







Towards Probabilistic Session-Type Monitoring

Christian Bartolo Burlò¹ , Adrian Francalanza² , Alceste Scalas³ ,
Catia Trubiani¹ , and Emilio Tuosto¹ 

¹ Gran Sasso Science Institute, L'Aquila, Italy
`christian.bartolo@gssi.it`

² Department of Computer Science, University of Malta, Msida, Malta

³ DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Abstract. We present a tool-based approach for the runtime analysis of communicating processes grounded on probabilistic binary session types. We synthesise a monitor out of a probabilistic session type where each choice point is augmented with a probability distribution. The monitor observes the execution of a process, infers its probabilistic behaviour and issues warnings when the observed behaviour deviates from the one specified by the probabilistic session type.

Keywords: Runtime Verification · Probabilistic session types · Monitor Synthesis

1 Introduction

Communication is central to present day computation. The expected communication protocol between two parties can be formalised as a (*binary*) *session type*, typically describing qualitative aspects such as the order and choice of service invocations at their corresponding payloads. In recent work, *quantitative* aspects of the communication protocol are also layered over a session type [8, 19].

Example 1. Consider a server hosting a guessing game by selecting an integer n between 1 and 100. A client can repeatedly (*i*) try to guess, (*ii*) ask for a hint, or (*iii*) quit the game. The expected interaction sequence of the guessing game server can be specified with the session type S_{game} below:

$$S_{game} = \text{rec } X. \& \left\{ \begin{array}{l} ?\text{Guess}(\text{Int})[0.75]. \oplus \left\{ \begin{array}{l} !\text{Correct}[0.01].X, \\ !\text{Incorrect}[0.99].X \end{array} \right\}, \\ ?\text{Help}[0.2]. !\text{Hint}(\text{Str})[1].X, \\ ?\text{Quit}[0.05].\text{end} \end{array} \right\}$$

The 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 MIUR projects PRIN 2017FTXR7S IT MATTERS and 2017TWR CNB SEDUCE; the EU Horizon 2020 project 830929 *CyberSec4Europe*; the Danish Industriens Fonds Cyberprogram 2020-0489 *Security-by-Design in Digital Denmark*.

© IFIP International Federation for Information Processing 2021

Published by Springer Nature Switzerland AG 2021

F. Damiani and O. Dardha (Eds.): COORDINATION 2021, LNCS 12717, pp. 106–120, 2021.

https://doi.org/10.1007/978-3-030-78142-2_7

The server waits for the client’s choice (at the external branching point $\&$) to either **Guess** a number, ask for **Help**, or **Quit**. If the client asks for help then the server replies with a **Hint** message including a string and the session loops. After the outcome of a guess, (described by the internal choice \oplus) is communicated to the client, the protocol recurs. The type for the client is *dual* and denoted by $\overline{S_{game}}$: each ‘ \oplus ’ is swapped with ‘ $\&$ ’ and each ‘!’ is swapped with ‘?’.

Besides enforcing that parties follow a certain communication pattern, our (augmented) session types also specify some quantitative aspects of the protocol. For instance, S_{game} above specifies that the server should give the client a realistic chance of guessing correctly (*i.e.*, 1%), while $\overline{S_{game}}$ specifies that the client should request for help 20% of the time. According to this augmented specification with quantitative requirements, a burst of client **Help** requests without any attempts to guess the correct answer (that far exceeds 20% of total requests) would constitute a violation of the protocol. More specifically, a substantial deviation from the expected behaviour could be seen as an indicator of abnormal behaviours such as an attempted denial-of-service attack. \diamond

Session types (and their probabilistic variants) are usually checked statically, by type checking the code of the interacting parties (*e.g.*, the clients and the server in this case). However, in an open network, it is common for one or more interacting parties *not* to be available for analysis in a classical pre-deployment fashion. There are even cases where, although we have full access to the participants, it is hard to statically verify their behaviour (*e.g.*, when a client is a human being, or generated via machine-learning techniques). This forces the verifier to carry out certain correctness checks in a post-deployment phase of software production. Recent work has shown that detections and enforcements of qualitative process properties expressed in terms of automata-like formalisms can be carried out effectively, at runtime, using monitors [1, 4–6, 16]. There are however limits to a monitoring approach for verification [2, 14]. At runtime, a monitor (*i*) cannot observe more than one execution, (*ii*) can only observe finite prefixes of a (possibly infinite) complete execution, and (*iii*) cannot detect execution branches that could never have been taken (*i.e.*, the analysis is evidence-based). These constraints make it unclear whether quantitative behavioural aspects such as the observation of branching probabilities, defined over complete (*i.e.*, potentially infinite) sequences of interactions, can be adequately monitored at runtime. Note that, in order to determine (with absolute certainty) whether the client-server interactions will observe the probabilities prescribed in S_{game} above at runtime, one needs access to the code for *both* the server *and* the client. Having access to just the source code of the server—as it is reasonable to expect when the specification is dictated by the party providing the service—does not help us in determining whether the probabilities at the external branching point $\&$ will be observed, since these depend on the choices made by the client (which is often determined at runtime).

