On the Monitorability of Session Types, in Theory and Practice

Christian Bartolo Burlò
Gran Sasso Science Institute, L'Aquila, Italy

Adrian Francalanza
Department of Computer Science, University of Malta, Msida, Malta

Alceste Scalas
DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Abstract

Software components are expected to communicate according to predetermined protocols and APIs. Numerous methods have been proposed to check the correctness of communicating systems against such protocols/APIs. *Session types* are one such method, used both for static type-checking as well as for run-time monitoring. This work takes a fresh look at the run-time verification of communicating systems using session types, in theory and in practice. On the theoretical side, we develop a formal model of session-monitored processes. We then use this model to formulate and prove new results on the *monitorability* of session types, defined in terms of *soundness* (*i.e.,* whether monitors only flag ill-typed processes) and *completeness* (*i.e.,* whether all ill-typed processes can be flagged by a monitor). On the practical side, we show that our monitoring theory is indeed *realisable*: we instantiate our formal model as a Scala toolkit (called `STMonitor`) for the automatic generation of session monitors. These executable monitors can be used as proxies to instrument communication across black-box processes written in any programming language. Finally, we evaluate the viability of our approach through a series of benchmarks.

2012 ACM Subject Classification Software and its engineering → Development frameworks and environments; Software and its engineering → Software verification and validation; Theory of computation → Concurrency

Keywords and phrases Session types, monitorability, monitor correctness, Scala

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.20

Related Version *Full Version*: `https://arxiv.org/abs/2105.06291`

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.2`

Funding This work has been partly supported by: the project MoVeMnt (No: 217987-051) under the Icelandic Research Fund; the BehAPI project funded by the EU H2020 RISE under the Marie Skłodowska-Curie action (No: 778233); the EU Horizon 2020 project 830929 *CyberSec4Europe*; the Danish Industriens Fonds Cyberprogram 2020-0489 *Security-by-Design in Digital Denmark.*

1 Introduction

Communication protocols and Application Programming Interfaces (APIs) [18] govern the interactions between concurrent and distributed software components by exposing the functionality of a component for others to use. Although the order of messages exchanged and methods invoked is crucial for correct API usage, this information is either outright omitted, or stated informally via textual descriptions [62, 61]. At best, protocols and temporal API usage are described semi-formally as message sequence charts [51]. This state of affairs is conducive to conflicting interactions, which may manifest themselves as run-time errors, deadlocks and livelocks. *Behavioural types* [11] provide a methodology

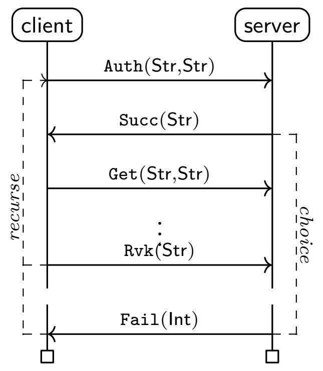


Figure 1 Authentication Protocol.

to address these shortcomings, by elevating protocols and flat API descriptions to *formal behavioural specifications* with explicit *sequences* and *choices* of operations. A prevalent form of behavioural types are *session types* [36, 37] which can ensure correct interactions that are free from communication errors, deadlocks and livelocks.

▶ **Example 1.** Consider a server that exposes the API calls `Auth` (authenticate), `Get` and `Rvk` (revoke). The intended use of this API is to invoke `Auth` followed with `Get` and finally `Rvk`, as depicted in Fig. 1. If authentication is successful, `Auth` returns a token that can be used for exclusive access to a resource with the service `Get`. After its use, the token should be revoked with the service `Rvk` to allow other parties to access the resource. For security reasons, the server is expected to only reply `Get` requests after it services an `Auth` request. However, if the order of invocation is not respected, a client may send a `Get` request before an `Auth` request. The resulting components' interaction will be incorrect, causing an error or deadlock. Even worse, the server may accept the `Get` request and let an unauthenticated client access sensitive information. The protocol from the viewpoint of the client can be described as the session type:

$$S \quad = \quad !\texttt{Auth}\,.\,\&\Big\{\ ?\texttt{Succ}\,.\,!\texttt{Get}\ldots!\texttt{Rvk}\,.\,S, \quad ?\texttt{Fail}\,.\,S\ \Big\}$$

Type S states that the client is expected to first invoke (!) the service `Auth` and then branch (&) according to the response received (?). If it receives `Success`, the client can invoke `Get` and eventually `Rvk` before restarting the protocol (S). Otherwise, if it receives `Fail`, the client may start following the type S from the beginning and retry authentication. ⌟

Run-time monitoring of session types: promise and challenges. In behavioural type frameworks (including session types), the conformance between the component under scrutiny and a desired protocol is commonly checked *statically*, via a type system that is tailored for the language used to develop the component. This avoids runtime overhead and allows for early error detection. However, there are cases where a (full) static analysis is not possible. For instance, within a distributed or collaborative system, not all system components are necessarily accessible for static analysis (*e.g.,* due to source obfuscation). Components may also be implemented using different programming languages, making it infeasible to develop bespoke type-checkers for every programming language used in development. In these cases, post-deployment techniques such as Runtime Verification (RV) [29, 13] can be used where protocol conformance is carried out *dynamically* via *monitors* [21, 50, 24, 42, 49, 17, 34]. Runtime monitoring of behavioural types comes with a set of challenges.

The realisability of effective monitoring: Restrictions such as inaccessible code and license agreements (regulating code modifications), may restrict the ways in which software components can be instrumented, thus hindering a monitor's capabilities for observation and intervention. Moreover, the runtime overhead induced by monitors should be kept within acceptable levels.

Monitor Correctness: Intuitively, a "correct" monitor for a session type S should carry out detections that correspond to the protocol represented by S. The recent results on *monitorability* help us unpack this intuition of "correctness" in terms of *soundness* and *completeness*: the monitor should not unnecessarily flag well-behaving code (*detection soundness* [30, 3]), while providing guarantees for recognising misbehaving components (*detection completeness* [4, 6]).

The aforementioned challenges are not independent of one another, and an adequate solution often needs to take both aspects into consideration. On the one hand, monitor correctness may require computations that increase runtime overheads; on the other hand, there are inherent limits to what can be detected at runtime (*i.e.,* the monitorability problem [13]) – and moreover, practical implementation concerns may restrict monitoring capabilities even further (*e.g.,* due to the need for low overheads). To our knowledge, the above aspects have not been fully investigated together for session types, in *one unified study*:

- there is no systematic examination for the *monitorability* of session types, determining the limits of runtime monitoring when verifying session-type conformance;
- no previous work tackles the design of a session monitoring system that is practically *realisable*, while also backed by formal detection soundness *and* completeness guarantees.

Contributions. We present the first formal analysis of the *monitorability of session types*, and use it to guide the design and implementation of a practical framework (written in Scala) for the run-time monitoring of concurrent and distributed applications. We focus on communication protocols that can be formalised as *(binary) session types* [36, 37] with two interacting parties (*e.g.,* a client and a server). Crucially, we tackle scenarios where at least one of the parties is a "black-box" process that may not be statically verified. After formalising a streamlined process calculus with session types (§ 2), we present our contributions:

1. We develop a formal model detailing how processes can be instrumented with monitors, to observe their interactions and flag violations on the offending party (§ 3). We then design an automated synthesis procedure from session types to monitors (in this operational model) to study the monitorability of session types (§ 3.4);
2. We carry out the first study on the *monitorability* of session types, formally linking their static and run-time verification (§ 4). We prove that our synthesised monitors are *detection-sound*, *i.e.,* components flagged by a monitor for session type S are indeed ill-typed for S (Theorem 15). We also prove a *weak detection-completeness* result (Theorem 19) showing to what degree can our synthesised monitors detect ill-typed components. Importantly, we show that these limits are not specific to our synthesis procedure by proving an impossibility result: under our "black-box" monitoring model, session monitoring cannot be both sound and complete (Theorem 21). The latter results are new to the area of behavioural types;
3. We show the *realisability* of our model, by implementing a toolkit (called `STMonitor`) that synthesises session monitors as executable Scala programs (§ 5). We provide `STMonitor` as companion artifact of this paper. We also provide evaluation benchmarks showing that our generated Scala monitors induce limited overheads, hence their usability in practice appears promising (§ 6).

   Proofs and additional details are available in the extended version of this paper [19].

Syntax

$$\begin{array}{rrcl} \text{Predicates} & A & ::= & \mathsf{tt} \mid \mathsf{ff} \mid v_1 == v_2 \mid v_1 >= v_2 \mid A_1 \mathbin{\&\&} A_2 \mid !A \mid \ldots \\ \text{Processes} & P, Q & ::= & \triangleleft \mathtt{l}(a).P \mid \triangleright \left\{ \mathtt{l}_i(x_i).P_i \right\}_{i \in I} \mid \mu_X.P \mid X \mid \mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid \mathbf{0} \end{array}$$

Semantics

$$[\text{pRec}]\ \frac{}{\mu_X.P \xrightarrow{\tau} P[^{\mu_X.P}/_X]} \qquad [\text{pSnd}]\ \frac{}{\triangleleft \mathtt{l}(v).P \xrightarrow{\triangleleft \mathtt{l}(v)} P}$$

$$[\text{pRcv}]\ \frac{}{\triangleright \left\{ \mathtt{l}_i(x_i).P_i \right\}_{i \in I} \xrightarrow{\triangleright \mathtt{l}_j(v)} P_j[^v/_{x_j}]}\ j \in I$$

$$[\text{pTru}]\ \frac{A \Downarrow \mathsf{tt}}{\mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} P} \qquad [\text{pFls}]\ \frac{A \Downarrow \mathsf{ff}}{\mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} Q}$$

Figure 2 Process Calculus Syntax and Semantics.

2 Process Calculus and Session Types

This section introduces the formalism at the basis of our work: a streamlined process calculus (§ 2.1) with standard session types (§ 2.2) and typing system (§ 2.3).

2.1 Process Calculus

Syntax. We adopt a streamlined process calculus that models a sequential process interacting on a single communication channel, similar to [33, 32, 60]. Our process calculus is defined in Figure 2. The syntax assumes separate denumerable sets of **values** $v, u, w \in \text{VAL}$ (including tuples), value **variables** $x, y, z \in \text{VAR}$ and **process variables** $X, Y \in \text{PVAR}$. We use a, b to range over the set $\text{VAL} \cup \text{VAR}$. The syntax also assumes a set of **predicates** A (used in conditionals). A process may communicate by sending or receiving **messages** of the form $\mathtt{l}(v)$, where $\mathtt{l}$ is a **label**, and v is the **payload value**. To this end, a process may perform **outputs** $\triangleleft \mathtt{l}(a).P$ (*i.e.,* send message $\mathtt{l}(v)$ and continue as P), or **inputs** $\triangleright \left\{ \mathtt{l}_i(x_i).P_i \right\}_{i \in I}$ (*i.e.,* receive a message with label $\mathtt{l}_i$ for any $i \in I$, and continue as P_i, with x_i replaced by the message payload). Loops are supported by the **recursion** construct $\mu_X.P$, and the **process variable** X. The process $\mathbf{0}$ represents a **terminated** process. The calculus also includes a standard **conditional** construct $\mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q$. We assume that all recursive processes are **guarded**, *i.e.,* process variables can only occur under an input or output prefix. The calculus has two **binders**: the input construct $\triangleright \left\{ \mathtt{l}_i(x_i).P_i \right\}_{i \in I}$ binds the free occurrences of the (value) variables x_i in the continuation process P_i, whereas the recursion construct $\mu_X.P$ binds the process variable X in the continuation process P.

Semantics. The dynamic behaviour of a process is described by the transition rules in Fig. 2. The rules take the form $P \xrightarrow{\mu} P'$, where the transition **action** μ can be either an **output action** $\triangleleft \mathtt{l}(v)$, an **input action** $\triangleright \mathtt{l}(v)$, or a **silent action** τ. Rule [pRec] allows the recursive process $\mu_X.P$ to unfold. Rules [pSnd] and [pRcv] enable communication:

- by [pSnd], process $\triangleleft \mathtt{l}(v).P$ sends a message by firing action $\triangleleft \mathtt{l}(v)$ and continuing as P;
- by [pRcv], process $\triangleright \left\{ \mathtt{l}_i(x_i).P_i \right\}_{i \in I}$ can receive a message $\mathtt{l}_j(v)$ $(j \in I)$ by firing action $\triangleright \mathtt{l}_j(v)$ and continuing as P_j, with the payload value v replacing the variable x_j.

The remaining two rules [pTru] and [pFls] define the silent transitions when the predicate in the process $\mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q$ evaluates to true ($A \Downarrow \mathsf{tt}$) or false ($A \Downarrow \mathsf{ff}$), respectively. For brevity, we often omit the trailing $\mathbf{0}$ and write $\triangleright \mathtt{l}(v).P$ for singleton input choices.

Syntax

$$\begin{array}{lrcl} & \text{Base types} & \mathsf{B} & ::= \mathsf{Int} \mid \mathsf{Str} \mid \mathsf{Bool} \mid \ldots \mid (\mathsf{B},\mathsf{B}) \\ & \text{Session types} & R, S & ::= \underbrace{\oplus\left\{!\mathtt{l}_i(\mathsf{B}_i).S_i\right\}_{i\in I} \mid \&\left\{?\mathtt{l}_i(\mathsf{B}_i).S_i\right\}_{i\in I}}_{\text{with } I\neq\emptyset \text{ and } \mathtt{l}_i \text{ pairwise distinct}} \mid \mathsf{rec}\ X.S \mid X \mid \mathsf{end} \end{array}$$

Dual types

$$\overline{\&\left\{?\mathtt{l}_i(\mathsf{B}_i).S_i\right\}_{i\in I}} = \oplus\left\{!\mathtt{l}_i(\mathsf{B}_i).\overline{S_i}\right\}_{i\in I} \qquad \overline{\mathsf{end}} = \mathsf{end} \qquad \overline{X} = X$$

$$\overline{\oplus\left\{!\mathtt{l}_i(\mathsf{B}_i).S_i\right\}_{i\in I}} = \&\left\{?\mathtt{l}_i(\mathsf{B}_i).\overline{S_i}\right\}_{i\in I} \qquad \overline{\mathsf{rec}\ X.S} = \mathsf{rec}\ X.\overline{S}$$

Figure 3 Session Types Syntax, and Definition of Dual Types.

