

Linear chan. typing ctx	Ψ	$::=$	$\cdot \mid \Psi, d{:}A$
Queue chan. typing ctx	Γ	$::=$	$\cdot \mid \Gamma, d{:}A \Leftarrow R$
Non-linear Chan. ctx	Θ	$::=$	$\cdot \mid \Theta, d$
Right endpoint	R	$::=$	$c{:}C \mid \_$

We define two functions $\mathsf{client}(\Gamma)$ and $\mathsf{provider}(\Gamma)$. The function $\mathsf{client}(\Gamma)$ returns the channel typing context that contains only client channels in Γ. That is, for every $d{:}A \Leftarrow R$ in Γ, $\mathsf{client}(\Gamma)$ returns $d{:}A$. Similarly, we define $\mathsf{provider}(\Gamma)$ to return the context that contains only provider channels in Γ: for every $d{:}A \Leftarrow R$ in Γ, $\mathsf{provider}(\Gamma)$ returns R. We write $\uplus$ to denote that two contexts are disjoint in their domains. We write $\Psi_1 \uplus \Psi_1 ... \uplus \Psi_n$ to mean that any two contexts Ψ_i and Ψ_j have disjoint domains, where $i, j \in [1, n]$.

The typing rules for queues, processes, and configurations are summarized in Figure 7. The judgment $\Psi \models_q \Omega_Q : \Gamma$ means that queues in Ω_Q are well-typed under Ψ. The context Γ contains the provider/client channel types of all the queues in Ω_Q. The rule for checking a single queue uses the judgment $\Psi \vdash \mathsf{queue}(a{:}A, q, R)$ (shown in Figure 8) to check that the elements in q are exactly the channel names in the domain of Ψ and that the types of elements in q, as specified in Ψ, respect the types of the two ends of the queue. The last rule ensures channels names are used linearly.

The typing judgment for unhavoced processes is $H; \Psi_i \models_u \Omega_P : \Psi_o$. The rules for this judgment check that processes in Ω_P are not in H and that the processes use channels in Ψ_i and provide services on channels in Ψ_o. The rule is similar to the last rule for queue typing: it ensures that processes only use channel names linearly.

The typing judgment for havoced processes is $H; \Theta \models_h \Omega_H : \Psi_o$. As before, H is the graph tracking havoc transitions and the context Ψ_o maps channel names to their types. The context Θ contains all the channels used by processes in Ω_H (not necessarily linearly). The rule for checking a single havoced process $\mathsf{proc}(c, P)$ checks that a free name used in the process is either in Θ, or the provider channel of the process, or a direct ancestor of c in H. The last rule type checks a set of havoced processes. Here, the variable contexts are shared across the premises of that rule. This means that havoced processes do not need to use channel names linearly.

The top-level typing rule for the entire state is $H; \Theta \models \Omega$. We write Ψ^* to denote the context resulted from erasing types in Ψ. The context Ψ^* only contains the channel names in Ψ. The key points of the above typing rule are that: (1) the queues, unhavoced processes, and havoced processes use mutually disjoint sets of channels, (2) havoced processes additionally use a set of dangling channels (in Θ), (3) the client channels, including their types, used by queues and processes should match exactly the set of client channels of the queues, and (4) the provider channels of the queues can be split into two disjoint sets, one that matches the set of provider channels, including their types, of unhavoced processes; the other that matches the provider channel names of havoced processes.

By restricting H and Θ to be empty, we obtain the stricter definition of well-typedness of configurations used in our property definitions. We define $\models \Omega : \mathsf{wf}$ as $\emptyset; \cdot \models \Omega$. That is, the configuration Ω can be typed under the assumption that no processes have havoced and all channel names are used linearly.