In this paper, we develop a tool-supported methodology for the runtime monitoring of quantitative behaviour for two interacting components. In particular, given a Probabilistic Session Type (PST) such as S_{game} above, we synthesise a

monitor that, at runtime, *(i)* observes the messages exchanged within the protocol to ensure that they follow the protocol prescribed by the session type, *(ii)* estimates the probabilistic behaviour of the interacting parties from the interactions observed at runtime, and *(iii)* determines whether to issue a warning for behavioural deviations from the branching probabilities prescribed in the PST, up to a pre-specified level of confidence. For generality, we target the scenario with the weakest level of assumptions, namely where the monitor and its instrumentation are oblivious to the actual implementation of *both* interacting parties participating in the session. Nevertheless, our solution still applies to cases where we have access to the source of the interacting parties.

The rest of the paper is structured as follows. Section 2 explains our methodology in detail. This lays the necessary foundations for the construction of our monitoring tool, described in Sect. 3. Comparisons to related work and discussions about possible future work are given in Sect. 4. To the best of our knowledge, the work we present here is the first attempt at verifying PSTs via runtime monitoring.

2 Methodology

Our proposed methodology operates post-deployment, where *both* participants in the session are analysed *at runtime*. As mentioned earlier, to maximise its applicability and generality, our methodology does not require any part of the participants' behaviour to be analysed pre-deployment, effectively treating them as *black boxes*. The definite verdict of whether an execution exhibited by a system abides by some probabilistic specification can only be given once it terminates (if at all). Nonetheless, our methodology is able to issue probabilistic judgements from incomplete executions, based on the interactions observed up to that point.

Our runtime analysis employs *online* passive monitors [13, 23]. These are computational entities that run live and observe the *incremental* behaviour of two communicating parties (as the execution proceeds) without affecting their interactions. Although our monitors have no prior information about the actual behaviour of the two parties, they are nevertheless able to:

- (a) approximate on-the-fly the probabilistic behaviour of the interacting parties, iteratively revising the approximation when a new interaction is observed, and
- (b) make (revocable) judgements that are based on the *probability distribution* described by our PSTs, for a preset *confidence level*.

Our methodology is supported by a tool, discussed in Sect. 3, that automatically synthesises a monitor from a PST S . At run-time, the monitor estimates the probabilities for each choice point of S (by observing the messages being sent and received), and determines whether such estimates respect the desired probabilities specified in S . This way, the monitor can apportion blame to the interacting party that has control at the choice point where a potential violation is detected. In the sequel, we present the technical details of our methodology; in Sect. 4 we discuss possible alternatives.

2.1 Probabilistic Session Types (PSTs)

In order to formalise probabilistic protocols, we adopt session types augmented with probability distributions over the choice points ($\&$ and \oplus): they allow us to specify the probability of a particular choice being taken by one of the components interacting in a session. The syntax of our PSTs (from which S_{game} in Sect. 1 is derived) is:

$$\begin{array}{l|l}
 S ::= & \&\{?l_i(s_i)[p_i].S_i\}_{i \in I} & (\textit{external choice}) \\
 & \oplus\{!l_i(s_i)[p_i].S_i\}_{i \in I} & (\textit{internal choice}) \\
 & \textit{rec } X.S & (\textit{recursion}) \\
 & X & (\textit{recursion variable}) \\
 & \textit{end} & (\textit{termination})
 \end{array}$$

In choice points ($\&$ and \oplus) the indexing set I is finite and non-empty, the *choice labels* l_i are pairwise distinct, and the *sorts* s_i range over basic data types (\textit{Int} , \textit{Str} , \textit{Bool} , *etc.*). Every choice point S_j is given a *multinomial distribution* interpretation. We assume that $\sum_{i \in I} p_i = 1$ where every p_i is positive, and represents the probability of selecting the branch labelled by l_i , over *every* choice point of interest S_j . The probabilities prescribed at a choice point impose a behavioural obligation on the interacting party who has control over the selection at that choice point. For instance, at the external choice in Example 1, it is the client that is required to adhere to the probabilities prescribed. As usual, we assume that recursion is guarded, *i.e.*, a recursion variable X can only appear under an external or internal prefix.

2.2 Monitoring Sessions

For the sake of the presentation, we assume that choice points of a PST S are indexed by a finite set of indices $j \in J$, which allows us to uniquely identify each choice point as S_j . Accordingly, we let I_j be a set indexing the labels $l_{i,j}$ of the choice point S_j , and denote the probability assigned at S_j to the branch labelled by $l_{i,j}$ as $p_{i,j}$. Our runtime analysis maintains the following counters:

- c_j : number of times the choice point S_j is observed at run-time;
- $c_{i,j}$: number of times the label $l_{i,j}$ ($i \in I_j$) of choice S_j is taken.