▶ **Example 2** (Process syntax and semantics). Recall the protocol depicted in Fig. 1. A corresponding client process for this protocol is defined as P_{auth} below.

$$P_{auth} = \mu_X.\triangleleft\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"}).P_{res} \qquad \text{where} \quad P_{res} = \triangleright\left\{\mathtt{Succ}(tok).P_{succ}\,,\ \mathtt{Fail}(code).P_{fail}\right\}$$

From the rules in Figure 2, the process P_{auth} executes as follows:

$$P_{auth} \xrightarrow{\tau} \left(\triangleleft\,\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"}).P_{res}\right)[^{P_{auth}}/_X] \qquad \text{using [pRec]}$$

$$\xrightarrow{\triangleleft\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"})} \triangleright\left\{\begin{array}{l}\mathtt{Succ}(tok).P_{succ}[^{P_{auth}}/_X]\,, \\ \mathtt{Fail}(code).P_{fail}[^{P_{auth}}/_X]\end{array}\right\} \qquad \text{using [pSnd]}$$

The process performs a silent action τ to unfold its recursion, and then sends a message with label `Auth` and tuple `"Bob"`,`"pwd"` as payload. If the authentication is successful, the process receives the message `Succ` including a token *tok* and proceeds according to P_{succ} (omitted):

$$\xrightarrow{\triangleright\mathtt{Succ}(321)} P_{succ}[^{P_{auth}}/_X][^{321}/_{tok}] \qquad \text{using [pRcv]}$$

Otherwise, if the authentication is unsuccessful, the process receives the message `Fail` including an error *code* from the server and proceeds according to P_{fail}. ⌟

2.2 Binary Session Types

Session types describe the structure of interaction among processes. They enable the verification of communicating systems against a stipulated communication protocol. Figure 3 formalises binary session types. We assume a set of standard base types B which includes tuples. The **selection type** (or **internal choice**) $\oplus\left\{!\mathtt{l}_i(\mathsf{B}_i).S_i\right\}_{i\in I}$ requires a component to send a message $\mathtt{l}_i(v)$ where the value v has base type B_i, for some $i \in I$. The **branching type** (or **external choice**) $\&\left\{?\mathtt{l}_i(\mathsf{B}_i).S_i\right\}_{i\in I}$ requires a component to receive a message of the form $\mathtt{l}_i(v)$, where the value v (*i.e.,* the message payload) is of the corresponding base type B_i for any $i \in I$. The **recursive** session type $\mathsf{rec}\ X.S$ binds the recursion variable X in S (we assume guarded recursion), while end describes a **terminated** session. For brevity, we often omit $\oplus$ and $\&$ for singleton choices, as well as trailing ends.

A process implementing a session type S can correctly interact with a process implementing the **dual type** of S, denoted as $\overline{S}$ (defined in Fig. 3). Intuitively, the dual type of a selection is a branching type with the same choices. Hence, every possible output from one component matches an input by the other component, and *vice versa.* Duality guarantees that the interaction between typed components is *safe* (*i.e.,* only expected messages are communicated) and *deadlock-free* (*i.e.,* the session terminates only if both components reach their end).

▶ **Example 3.** The session type S_{auth} below formalises the first part of the protocol that the *client* in Fig. 1 is expected to follow (*i.e.,* the type S in Example 1).

$$S_{auth} = \mathsf{rec}\ Y.!\mathtt{Auth}(\mathsf{Str},\mathsf{Str}).\&\left\{?\mathtt{Succ}(\mathsf{Str}).S_{succ}\,,\ ?\mathtt{Fail}(\mathsf{Int}).Y\right\}$$

Identifier Typing

$$[\textsc{tVar}]\,\frac{\Gamma(x) = \mathsf{B}}{\Gamma \vdash x : \mathsf{B}} \qquad [\textsc{tVal}]\,\frac{v \in \mathsf{B}}{\Gamma \vdash v : \mathsf{B}}$$

Process Typing

$$[\textsc{tBra}]\,\frac{\forall i \in I \qquad \Theta \cdot \Gamma, x_i : \mathsf{B}_i \vdash P_i : S_i}{\Theta \cdot \Gamma \vdash \triangleright\{\mathtt{l}_i(x_i).P_i\}_{i \in I \cup J} : \&\{?\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i \in I}} \qquad [\textsc{tRec}]\,\frac{\Theta, X : S \cdot \Gamma \vdash P : S}{\Theta \cdot \Gamma \vdash \mu_X.P : S}$$

$$[\textsc{tSel}]\,\frac{\exists i \in I \qquad \mathtt{l} = \mathtt{l}_i \qquad \Gamma \vdash a : \mathsf{B}_i \qquad \Theta \cdot \Gamma \vdash P : S_i}{\Theta \cdot \Gamma \vdash \triangleleft\mathtt{l}(a).P : \oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i \in I}} \qquad [\textsc{tPVar}]\,\frac{\Theta(X) = S}{\Theta \cdot \Gamma \vdash X : S}$$

$$[\textsc{tIf}]\,\frac{\Gamma \vdash A : \mathsf{Bool} \qquad \Theta \cdot \Gamma \vdash P : S \qquad \Theta \cdot \Gamma \vdash Q : S}{\Theta \cdot \Gamma \vdash \mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q : S} \qquad [\textsc{tNil}]\,\frac{}{\Theta \cdot \Gamma \vdash \mathbf{0} : \mathsf{end}}$$

Figure 4 Session Typing Rules.

The *server* should follow $\overline{S_{auth}} = \mathsf{rec}\ Y.?\mathtt{Auth}(\mathsf{Str}, \mathsf{Str}). \oplus \{!\mathtt{Succ}(\mathsf{Str}).\overline{S_{succ}}\,,\ !\mathtt{Fail}(\mathsf{Int}).Y\}$, its dual. According to S_{auth}, the *client* initiates interaction by sending a message with label `Auth`, carrying a tuple of strings (username and password) as payload. The *server* should then reply with either `Success` (carrying a string), or `Failure` (with an integer error code). In case of `Success`, the *client* continues along S_{succ}. In case of `Failure`, the session loops. ⌟

2.3 Session Typing System

Our session typing system (in Fig. 4) is standard. It uses two **typing environments** Θ and Γ, where Θ is a partial mapping from process variables to session types, while Γ is a partial mapping from value variables to base types. We represent them syntactically as:

$$\Theta \ ::= \ \emptyset \mid \Theta, X : S \qquad\qquad \Gamma \ ::= \ \emptyset \mid \Gamma, x : \mathsf{B}$$

The type system is *equi-recursive* [53]: when comparing two types, we consider a recursive type $\mathsf{rec}\ X.S$ to be equivalent to its unfolding $S[^{\mathsf{rec}\ X.S}/_{X}]$ (*i.e.*, interchangeable in all contexts).

The typing judgement for values and variables is $\Gamma \vdash a : \mathsf{B}$, defined by rules [tVar] and [tVal]. The process typing judgement, $\Theta \cdot \Gamma \vdash P : S$, states that process P communicates according to session type S, given the typing assumptions in Θ and Γ. In the *branching rule* [tBra], an input process has a branching type $\&\{?\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i \in I}$ if all the possible branches in the type are present as choices in the process, with matching labels. Hence, the process must have the form $\triangleright\{\mathtt{l}_i(x_i).P_i\}_{i \in I \cup J}$ (notice that if $J \neq \emptyset$, the process has more input branches than the type). Moreover, for each matching branch, each continuation process P_i (for $i \in I$) must be typed with the corresponding continuation type S_i, assuming that the received message payload x_i has the expected type B_i. The *selection rule* [tSel] states that $\triangleleft\mathtt{l}(a).P$ follows a selection type of the form $\oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i \in I}$ if there exists a possible choice in the type that matches the message $\mathtt{l}(a)$. To match, the labels must be identical, and the type of the payload a must be of the type B_i stated in the session type, and the continuation process P must be of the continuation type S_i. The remaining rules are fairly standard.

▶ Remark 4. Although we do not fix the boolean predicates A, we assume that:

1. boolean predicates can be type-checked with standard rules;
2. base types B come with a predicate $\mathsf{isB}(v)$ that returns tt if v is of type B, and ff otherwise (akin to `instanceof` in Java.) ⌟

▶ **Example 5.** Recall the process P_{auth} (Example 2) and the session type S_{auth} (Example 3):

$$P_{auth} \;=\; \mu_X.\triangleleft\,\texttt{Auth("Bob","pwd")}.P_{res} \qquad S_{auth} \;=\; \mathsf{rec}\;Y.!\texttt{Auth}(\mathsf{Str},\mathsf{Str}).S_{res}$$

One can show that P_{auth} type-checks with S_{auth}, *i.e.,* $\emptyset \cdot \emptyset \vdash P_{auth} : S_{auth}$. ⌟

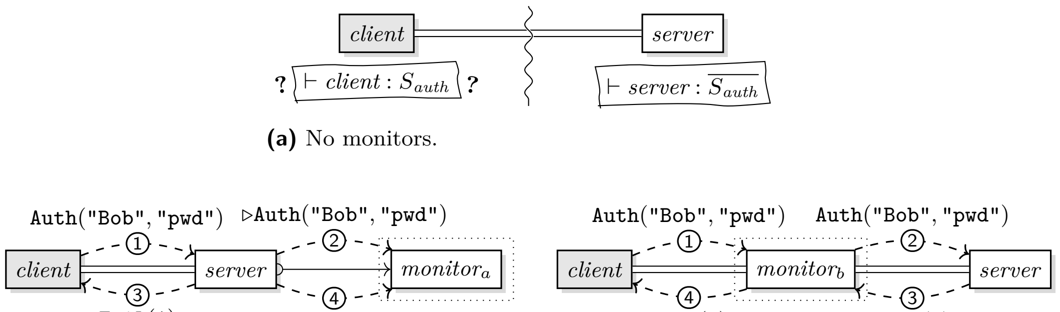
3 A Formal Model for Monitoring Sessions

We now formalise an operational setup that enables us to verify the (binary) session types of § 2 at runtime. Our runtime analysis is conducted by *uni-verdict* rejection monitors, whose purpose is to flag any session type violations detected (*i.e., violation monitors* [30, 3]).

3.1 Monitor and Instrumentation Design

We now illustrate the design decisions behind our formal monitoring framework. To this end, we use as a reference the client-server system outlined in Example 1. Consider, in particular, the scenario depicted in Fig. 5a, where a *client* is expected to interact with a *server* following the prescribed protocol S_{auth}; the *server* is *trusted* and guaranteed to adhere to the dual type $\overline{S_{auth}}$ (*e.g.,* because it has been statically typechecked against $\overline{S_{auth}}$ using the type system in § 2.3) – but we have limited control over the *client*, which might be untyped, hence its interactions are potentially unsafe.

Our setup should place no assumptions on the *client*, largely treating it as a black box. In fact, we target scenarios where the *client* source is inaccessible, possibly remote, interacting with the server via a generic channel of communication (*e.g.,* TCP sockets or HTTP addresses). This precludes the possibility of weaving the monitor within the *client* component. To achieve a model that can handle these requirements, we restrict ourselves to *outline monitors* [13, 7], which are decoupled from the process-under-scrutiny as concurrent units of code that can be more readily deployed over a black-box component; outline monitors are also easier to verify for correctness via compositional techniques [23, 26, 27, 14, 31, 28].



(a) No monitors.

(b) Server side instrumentation.

(c) Channel instrumentation.

Figure 5 Design choices for instrumentation.

Our model focusses on the communication occurring on the channel between the *client* and the *server* – and we assume such communication to be *synchronous* and *reliable*. Outline monitors can typically only analyse the externally *observable* actions of a monitored component. In our case, monitored processes follow the semantics of Fig. 2, hence the only observable actions are send ($\triangleleft\texttt{l}(v)$) and receive ($\triangleright\texttt{l}(v)$); τ-moves are unobservable.

We consider two potential instrumentation setup designs for an outline approach. In the setup in Fig. 5b, the *server* is instrumented with a *sequence-recogniser monitor* [57, 43] ($monitor_a$). The *server* is required to notify $monitor_a$ about every send and receive action it performs – this can be achieved via listeners added through mechanisms such as class-loaders, agents and VM-level tracers. For $monitor_a$, every receive action the *server* performs indicates a send action by the *client* and vice-versa (*i.e.,* every send indicates a receive). In Fig. 5b, The *client* sends the message `Auth("Bob","pwd")` to the *server* ①. Once received, ② the *server* notifies $monitor_a$ with the message contents and the direction of the message ($\triangleright$). For $monitor_a$ this indicates that the *client* sent the particular message. After the *server* replies with the message `Fail(1)` ③, it notifies $monitor_a$ with the message contents and the direction ($\triangleleft$) ④, indicating that the *client* received the message.

In the alternative setup depicted in Fig. 5c, the monitor ($monitor_b$) is instrumented on the communication channel and acts as a *proxy* (or a *partial-identity monitor* [34]) between the two components. Any communicated messages must pass through $monitor_b$ in order for it to analyse them. In the execution of Fig. 5c the *client* sends the message `Auth("Bob","pwd")` to $monitor_b$ ①. The monitor checks that its contents conform with the protocol before proceeding to forward the message to the *server* ②. The *server* replies by sending the message `Fail(1)` to $monitor_b$ ③, which forwards it straight to the *client* ④.