We make the implicit well-formedness assumption that the context Γ's provider and client channels are disjoint from each other. That is $\mathsf{client}(\Gamma) \uplus \mathsf{provider}(\Gamma)$. The queue typing rules only generate well-formed Γ's.

$$\boxed{\Psi \models \Omega_Q : \Gamma}$$

$$\frac{}{\cdot \models_q \cdot : \cdot} \qquad \frac{\Psi \vdash_q \mathsf{queue}(a{:}A, q, R) \quad a \notin \mathsf{dom}(R)}{\Psi \models_q \mathsf{queue}(a{:}A, q, R) : a{:}A {\Leftarrow} R}$$

$$\frac{\Psi \models_q \Omega_Q : \Gamma \quad \Psi' \models_q \Omega'_Q : \Gamma' \quad \Psi \uplus \Psi' \quad \Gamma \uplus \Gamma'}{\Psi, \Psi' \models_q \Omega_Q, \Omega'_Q : \Gamma, \Gamma'}$$

$$\boxed{H; \Psi_i \models_u \Omega_P : \Psi_o}$$

$$\frac{}{H; \cdot \models_u \cdot : \cdot} \qquad \frac{c \notin H \quad \Psi \vdash P :: (c{:}A)}{H; \Psi \models_u \mathsf{proc}(c, P) : c{:}\, A}$$

$$\frac{\Psi \models_u \Omega_P : \Psi_o \quad \Psi' \models_u \Omega'_P : \Psi'_o \quad \Psi \uplus \Psi' \quad \Psi_o \uplus \Psi'_o}{\Psi, \Psi' \models_u \Omega_P, \Omega'_P : \Psi_o, \Psi'_o}$$

$$\boxed{H; \Theta \models_h \Omega_H : \Psi_o}$$

$$\frac{}{H; \cdot \models_h \cdot : \cdot}$$

$$\frac{c \in H \quad \mathsf{fn}(P) \subseteq \Theta \cup \mathsf{ancestors}(H, c) \cup \{c\}}{H; \Theta \models_h \mathsf{proc}(c, P) : c{:}\, \mathsf{havoc}}$$

$$\frac{H; \Theta \models_h \Omega_P : \Psi_o \quad H; \Theta \models_h \Omega'_P : \Psi'_o \quad \Psi_o \uplus \Psi'_o}{H; \Theta \models_h \Omega_P, \Omega'_P : \Psi_o, \Psi'_o}$$

$$\boxed{H; \Theta \models \Omega}$$

$$\frac{\begin{array}{l} \Psi^i_Q \models_q \Omega_Q : \Gamma^o_Q \\ H; \Psi^i_U \models_u \Omega_U : \Psi^o_U \\ H; \Theta, (\Psi^i_H)^* \models_h \Omega_H : \Psi^o_H \\ \mathsf{dom}(\Theta) \cap \mathsf{dom}(\Psi^i_Q, \Psi^i_U, \Psi^i_H) = \emptyset \\ \Gamma^o_Q = \Gamma_Q, \Gamma_U, \Gamma_H \quad \Psi^i_Q = \mathsf{client}(\Gamma_Q) \\ \Psi^i_U = \mathsf{client}(\Gamma_U) \quad \Psi^i_H = \mathsf{client}(\Gamma_H) \\ \Gamma^o_Q = \Gamma_1, \Gamma_2 \quad \Psi^o_U = \mathsf{provider}(\Gamma_1) \\ \mathsf{dom}(\Psi^o_H) = \mathsf{dom}(\mathsf{provider}(\Gamma_2)) \end{array}}{H; \Theta \models \Omega_Q, \Omega_U, \Omega_H}$$

Figure 7: Typing the process, queues, and configurations

Blame Correctness Theorem The fact that our monitor assigns blame correctly (Definition 1) is a collorary of Theorem 1 (see below).

Theorem 1 (Alarm)**.**

1. *If* $\emptyset; \cdot \models \Omega$ *and* $\Omega, \emptyset, \emptyset, \cdot \longrightarrow^* \Omega', G, H, \Theta$ *then* $H; \Theta \models \Omega'$
2. *If* $\emptyset; \cdot \models \Omega$ *and* $\Omega, \emptyset, \emptyset, \cdot \longrightarrow^* \_, H, \mathsf{alarm}(a)$ *then* $a \in H$.