For each $j \in J$, these counters yield the *estimated probability*:

$$\widehat{p}_{i,j} = \frac{c_{i,j}}{c_j} \quad j \in J, i \in I_j \quad (1)$$

Namely, $\widehat{p}_{i,j}$ is the frequency with which the i -th branch $l_{i,j}$ of choice point S_j has been taken *so far*. The monitor continuously updates the estimated probabilities as it observes the interactions taking place while the execution unfolds.

The monitor cannot base its decision to issue a warning only on these estimated probabilities. These could potentially be very inaccurate if either of the

components briefly exhibits sporadic behaviour at any point in time of execution. To assess whether the monitored sequence of interactions has *substantially* deviated from the probabilistic behaviour specified in a session type, the runtime analysis needs to consider how accurate these estimated probabilities are in conveying the observed behaviour of the components. We relate this problem to *statistical inference*, where the sequence of interactions observed up to the current point of execution is a *sample* of the larger population, being the entire (possibly infinite) execution.

There are various established paradigms for statistical inference. Our proposed methodology takes a *frequentist* approach. In particular, we calculate *confidence intervals* (CIs) [20] around each desired probability $p_{i,j}$ in a session type to give an approximation of the expected probabilistic behaviour based on the sample size and a *confidence level* $0 \leq \ell < 1$. For any S-abiding execution that iterates through choice point S_j for c_j times, the interval would contain the acceptable range of values for the estimated probabilities with confidence ℓ . To calculate the CI for a choice point S_j , we first calculate the *standard error* SE on the specified probabilities $p_{i,j}$, which depends on the number of times that choice point c_j has iterated (*i.e.*, the sample size). This is then used to calculate the *maximum acceptable error* E (2) based on the given confidence level ℓ , where the multiplier $Z(\ell)$ is the number of standard deviations of a normal distribution representing the particular branch, covering $(\ell \times 100)\%$ of its values [20].

$$E_{i,j} = Z(\ell) \cdot SE_{i,j} \quad \text{where } SE_{i,j} = \sqrt{\frac{p_{i,j}(1-p_{i,j})}{c_j}} \quad \text{for } j \in J, i \in I_j \quad (2)$$

Having calculated the error $E_{i,j}$, the runtime analysis calculates the confidence interval around $p_{i,j}$ as:

$$[p_{i,j} - E_{i,j}, p_{i,j} + E_{i,j}] \quad (3)$$

If and when an estimated probability $\widehat{p}_{i,j}$ (1) falls *outside* this interval, the proposed runtime analysis for our methodology issues a *warning* implying that:

The estimated probability $\widehat{p}_{i,j}$ has deviated enough from the specified probability $p_{i,j}$ to conclude, with confidence ℓ , that the interacting party responsible for the choice point S_j violates the prescribed probability.

The higher the confidence level ℓ specified, the longer it takes for the maximum error $E_{i,j}$ in (2) to converge [20]. Consequently:

- when a higher confidence ℓ is required, the monitor will have *wider* confidence intervals, hence it needs to collect more evidence (*i.e.*, a larger sample size) in order to issue a warning;
- when a lower confidence ℓ is required, the monitor will have *narrower* confidence intervals, hence it might deem an observed session to deviate substantially from the probabilities specified in S at an earlier point in execution. This means that the monitor may potentially issue spurious warnings.

Importantly, after a warning is issued, the subsequent behaviour of the monitored components might cause the (updated) estimated probabilities to fall back within the confidence intervals. As a result, the monitor may *retract* the warning. The warnings issued by the monitor become irrevocable verdicts only when the session terminates (if at all).

Example 2. Recall S_{game} from Example 1. Assume that a monitor for S_{game} is instantiated with confidence level $\ell = 99.999\%$, and that in the running session, the client's choice ($\&$) has iterated nine times, with the client choosing `Help` five times. Thus, the runtime analysis counters are:

$$c_{\&} = 9 \qquad c_{\text{Help},\&} = 5$$

The monitor calculates the estimated probability $\widehat{p}_{\text{Help},\&} = 0.56$ from these counter values using (1). It then calculates the error $E_{\text{Help},\&} = 0.59$ from (2) (with $Z(\ell) = 4.4172$) for the `Help` branch in S . Using the specified probability $p_{\text{Help},\&} = 0.2$ and $E_{\text{Help},\&}$, it calculates the confidence interval from (3) 0.2 ± 0.59 , that is $[-0.39, 0.79]$. Since the estimated probability $\widehat{p}_{\text{Help},\&} = 0.56$ falls within this confidence interval, the monitor does *not* issue a warning.