On the one hand, the monitor in Fig. 5b is completely passive: it performs analysis in response to the events received. On the other hand, the monitor in Fig. 5c is also responsible for *forwarding* messages between the *client* and the *server*. Thus, the communication between the two components in Fig. 5c *relies* on $monitor_b$: should the monitor crash or terminate abruptly, the *client* and the *server* will stop interacting. Moreover, the setup in Fig. 5c introduces additional delays when every communicated message passes through $monitor_b$; these are avoided in Fig. 5b. The main drawback of the setup in Fig. 5b is that the *server* is directly exposed to an untrusted client, with additional responsibility of reporting events. In contrast, the instrumentation in Fig. 5c provides a layer of protection to the *server* from potentially malicious interactions: if the *client* sends a message that violates the protocol, $monitor_b$ is able to flag the message without forwarding it to the *server*. Moreover, the setup in Fig. 5c provides more flexibility for reasoning on the run-time monitoring of systems where *both* the *client* and the *server* are black boxes. This work opts for the setup in Fig. 5c.

3.2 A Monitor Calculus

Fig. 6 describes the structure and behaviour of a partial-identity monitor operating as in Fig. 5c. Monitors are similar to the processes defined in Fig. 2, with a few key additions. Since monitors need to interact with the environment, they also include the constructs $\blacktriangle\mathtt{l}(a).M$ and $\blacktriangledown\{\mathtt{l}_i(x_i : B_i).M_i\}_{i\in I}$, and rules [mOut] and [mIn]: they are analogous to the process output and input constructs, where interaction takes place between the environment and the monitor instead. We use the terms **internal** and **external** to differentiate between actions involving the monitored process and the environment, respectively.

As shown in Figure 7, monitors can reach two kinds of **rejection verdicts**, namely $\mathtt{no}_P$ and $\mathtt{no}_E$; the P and E tags distinguish between violations committed by the *monitored process* (P) and the *environment* (E). The rules [mIV] and [mEV] specify how the monitor reaches a verdict. Rule [mIV] represents the case when the monitor receives a violating message $\mathtt{l}(v)$ and consequently reaches the verdict $\mathtt{no}_P$; the message is deemed violating since its label is not among those that the monitor expects to receive. Symmetrically, in rule [mEV] the monitor reaches $\mathtt{no}_E$ when it receives a violating message from the external environment. The following example outlines the scenarios in which monitors reach a verdict.

Syntax

$$\text{Monitor}\quad M, N \;\coloneqq\; \triangleleft\mathtt{l}(a).M \mid \triangleright\big\{\mathtt{l}_i(x_i).M_i\big\}_{i\in I} \mid \blacktriangle\mathtt{l}(a).M \mid \blacktriangledown\big\{\mathtt{l}_i(x_i).M_i\big\}_{i\in I} \mid \mu_X.M \mid X \mid \mathsf{if}\ A\ \mathsf{then}\ M\ \mathsf{else}\ N \mid \mathbf{0} \mid \mathsf{no}_P \mid \mathsf{no}_E$$

Semantics

$$[\text{MSND}]\ \frac{}{\triangleleft\mathtt{l}(v).M \xrightarrow{\triangleleft\mathtt{l}(v)} M} \qquad [\text{MOUT}]\ \frac{}{\blacktriangle\mathtt{l}(v).M \xrightarrow{\blacktriangle\mathtt{l}(v)} M} \qquad [\text{MREC}]\ \frac{}{\mu_X.M \xrightarrow{\tau} M[^{\mu_X.M}/_X]}$$

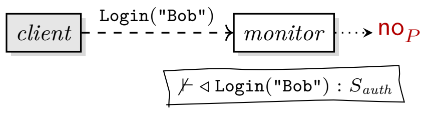
$$[\text{MRCV}]\ \frac{}{\triangleright\big\{\mathtt{l}_i(x_i).M_i\big\}_{i\in I} \xrightarrow{\triangleright\mathtt{l}_j(v)} M_j[^v/_{x_j}]}\ j\in I \qquad [\text{MIN}]\ \frac{}{\blacktriangledown\big\{\mathtt{l}_i(x_i).M_i\big\}_{i\in I} \xrightarrow{\blacktriangledown\mathtt{l}_j(v_j)} M_j[^{v_j}/_{x_j}]}\ j\in I$$

$$[\text{MTRU}]\ \frac{A \Downarrow \mathsf{tt}}{\mathsf{if}\ A\ \mathsf{then}\ M\ \mathsf{else}\ N \xrightarrow{\tau} M} \qquad [\text{MFLS}]\ \frac{A \Downarrow \mathsf{ff}}{\mathsf{if}\ A\ \mathsf{then}\ M\ \mathsf{else}\ N \xrightarrow{\tau} N}$$

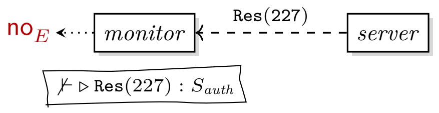
Violation Semantics

$$[\text{MIV}]\ \frac{}{\triangleright\big\{\mathtt{l}_i(x_i).M_i\big\}_{i\in I} \xrightarrow{\triangleright\mathtt{l}(v)} \mathsf{no}_P}\ \forall i\in I:\mathtt{l}\neq\mathtt{l}_i \qquad [\text{MEV}]\ \frac{}{\blacktriangledown\big\{\mathtt{l}_i(x_i).M_i\big\}_{i\in I} \xrightarrow{\blacktriangledown\mathtt{l}(v)} \mathsf{no}_E}\ \forall i\in I:\mathtt{l}\neq\mathtt{l}_i$$

Figure 6 Monitor Syntax and Semantics.



(a) Internal violation.



(b) External violation.

Figure 7 Monitor violations.

▶ **Example 6.** Fig. 7 depicts a monitor verifying the conformity of a *client* with the session type S_{auth} (from Example 3). In Fig. 7a, the *client* sends the message `Login("Bob")`. Since the type S_{auth} states that the *client* should send a message with label `Auth`, the monitor reaches the verdict no_P by rule [MIV]. In Fig. 7b, the monitor receives `Res(227)` from the environment (which represents a buggy server). In this case the monitor reaches the verdict no_E (by rule [MEV]) since the message does not conform with S_{auth} which states that the *client* should receive either `Succ` or `Fail`. ⌟

▶ Remark 7. According to Fig. 6, our monitors can reach a verdict explicitly in their syntax (by having $\mathsf{no}_P/\mathsf{no}_E$ in their body), or by just transitioning to a verdict via rules [MIV] or [MEV]. We will make use both methods for our synthesised monitors (see § 3.4). ⌟

3.3 Composite Monitored System

The rules in Fig. 8 formalise the behaviour of the monitor when composed with the process to monitor, while also interacting with an environment (*i.e.,* another process). This setup is depicted in Fig. 9. We refer to a process P instrumented with a monitor M as a **composite (monitored) system**, denoted as $\langle P; M\rangle$. The rules [IRCV] and [ISND] model the interaction *within* the composite system, (*i.e.,* between the monitored process P and the monitor M in Fig. 9). Note that the interaction between the two is *synchronous*: the monitor (resp. process) can only send a message when the process (resp. monitor) can receive the same message. If P sends a message (by [ISND]) that violates the monitor's inputs, M is able to flag the violation by rule [MIV]. The rules [IOUT] and [IIN] model the interaction between the composite system and the environment. As shown in Fig. 9, the monitor is the entity that interacts with the environment (represented as a process Q). Accordingly, the monitor can flag a message sent by the environment if the message violates the monitor's expected

$$[\textsc{iSnd}]\ \frac{P \xrightarrow{\triangleleft \mathtt{l}(v)} P' \qquad M \xrightarrow{\triangleright \mathtt{l}(v)} M'}{\langle P; M\rangle \xrightarrow{\tau} \langle P'; M'\rangle} \qquad [\textsc{iRcv}]\ \frac{P \xrightarrow{\triangleright \mathtt{l}(v)} P' \qquad M \xrightarrow{\triangleleft \mathtt{l}(v)} M'}{\langle P; M\rangle \xrightarrow{\tau} \langle P'; M'\rangle}$$

$$[\textsc{iOut}]\ \frac{M \xrightarrow{\blacktriangle \mathtt{l}(v)} M'}{\langle P; M\rangle \xrightarrow{\blacktriangle \mathtt{l}(v)} \langle P; M'\rangle} \qquad [\textsc{iIn}]\ \frac{M \xrightarrow{\blacktriangledown \mathtt{l}(v)} M'}{\langle P; M\rangle \xrightarrow{\blacktriangledown \mathtt{l}(v)} \langle P; M'\rangle}$$

$$[\textsc{iProc}]\ \frac{P \xrightarrow{\tau} P'}{\langle P; M\rangle \xrightarrow{\tau} \langle P'; M\rangle} \qquad [\textsc{iMon}]\ \frac{M \xrightarrow{\tau} M'}{\langle P; M\rangle \xrightarrow{\tau} \langle P; M'\rangle}$$

Figure 8 Composite monitored system semantics.

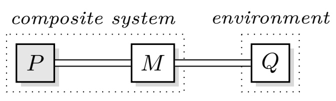


Figure 9 The composite system interacting with the environment.

inputs, by rule [mEv]. The rules [iProc] and [iMon] allow the monitored process and the monitor respectively to perform actions independent of each other (*e.g.*, to recurse or branch internally).

Our partial identity monitors halt upon reaching a verdict, in contrast to instrumented sequence recognisers that operationally continue to process events without changing their (irrevocable) verdict [26, 28]. As a result, our monitors also halt any interactions between the composite system and the environment. Because of this, monitor correctness is of paramount importance. The following example outlines the impact of a poorly constructed monitor.

▶ **Example 8.** Recall process $P_{auth} = \mu_X . \triangleleft \mathtt{Auth}(\texttt{"Bob"}, \texttt{"pwd"}).P_{res}$ (Example 2), which adheres to the session type $S_{auth} = \mathsf{rec}\ Y.!\mathtt{Auth}(uname : \mathsf{Str}, pwd : \mathsf{Str}).S_{res}$ (Example 5). A monitor corresponding to S_{auth} should *receive* from P_{auth}, analyse the message, and forward it to the environment. The following (erroneous) monitor might seem to monitor S_{auth}:

$$M_{bad} \;=\; \triangleright\mathtt{Login}(uname).\blacktriangle\mathtt{Login}(uname).N_{bad}$$

If process P_{auth} is instrumented with monitor M_{bad}, we observe the following behaviour:

$$\langle P_{auth}; M_{bad}\rangle \xrightarrow{\tau} \langle \triangleleft\mathtt{Auth}(\texttt{"Bob"}, \texttt{"pwd"}).P_{res}[^{P_{auth}}/_X]; M_{bad}\rangle \xrightarrow{\tau} \langle P_{res}[^{P_{auth}}/_X]; \mathsf{no}_P\rangle$$

After P_{auth} unfolds, it sends the message $\mathtt{Auth}(\texttt{"Bob"}, \texttt{"pwd"})$ to the monitor as per S_{auth}. However, M_{bad} can only receive messages with label $\mathtt{Login}$, hence it transitions to no_P. ⌟

3.4 Monitor Synthesis

Def. 9 presents a synthesis procedure from session types (Fig. 3) to monitors (Fig. 6). The monitors generated are meant to act as a proxy between the monitored process and the environment process, as outlined in Fig. 5c. There are various practical advantages in having an automated synthesis function: it is less error prone, expedites development and improves the maintainability of the verification framework.

▶ **Definition 9.** *The monitor synthesis function* $[\![-]\!] : S \mapsto M$ *takes as input a session type* S *and returns a monitor* M*. It is defined inductively, on the structure of the session type* S*:*

$$[\![\oplus\{!\mathtt{l}_i(B_i).S_i\}_{i\in I}]\!] \triangleq \triangleright\{\mathtt{l}_i(x_i).\mathit{if}\ \mathsf{isB}_i(x_i)\ \mathit{then}\ \blacktriangle\mathtt{l}_i(x_i).[\![S_i]\!]\ \mathit{else}\ \mathsf{no}_P\}_{i\in I}$$

$$[\![\&\{?\mathtt{l}_i(B_i).S_i\}_{i\in I}]\!] \triangleq \blacktriangledown\{\mathtt{l}_i(x_i).\mathit{if}\ \mathsf{isB}_i(x_i)\ \mathit{then}\ \triangleleft\mathtt{l}_i(x_i).[\![S_i]\!]\ \mathit{else}\ \mathsf{no}_E\}_{i\in I}$$

$$[\![\mathsf{rec}\ X.S]\!] \triangleq \mu_X.[\![S]\!] \qquad\qquad [\![X]\!] \triangleq X \qquad\qquad [\![\mathsf{end}]\!] \triangleq \mathbf{0}$$ ⌟

The main cases of Def. 9 are those for the selection and branching types. In the case of $S = \oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$, the synthesised monitor first waits to receive a message from the monitored process, with one of the labels specified in the type. Once the message is received, the monitor checks whether its payload is of the correct base type B_i, *i.e.*, $\mathsf{isB}_i(x_i)$ (see Remark 4), raising no_P if it is not. If $\mathsf{isB}_i(x_i)$ is true, the monitor forwards the message towards the environment, and proceeds according to $[\![S_i]\!]$. The synthesis for $S = \&\{?\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$ is analogous, but the generated monitor receives a message from the environment, analyses it, and forwards it to the monitored process; any violations are attributed to the environment.