The above theorem states that from an initial configuration, a well-typed configuration can make a series of transitions to either another well-formed configuration, or a state where an alarm is raised on a process a that is in the H graph. Based on the semantics, H only contains processess that either have made a havoc transition or are descendants of one that has made a havoc transition, and H is a subgraph of G, and therefore, the blame assigned to a and a's direct ancestors in G is correct.

Another collorary of Theorem 1 is that well-typed processes are not blamed: if the configuration is well-formed and no process is compromised, then the monitor will not raise any alarm. This is easy to prove because if there is an alarm associated with a, then a must be in H. However, when no process havocs, H remains empty; a contradiction.

$$\frac{}{\cdot \vdash \mathsf{queue}(a{:}A^+, \overleftarrow{\cdot}, c{:}A^+)}$$

$$\frac{}{\cdot \vdash \mathsf{queue}(a{:}A^-, \overrightarrow{\cdot}, c{:}A^-)}$$

$$\frac{}{\cdot \vdash \mathsf{queue}(a{:}\mathbf{1}, \overleftarrow{\mathsf{end}}, \_)}$$

$$\frac{\Psi \vdash \mathsf{queue}(a{:}A^+, \overleftarrow{p}, R)}{\Psi, b{:}B^+ \vdash \mathsf{queue}(a{:}B^+ \otimes A^+, \overleftarrow{b \cdot p}, R)}$$

$$\frac{\Psi \vdash \mathsf{queue}(a{:}A_j^+, \overleftarrow{p}, R)}{\Psi \vdash \mathsf{queue}(a{:}\oplus\{lab_i : A_i^+\}_i, \overleftarrow{lab_j \cdot p}, R)}$$

$$\frac{}{\cdot \vdash \mathsf{queue}(a{:}{\downarrow}A^-, \overleftarrow{\mathsf{shift}}, c{:}A^-)}$$

$$\frac{\Psi \vdash \mathsf{queue}(a{:}A, \overrightarrow{q}, c{:}C^-)}{\Psi, b{:}B^+ \vdash \mathsf{queue}(a{:}A, \overrightarrow{q \cdot b}, c{:}B^+ \multimap C^-)}$$

$$\frac{\Psi \vdash \mathsf{queue}(a{:}A, \overrightarrow{q}, c{:}C_j^-\})}{\Psi \vdash \mathsf{queue}(a{:}A, \overrightarrow{q \cdot lab_j}, c{:}\&\{lab_i : C_i^-\})}$$

$$\frac{}{\cdot \vdash \mathsf{queue}(a{:}A^+, \overrightarrow{\mathsf{shift}}, c{:}{\uparrow}A^+)}$$

Figure 8: Typing of Queues

The correctness proof for the blame assignment is similar to that of a preservation proof. The key lemma is Lemma 2, which states that if a well-typed configuration makes a transition, then it either steps to another well-formed configuration, or an alarm is raised on a process a that is in the H graph.

Lemma 2 (Alarm (one step)). *If $H; \Theta \models \Omega$ and $\Omega, G, H, \Theta \longrightarrow$ then*

1. *$\Omega, G, H, \Theta \longrightarrow \Omega', G', H', \Theta'$ implies $H'; \Theta' \models \Omega'$*
2. *$\Omega, G, H, \Theta \longrightarrow \_, H, \mathsf{alarm}(a)$ implies $a \in H$.*

Using the above lemma, we can prove Theorem 1 which considers a sequence of transitions. The proof is done by induction on the length of the trace.

Minimality We show an example scenario (Figure 9) where the monitor cannot tell which process in the blame set is compromised. Initially, no process is compromised. The initial configuration is Ω_0, which contains a process b and a queue. We omit other processes and queues, as they are not important for this example. The process offering along b spawns a process, which spawns another process that sends an integer to a'. We consider three execution traces $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$. We mark the havoc step using a superscript $H(x)$, where x is the channel name of the process that makes the havoc step. The configurations Ω_i and Ω_i' are the intermediary configurations in those traces. We use the convention that Ω_i' contains a compromised process; instead of sending 1 to a', the compromised process tries to send a channel to a'. When this send is executed, the monitor will raise an alarm.