Now, assume that the session continues, the external choice point $\&$ is iterated, and the client chooses `Help` eight consecutive times more. This means that the counters of our runtime analysis become:

$$c_{\&} = 17 \qquad c_{\text{Help},\&} = 13$$

From (1), the estimated probability is updated to $\widehat{p}_{\text{Help},\&} = 0.76$. From (2), the monitor also updates the confidence interval. Since $c_{\&}$ is now larger, it yields $E_{\text{Help},\&} = 0.43$ which results in the narrower confidence interval $[-0.23, 0.63]$. At this point, our runtime analysis detects that the estimated $\widehat{p}_{\text{Help},\&}$ falls outside this confidence interval and the corresponding warning is issued.

Note that a monitor for S_{game} with lower confidence level $\ell = 95\%$ (*i.e.*, $Z(\ell) = 1.9599$) would issue a warning earlier, *e.g.*, when $c_{\&} = 9$ and $c_{\text{Help},\&} = 5$. In fact, the lower confidence level would yield $E_{\text{Help},\&} = 0.26$ giving the tighter confidence interval $[-0.06, 0.46]$ which does not include $\widehat{p}_{\text{Help},\&} = 0.56$. \diamond

3 The Tool

We extend the monitoring framework in [5] to implement our probabilistic session type monitors. The implementation is available at:

<https://github.com/chrisbartoloburlo/stmonitor>

The overall approach is depicted in Fig. 1: our tool `synth` generates a passive monitor `mon` (written in Scala) from a probabilistic session type S that behaves as a partial-identity [17]. In addition to carrying out the runtime analysis, such monitors are also tasked with forwarding the messages analysed, offering higher degrees of control for stopping execution once a violation is detected. Accord-

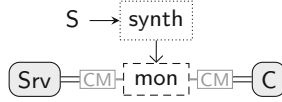


Fig. 1. Outline of monitor synthesis and instrumentation

ingly, the synthesised executable analysis `mon`, is instrumented to act as an intermediary proxy between two interacting components participating in the session, *e.g.*, between a client (`C`) and a server (`Srv`) for our PST S_{game} from Example 2. Internally, the synthesised monitor `mon` uses the `lchannels` library [22] to represent the session type within Scala. To interact with the components, the monitor makes use of user-supplied *connection managers* (`CM`) that sit between the monitor and the components.¹ The connection managers act as *translators* and *gatekeepers* by transforming messages from the transport protocols supported by `C` and `Srv` to the session type representation used by `mon`, and *vice versa*. These allow the monitor synthesis to abstract over the communication protocols in use: *i.e.*, the synthesis is agnostic to the message transportation being used.

The quantitative analysis of the communicated messages applies only if the qualitative aspects of the type are being respected. Similarly to the monitors produced in [5], the code synthesised by our tool can be seen as communicating finite-state machines [7] where states correspond to choice points in a session type. Upon receiving a message, the actual direction of the choice point triggers the analysis of a state’s transition modelling such a choice, potentially producing a warning (or a warning retraction) as a side-effect. Algorithm 1 outlines the logic inside a single state representing a choice point S_j . The synthesis in [5] generates the code that conducts the dynamic typechecking on the messages received, and issues violation verdicts committed to S (line 12). In this work, we augment the synthesis to equip the monitor logic with the ability to conduct the quantitative analysis (lines 3 to 8) as discussed in Sect. 2.

¹ This design is conveniently inherited from [4, 5] but is orthogonal to our approach.

Algorithm 1: Synthesised state of a monitor

```

1 receive choice  $i$  at choice point  $S_j$ 
2   if  $i \in \text{choices } I_j$  then
3     increment counters  $c_j$  and  $c_{i,j}$ 
4     forall choices  $i \in I_j$  calculate
5       if not  $\text{checkInterval}(c_i, c_{i,j}, p_{i,j})$  then
6         | issue a warning blaming the sender
7       else
8         | retract warning
9     forward choice  $i$  to the other side
10    proceed according to the continuation of choice  $i$  in choice point  $j$ 
11    of  $S$ 
12  else
13    | issue a violation verdict

```

The monitors generated by our tool include message counters along with the logic necessary to estimate the probabilities of the choices in a running session. This logic is used to issue warnings whenever the observed (partial) execution deviates from the probabilities in each state S_j . If the message received by the monitor respects the choice point S_j (line 2 in Algorithm 1), it increments the counters of the current choice point, c_j , and that of the choice taken $c_{i,j}$. For every choice within the choice point, the monitor invokes the function *checkInterval* (described in Algorithm 2) to *calculate* and *test* whether the estimated probability falls within the respective confidence interval (line 5). If the estimated probability is not within the interval, the monitor issues a warning and assigns blame to the sender of the current message (line 6). Otherwise, if the probability lies within the interval, the monitor retracts a previously issued warning (line 8). In an effort to minimise unnecessary (repeated) notifications, the monitors generated by our tool only issue (resp. retract) a warning the *first time* an estimated probability transitions outside (resp. inside) the calculated confidence interval. All subsequent notifications are suppressed in case of estimated probabilities that *remain* outside (resp. inside) the interval.