▶ **Example 10** (Session Monitor Synthesis). Recall the session type S_{auth} in Example 3:

$$S_{auth} \;=\; \mathsf{rec}\; Y.!\mathtt{Auth}(\mathsf{Str},\mathsf{Str}).S_{res} \quad \text{where} \quad S_{res} \;=\; \&\{?\mathtt{Succ}(\mathsf{Str}).S_{succ}, ?\mathtt{Fail}(\mathsf{Int}).Y\}$$

The synthesis for this type first generates the recursion construct μ_Y followed by the synthesis for the selection type:

$$M_{auth} \;=\; [\![S_{auth}]\!] \;=\; \begin{cases} \mu_Y.\triangleright\mathtt{Auth}(uname,pwd).\ \mathsf{if}\ \big(\mathsf{isB_{Str}}(uname)\wedge\mathsf{isB_{Str}}(pwd)\big) \\ \qquad\qquad \mathsf{then}\ \blacktriangle\mathtt{Auth}(uname,pwd).[\![S_{res}]\!]\ \mathsf{else}\ \mathsf{no}_P \end{cases}$$

Monitor M_{auth} first waits to receive a message with label `Auth` from the monitored process (via $\triangleright$), checks the types of the payload $\big(\mathsf{isB_{Str}}(uname)\wedge\mathsf{isB_{Str}}(pwd)\big)$, and proceeds to forward the message to the environment (via $\blacktriangle$), continuing as the monitor of S_{res}:

$$[\![S_{res}]\!] \;=\; \blacktriangledown\left\{\begin{array}{l} \mathtt{Succ}(tok).\mathsf{if}\ \mathsf{isB_{Str}}(tok)\ \mathsf{then}\ \triangleleft\mathtt{Succ}(tok).[\![S_{succ}]\!]\ \mathsf{else}\ \mathsf{no}_E, \\ \mathtt{Fail}(code).\mathsf{if}\ \mathsf{isB_{Int}}(code)\ \mathsf{then}\ \triangleleft\mathtt{Fail}(code).Y\ \mathsf{else}\ \mathsf{no}_E \end{array}\right\}$$

Observe that $[\![S_{res}]\!]$ inputs from the environment and outputs to the monitored process. ⌟

If process P_{auth} is instrumented with monitor M_{auth} as the composite system $\langle P_{auth}; M_{auth}\rangle$, we observe the behaviour outlined in Fig. 5c, as we show in the following example.

▶ **Example 11.** Recall P_{auth} defined in Example 2:

$$P_{auth} \;=\; \mu_X.(\triangleleft\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"})).P_{res} \quad \text{where} \quad P_{res} \;=\; \triangleright\{\mathtt{Succ}(tok).P_{succ}, \mathtt{Fail}(code).P_{fail}\}$$

When P_{auth} is instrumented with the monitor $M_{auth} = [\![S_{auth}]\!]$ we observe the behaviour:

$$\langle P_{auth}; M_{auth}\rangle \xrightarrow{\tau} \langle P'_{auth}; M_{auth}\rangle \qquad \text{where } P'_{auth} = \triangleleft\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"}).P_{res}[^{P_{auth}}/_{X}]$$

$$\langle P'_{auth}; M_{auth}\rangle \xrightarrow{\tau} \langle P'_{auth}; M'_{auth}\rangle \qquad\qquad \text{using [iMon]}$$

$$\text{where } M'_{auth} = \big(\triangleright\mathtt{Auth}(uname,pwd).\mathsf{if}\ \big(\mathsf{isB_{Str}}(uname)\wedge\mathsf{isB_{Str}}(pwd)\big)$$
$$\mathsf{then}\ \blacktriangle\mathtt{Auth}(uname,pwd).[\![S_{res}]\!]\ \mathsf{else}\ \mathsf{no}_P\big)[^{M_{auth}}/_{Y}]$$

After unfolding, using the rules [iProc] and [iMon] respectively, the monitor can receive and the process can send, and they can transition together to communicate: (see ① in Fig. 5c)

$$P'_{auth} \xrightarrow{\triangleleft\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"})} P''_{auth} \text{ where } P''_{auth} = \triangleright\{\mathtt{Succ}(tok).P_{succ}, \mathtt{Fail}(code).P_{fail}\}[^{P_{auth}}/_{X}]$$

$$M'_{auth} \xrightarrow{\triangleright\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"})} M''_{auth} \text{ where}$$

$$M''_{auth} = \mathsf{if}\ \big(\mathsf{isB_{Str}}(\texttt{"Bob"})\wedge\mathsf{isB_{Str}}(\texttt{"pwd"})\big)\ \mathsf{then}\ \blacktriangle\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"}).[\![S_{res}]\!][^{M_{auth}}/_{Y}]\ \mathsf{else}\ \mathsf{no}_P$$

$$\langle P'_{auth}; M'_{auth}\rangle \xrightarrow{\tau} \langle P''_{auth}; M''_{auth}\rangle$$

The monitor proceeds by checking the values of the payload values using the rule [iMon].

$$M''_{auth} \xrightarrow{\tau} M'''_{auth} \quad \text{where } M'''_{auth} \;=\; \blacktriangle\mathtt{Auth}(\texttt{"Bob"},\texttt{"pwd"}).[\![S_{res}]\!][^{M_{auth}}/_{Y}]$$

$$\langle P''_{auth}; M''_{auth}\rangle \xrightarrow{\tau} \langle P''_{auth}; M'''_{auth}\rangle$$

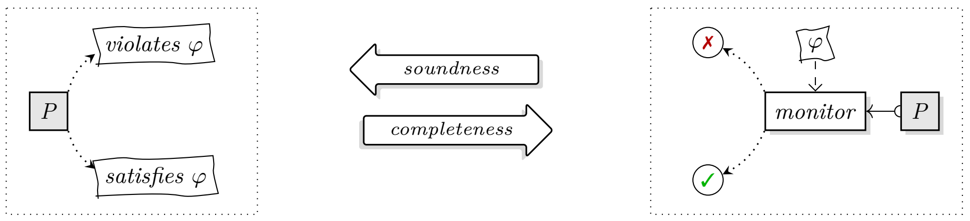


Figure 10 Monitoring soundness and completeness, from a logic-based viewpoint [30, 3, 4].

M'''_{auth} now forwards the message to the environment by rule [IOUT]: (see ② in Fig. 5c)

$$\langle P''_{auth}; M'''_{auth}\rangle \xrightarrow{\blacktriangle\texttt{Auth("Bob","pwd")}} \langle P''_{auth}; [\![S_{res}]\!][{}^{M_{auth}}/_{Y}]\rangle$$

The monitor is currently waiting to receive from the environment, since:

$$[\![S_{res}]\!][{}^{M_{auth}}/_{Y}] = \blacktriangledown \begin{Bmatrix} \mathsf{Succ}(tok).\mathsf{if}\ \mathsf{isB_{Str}}(tok)\ \mathsf{then} \triangleleft \mathtt{Succ}(tok).[\![S_{succ}]\!][{}^{M_{auth}}/_{Y}]\ \mathsf{else}\ \mathsf{no}_E \\ \mathtt{Fail}(code).\mathsf{if}\ \mathsf{isB_{Int}}(code)\ \mathsf{then} \triangleleft \mathtt{Fail}(code).M_{auth}\ \mathsf{else}\ \mathsf{no}_E \end{Bmatrix}$$

If the monitor receives the message `Succ(321)`, it forwards the message to the monitored process and proceeds according to $[\![S_{succ}]\!][{}^{M_{auth}}/_{Y}]$. If the monitor receives the message `Fail(1)` (see ③ in Fig. 5c) it forwards the message to the process P''_{auth} (see ④ in Fig. 5c):

$$\triangleleft\mathtt{Fail}(1).M_{auth} \xrightarrow{\triangleleft\mathtt{Fail}(1)} M_{auth} \qquad\qquad P''_{auth} \xrightarrow{\triangleright\mathtt{Fail}(1)} P_{fail}[{}^{P_{auth}}/_{X}][{}^{1}/_{code}]$$
$$\langle P''_{auth}; \triangleleft\mathtt{Fail}(1).M_{auth}\rangle \xrightarrow{\tau} \langle P_{fail}[{}^{P_{auth}}/_{X}][{}^{1}/_{code}]; M_{auth}\rangle$$

The composite system can now proceed with the monitor restarting as M_{auth}. ⌟

Should the monitored process send a message that violates the session type, the monitor can flag the violation upon receiving a message, as the following example shows.

▶ **Example 12.** Consider the scenario in Fig. 7a, where the *client* is the process P_{bad}:

$$P_{bad} \;=\; \triangleleft\mathtt{Login}(\texttt{"Bob"}).\triangleright\mathtt{Res}(tok:\mathsf{Str}).P_{res}$$

and recall the monitor M_{auth} (from Examples 10 and 11) obtained from the session type S_{auth}. When P_{bad} is instrumented with M_{auth}, we observe the following behaviour:

$$\langle P_{bad}; M_{auth}\rangle \xrightarrow{\tau} \langle P_{bad}; M'_{auth}\rangle \xrightarrow{\tau} \langle\triangleright\mathtt{Res}(tok:\mathsf{Str}).P_{res}; \mathsf{no}_P\rangle \qquad \text{using [IMON],[ISND]}$$

$$\text{where } M'_{auth} = \left\{ \begin{array}{l} \big(\triangleright\mathtt{Auth}(uname,pwd).\mathsf{if}\ \big(\mathsf{isB_{Str}}(uname) \wedge \mathsf{isB_{Str}}(pwd)\big) \\ \qquad\qquad \mathsf{then}\ \blacktriangle\mathtt{Auth}(uname,pwd).[\![S_{res}]\!]\mathsf{else}\ \mathsf{no}_P\big)[{}^{M_{auth}}/_{Y}] \end{array}\right.$$

i.e., M_{auth} unfolds, receives `Login("Bob")` from P_{bad}, and flag the rejection verdict no_P. ⌟

4 Formal Analysis and Results

In § 3 we argued for the importance of monitor correctness. This has also been recognised by other works that study monitoring techniques for session types [17, 42, 34]. However, these attempts all propose their own bespoke notion of monitor correctness that is often hard to relate to the others. Instead, we strive towards a more systematic approach for monitor correctness and study monitor correctness in relation to an independent characterisation of process correctness. More concretely, we assess the correctness of *session monitors* in relation to *session typing*. We draw inspirations from a recent body of work that captures this relationship in terms of *soundness* and *completeness* [30, 3], as depicted in Fig. 10. In such body of work, *monitor soundness* states that if a monitor M is monitoring a process P for a property φ, and M reaches a rejection (resp. acceptance) verdict, then such a verdict must correspond to P's violations (resp. satisfactions) of property φ. *Monitor completeness* is the dual property: if a process P violates (resp. satisfies) a property φ, then the monitor that runtime-checks P for φ must reach a rejection (resp. acceptance) verdict. This formulation is appealing to our study for a number of reasons:

- The touchstone logic used to specify process correctness is the Hennessy-Milner Logic with minimal and maximal fixpoints (recHML) [45]; like session types, it has a tight relation to (ω-)regular properties, and a long tradition of automata-based interpetations.
- Recent work [4, 6] has extended this framework to a spectrum of correctness criteria. This gives us the flexibility of identifying the criteria that best fit our concerns.

To study session types monitorability, we adapt this theoretical framework to our setting:

M1 instead of logic formulas as specifications, we adopt session types as specifications; and
M2 to characterise processes satisfying a specification, we use the session typing system.

This leads to important differences between our approach and [30, 3]:

D1 by item M2, our processes characterisation is syntactic (rather than semantic), which is further removed from the runtime behaviour observed by the monitor;
D2 session types describe interactions between two parties, and our monitors can attribute a violation to a party. By contrast, monitors for recHML formulas flag generic rejections;
D3 we here limit our analysis to rejection monitors and do not consider acceptance verdicts.

Consequently, we formalise our notions of monitoring soundness and completeness as follows. Here, t represents a *trace*, *i.e.,* finite a sequence of environment send/receive actions $\blacktriangle\mathtt{l}(v)$ and $\blacktriangledown\mathtt{l}(v)$ (from Fig. 8); moreover, $\overset{t}{\Rightarrow}$ is a sequence of transitions where the actions in t are interleaved with finite sequences of τ-transitions.

▶ **Definition 13** (*Session Monitor Soundness*). *A monitor M* soundly monitors *for a session type S iff, for all P, if there is a trace t such that $\langle P; M\rangle \overset{t}{\Rightarrow} \langle P'; \mathsf{no}_P\rangle$, then $\emptyset \cdot \emptyset \vdash P : S$ does* not *hold.* ⌟

▶ **Definition 14** (*Session Monitor Completeness*). *A monitor M monitors for a session type S in a* complete *manner, iff for all processes P, whenever $\emptyset \cdot \emptyset \vdash P : S$ does* not *hold, then there exists a trace t such that $\langle P; M\rangle \overset{t}{\Rightarrow} \langle P'; \mathsf{no}_P\rangle$.* ⌟

4.1 Soundness of Session Type Monitoring

A tenet of [30, 3, 4] is that, in order to have monitor correctness, soundness (Def. 13) is not negotiable. We here show that our monitor synthesis procedure is sound, *i.e.,* we show that for any session type S, monitor $[\![S]\!]$ observes Def. 13 w.r.t. specification S.

▶ **Theorem 15** (*Synthesis Soundness*). *For all session types S and processes P, if there exists a trace t such that $\langle P; [\![S]\!]\rangle \overset{t}{\Rightarrow} \langle P'; \mathsf{no}_P\rangle$, then $\emptyset \cdot \emptyset \vdash P : S$ does* not *hold.*

Proof. Instead of proving the statement directly, we prove its contrapositive:

For all session types S and processes P such that $\emptyset \cdot \emptyset \vdash P : S$, if $\langle P; [\![S]\!]\rangle \overset{t}{\Rightarrow} \langle P'; M'\rangle$ then $M' \neq \mathsf{no}_P$.

To this end, we first establish a *subject reduction* result, relying on standard properties of our type system: this determines how process P evolves w.r.t. its session type S. Then, we prove the contrapositive statement above by *lexicographical induction* on the derivation of

$\emptyset \cdot \emptyset \vdash P : S$ and the number of transitions in the trace $\langle P; [\![S]\!]\rangle \xRightarrow{t} \langle P'; M'\rangle$. This requires some sophistication, because as the instrumented system $\langle P; [\![S]\!]\rangle$ evolves, for each step of P the monitor $[\![S]\!]$ (as generated by Def. 9) may take multiple steps to evaluate synthesised conditions before it can forward messages. Hence, we prove additional results to handle such cases, and formulate a suitable induction hypothesis allowing us to complete the proof of the contrapositive statement. Theorem 15 follows as a corollary. ◀

As a by-product of Theorem 15 we also deduce that if a process P has type S, then the instrumented process $\langle P; [\![S]\!]\rangle$ can only get stuck due to an *external* violation, *i.e.,* no_E; this arises when the environment sends a message with a wrong label or payload type. This result is formalised in Corollary 16 below, and is reminiscent of the notion of *blaming* in gradual types (*i.e.,* untyped components can always be blamed in case of errors [10, 41]).