All of the three traces end up raising the same alarm. When the alarm is raised, the graph G that the monitor maintains contains a chain denoting that b has spawned b_1 and b_1 has spawned b_2. Here, $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$ are observationally equivalent to the monitor. We

$$\begin{array}{ll}
P & = \mathsf{send}\ a'\mathsf{shift}; \mathsf{wait}\ a' \\
\Omega_0 & = \mathsf{proc}(b, x_1 \leftarrow (x_2 \leftarrow (\mathsf{send}\ a'\ 1; P; \mathsf{close}\ x_2); \\
 & \qquad\qquad \mathsf{wait}\ x_2; \mathsf{close}\ x_1); \\
 & \qquad \mathsf{wait}\ x_1; \mathsf{close}\ b) \\
 & \quad \otimes \mathsf{queue}(a' : \mathsf{int} \to {\uparrow}\mathbf{1}, \overrightarrow{\cdot}, a : \mathsf{int} \to {\uparrow}\mathbf{1}) \\
\Omega_0' & = \mathsf{proc}(b, x_1 \leftarrow (x_2 \leftarrow (\mathsf{send}\ a'\ (y \leftarrow \mathsf{close}\ y); \\
 & \qquad\qquad\qquad P; \mathsf{close}\ x_2); \\
 & \qquad\qquad \mathsf{wait}\ x_2; \mathsf{close}\ x_1); \\
 & \qquad \mathsf{wait}\ x_1; \mathsf{close}\ b) \\
 & \quad \otimes \mathsf{queue}(a' : \mathsf{int} \to {\uparrow}\mathbf{1}, \overrightarrow{\cdot}, a : \mathsf{int} \to {\uparrow}\mathbf{1}) \\
\Omega_1 & = \mathsf{proc}(b, \mathsf{wait}\ b_1'; \mathsf{close}\ b) \\
 & \quad \otimes \mathsf{queue}(b_1' : \mathbf{1}, \overleftarrow{\cdot}, b_1 : \mathbf{1}) \\
 & \quad \otimes \mathsf{proc}(b_1, x_2 \leftarrow (\mathsf{send}\ a'\ 1; P; \mathsf{close}\ x_2); \\
 & \qquad\quad \mathsf{wait}\ x_2; \mathsf{close}\ b_1); \\
 & \quad \otimes \mathsf{queue}(a' : \mathsf{int} \to {\uparrow}\mathbf{1}, \overrightarrow{\cdot}, a : \mathsf{int} \to {\uparrow}\mathbf{1}) \\
\Omega_1' & = \mathsf{proc}(b, \mathsf{wait}\ b_1'; \mathsf{close}\ b) \\
 & \quad \otimes \mathsf{queue}(b_1' : \mathbf{1}, \overleftarrow{\cdot}, b_1 : \mathbf{1}) \\
 & \quad \otimes \mathsf{proc}(b_1, x_2 \leftarrow (\mathsf{send}\ a'\ (y \leftarrow \mathsf{close}\ y); P; \mathsf{close}\ x_2); \\
 & \qquad\quad \mathsf{wait}\ x_2; \mathsf{close}\ b_1); \\
 & \quad \otimes \mathsf{queue}(a' : \mathsf{int} \to {\uparrow}\mathbf{1}, \overrightarrow{\cdot}, a : \mathsf{int} \to {\uparrow}\mathbf{1}) \\
\Omega_2 & = \mathsf{proc}(b, \mathsf{wait}\ b_1'; \mathsf{close}\ b) \\
 & \quad \otimes \mathsf{queue}(b_1' : \mathbf{1}, \overleftarrow{\cdot}, b_1 : \mathbf{1}) \\
 & \quad \otimes \mathsf{proc}(b_1, \mathsf{wait}\ b_2'; \mathsf{close}\ b_1); \\
 & \quad \otimes \mathsf{queue}(b_2' : \mathbf{1}, \overleftarrow{\cdot}, b_2 : \mathbf{1}) \\
 & \quad \otimes \mathsf{proc}(b_2, \mathsf{send}\ a'\ 1; P; \mathsf{close}\ b_2) \\
 & \quad \otimes \mathsf{queue}(a' : \mathsf{int} \to {\uparrow}\mathbf{1}, \overrightarrow{\cdot}, a : \mathsf{int} \to {\uparrow}\mathbf{1}) \\
\Omega_2' & = \mathsf{proc}(b, \mathsf{wait}\ b_1'; \mathsf{close}\ b) \\
 & \quad \otimes \mathsf{queue}(b_1' : \mathbf{1}, \overleftarrow{\cdot}, b_1 : \mathbf{1}) \\
 & \quad \otimes \mathsf{proc}(b_1, \mathsf{wait}\ b_2'; \mathsf{close}\ b_1); \\
 & \quad \otimes \mathsf{queue}(b_2' : \mathbf{1}, \overleftarrow{\cdot}, b_2 : \mathbf{1}) \\
 & \quad \otimes \mathsf{proc}(b_2, \mathsf{send}\ a'\ (y \leftarrow \mathsf{close}\ y); P; \mathsf{close}\ b_2) \\
 & \quad \otimes \mathsf{queue}(a' : \mathsf{int} \to {\uparrow}\mathbf{1}, \overrightarrow{\cdot}, a : \mathsf{int} \to {\uparrow}\mathbf{1}) \\
G & = b \to_{sp} b_1 \to_{sp} b_2
\end{array}$$