Algorithm 2 is the function implementing the methodology outlined in Algorithm 2. It calculates the estimated probability and confidence interval around the probability specified in the type, based on the counters maintained by the monitor itself. We note that this function can be adapted to other techniques that test whether the behaviour is being respected without affecting the main synthesis of the monitor; see Sect. 4 for further discussions on this point.

Algorithm 2: Function to calculate intervals

```

1 def checkInterval( $c_i, c_{i,j}, p_{i,j}$ ):
2   calculate the estimated probability  $\widehat{p_{i,j}}$  (1) using  $c_i$  and  $c_{i,j}$ 
3   calculate the error  $E_{i,j}$  (2) using  $c_j$  and  $p_{i,j}$ 
4   return ( $\widehat{p_{i,j}}$  in  $[p_{i,j} - E_{i,j}, p_{i,j} + E_{i,j}]$ )

```

Example 3. Recall the PST S_{game} from Example 1 and assume that the monitor synthesised from S_{game} is instantiated with the confidence level $\ell = 99.999\%$. Consider the extended case from Example 2 whereby, after guessing incorrectly 4 times, the client asks for **Help** 13 times; it then guesses correctly 2 consecutive times, which brings the monitor counters of the respective internal choice point (\oplus) to:

$$c_{\oplus} = 6 \qquad c_{\text{Correct},\oplus} = 2 \qquad c_{\text{Incorrect},\oplus} = 4$$

Following the logic in Algorithm 1, after having incremented the counters (line 3), the monitor checks the confidence intervals for *every* choice present in the choice point (line 4). It invokes the function *checkInterval* with the arguments $c_{\oplus}, c_{\text{Correct},\oplus}, p_{\text{Correct},\oplus}$ where $p_{\text{Correct},\oplus} = 0.01$ for the choice **Correct** in S_{game} . The monitor calculates:

$$p_{\text{Correct},\oplus}^{\widehat{\phantom{p_{\text{Correct},\oplus}}}} = 0.33 \qquad E_{\text{Correct},\oplus} = 0.18 \qquad p_{\text{Correct},\oplus} \pm E_{\text{Correct},\oplus} = [-0.17, 0.19]$$

Since $p_{\text{Correct},\oplus}^{\widehat{\phantom{p_{\text{Correct},\oplus}}}}$ is not included in the interval, the function returns **False**. Consequently, the monitor issues a warning blaming the server for sending **Correct** with a probability higher than that specified in S_{game} . Next, the monitor invokes *checkInterval* for the choice **Incorrect** with the arguments $c_{\oplus}, c_{\text{Incorrect},\oplus}, p_{\text{Incorrect},\oplus}$ where $p_{\text{Incorrect},\oplus} = 0.99$. Similarly, the monitor calculates:

$$p_{\text{Incorrect},\oplus}^{\widehat{\phantom{p_{\text{Incorrect},\oplus}}}} = 0.67 \qquad E_{\text{Incorrect},\oplus} = 0.18 \qquad p_{\text{Incorrect},\oplus} \pm E_{\text{Incorrect},\oplus} = [0.81, 1.17]$$

and since $p_{\text{Incorrect},\oplus}^{\widehat{\phantom{p_{\text{Incorrect},\oplus}}}}$ does not lie within the interval, the monitor issues another warning, again blaming the server for this choice.

Consider now the case where the client sends 6 further guesses, to which the server replies with **Incorrect** for all. Therefore, the monitor counters for this choice point are now updated as follows:

$$c_{\oplus} = 12 \qquad c_{\text{Correct},\oplus} = 2 \qquad c_{\text{Incorrect},\oplus} = 10$$

Similarly, the monitor calculates the intervals for both choices:

$$\begin{aligned}
p_{\text{Correct},\oplus}^{\widehat{\phantom{p_{\text{Correct},\oplus}}}} &= 0.17 & E_{\text{Correct},\oplus} &= 0.13 & p_{\text{Correct},\oplus} \pm E_{\text{Correct},\oplus} &= [-0.12, 0.14] \\
p_{\text{Incorrect},\oplus}^{\widehat{\phantom{p_{\text{Incorrect},\oplus}}}} &= 0.67 & E_{\text{Incorrect},\oplus} &= 0.13 & p_{\text{Incorrect},\oplus} \pm E_{\text{Incorrect},\oplus} &= [0.86, 1.12]
\end{aligned}$$

Note that after the monitor observes more messages for this choice point, the intervals shrink for both choices, gradually converging to the specified probabilities. Nonetheless, the estimated probabilities from the observed behaviour

still do not fall within the confidence intervals in both cases. Accordingly, the monitor does not issue any warnings since it had already issued one previously when the potential violation was originally detected.