▶ **Corollary 16** (Monitor Blaming). *For any process P and session types S where $\emptyset \cdot \emptyset \vdash P : S$, for any trace t such that $\langle P; [\![S]\!]\rangle \xRightarrow{t} \langle P'; M'\rangle \not\rightarrow$ where $P \neq \mathbf{0}$, we have $M' = \mathsf{no}_E$.* ⌟

4.2 On the Completeness of Session Type Monitoring

Monitor soundness, by itself, is a weak result. For instance, the monitor that merely acts as a forwarder between the monitored process and the environment, *never raising any detections*, is trivially sound but, arguably, not very useful. One way to force the monitor to produce useful detections is via *completeness*, as per Def. 14 above. We investigate completeness for our synthesised monitors by first establishing a "weak" completeness result (§ 4.2.1) showing how ill-typed processes can misbehave when instrumented. Then, we prove that a "full" completeness result is impossible in our black-box monitoring model (§ 4.2.2).

4.2.1 Weak Monitor Synthesis Completeness

To achieve our completeness result, in this section we need a *precise typing* assumption on predicates A: ill-typed predicates do not evaluate to a boolean – *i.e.,* if $\Gamma \vdash A : \mathsf{Bool}$ does *not* hold, then $A \not\Downarrow \mathsf{tt}$ and $A \not\Downarrow \mathsf{ff}$. Furthermore, we need to limit our analysis to processes without *dead code* (Def. 18 below). For the process language of Fig. 2, this means: for every "if" statement occurring in a process P, there are executions of P where the left branch is taken, and executions where the right branch is taken. These executions depend on P's inputs, which may cause different instantiations to P's variables. Example 17 illustrates why we need this assumption; note that these assumptions are *not* needed for monitor soundness.

▶ **Example 17.** The process $P = \mathsf{if}\ \mathsf{tt}\ \mathsf{then}\ \triangleleft\, \mathtt{l}_1(v_1).\mathbf{0}\ \mathsf{else}\ \triangleleft\, \mathtt{l}_2(v_2).\mathbf{0}$ is *not* typable with $S = \oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in\{1\}}$ (for any S_i): it is only typable with internal choices of the form $\oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in 1..n}$, with $n \geq 2$. Yet, P would operate correctly if instrumented with monitor $[\![S]\!]$, because its "else" branch is dead code. If we remove the dead code from P, the remaining process $\triangleleft \mathtt{l}_1(v_1).\mathbf{0}$ is typable with S, and behaves like P. ⌟

▶ **Definition 18.** A process P has no dead code *iff for all its subterms of the form* $P' = \mathsf{if}\ A\ \mathsf{then}\ Q\ \mathsf{else}\ Q'$, *there exist traces t and t' and substitutions σ and σ' such that* $P \xRightarrow{t} P'\sigma \xrightarrow{\tau} Q\sigma$ *(hence, $A\sigma \Downarrow \mathsf{tt}$) and* $P \xRightarrow{t'} P'\sigma' \xrightarrow{\tau} Q'\sigma'$ *(hence, $A\sigma \Downarrow \mathsf{ff}$).* ⌟

With the "no dead code" assumption, we can formulate our weak completeness result. It states that when a process P is ill-typed for a session type S, then the monitored system $\langle P; [\![S]\!]\rangle$ exhibits at least one execution that gets stuck due to P's behaviour, without any violation by the environment.

▶ **Theorem 19** (Weak Monitor Synthesis Completeness). *Take any closed process P without dead code such that* $\emptyset \cdot \emptyset \vdash P : S$ *does* not *hold. Then, there exists a trace t such that* $\langle P; [\![S]\!] \rangle \overset{t}{\Rightarrow} \langle P'; M' \rangle \not\rightarrow$ *, with $P' \neq \mathbf{0}$ or $M' \neq \mathbf{0}$; moreover, $M' \neq \mathsf{no}_E$.*

Proof. The proof is based on *failing derivations*, inspired by [16, 44]. It consists of 6 steps.

1. We define the *rule function* Φ that, following the typing rules in Fig. 4, maps a judgement of the form $J = \Theta \cdot \Gamma \vdash P : S$ to either the set of all judgements in J's premises (for inductive rules), or $\{\mathtt{tt}\}$ (for axioms), or $\emptyset$ (if J does not match any rule);
2. we formalise a *failing derivation* of a session typing judgement $\Theta \cdot \Gamma \vdash P : S$ as a finite sequence of judgements $\mathcal{D} = (J_0, J_1, \ldots, J_n)$ such that:
   - **(i)** for all $i \in 0..n$, J_i is a judgement of the form $\Theta_i \cdot \Gamma_i \vdash P_i : S_i$;
   - **(ii)** $J_0 = \Theta \cdot \Gamma \vdash P : S$ (*i.e.,* the failing derivation $\mathcal{D}$ begins with the judgement of interest);
   - **(iii)** $\forall i \in 1..n$, $J_i \in \Phi(J_{i-1})$ (*i.e.,* each judgement in $\mathcal{D}$ is followed by one of its premises);
   - **(iv)** $\Phi(J_n) = \emptyset$ (*i.e.,* the last judgement in $\mathcal{D}$ does not match any rule in Fig. 4)
3. we prove there is a failing derivation of $J = \Theta \cdot \Gamma \vdash P : S$ if and only if J is *not* derivable;
4. we formalise a *negated* typing judgement $\Theta \cdot \Gamma \nvdash P : S$ and prove that it holds if and only if there is a corresponding failing derivation of $\Theta \cdot \Gamma \vdash P : S$;
5. thus, from items 3 and 4 above, we know that $\Theta \cdot \Gamma \vdash P : S$ is *not* derivable if and only if $\Theta \cdot \Gamma \nvdash P : S$ is derivable. Consequently, the judgement $\Theta \cdot \Gamma \nvdash P : S$ tells us exactly what are the possible shapes of P and S covered by the theorem's statement;
6. finally, we use all ingredients above to prove the thesis. From a failing derivation of $\Theta \cdot \Gamma \vdash P : S$ (item 3), we construct a trace t leading from $\langle P; M \rangle$ to some $\langle P'; M' \rangle$; further, using the corresponding derivation of $\Theta \cdot \Gamma \nvdash P : S$ (items 4, 5), we prove that t is a valid trace, and $\langle P'; M' \rangle \not\rightarrow$ with $P' \neq \mathbf{0}$ or $M' \neq \mathbf{0}$, and $M' \neq \mathsf{no}_E$. ◀

Although Theorem 19 is weaker than the ideal requirement set out in Def. 14, its proof sheds light on all the possible reasons why an ill-typed monitored process gets stuck:

1. the monitor reaches a process rejection verdict, $M' = \mathsf{no}_P$, because the process sends a message with a wrong label, or payload value of a wrong base type.
2. the monitor blocks waiting for the process to send a message, but:
   - **a.** P' is attempting to receive a message itself or
   - **b.** $P' = \mathbf{0}$ (i.e., P' has terminated its execution);
3. the monitor blocks waiting for the process to receive a message, but:
   - **a.** the process is also waiting to receive a message but does not support the required message label being sent or
   - **b.** P' is attempting to send a message itself or
   - **c.** $P' = \mathbf{0}$;
4. the monitor expects the process to end, but P' is trying to send/receive more messages;
5. P' is stuck on an ill-typed expression.

▶ Remark 20. Process violations are only flagged no_P (as required in Def. 14) is case 1. We now discuss how a practical monitor implementation could, in principle, detect violations in other cases, and highlight when this additional detection power would require additional assumptions that go beyond our black-box monitoring design.

- In cases 3a and 5, the trace t may lead to a run-time error; this could be flagged by assuming that the monitor can detect whether the monitored process has crashed;

- In case 4, the monitor expects the session to be ended. This could be handled by assuming and end-of-session signal: the monitor can wait for such a signal, and flag any other message sent by the process. However, if the process is attempting to receive (instead of ending the session), the detection is more subtle, as in case 2a below;
- Cases 2b and 3c could be similarly handled by assuming an end-of-session signal;
- Case 2a is more subtle: both the process and monitor are waiting for a message. Reception timeouts from the monitor side are inadequate because they lead to unsound detections. To accurately handle this case, we would need to instrument the process executable, which breaks our black-box assumptions from § 3.1. Similarly, flagging a violation in case 3b also requires access to the process code, again breaking our black-box design. ⌟

4.2.2 Impossibility of Sound and Complete Session Monitoring

The weakness of our completeness result in Theorem 19 is not specific to our monitor synthesis function. Rather, we show that this is an inherent limit of the operational model (Figures 6 and 8) that captures the black-box monitor design decisions of § 3.1. Similar impossibility results often arise for reasonably expressive specification languages (such as the logics in [30, 1, 3, 4]), where it is usually the case that only a subset of specifications can be monitored in a sound and complete way.

▶ **Theorem 21** (Impossibility of Sound and Complete Session Monitoring). *A (closed) session type $S \neq \mathsf{end}$ cannot have a sound and complete monitor under the semantics of Fig. 6.*

Proof. We proceed by case analysis on the structure of S:

Case $S = \&\{?\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$: We assume that a complete monitor M for S exists and proceed to show that such a monitor is necessarily unsound for S. Fix a complete monitor M for S. Consider the process $P_2 = \triangleright\{\mathtt{l}_i(x_i).Q_i\}_{i\in I}$ that is well-typed w.r.t. the session type S. Then, consider the process P_1 obtained by pruning some of the top-level external choices of P_2, *i.e.,* $P_1 = \triangleright\{\mathtt{l}_j(x_j).Q_j\}_{j\in J}$ where $J \subset I$ (a strict inclusion). Observe that P_1 is ill-typed for S, and thus, by completeness (Def. 14), M should reject P_1, (*i.e.,* there must exists a trace t such that $\langle P_1; M\rangle \overset{t}{\Rightarrow} \langle P_1'; \mathsf{no}_P\rangle$). There are two ways for M to reach such a verdict:

- $M \overset{t}{\Rightarrow} \mathsf{no}_P$ without interacting with P_1. In this case, the same rejection verdict is reached by the composite system $\langle P_2; M\rangle$. Since P_2 is well-typed for S, this means that M is unsound for S by Def. 13;
- M reaches the rejection verdict after interacting (at least once) with P_1. In this case, we have $P_1 \xrightarrow{\triangleright\mathtt{l}_j(v)} Q_j$ (for some $j \in J$), and there are t_1, t_2, P_1' such that $t = t_1.t_2$ and $M \xRightarrow{t_1.\triangleleft\mathtt{l}_j(v)} M'$ and $\langle Q_j; M'\rangle \overset{t_2}{\Rightarrow} \langle P_1'; \mathsf{no}_P\rangle$. But then, since $j \in J \subseteq I$, we also have $\langle P_2; M\rangle \overset{t}{\Rightarrow} \langle P_1'; \mathsf{no}_P\rangle$. Since M rejects the well-typed process P_2, this again makes M unsound for S by Def. 13.

We have thus shown that a complete monitor M for S is necessarily unsound.

Case $S = \oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$: Assume that a complete monitor M for S exists. The process $P_1 = \mathbf{0}$ is ill-typed for S (since it does not produce any of the expected outputs). By Def. 14 (Completeness), there must exist a trace t such that $\langle P_1; M\rangle \overset{t}{\Rightarrow} \langle P_1'; \mathsf{no}_P\rangle$. From the structure of P_1 it is clear that M reaches its rejection verdict without interacting with P_1, *i.e.,* $M \overset{t}{\Rightarrow} \mathsf{no}_P$. This also means that M would also reach a rejection verdict when instrumented with $P_2 = \triangleleft\mathtt{l}_k(v_k).Q_2'$ with $k \in I$ and is well-typed w.r.t. S. This makes M unsound by Def. 13.

Recall that all session types are assumed to be guarded. Since the above two cases rule out all the guarding constructs, $\oplus\{!\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$ and $\&\{?\mathtt{l}_i(\mathsf{B}_i).S_i\}_{i\in I}$, we conclude that there is no closed (guarded) session type that can be monitored for soundly and completely, except for all the trivial session types that equate to `end`. ◀

5 Realisability and Implementation

Up to this point we have considered a level of abstraction that allows us to model session monitors and monitored components, reason about their behaviour, and prove their properties. We now illustrate how our theoretical developments can be translated into an actual implementation of session monitoring, targeting the Scala programming language. The key idea is to turn our monitor synthesis procedure (Def. 9) into a code generation tool that, given a protocol specification (as a session type), produces the Scala code of a corresponding executable monitor. The tool is called `STMonitor`, and is provided as companion artifact to this paper. It is also available at:

`https://github.com/chrisbartoloburlo/stmonitor` (release tag `v0.0.1`)

We describe `STMonitor` in § 5.2 – but first, we augment session types with *assertions* (§ 5.1).