For simplicity, we omit G from the traces.

$$\begin{array}{ll}
\mathcal{T}_1 & = \Omega_0 \longrightarrow^{H(b)} \Omega_0' \longrightarrow \Omega_1' \longrightarrow \Omega_2' \longrightarrow \_, \mathsf{alarm}(b_2) \\
\mathcal{T}_2 & = \Omega_0 \longrightarrow \Omega_1 \longrightarrow^{H(b_1)} \Omega_1' \longrightarrow \Omega_2' \longrightarrow \_, \mathsf{alarm}(b_2) \\
\mathcal{T}_3 & = \Omega_0 \longrightarrow \Omega_1 \longrightarrow \Omega_2 \longrightarrow^{H(b_2)} \Omega_2' \longrightarrow \_, \mathsf{alarm}(b_2) \\
\mathcal{N} & = \{b, b_1, b_2\}
\end{array}$$

Figure 9: An example illustrating minimality of blame assignment.

define two traces to be observationally equivalent to the monitor if the lengths of the traces are the same and for any two states at the same position on the traces, all the channel names of the queues are the same and the graphs G are the same. This definition is reasonable because the monitor state consists entirely of the queue typing and the graph. In all three scenarios, the monitor assigns blame to the set $\{b, b_1, b_2\}$. This blame is minimal as the monitor cannot distinguish which trace it is on when an alarm is raised.

Transparency Our monitor is transparent; it does not alter the behavior of well-formed configurations. The proof is done by examining how each monitor check is applied to well-formed configurations. Since well-formed configurations do not have havoced processes (H is empty), all processes and queues are well-typed using the stricter typing rules. The fact that the monitor checks never fail can be obtained by inverting the typing judgments of the relevant queues and processes.

Preservation and progress? The above theorem is not the same as the preservation lemma proven in Pfenning and Griffith (2015).

Circular dependency:

$$\begin{aligned}
&\mathsf{proc}(c, \mathsf{send}\ a'\ (x{\leftarrow}(y{\leftarrow}\mathsf{recv}\ a'\ ;\ P_{\{x,y\}}))\ ;\ Q)\otimes\\
&\mathsf{queue}(a'{:}A^+{\multimap}B^-, \overrightarrow{\cdot}, a{:}A^+{\multimap}B^-)\otimes\\
&\mathsf{proc}(a, z{\leftarrow}\mathsf{recv}\ a\ ;\ z'{\leftarrow}\mathsf{recv}\ z\ ;\ R_{z'})\\
&\multimap \exists d.\exists d'.\ \mathsf{proc}(c, Q) \otimes \mathsf{queue}(a'{:}B^-, \overrightarrow{d'}, a{:}A^+{\multimap}B^-)\otimes\\
&\qquad\qquad \mathsf{proc}(a, z{\leftarrow}\mathsf{recv}\ a\ ;\ z'{\leftarrow}\mathsf{recv}\ z\ ;\ R_{z'})\otimes\\
&\qquad\qquad \mathsf{queue}(d'{:}A^+, \overleftarrow{\cdot}, d{:}A^+)\otimes\\
&\qquad\qquad \mathsf{proc}(d, y{\leftarrow}\mathsf{recv}\ a'\ ;\ P_{\{d,y\}})\\
&\multimap \mathsf{proc}(c, Q) \otimes \mathsf{queue}(a'{:}B^-, \overrightarrow{\cdot}, a{:}B^-)\otimes\\
&\quad \mathsf{proc}(a, z'{\leftarrow}\mathsf{recv}\ d'\ ;\ R_{z'})\otimes\\
&\quad \mathsf{queue}(d'{:}A^+, \overleftarrow{\cdot}, d{:}A^+) \otimes \mathsf{proc}(d, y{\leftarrow}\mathsf{recv}\ a'\ ;\ P_{\{d,y\}})
\end{aligned}$$

Dangling channel:

$$\begin{aligned}
&\mathsf{queue}(c'{:}A^+, \overleftarrow{p}, c{:}A^+) \otimes \mathsf{proc}(c, c{\leftarrow}d')\otimes\\
&\mathsf{queue}(d'{:}A^+, \overleftarrow{q}, d{:}A^+) \otimes \mathsf{proc}(a, \mathsf{send}\ d'\ ;\ P)\\
&\multimap \mathsf{queue}(c'{:}A^+, \overleftarrow{p \cdot q}, d{:}A^+) \otimes \mathsf{proc}(a, \mathsf{send}\ d'\ ;\ P)
\end{aligned}$$

Figure 10: Example configurations.

This is because the configuration typing rules maintain looser invariants than those in Pfenning and Griffith (2015). More concretely, the preservation allows violation of linearity by havoced processes, such as sharing channels, using dangling channels, and creating circular dependencies of queues. These violations are not possible if no havoc transitions are made.

For similar reasons, well-typed configurations may be stuck (do not have progress). We show a few examples of violations that result in stuck system states in Figure 10. In the first example, process c has havoced. Notice that c uses the client channel a' multiple times. In the first step, c spawns a new process d and sends the client channel d' of that new process to the queue a'. In the second step, process a receives the client channel d', and waits to receive from d'. At the same time process d waits on a'. Now processes a and d wait on each other and the system is stuck. In the second example, both processes a and c have havoced. Otherwise, they can't both have access to the client channel d'. (For a and c to share a channel, we can use the same trick used in the previous example: c can be spawned by a and contains all the channels that a can access). Process c forwards d' to c, now d' disappears. However, process a can still attempt to operate on d'. Since d' doesn't exist anymore, a is stuck.

5. Related Work

There is a rich body of work on higher-order contracts and the correctness of blame assignments in the context of lambda calculus since Findler and Felleisen (2002) first introduced higher-order contracts and the concept of blame. Wadler and Findler (2009) defined the first blame calculus and established that blame always lies with the less-precisely typed-code. More comprehensive theorems about the correctness of blame assignment have been proposed by Dimoulas et al. (2012, 2011).

Subsequent work on gradual typing that considers systems with both static and dynamic typing also uses "blame always lies with the less-precisely typed code" as a criteria for correctness. For instance, Ahmed et al. (2011) developed a blame calculus for a language that integrates parametric polymorphism with static and dynamic typing. Fennell and Thiemann (2012) proved a blame theorem for a linear lambda calculus with type Dynamic. Most recently, Wadler (2015) surveys the history of the blame calculus and presents the latest developments. Keil and Thiemann (2015) develop a blame assignment for higher order contracts that includes intersection and union contracts. Siek et al. (2015) develop three calculi for gradual typing and relate them in an effort to unite the concepts of blame and coercion.