In fact, the monitor only *retracts* the warnings when the estimated probabilities fall within the interval. Concretely, the client would have to guess incorrectly 6 more times, setting the counters to:

$$c_{\oplus} = 18 \qquad c_{\text{Correct},\oplus} = 2 \qquad c_{\text{Incorrect},\oplus} = 16$$

These counters result in the intervals:

$$\begin{aligned} \widehat{p_{\text{Correct},\oplus}} &= 0.11 & E_{\text{Correct},\oplus} &= 0.1 & p_{\text{Correct},\oplus} \pm E_{\text{Correct},\oplus} &= [-0.09, 0.11] \\ \widehat{p_{\text{Incorrect},\oplus}} &= 0.89 & E_{\text{Incorrect},\oplus} &= 0.1 & p_{\text{Incorrect},\oplus} \pm E_{\text{Incorrect},\oplus} &= [0.89, 1.09] \end{aligned}$$

that both include the estimated probability, causing the monitor to retract the warnings. \diamond

It is often the case that warnings for certain branches have little significance. For instance, in Example 3 above, the monitor also issues a warning for the choice **Incorrect** in addition to that for **Correct**. In practice, one might only be interested in knowing that the server replied **Correct** with a *higher* probability than that specified. To enable such specifications, we enrich the syntax of the probabilistic session types in Sect. 2 with the possibility of using $*$ which specifies to the monitor to not issue warnings for the respective branch or interval boundary.

Example 4. The PST S_{game} from Example 1 can be modified to the type description S'_{game} below:

$$S'_{\text{game}} = \text{rec } X.\& \left\{ \begin{array}{l} ?\text{Guess}(\text{Int})[0.75, *].\oplus \left\{ \begin{array}{l} !\text{Correct}[0.01].X, \\ !\text{Incorrect}[*].X \end{array} \right\}, \\ ?\text{Help}[*, 0.2].!\text{Hint}(\text{Str})[*].X, \\ ?\text{Quit}[*].\text{end} \end{array} \right\}$$

The new type indicates to the monitor *exactly* which choices it should issue warnings for. For the external choice, the monitor should only issue a warning when the estimated probability of the client sending **Guess** is *lower* than 0.75 and that of sending **Help** is *higher* than 0.2, completely ignoring the probability of the choice **Quit**. Similarly, for the internal choice, the monitor should only issue a warning for the choice **Correct**, and suppress those for **Incorrect**. \diamond

With this minor extension to the probabilistic session types we reduce the number of warnings issued and retracted by the monitor. Moreover, we also decrease the amount of computation performed by the monitor at runtime to only those choices that are deemed important. Effectively, this improves the overheads induced by the monitor.

4 Conclusions and Discussion

We have presented a tool-based methodology to analyse specifications augmented with quantitative requirements *at runtime*. More specifically, we extend existing work to implement the synthesis of monitors from probabilistic session types that conduct analysis of the interaction between two parties at runtime. The synthesised monitors issue warnings based only on evidence observed up to the current point of execution while taking into account its accuracy. Notably, the proposed methodology can serve as the basis for other runtime analysis techniques in which the specifications describe any quantitative behaviour.

We conjecture that our approach can be used for systems where protocol-based interactions are replicated in large numbers, and where human intervention is required to ensure their correct execution (*e.g.*, healthcare and fraud detection in e-payments or online gambling). In such applications, our monitors would direct human operation (*i.e.*, the scarce resource) to the cases that have the highest likelihood of exhibiting anomalous behaviour. Another potential application is that of control software that is derived using AI learning techniques. Although effective, such software is often not fully understood and notorious for sudden unexpected behaviour. With our approach, we can automate the monitoring of its interactions and shut off communication whenever the approximated runtime behaviour deviates considerably from that projected.

4.1 Related Work

Our methodology uses PSTs akin to those introduced in [19]. The authors use a type system to *statically* estimate the probability of well-typed processes *reaching successful states*. Notably, types are dynamically checked in our approach and we do not guarantee probabilistic properties; the proposed runtime analysis only issues warnings when the observed behaviour at runtime substantially deviates from the specification. Moreover, our PSTs can also specify behaviour of deterministic systems and are not restricted to probabilistic systems.

Several works apply probabilistic monitoring to minimise the number of runs to be monitored, based on predefined confidence levels [18, 21, 26]. In [24], probability is estimated to check whether the system's behaviour modelled as a hidden Markov model satisfies a temporal property in cases where gaps are present in the execution trace. Unlike these approaches, we use probabilities to specify quantitative aspects of communication protocols which we then check whether they are being respected at runtime.