5.1 Introducing Assertions in Session Types Specifications

Since we use session types as specifications for a tool that generates executable monitors, it is convenient to enrich them with *assertions* on the values being sent or received. We augment the session types syntax (Fig. 3) by extending selection and branching as follows:

$$S \quad ::= \quad \oplus\{!\mathtt{l}_i(x_i : \mathsf{B}_i)[A_i].S_i\}_{i\in I} \quad | \quad \&\{?\mathtt{l}_i(x_i : \mathsf{B}_i)[A_i].S_i\}_{i\in I} \quad | \quad \ldots$$

The assertions A_i are predicates of the process calculus (Fig. 2, Remark 4), and they can refer to the named payload variables x_i. Such assertions do not influence type-checking: they are copied in the synthesised monitors, where they are used to flag the new violations no_P^A (assertion violation by the process) and no_E^A (external assertion violation). To achieve this, we update our monitor synthesis function (Def. 9) as follows:

$$[\![\oplus\{!\mathtt{l}_i(x_i : \mathsf{B}_i)[A_i].S_i\}_{i\in I}]\!] \triangleq \triangleright\{\mathtt{l}_i(x_i).\mathsf{if}\ \mathsf{isB}_i(x_i)\ \mathsf{then}\ (\mathsf{if}\ A_i\ \mathsf{then}\ \blacktriangle\mathtt{l}_i(x_i).[\![S_i]\!]\ \mathsf{else}\ \mathsf{no}_P^A)\ \mathsf{else}\ \mathsf{no}_P\}_{i\in I}$$

$$[\![\&\{?\mathtt{l}_i(x_i : \mathsf{B}_i)[A_i].S_i\}_{i\in I}]\!] \triangleq \blacktriangledown\{\mathtt{l}_i(x_i).\mathsf{if}\ \mathsf{isB}_i(x_i)\ \mathsf{then}\ (\mathsf{if}\ A_i\ \mathsf{then}\ \triangleleft\mathtt{l}_i(x_i).[\![S_i]\!]\ \mathsf{else}\ \mathsf{no}_E^A)\ \mathsf{else}\ \mathsf{no}_E\}_{i\in I}$$

The only changes are highlighted: if the monitored process sends a message that violates the assertion, it is flagged with no_P^A; symmetrically, if a message that violates the assertion is received from the environment, then the message is flagged with no_E^A.

▶ **Example 22.** Recall S_{auth} from Example 3. We can refine it with assertions to check the validity of the data being transmitted and received:

$$\begin{aligned} S_{auth}^A &= \mathsf{rec}\ Y.!\mathtt{Auth}(\mathit{uname} : \mathsf{Str}, \mathit{pwd} : \mathsf{Str})[\mathit{validUname}(\mathit{uname})].S_{res}^A \\ S_{res}^A &= \&\{?\mathtt{Succ}(\mathit{tok} : \mathsf{Str})[\mathit{validTok}(\mathit{tok}, \mathit{uname})].S_{succ}^A, ?\mathtt{Fail}(\mathit{code} : \mathsf{Int})[\mathtt{tt}].Y\} \end{aligned}$$

In S_{auth}^A, when the *client* sends $\mathtt{Auth}(\mathit{uname}, \mathit{pwd})$, the value of *uname* is passed to the predicate *validUname* which ensures that the supplied *uname* is given in the correct format. If the *server* replies with $\mathtt{Succ}(\mathit{tok})$, the token *tok* and username *uname* are validated by the cryptographic predicate *validTok*, which tests whether the token is correct for the given username. If so, the *client* continues along session type S_{succ}^A. Otherwise, if the *server* chooses to send `Fail` with the error *code*, the trivial assertion check `tt` is performed. ⌟

Notice that, when all assertions are trivially true, the augmented monitor synthesis is equivalent to the original Def. 9. Otherwise, the synthesised monitors with assertions are more restrictive: executions where no violations no_P nor no_E were detected might now violate an assertion and result in no_P^A or no_E^A. The introduction of such assertions in our theory changes our monitorability results as follows:

- soundness (Theorem 15) is preserved – which is crucial for practical usability;
- blaming (Corollary 16) is weakened: an instrumented well-typed process may violate an assertion, and be flagged with no_P^A;
- weak detection completeness (Theorem 19) is *not* preserved: assertions can in principle be unsatisfiable, hence some ill-typed processes may not be flagged because all their traces end with an environment assertion violation no_E^A.

5.2 Implementation

We now illustrate the implementation of our session monitor synthesis tool. It generates runnable Scala code from session types, possibly including the assertions discussed in § 5.1.

Implementation framework. Our synthesised monitors uses the session programming library `lchannels` [56]. It allows for implementing a session type S in Scala, by

1. defining a set of *Continuation-Passing-Style Protocol classes* (CPSPc) corresponding to S, and
2. using a communication API that, by leveraging such CPSPc, lets the Scala compiler spot protocol violations.

By using `lchannels`, we are more confident that if a syntesised monitor for session type S compiles, then it correctly sends/receives messages according to S. Moreover, `lchannels` abstracts communication from the underlying message transport, hence it allows our monitors to interact with clients or servers written in any programming language.

Implementation of the session monitor synthesis. Overall, our Scala monitor generation requires 3 user-supplied inputs:

i1 a session type S (with or without assertions) describing the desired protocol;
i2 for each assertion in S (if any), a corresponding Scala function returning true/false; and
i3 a *Connection Manager* class (discussed below) to interact with the monitored process.

Given a session type (input 1) our monitor synthesiser tool generates:

1. the protocol classes (CPSPc) for representing the session type in Scala + `lchannels`, and
2. the Scala source code of a runtime monitor (requiring inputs 2 and 3 to compile).

The generated monitor acts as a mediator between client and server: one is on the *internal* side of the monitor (*i.e.,* the instrumented process), while the other is on the external side. The internal side is untrusted: its messages are run-time checked, to ensure they follow the desired protocol (*e.g.,* session type S_{auth}^A in Example 22). Instead, the *external* side is trusted: it is (mostly) expected to follow the dual protocol (*e.g.,* the dual session type $\overline{S_{auth}^A}$). This design choice allows us to simplify the monitor implementation, as its communication with the external side are handled by `lchannels`. However, our design does not limit the flexibility of the approach, since an untrusted peer can be made trusted by instrumenting it with a monitor (see discussion below).

Monitor synthesis in practice. We now illustrate the scenario depicted in Fig. 11 where:

1. we want a client/server system to implement the session type (with assertions) S^A_{auth} (Example 22);
2. we trust the *server* (*e.g.,* because it is type-checked), and
3. we want to instrument a *client* whose source code is inaccessible or cannot be verified.

Other variations of this scenario are possible. For instance, we could similarly instrument an untrusted server, by running our monitor synthesiser on the dual session type $\overline{S_{auth}}$. The resulting combination of monitor-and-server is then trusted and can interact via `lchannels`. As a result, it could then be used as the trusted *server* in Fig. 11.

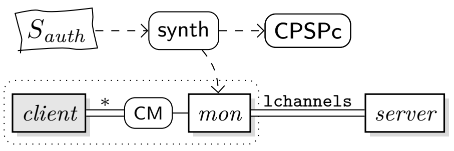


Figure 11 The composite system interacting with the environment.

The generated monitor (*mon*) intercepts all messages between *client* and *server*. The communication between *mon* and *server* occurs via `lchannels`; instead, the communication between the monitor and the client is handled by a *Connection Manager* (CM): a user-supplied Scala class, input 3, which acts as a *translator* and *gatekeeper*, by transforming each messages from the monitor-client transport protocol into a corresponding CPSP class, and *vice versa*. With this design, the code generated for the monitor is abstracted from the low-level details of the protocols used by both the *client* and *server*.

There is a tight correspondence between the monitors generated by our tool, and our formal monitor synthesis. This increases our confidence that the results in § 4 carry over to our implementation and that our tool is indeed correct. In the sequel, we illustrate the generated monitoring code for Example 22 above, showing the monitoring of a selection type (§ 5.2) and branching type (§ 5.2).

$$
\begin{aligned}
&[\![S^A_{auth}]\!] \;=\\
&\quad \mu_Y . \triangleright \Big\{ \mathtt{Auth}(uname : \mathsf{Str}, pwd : \mathsf{Str}).\\
&\quad \text{if } \mathsf{isB}_{\mathsf{Str}}(uname) \wedge \mathsf{isB}_{\mathsf{Str}}(pwd)\\
&\quad\;\; \text{then if } validUname(uname)\\
&\quad\;\;\;\; \text{then } \blacktriangle\mathtt{Auth}(uname, pwd).[\![S_{res}]\!]\\
&\quad\;\;\;\; \text{else } \mathsf{no}^A_P\\
&\quad\;\; \text{else } \mathsf{no}_P \Big\}
\end{aligned}
$$

```
1  def receiveAuth(srv:Out[Auth],client:CM): Unit ={
2    client.receive() match {
3      case msg @ Auth(_, _) =>
4        if (validUname(msg.uname)) {
5          val cont = srv !! Auth(msg.uname,
6                                 msg.pwd)_
7          payloads.Auth.uname = msg.uname
8          sendChoice1(msg.cont, client)
9        } else {
10         /* INTERNAL VIOLATION (assertion) */
11       }
12     case _ =>
13       /* INTERNAL VIOLATION: invalid message */
14 } }
```

Figure 12 Comparison between the formal and implementation synthesis of the internal choice.

The internal receive operator of the monitor calculus ($\triangleright$) corresponds to line 2 in § 5.2, where the monitor invokes the `receive` method of the CM. Depending on the type of message received, the monitor performs a series of checks. By default, a catch-all case (line 12) handles any messages violating the protocol: this is similar to rule [MIV] of the formal monitor (Fig. 6), which flags the violation no_P. If `Auth` is received, the monitor initially invokes the function `validUname()` with argument `uname`; such a function is user-supplied (see input 2 above). If the function returns `false`, the monitor flags the violation and halts (line 10): this corresponds to the external assertion violation no_P^A in $[\![S_{auth}^A]\!]$. Otherwise, if `validUname()` returns `true`, the message is forwarded to the *server* (line 5). The function used to forward the message (`!!`), which is part of `lchannels`, corresponds to the external output operator $\blacktriangle$ of $[\![S_{auth}^A]\!]$; it returns a continuation channel that is stored in `cont`. To associate the payload identifiers of S_{auth}^A to their current values, the monitors maintain a mapping, called `payloads`. In this case, the value of `uname` is stored (line 7) since it is used later on in S_{auth}. Finally, the monitor moves to the next state `sendChoice1` (§ 5.2), passing the channel stored in `cont` to continue the protocol (line 8).

$$
\begin{aligned}
&[\![S_{res}^A]\!] = \\
&\quad \blacktriangledown\Big\{\mathtt{Succ}(tok : \mathsf{Str}).\mathsf{if}\ \mathsf{isB}_{\mathsf{Str}}(tok) \\
&\qquad\quad \mathsf{then\ if}\ validTok(tok, uname) \\
&\qquad\quad\ \mathsf{then}\ \triangleleft\,\mathtt{Succ}(tok).[\![S_{succ}]\!] \\
&\qquad\quad\ \mathsf{else}\ \mathsf{no}_E^A\ \mathsf{else}\ \mathsf{no}_E, \\
&\qquad \mathtt{Fail}(code : \mathsf{Int}).\mathsf{if}\ \mathsf{isB}_{\mathsf{Int}}(code) \\
&\qquad\ \mathsf{then\ if}\ \mathsf{tt}\ \mathsf{then}\ \triangleleft\,\mathtt{Fail}(code).Y \\
&\qquad\ \mathsf{else}\ \mathsf{no}_E^A\ \mathsf{else}\ \mathsf{no}_E\Big\}
\end{aligned}
$$

```scala
1  def sendChoice1(srv:In[Choice1],Client:CM):Any = {
2    srv ? {
3      case msg @ Succ(_) =>
4        if (validTok(msg.tok, payloads.Auth.uname)) {
5          Client.send(msg)
6          /* Continue according to S_succ */
7        } else {
8          /* EXTERNAL VIOLATION (assertion) */
9        }
10     case msg @ Fail(_) =>
11       Client.send(msg)
12       receiveAuth(msg.cont, External)
13 } }
```

Figure 13 Comparison between the formal and implementation synthesis of the external choice.

According to S_{res}^A, the server can choose to send either `Succ` or `Fail`. The monitor waits to receive either of the options from the *server*, using the method `?` from `lchannels` (line 2). This corresponds to the external input operator of the monitor calculus ($\blacktriangledown$) used in $[\![S_{res}^A]\!]$, which can also receive both options from the *server*.

- If the *server* sends `Succ`(toc), the first case is selected (line 3). The monitor evaluates the assertion `validTok` on *tok* and *uname* (stored in § 5.2, and now retrieved from the `payloads` mapping). If it is satisfied, the message is forwarded to the client (line 5) via CM's `send` method, which corresponds to the internal send operator ($\triangleleft$) in the monitor calculus. The monitor then proceeds according to the monitor $[\![S_{succ}]\!]$. Otherwise, the monitor logs a violation and halts (line 8); similarly, $[\![S_{res}^A]\!]$ flags the violation no_E^A indicating an external assertion violation.
- Instead, if the *server* sends `Fail` (line 10), the monitor forwards it to the client; there are no assertion checks here, as the assertion after `Fail` in $[\![S_{res}^A]\!]$ is tt. Then, following the recursion in $[\![S_{res}^A]\!]$, the monitor (on line 12) loops to `receiveAuth` (§ 5.2).

Unlike the synthesised code of `receiveAuth` (that handles the previous external choice, in § 5.2), there is no catch-all case for unexpected messages from the *server*. This is by design. As explained above we use `lchannels` to interact with the "trusted" external side, hence the interaction with the server is typed, and a catch-all case would be unreachable code. Still, `lchannels` throws an exception (crashing the monitor) if it receives an invalid message – which corresponds to the monitor $[\![S_{res}^A]\!]$ flagging an external violation via rule [MEV].

6 Empirical Evaluation

We evaluate the feasibility of our implementation by measuring the overheads induced by the run-time checks of our synthesised monitors (§ 5). We consider 3 application protocols, modelled as session types, as our benchmarks:

1. A ping-pong protocol, based on a request-response over HTTP (a style of protocol that is typical, *e.g.,* in applications based on web services). Although it is a fairly simple protocol, our implementation uses HTTP to carry ping/pong messages, highlighting the fact that our generated monitors are independent from the message transport in use;
2. A fragment of the Simple Mail Transfer Protocol (SMTP) [47]. This benchmark represents a more complex protocol featuring nested internal/external choices;
3. A fragment of the HTTP protocol, also featuring sequences of nested internal/external choices.