Compared to the body of work mentioned above, our work focuses on distributed systems, where processes communicate with each other via message queues. At a high-level, we can relate our adversary model to the work on blame assignment as follows. Each process can be viewed as a program written in dynamically typed language. Our monitor enforces the coercion of session types by observing the communications between the processes. Our blame assignment is the ancestor closure of the process that has directly caused an alarm to be raised. The set of processes that is blamed always includes a compromised process (all of its descendants are likely to have been compromised too). If we view the compromised process as a less-precisely-typed program, our correctness of blame property is similar to the notion proposed in Wadler and Findler (2009): blame always falls on less-precisely-typed programs.

Disney et al. investigated behavior contracts that enforce temporal properties for modules (Disney et al. 2011). Our contracts (i.e., session types) enforce temporal properties as well; the session types specify the order in which messages are sent and received by the processes. Our contracts are not as expressive as the temporal higher-order contracts proposed by Disney et al, but our system is concurrent, while their system does not consider concurrency.

The work most closely related to ours is on multi-party session types (Bocchi et al. 2013; Chen et al. 2011; Thiemann 2014). Bocchi et al. (2013)'s work and Chen et al. (2011)'s work assume a similar asynchronous message passing model as ours. Their monitor architecture is also similar to ours; monitors are placed at the ends of the communication channels and monitor communication patterns. One key difference is that their monitors do not raise alarms; instead, the monitors suppress "bad" messages and move on. Our monitors halt the execution and assign blame. Consequently, this work does not have theorems about blame assignment which are central to our work. Using global types, their monitors can additionally enforce global properties such as deadlock freeness, which our monitors cannot. Our work supports higher-order processes, while their work is strictly first-order. Recently, Thiemann (2014) has taken steps towards integrating gradual typing and a two-party session-type system. There, the communications are synchronous. A *proxy* process acts as a monitor for a forked dynamically typed process, mediating the communications of the forked process. This proxy can be viewed as a partial identity process that eta-expands the coerced session type. As discussed in Section 3.2, we moved away from that design and chose a more local monitoring strategy to achieve more precise blame assignment. Neither our system nor their system prevents deadlock. We additionally proved both the correctness of blame and the monitor transparency properties of our monitor.

6. Conclusion

We have presented a system for monitoring and blame assignment for session types based on a Curry-Howard interpretation of linear logic. Communication is asynchronous, through message queues, and our monitors manage these queues based on a types-as-contracts interpretations of the session types. Monitored communications include spawning of processes (logically the cut rule), forwarding between processes (logically the identity rule), internal and external choice, as well as passing of channels along channels which gives the language some higher-order aspects. Our adversary or failure model allows a process to make arbitrary transitions to ill-typed code in a step we call *havoc*, subject to the constraint that a havoced process can not gain access to other private channels. Proving correctness of the blame assignment has been technically challenging because execution may continue with havoced processes for many steps before an observable type violation occurs (if at all). Process configurations with rogue processes are complex and, among other invariants, may violate the linearity constraints which force channels to have exactly one provider and one client.

We have omitted shared channels from this presentation. An in-depth discussion of their typing and operational reading can be found in Pfenning and Griffith (2015). A shared channel is always provided by a unique persistent process and can be used by arbitrarily many other processes. We can only use a shared channel of type $\uparrow^{\mathsf{U}} A$ by spawning a new linear session of type A, leaving the persistent process in place. This new linear session would be monitored with the techniques described here. Monitoring and blame assignment of the shared channels themselves is actually easier, since persistent message "queues" are always empty or singletons.

We do not consider the mechanics of how to connect to offered services, be they linear or persistent. As long as offered services come with a type (which is required in any case for session-typed communication), monitoring can then commence as described in this paper once a connection has been established.

One remaining interesting technical question is whether our blame assignment identifies a minimal set of potential culprits, given the kinds of observations we allow the monitor to make. We conjecture that this is so, but currently we lack the theory of observational equivalence in this setting to answer it definitively. Another interesting question is whether we can be more precise if we allow more global observations about the state of the computation, for example, the geometry of the connections. Precision here can refer both to smaller blame sets and to discovering more violations which are undetected by simple type-checking. Along the latter dimension, it would also be interesting to add contracts or dependent types to the language so that more violations can be discovered (potentially both statically and dynamically).