On the runtime verification of probabilistic systems, the work in [11] models systems as discrete-time Markov chains and expresses requirements using Probabilistic Computation Tree Logic. Their aim is to adapt the behaviour of the underlying system to satisfy non-functional requirements, such as reliability or energy consumption. The work in [10] also monitors Markov Chains whereby monitors are able to verify if a property is satisfied by executing the system and steer it to take certain paths. Similarly to [11], Markov decision processes are used in [12] to model probabilistic systems and optimise the performance

of their verification with the aim of using the results obtained to steer the system. In [10, 11] and [12], the authors adopt incremental verification techniques that exploit the results of previous analyses of the system, whereas our runtime analysis only considers the *current* execution without any prior knowledge on the system. Moreover, we employ monitors that are passive and do not alter nor control the behaviour of the monitored system in any way.

4.2 Future Work

Improving Confidence Interval Estimation. The proposed approach using on confidence intervals described in Sect. 2 serves the goal of instantiating our interpretation of probabilistic session types (Sect. 2.1) on a concrete mechanism for monitors to emit judgements on the probabilistic behaviour of components at runtime. Our approach is not limited, nor bound, to the current statistical inference technique. For instance, we can improve our CI estimation by utilising the Wilson score interval [25], which is more costly but also more reliable than normal approximation when the sample size (observed messages up to the current point of execution) is small or the specified probability is close to 0 or 1. With an easy extension, we can additionally support different confidence levels per choice points. We also plan to study alternative statistical inference paradigms apart from the frequentist approach considered here, such as the Bayesian intervals [3] which would potentially give different interpretations to the PSTs. In turn, this would enable monitors to issue warnings based on the *aggregate recommendations* of the different estimators.

Alternative Interpretations of Probabilistic Session Types. In the proposed approach we opted for an interpretation of probabilistic session types that only considers probabilities at each individual choice point rather than the *global probabilistic behaviour*. We plan to study different interpretations that consider the *dependencies* among choice points in a session type, and also probabilities that are data (payload) dependent.

An intricate aspect of our approach is that it may lead to a potential trade-off between two extremes: taking longer to issue a warning with high confidence, or issue warnings earlier with low confidence. The first extreme corresponds to an increased risk of false negatives: a monitored application could substantially diverge from the observed session type, without being flagged. The second extreme corresponds to an increased risk of false positives: a statistically well-behaving monitored application could be flagged after a minor divergence from the expected frequency of choices. Finding the right value for the confidence level requires careful calibration since it depends on the application and on the relative cost of false negatives when compared to that of false positives.

Application-dependent heuristics together with more advanced interpretations of probabilistic session types can be used to overcome these difficulties. For instance, we could consider the introduction of an observation window of length w , and the probabilities in the session type could refer to a limited number of observed communications regulated by w . Accordingly, the monitor could

keep track of the number of communications observed in the desired window, and use this information to issue warnings in case deviations persist. Concretely, with the interpretation of probabilities described in Sect. 2.1, a client in Example 1 is allowed to send any number of consecutive requests for help, as long as their frequency is not far from 20%, calculated by considering the *entire history* of the session. By introducing a small observation window, we can ensure that two consecutive requests of help would be flagged at any point during the session, regardless of their frequency.

Other Extensions and Improvements. We are in the process of conducting empirical evaluations to assess the effectiveness of our methodology. There are a number of extensions that can be realised relatively easily to improve the tool's flexibility and applicability. For instance, our methodology can be extended so that the probabilities within the session type are (machine-)learnt from a series of observed interactions of different parties. Moreover, monitor (verdict) *explainability* [9, 15] is rapidly gaining importance: our tool can be readily extended to provide useful explanations to support the warnings raised. We are also considering the adaptation of our methodology to the pre-deployment phase of development, thus turning monitors into test drivers that can steer-and-verify implementations.

References

1. Aceto, L., Achilleos, A., Francalanza, A., Ingólfssdóttir, A., Lehtinen, K.: Adventures in monitorability: from branching to linear time and back again. *Proc. ACM Program. Lang.* **3**(POPL), 52:1–52:29 (2019). <https://doi.org/10.1145/3290365>
2. Aceto, L., Achilleos, A., Francalanza, A., Ingólfssdóttir, A., Lehtinen, K.: An operational guide to monitorability with applications to regular properties. *Softw. Syst. Model.* **20**(2), 335–361 (2021). <https://doi.org/10.1007/s10270-020-00860-z>
3. Agresti, A., Hitchcock, D.B.: Bayesian inference for categorical data analysis. *Stat. Methods Appl.* **14**(3), 297–330 (2005). <https://doi.org/10.1007/s10260-005-0121-y>
4. Bartolo Burlò, C., Francalanza, A., Scalas, A.: On the monitorability of session types, in theory and practice. In: 35th European Conference on Object-Oriented Programming, ECOOP 2021, 12–17 July 2021 (2021, to appear)
5. Bartolo Burlò, C., Francalanza, A., Scalas, A.: Towards a hybrid verification methodology for communication protocols (short paper). In: Gotsman, A., Sokolova, A. (eds.) FORTE 2020. LNCS, vol. 12136, pp. 227–235. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50086-3_13
6. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. *Theor. Comput. Sci.* **669**, 33–58 (2017). <https://doi.org/10.1016/j.tcs.2017.02.009>
7. Brand, D., Zafropulo, P.: On communicating finite-state machines. *J. ACM* **30**(2), 323–342 (1983). <https://doi.org/10.1145/322374.322380>
8. Das, A., Wang, D., Hoffmann, J.: Probabilistic resource-aware session types. *CoRR* abs/2011.09037 (2020). <https://arxiv.org/abs/2011.09037>
9. Dawes, J.H., Reger, G.: Explaining violations of properties in control-flow temporal logic. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 202–220. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_12