Ping-pong over HTTP. In this protocol, a client is expected to recursively send messages with label `Ping` to the server which, in turn, replies with `Pong`. The protocol proceeds until the client sends `Quit`. The client-side protocol is shown below (the server-side is dual).

$$S_{pong} \;=\; \mathsf{rec}\ X.(\oplus\big\{!\mathtt{Ping}().?\mathtt{Pong}().X,\ !\mathtt{Quit}()\big\})$$

Notice that the protocol has no explicit reference to HTTP. In fact, we use HTTP as a mere message transport, by providing a suitable Connection Manager to the synthesised monitor (which is transport-agnostic). Concretely, the ping-pong is implemented with the server handling requests on an URL like `http://127.0.0.1/ping`, and the client performing a `GET` request on that URL, and reading the response. For this benchmark, the setup is:

- the client is on the internal side of the generated monitor, hence subject to scrutiny;
- the server is on the external side of the generated monitor.

As untrusted client we use a standard, unmodified load testing tool: Apache JMeter (`https://jmeter.apache.org/`) configured to send HTTP requests at an increasing rate.

SMTP. We model a fragment of the SMTP protocol (server-side) as the session type S_{smtp}:

$$\begin{aligned}
S_{smtp} &= !\mathtt{M220}(msg:\mathsf{Str}).\&\big\{?\mathtt{Helo}(host:\mathsf{Str}).!\mathtt{M250}(msg:\mathsf{Str}).S_{mail},\,?\mathtt{Quit}().!\mathtt{M221}(msg:\mathsf{Str})\big\}\\
S_{mail} &= \mathsf{rec}\ X.(\&\big\{?\mathtt{MailFrom}(addr:\mathsf{Str}).!\mathtt{M250}(msg:\mathsf{Str}).\mathsf{rec}\ Y.(\&\big\{ && (7)\\
&\qquad ?\mathtt{RcptTo}(addr:\mathsf{Str}).!\mathtt{M250}(msg:\mathsf{Str}).Y, && (8)\\
&\qquad ?\mathtt{Data}().!\mathtt{M354}(msg:\mathsf{Str}).?\mathtt{Content}(txt:\mathsf{Str}).!\mathtt{M250}(msg:\mathsf{Str}).X, && (9)\\
&\qquad ?\mathtt{Quit}().!\mathtt{M221}(msg:\mathsf{Str})\big\}),\,?\mathtt{Quit}().!\mathtt{M221}(msg:\mathsf{Str})\big\})
\end{aligned}$$

When a client establishes a connection, the server sends a welcome message (`M220`), and waits for the client to identify itself (`Helo`). Then, the client can recursively send emails by specifying the sender and recipient address(es), followed by the mail contents. The client can send multiple emails by repeating the loop on "X" between lines (7) and (9).

The SMTP protocol runs over TCP/IP. The specification above (and the synthesised monitors) are again transport-agnostic: we handle TCP/IP sockets by providing a suitable Connection Manager to the synthesised monitor.

For this benchmark, the setup used is "dual" to that of the HTTP ping-pong benchmark above:

- the server is on the internal side of the generated monitor, hence subject to scrutiny;
- the client is on the external side of the generated monitor.

For this experiment, we implement an SMTP client that sends emails to the server, and measures the response time. We take such measurements against two (untrusted and monitored) servers, both configured to accept incoming emails and discard them:

1. a default instance of `smtpd` from the Python standard library;[1]
2. a default instance of Postfix,[2] one of the most used SMTP servers [59].

HTTP. In this benchmark, we do *not* use HTTP as a mere message transport (unlike the ping-pong benchmark above). Rather, we model HTTP headers, requests, and responses with a session type, which we use to synthesise a monitor that checks the interactions between a trusted server and an untrusted client. We focus on a fragment of HTTP that is sufficient for supporting typical client-server interactions (*e.g.,* when the client is the Mozilla Firefox browser). The HTTP session type (here omitted due space reasons) and its (trusted) server implementation are adapted from the `lchannels` examples [55]. For benchmarking, we use Apache JMeter (`https://jmeter.apache.org/`) as untrusted client.

Benchmarking setups and measurements. In all of our benchmarks, we study the overhead of our synthesised monitors by comparing:

- an *unsafe* setup: the client and server interact directly;
- a *monitored* setup: communication between the trusted and untrusted components is mediated by our synthesised monitors, which halts when it detects a violation – as described earlier in Fig. 11.

We follow a multi-faceted approach, as advocated by [8], and base our study on three measurements: average response time, average CPU utilisation, and maximum memory consumption. The response time is arguably the most important measurement, since slower response times can be immediately perceived when interacting with a monitored system. We measure them by running experiments of increasing length: for ping-pong and HTTP, we perform an increasing number of request-response loops, whereas for SMTP, we send an increasing number of emails. The general expectation is: for longer experiments, the average response time and CPU usage should decrease, while the maximum memory consumption should increase. We repeat each experiment 30 times, and we plot the average of all results.

In our benchmarks, overheads can have two forms:

Overhead 1: the translation and duplication of messages being forwarded between client and server;

Overhead 2: the run-time checks needed to ensure that the desired session type is being respected.

Overhead 1 is unavoidable for the most part. By their own nature, partial identity monitors (like ours) must receive and forward all messages. This overhead can only be minimised by using more efficient message transports. By contrast, overhead 2 is specifically caused by our monitor synthesis. Our benchmarks were specifically designed to accurately capture this latter form of overhead. In order to better distinguish overhead 1 from overhead 2, our benchmarks run the trusted side (client or server) and the synthesised monitors on a same JVM instance, where they interact in the most efficient way (*i.e.,* through the `LocalChannel` transport provided by the `lchannels` library). This minimises overhead 1,

[1] `https://docs.python.org/3/library/smtpd.html`
[2] `http://www.postfix.org/`

and allows us to better observe the impact of overhead 2. Clearly, the untrusted side of each benchmark (*i.e.,* the black-box client or server being monitored) always runs as a separate process.

Despite this, our synthesised monitors can still be deployed independently of the trusted side (*i.e.,* on their own JVM, possibly across a network) because they are agnostic to the message transports in use; this is made possible by the use of connection managers and `lchannels`. We demonstrate this capability by also taking measurements for a *detached* setup, where the trusted component and monitor run on separate JVMs (on a same host), and interact via TCP/IP (through a suitable message transport for `lchannels`). This setup is more flexible, but the slower message transport increases overhead 1. We implemented this setup for ping-pong and SMTP, measuring their response times.

Results and analysis. The benchmark results are reported Fig. 14. For the ping-pong benchmark (Fig. 14a), the impact of monitors is noticeable but limited: for the "monitored" setup (which highlights overhead 2), the response times are less than 14% slower; the "detached" monitor setup is unsurprisingly slower, due to its slower message transport (which increases overhead 1). For the SMTP benchmark (Figures 14b and 14c), we can observe different behaviours:

- the Python `smtpd` server (Fig. 14b) has extremely fast response times: it is essentially a dummy server that receives emails and does nothing with them. This is also evident from the CPU usage: it constantly increases, because the SMTP client receives immediate responses, no matter how many emails it sends, with or without a monitor. Consequently, our monitors cause a relatively high impact on such fast response times (almost 34%);
- the Postfix SMTP server (Fig. 14c) is more realistic: unlike Python `smtpd`, it takes some time (with fluctuations) to process each email and respond to the client. Consequently, our monitors have a relatively small impact on the response times (less than 7%).

As in the case of ping-pong, the "detached" monitor setup for both SMTP benchmarks is slower, as it uses a slower message transport (which increases overhead 1). Finally, the HTTP benchmark (Fig. 14d) shows a response time overhead that is below 5%. By and large, these overhead levels are tolerable for many applications that are not mission critical, and are comparable to the overhead experienced when running state-of-the-art RV tools [13].

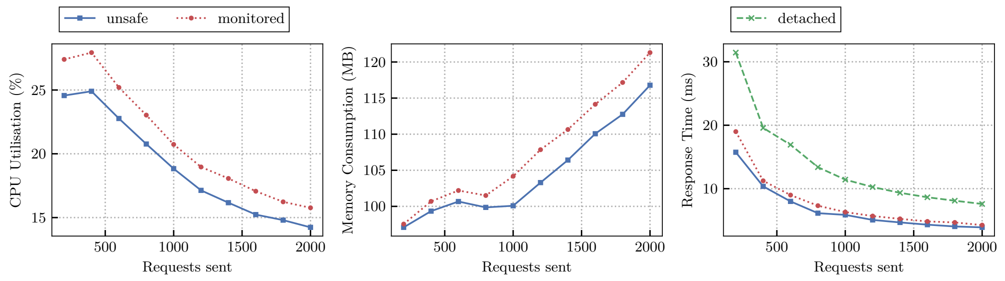
7 Conclusion

We presented a formal analysis for the *monitorability limits* of (binary) session types w.r.t. a partial-identity monitor model; to wit, this is the first monitorability assessment of session types. We couple this study with an implementation of session monitor synthesis.

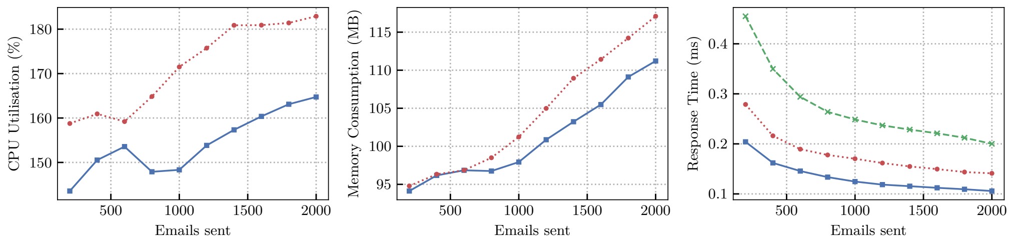
More in detail, our contributions are the following. On the the theoretical side, we provide the first treatment of the *monitorability* of session types, and *detection-soundness* and *detection-completeness* properties of session monitors, and we prove that our autogenerated session monitors enjoy both the former and (to a lesser extent) the latter. We also present an impossibility result of completeness for our black-box monitoring setup – which is a novel result to the area of session type monitoring. On the practical side, we evaluate the viability of our implementation (called `STMonitor`) via benchmarks. The results show that our monitor synthesis procedure only introduces limited overheads.

7.1 Related Work

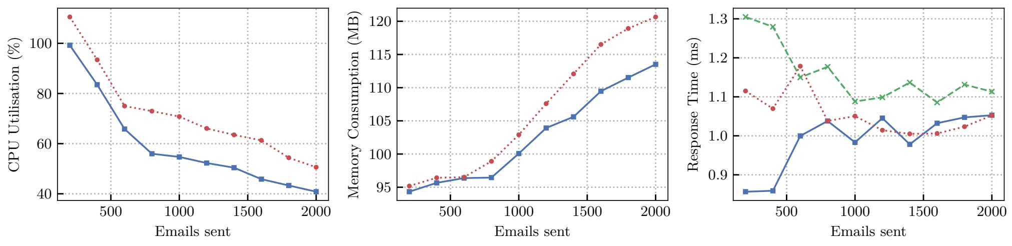
Several papers address the monitoring of session-types-based protocols – but no previous work studies the formal problem of session monitorability; furthermore, their approaches differ from ours in various ways, as we now discuss.



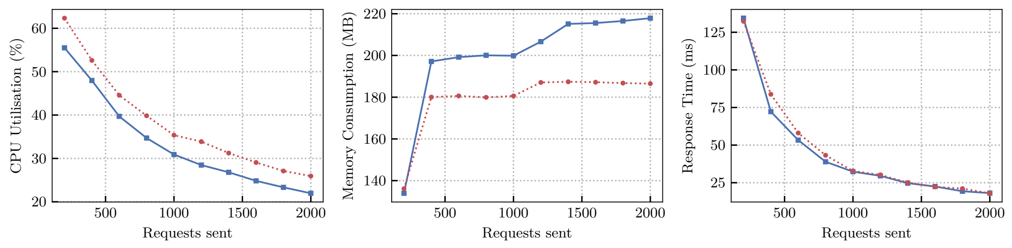
(a) Ping-pong over HTTP (trusted server, untrusted client). Monitored response time overhead: 13.82%.



(b) SMTP Python session (trusted client, untrusted server). Monitored response time overhead: 33.98%.



(c) SMTP Postfix session (trusted client, untrusted server). Monitored response time overhead: 6.68%.



(d) HTTP session (trusted server, untrusted client). Monitored response time overhead: 4.81%.

Figure 14 Benchmark results: average CPU usage, maximum memory consumption, and average response time (30 runs, 2 CPUs (Intel Pentium Gold G5400 @ 3.70GHz), 8 GB RAM, Ubuntu 20.04).

The work [17] formalises a theory of process networks including monitors generated from (multiparty) session types. The main differences with our work are:

1. the design of [17] is based on a global, centralised router providing a *safe transport network* that dispatches messages between participant processes; correspondingly, its implementation [24, 50, 40] includes a Python library for monitored processes to access the safe transport network. By contrast, we do not assume a specific message routing system, and our theory and implementation address the monitoring of black-box components;
2. the results in [17] do not consider limits related to session monitorability. Their results (*e.g., transparency*) are analogous to our *detection soundness* (Theorem 15), *i.e.,* synthesised monitors do not disrupt communications of well-typed processes; they do not address *completeness* (Theorem 19 and Theorem 21), *i.e.,* to what extent can a monitor detect ill-typed processes.

Furthermore, our work and [17] differ in a fundamental design choice: when our monitors detect an invalid message, they flag a violation and halt – whereas monitors in [17] drop invalid messages, and keep forwarding the rest. The latter is akin to *runtime enforcement via suppressions* [9]; studying this design with our theory is interesting future work.

Our protocol assertions (§ 5.1) are reminiscent of *interaction refinements* in [48], that are also statically generated (by an F# type provider), and dynamically enforced when messages are sent/received. However, our approach and design are different from [48]: we synthesise session monitoring processes that can be deployed over a network, to instrument black-box processes – whereas [48] expects the runtime-verified code to be written with a specific language and framework, and injects dynamic checks in the program executable. Furthermore, the work [48] does not address session monitorability limits.