Acknowledgment

This research was supported in part by NSF grant CNS1423168.

References

A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, 2011.

L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. In *Formal Techniques for Distributed Systems (FMOODS 2013)*, 2013.

L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *21st International Conference on Concurrency Theory (CONCUR 2010)*, 2010.

L. Caires, F. Pfenning, and B. Toninho. Towards concurrent type theory. In *7th Workshop for Types in Language Design and Implementation (TLDI 2012)*, 2012. Notes for an invited talk.

L. Caires, F. Pfenning, and B. Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, pages 1–57, 2013. Special Issue on Behavioural Types.

I. Cervesato and A. Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10): 1044–1077, 2009.

T. Chen, L. Bocchi, P. Deniélou, K. Honda, and N. Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *6th International Symposium on Trustworthy Global Computing (TGC 2011)*, 2011.

H. DeYoung, L. Caires, F. Pfenning, and B. Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In *21st Conference on Computer Science Logic (CSL 2012)*, 2012.

C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct Blame for Contracts: No More Scapegoating. In *38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, 2011.

C. Dimoulas, S. T. Hochstadt, and M. Felleisen. Complete Monitors for Behavioral Contracts. In *21st European Conference on Programming Languages and Systems (ESOP 2012)*, 2012.

T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In *16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*, 2011.

L. Fennell and P. Thiemann. The blame theorem for a linear lambda calculus with type dynamic. In *13th International Symposium on Trends in Functional Programming (TFP 2012)*, 2012.

R. B. Findler and M. Felleisen. Contracts for Higher-order Functions. *SIGPLAN Not.*, 37(9):48–59, 2002.

D. Griffith and E. L. Gunter. Liquid pi: Inferrable dependent session types. In *5th NASA Formal Methods Symposium (NSM 2013)*, 2013.

K. Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory (CONCUR 1993)*, 1993.

K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming Languages and Systems (ESOP 1998)*, 1998.

L. Jia, H. Gommerstadt, and F. Pfenning. Monitors and blame assignment for higher-order session types. Technical Report CMU-CyLab-15-004, CyLab, Carnegie Mellon University, Nov. 2015.

M. Keil and P. Thiemann. Blame assignment for higher-order contracts with intersection and union. In *20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*, 2015.

O. Laurent. Polarized proof-nets: Proof-nets for LC. In *4th International Conference on Typed Lambda Calculi and Applications (TLCA 1999)*, 1999.

J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation*, 239:254–302, 2014.

F. Pfenning. Substructural operational semantics and linear destination-passing style. In *2nd Asian Symposium on Programming Languages and Systems (APLAS 2004)*, 2004. Abstract of invited talk.

F. Pfenning and D. Griffith. Polarized substructural session types. In *18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, 2015. Invited talk.

F. Pfenning, L. Caires, and B. Toninho. Proof-carrying code in a session-typed process calculus. In *1st International Conference on Certified Programs and Proofs (CPP 2011)*, 2011.

C. Scholliers, Éric Tanter, and W. D. Meuter. Computational contracts. *Science of Computer Programming*, 98, Part 3:360 – 375, 2015. ISSN 0167-6423. Special Issue on Advances in Dynamic Languages.

J. Siek, P. Thiemann, and P. Wadler. Blame and Coercion: Together Again for the First Time. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*, 2015.

R. J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, Nov. 2012. Available as Technical Report CMU-CS-12-142.

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *17th International Conference on Functional Programming (ICFP 2011)*, 2011.

P. Thiemann. Session Types with Gradual Typing. In *9th International Symposium on Trustworthy Global Computing (TGC 2014)*. 2014.

B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *22nd European Symposium on Programming (ESOP 2013)*, 2013.

P. Wadler. Propositions as sessions. In *17th International Conference on Functional Programming (ICFP 2012)*, 2012.

P. Wadler. A Complement to Blame. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, 2015.

P. Wadler and R. B. Findler. Well-Typed Programs Can't Be Blamed. In *18th European Symposium on Programming Languages and Systems (ESOP 2009)*, 2009.