10. Esparza, J., Kiefer, S., Kretínský, J., Weininger, M.: Online monitoring ω -regular properties in unknown Markov chains. CoRR abs/2010.08347 (2020). <https://arxiv.org/abs/2010.08347>
11. Filieri, A., Tamburrelli, G., Ghezzi, C.: Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. *IEEE Trans. Softw. Eng.* **42**(1), 75–99 (2016). <https://doi.org/10.1109/TSE.2015.2421318>
12. Forejt, V., Kwiatkowska, M., Parker, D., Qu, H., Ujma, M.: Incremental runtime verification of probabilistic systems. In: Qadeer, S., Tasiran, S. (eds.) *RV 2012*. LNCS, vol. 7687, pp. 314–319. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_30
13. Francalanza, A.: A theory of monitors. *Inf. Computa.* 104704 (2021, to appear). <https://doi.org/10.1016/j.ic.2021.104704>
14. Francalanza, A., Aceto, L., Ingólfssdóttir, A.: Monitorability for the Hennessy-Milner logic with recursion. *Formal Methods Syst. Des.* **51**(1), 87–116 (2017). <https://doi.org/10.1007/s10703-017-0273-z>
15. Francalanza, A., Cini, C.: Computer says no: verdict explainability for runtime monitors using a local proof system. *J. Log. Algebraic Methods Program.* **119**, 100636 (2021). <https://doi.org/10.1016/j.jlamp.2020.100636>
16. Francalanza, A., Mezzina, C.A., Tuosto, E.: Towards choreographic-based monitoring. In: Ulidowski, I., Lanese, I., Schultz, U.P., Ferreira, C. (eds.) *RC 2020*. LNCS, vol. 12070, pp. 128–150. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-47361-7_6
17. Gommerstadt, H., Jia, L., Pfenning, F.: Session-typed concurrent contracts. In: Ahmed, A. (ed.) *ESOP 2018*. LNCS, vol. 10801, pp. 771–798. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_27
18. Grunske, L.: An effective sequential statistical test for probabilistic monitoring. *Inf. Softw. Technol.* **53**(3), 190–199 (2011). <https://doi.org/10.1016/j.infsof.2010.10.003>
19. Inverso, O., Melgratti, H.C., Padovani, L., Trubiani, C., Tuosto, E.: Probabilistic analysis of binary sessions. In: *CONCUR. LIPIcs*, vol. 171, pp. 14:1–14:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). <https://doi.org/10.4230/LIPIcs.CONCUR.2020.14>
20. Newcombe, R.G.: *Confidence Intervals for Proportions and Related Measures of Effect Size*. CRC Biostatistics Series. CRC Press, Chapman & Hall, Boca Raton (2012)
21. Ruchkin, I., Sokolsky, O., Weimer, J., Hedao, T., Lee, I.: Compositional probabilistic analysis of temporal properties over stochastic detectors. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst.* **39**(11), 3288–3299 (2020). <https://doi.org/10.1109/TCAD.2020.3012643>
22. Scalas, A., Yoshida, N.: Lightweight session programming in scala. In: *ECOOP. LIPIcs*, vol. 56, pp. 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). <https://doi.org/10.4230/LIPIcs.ECOOP.2016.21>
23. Schneider, F.B.: Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* **3**(1), 30–50 (2000). <https://doi.org/10.1145/353323.353382>
24. Stoller, S.D., et al.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) *RV 2011*. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_15

25. Wilson, E.B.: Probable inference, the law of succession, and statistical inference. *J. Am. Stat. Assoc.* **22**(158), 209–212 (1927). <https://www.tandfonline.com/doi/abs/10.1080/01621459.1927.10502953>
26. Zhu, Y., Xu, M., Zhang, P., Li, W., Leung, H.: Bayesian probabilistic monitor: a new and efficient probabilistic monitoring approach based on Bayesian statistics. In: *QSIC*, pp. 45–54. IEEE (2013). <https://doi.org/10.1109/QSIC.2013.55>