The work [49] proposes a methodology to supervise (multiparty) session protocols, and recover them in case of failure of some component; it also includes an implementation in Erlang. Similarly to this work, in [49] each component is observed by a session monitor; unlike this work, [49] does not address any aspect of session monitorability, and focuses on proving that its recovery strategy does not deadlock.

The work by Gommerstadt *et al.* [34] considers a partial identity monitor model for session types that is close to the one discussed in § 3. They however do not provide any synthesis function and assume that monitors are constructed by hand. To complement this, they define a dedicated type system to prove that the monitor code behaves as a partial identity, *e.g.,* it forwards messages in the correct order, without dropping them. They do not study session monitorability. To our knowledge, their approach has not been implemented as a tool nor has it been assessed empirically either.

Melgratti and Padovani [46] propose monitors that act as wrappers around a session library. This technique effectively *inlines* the monitors in the monitored process code. In fact, their implementation assumes that the processes under scrutiny are written in OCaml using the FuSE library. In contrast, we synthesise *outline* monitors as independent processes that observe black-box implementations written in *any* language/library. The work proves a series of results that are akin to our notion of monitoring soundness, without addressing completeness.

In separate work, Waye *et al.* [63] monitor black-box services, focusing exclusively on *request-response* protocols. Unlike our session-type monitors, they do not support protocols with prescribed sequences of internal/external choices and recursion. In fact, their contracts are analogous to enhanced assertions on transmitted/received values (reminiscent of the assertion introduced in § 5.1). Although they provide soundness results for their monitoring framework, they do not consider any further monitorability issues.

The recent work [35] presents a runtime verification framework for communication protocols (based on multiparty session types) in Clojure. Unlike this work, [35] expects monitored applications to be written in a specific language and framework – whereas we address the monitoring of black-box processes. Again, [35] does not study session monitorability.

7.2 Future Work

This work is our first step along a new line of research on the relative power of static versus run-time verification methods. In general terms, given a calculus C with a type system T and run-time monitoring system M, *monitoring soundness* tells us whether M is flagging "real" errors according to T. Dually, *monitoring completeness* tells us whether T is too restrictive w.r.t. M (*i.e.,* whether T is rejecting too many processes that M deems well-behaved). In this work, we demonstrate a rather tight connection between the chosen process calculus (C) and session type system (T), and our session monitors (M): our synthesised monitors are sound (Theorem 15), and most processes rejected by the type system behave incorrectly (Theorem 19). Our plan is to study more instances of C, T and M – both in theory, and in practice.

One avenue worth exploring is that of increasing the observational powers of the monitoring setup considered, in order to extend session monitorability. The work by Aceto *et al.* [2] is a systematic study that considers a variety of extensions to the traditional monitoring setup (consisting of one monitor observing events describing the computation effected by the process under scrutiny). The extensions considered include traces that report process termination and events that could not have been produced at different stages of the computation (*i.e.,* refusals [52]). They also consider monitoring setups where a process is monitored over multiple runs. In each case, they show the maximal properties that can be monitored for in a sound and complete manner, characterised a syntactic fragments of the modal μ-calculus. We intend to consider how any of the proposed extensions would affect our monitorability results and the extent to which they are implementable in practice. Other bodies of work take a slightly different approach to monitorability, by weakening the completeness requirement from their notion of adequate monitoring [5, 6]. It would be worthwhile exploring the effect of having such weakened completeness requirements on the monitorability of session types.

Although we have limited ourselves to binary session, we plan to extend the framework above to the static and run-time verification of multiparty and asynchronous sessions [38, 39]. This will most likely require us to consider communicating monitors, that cooperate to aggregate observations made from analysing communications on distinct channels. For multiparty sessions, we can benefit from previous work [54, 55] where `lchannels` is used to implement multiparty protocols written in Scribble [58, 64]. Our implementations should also benefit from insights gained from numerous work on decentralised runtime verification [15, 12, 25]. For both multiparty and asynchronous sessions, we can benefit from the research on precise session subtyping [33, 32, 22].

In this work, our session monitors adhere to the *"fail-fast"* design methodology: if a protocol violation occurs, the monitor flags the violation and halts. In the practice of distributed systems, "fail-fast" is advocated as an alternative to defensive programming [20]; it is also in line with existing literature on runtime verification [13]. As mentioned above, an interesting research direction is to adapt our session monitorability framework to *suppressions* [9], *i.e.,* by dropping invalid messages without halting the monitor, as in [17].

Finally, we plan to investigate how to handle violations by adding *compensations* to our session types – *i.e.,* by formalising how the protocol should proceed if a violation is detected at a certain stage. In this setting, the monitors would play a more active role in handling violations, and their synthesis would need to be more sophisticated; this new research could be related to the work on session recovery [49].

References

1 Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. Monitoring for silent actions. In *FSTTCS*, volume 93 of *LIPIcs*, pages 7:1–7:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

2 Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. A framework for parameterized monitorability. In *FoSSaCS*, volume 10803 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2018.

3 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Adventures in monitorability: from branching to linear time and back again. *Proc. ACM Program. Lang.*, 3(POPL):52:1–52:29, 2019. `doi:10.1145/3290365`.

4 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. An operational guide to monitorability. In *SEFM*, volume 11724 of *Lecture Notes in Computer Science*, pages 433–453. Springer, 2019. `doi:10.1007/978-3-030-30446-1_23`.

5 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. The best a monitor can do. In *CSL*, volume 183 of *LIPIcs*, pages 7:1–7:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

6 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. An operational guide to monitorability with applications to regular properties. *Softw. Syst. Model.*, 20(2):335–361, 2021. `doi:10.1007/s10270-020-00860-z`.

7 Luca Aceto, Duncan Paul Attard, Adrian Francalanza, and Anna Ingólfsdóttir. A choreographed outline instrumentation algorithm for asynchronous components. *CoRR*, abs/2104.09433, 2021. URL: `https://arxiv.org/abs/2104.09433`.

8 Luca Aceto, Duncan Paul Attard, Adrian Francalanza, and Anna Ingólfsdóttir. On benchmarking for concurrent runtime verification. In *FASE*, volume 12649 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2021.

9 Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir. On runtime enforcement via suppressions. In *CONCUR*, volume 118 of *LIPIcs*, pages 34:1–34:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.CONCUR.2018.34`.

10 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *POPL*, pages 201–214. ACM, 2011. `doi:10.1145/1926385.1926409`.

11 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016. `doi:10.1561/2500000031`.

12 Duncan Paul Attard and Adrian Francalanza. Trace partitioning and local monitoring for asynchronous components. In *SEFM*, volume 10469 of *Lecture Notes in Computer Science*, pages 219–235. Springer, 2017.

13 Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In *Lectures on Runtime Verification*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018. `doi:10.1007/978-3-319-75632-5_1`.

14 David A. Basin, Thibault Dardinier, Lukas Heimes, Srdan Krstic, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. A formally verified, optimized monitor for metric first-order dynamic logic. In *IJCAR (1)*, volume 12166 of *Lecture Notes in Computer Science*, pages 432–453. Springer, 2020. `doi:10.1007/978-3-030-51074-9_25`.

15 Andreas Bauer and Yliès Falcone. Decentralised LTL monitoring. *Formal Methods Syst. Des.*, 48(1-2):46–93, 2016.

16 Jeremy Blackburn, Ivory Hernandez, Jay Ligatti, and Michael Nachtigal. Completely subtyping iso-recursive types. Technical Report CSE-071012, University of South Florida, 2012.

17 Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theor. Comput. Sci.*, 669:33–58, 2017. doi:10.1016/j.tcs.2017.02.009.

18 Alan Brown, Jerry Fishenden, and Mark Thompson. *API Economy, Ecosystems and Engagement Models*, pages 225–236. Palgrave Macmillan UK, London, 2014. doi:10.1057/9781137443649_13.

19 Christian Batrolo Burlò, Adrian Francalanza, and Alceste Scalas. On the monitorability of session types, in theory and practice (extended version), 2021. arXiv:2105.06291.

20 Francesco Cesarini and Simon Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

21 Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, volume 7173 of *Lecture Notes in Computer Science*, pages 25–45. Springer, 2011. doi:10.1007/978-3-642-30065-3_2.

22 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the preciseness of subtyping in session types. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:12)2017.

23 Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *IEEE Symposium on Security and Privacy*, pages 813–830. IEEE Computer Society, 2015. doi:10.1109/SP.2015.55.

24 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. *Formal Methods Syst. Des.*, 46(3):197–225, 2015. doi:10.1007/s10703-014-0218-8.

25 Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. *Formal Methods Syst. Des.*, 54(3):336–363, 2019.

26 Adrian Francalanza. A theory of monitors - (extended abstract). In *FoSSaCS*, volume 9634 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2016. doi:10.1007/978-3-662-49630-5_9.

27 Adrian Francalanza. Consistently-detecting monitors. In *CONCUR*, volume 85 of *LIPIcs*, pages 8:1–8:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CONCUR.2017.8.

28 Adrian Francalanza. A Theory of Monitors. *Information and Computation*, page 104704, 2021. doi:10.1016/j.ic.2021.104704.

29 Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. A foundation for runtime monitoring. In *RV*, volume 10548 of *Lecture Notes in Computer Science*, pages 8–29. Springer, 2017. doi:10.1007/978-3-319-67531-2_2.

30 Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. Monitorability for the hennessy-milner logic with recursion. *Formal Methods Syst. Des.*, 51(1):87–116, 2017. doi:10.1007/s10703-017-0273-z.

31 Adrian Francalanza and Jasmine Xuereb. On implementing symbolic controllability. In *COORDINATION*, volume 12134 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2020. doi:10.1007/978-3-030-50029-0_22.

32 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *J. Log. Algebraic Methods Program.*, 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.

33 Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for asynchronous multiparty sessions. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi:10.1145/3434297.

34 Hannah Gommerstadt, Limin Jia, and Frank Pfenning. Session-typed concurrent contracts. In *ESOP*, volume 10801 of *Lecture Notes in Computer Science*, pages 771–798. Springer, 2018. `doi:10.1007/978-3-319-89884-1_27`.

35 Ruben Hamers and Sung-Shik Jongmans. Discourje: Runtime verification of communication protocols in clojure. In *TACAS (1)*, volume 12078 of *Lecture Notes in Computer Science*, pages 266–284. Springer, 2020. `doi:10.1007/978-3-030-45190-5_15`.

36 Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. `doi:10.1007/3-540-57208-2_35`.

37 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. `doi:10.1007/BFb0053567`.

38 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008. `doi:10.1145/1328438.1328472`.

39 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. `doi:10.1145/2827695`.

40 Raymond Hu, Rumyana Neykova, Nobuko Yoshida, Romain Demangeon, and Kohei Honda. Practical interruptible conversations - distributed dynamic verification with session types and python. In *RV*, volume 8174 of *Lecture Notes in Computer Science*, pages 130–148. Springer, 2013. `doi:10.1007/978-3-642-40787-1_8`.

41 Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *Proc. ACM Program. Lang.*, 1(ICFP):38:1–38:28, 2017. `doi:10.1145/3110282`.

42 Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and blame assignment for higher-order session types. In *POPL*, pages 582–594. ACM, 2016. `doi:10.1145/2837614.2837662`.

43 Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005. `doi:10.1007/s10207-004-0046-8`.

44 Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. On subtyping-relation completeness, with an application to iso-recursive types. *ACM Trans. Program. Lang. Syst.*, 39(1):4:1–4:36, 2017. `doi:10.1145/2994596`.

45 Kim Guldstrand Larsen Luca Aceto, Anna Ingólfsdóttir and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press, 2007.

46 Hernán C. Melgratti and Luca Padovani. Chaperone contracts for higher-order sessions. *Proc. ACM Program. Lang.*, 1(ICFP):35:1–35:29, 2017. `doi:10.1145/3110279`.

47 Network Working Group. RFC 5321: Simple Mail Transfer Protocol. `https://tools.ietf.org/html/rfc5321`, 2008.

48 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in f#. In *CC*, pages 128–138. ACM, 2018. `doi:10.1145/3178372.3179495`.

49 Rumyana Neykova and Nobuko Yoshida. Let it recover: multiparty protocol-induced recovery. In *CC*, pages 98–108. ACM, 2017. `doi:10.1145/3033019`.

50 Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: local verification of global protocols. In *RV*, volume 8174 of *Lecture Notes in Computer Science*, pages 358–363. Springer, 2013. `doi:10.1007/978-3-642-40787-1_25`.

51 Doron A. Peled. Specification and verification using message sequence charts. *Electron. Notes Theor. Comput. Sci.*, 65(7):51–64, 2002. `doi:10.1016/S1571-0661(04)80484-5`.

52 Iain Phillips. Refusal testing. *Theor. Comput. Sci.*, 50:241–284, 1987.

53 Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

54 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *ECOOP*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.ECOOP.2017.24`.

55 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming (artifact). *Dagstuhl Artifacts Ser.*, 3(2):03:1–03:2, 2017. doi:10.4230/DARTS.3.2.3.

56 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *ECOOP*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.21.

57 Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000. doi:10.1145/353323.353382.

58 Scribble homepage, 2020. URL: http://www.scribble.org.

59 SecuritySpace. Mail (MX) server survey, 2021. URL: http://www.securityspace.com/s_survey/data/man.202103/mxsurvey.html.

60 Paula Severi and Mariangiola Dezani-Ciancaglini. Observational equivalence for multiparty sessions. *Fundam. Informaticae*, 170(1-3):267–305, 2019. doi:10.3233/FI-2019-1863.

61 Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *ISSTA*, pages 174–184. ACM, 2007. doi:10.1145/1273463.1273487.

62 Fu Song and Tayssir Touili. Model-checking software library API usage rules. In *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2013. doi:10.1007/978-3-642-38613-8_14.

63 Lucas Waye, Stephen Chong, and Christos Dimoulas. Whip: higher-order contracts for modern services. *Proc. ACM Program. Lang.*, 1(ICFP):36:1–36:28, 2017.

64 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *TGC*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi:10.1007/978-3-319-05119-2_3.