

PSTMonitor: Monitor Synthesis from Probabilistic Session Types

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

¹*Gran Sasso Science Institute, Italy*, ²*Department of Computer Science, University of Malta, Malta*, ³*DTU Compute, Technical University of Denmark, Denmark*

Abstract

We present **PSTMonitor**, a tool for the run-time verification of quantitative specifications of message-passing applications, based on probabilistic session types. The key element of **PSTMonitor** is the detection of executions that deviate from expected probabilistic behaviour. Besides presenting **PSTMonitor** and its operation, the paper analyses its feasibility in terms of the runtime overheads it induces.

Keywords: Runtime Verification, Probabilistic Session Types, Monitor Synthesis

1. Introduction

This paper showcases **PSTMonitor**, a tool supporting the run-time monitoring of quantitative properties of message-passing applications. The underlying methodology of **PSTMonitor**, recently described in the companion paper [1], relies on *(probabilistic) session types* (PST for short). PSTs specify both *(i)* the expected communication protocol; and *(ii)* quantitative constraints on the branching behaviour of the protocol through probabilities, within one unified formalism.

Example 1. Consider a server hosting a guessing game. The server picks an integer n between 1 and 100, and the client is expected to either attempt to guess n , or ask for a hint. In such setup, one might want to assess whether the server behaves fairly and gives the client a chance of guessing correctly. On the other end, one might also want to assess whether the client is asking for too many hints without attempting to guess. \square

The methodology proposed in [1] explores the post-deployment verification of systems such as the one in Example 1 against PSTs. In particular, monitors are used to detect deviations of the current run-time execution of a system from the probabilistic behaviour specified by the PST. Given a PST, our tool **PSTMonitor** automates the synthesis of a monitor that is able to both

20 (i) report violations in case of miscommunications; and (ii) issue warnings
21 if the observed communications deviate substantially from the probabilis-
22 tic behaviour expressed in the PST. One complication is that PSTs specify
23 probabilistic behaviours over *complete* executions, whereas our synthesised
24 monitors need to detect deviations in real-time, hence they can only make
25 decisions based on a *partial* execution observed up to the current instant. For
26 this reason, `PSTMonitor` relies on *confidence intervals* to gauge whether the
27 observed behaviour (up to the current execution point) is compatible with a
28 given PST.

29 The rest of the paper is structured as follows. Section 2 illustrates key
30 functionalities of our tool and the workflow for deploying monitors derived
31 from PSTs. Section 3 discusses the feasibility of our probabilistic monitoring
32 by considering a fragment of the Simple Mail Transfer Protocol (SMTP) [2].
33 Finally, Section 4 draws some conclusions and sketches future work.

34 2. Workflow and Tool Functionality

35 The workflow of `PSTMonitor` starts with a client-server communication
36 protocol expressed as a *probabilistic session type (PST)* [3, 4] — *i.e.*, a (bi-
37 nary) session type where the choice-points are augmented with quantitative
38 information, namely probabilities describing the frequencies of the choices
39 to be taken. The PST is then used to automatically synthesise a monitor
40 which operates w.r.t. a given *confidence level*, as detailed in [1, Section 2.2].
41 Intuitively, the synthesised monitor iteratively calculates *confidence intervals*
42 around all choice-points specified in the PST, and computes frequency-based
43 estimates of the probabilities of the choices observed in the monitored sys-
44 tem. The monitor continuously checks whether the run-time-observed choice
45 probabilities fall within the corresponding PST confidence intervals: if not,
46 the monitor issues a warning, meaning that the observed behaviour is signif-
47 icantly deviating from the PST specification. The monitor can later retract
48 the warning, if the run-time-observed choice probabilities return within the
49 PST confidence intervals.

50 2.1. Specifying behaviour via Probabilistic Session Types (PSTs)

Session types formalise communication protocols by specifying the order
and choice of messages together with their corresponding payloads. We take
binary session types (supporting client-server protocols) and augment the
choice points with probability distributions that specify the frequency with
which a choice should be taken by one of the components interacting in a

session. The syntax of our Probabilistic Session Types (PSTs) is thus:

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

51 In choice points ($\&$ and $+$) the indexing set I is finite and non-empty, the
52 *choice labels* l_i are pairwise distinct, and the *sorts* s_i range over basic data
53 types (`Int`, `Str`, `Bool`, etc.) for typing *variables* x_i . We give a *multinomial*
54 *distribution* interpretation to each choice point ($\&$ and $+$) in a PST: we
55 require that $\sum_{i \in I} p_i = 1$, where every $0 \leq p_i \leq 1$ is the probability of
56 selecting the branch labelled by l_i . The probabilities prescribed at a choice
57 point represent a behavioural obligation on the interacting party that has
58 control over the selection at that choice point. As usual, we require that
59 recursion is guarded, *i.e.*, a recursion variable X can only appear under an
60 external or internal choice prefix.

61 **Example 1.** We formalise the communication protocol outlined in Ex-
62 ample 1 as the PST S_{game} in Figure 1, written from the perspective of the
63 server. The type specifies that the server should wait for the client's choice
64 (at the external branching point $\&$) to either `Guess` a number, ask for `Help`,
65 or `Quit`. If the client asks for help, the server should reply with a `Hint` mes-
66 sage including a string, and the session loops. The session should also loop
67 after the outcome of a guess (`Correct` or `Incorrect`, at the internal choice
68 $+$) is communicated to the client.

69 The probability annotations in S_{game} specify the expected frequency of
70 each choice, and rule out unwanted behaviours. In particular, they require
71 the server to reply with `Correct` 1% of the time, and the client to only
72 request for help 20% of the time. \square

```

1   S_game = rec X.(+{!Guess(num: Int) [0.75].
2       &{?Correct() [0.01].X, ?Incorrect() [0.99].X},
3       !Help() [0.2].?Hint(info: String) [1].X,
4       !Quit() [0.05].end})

```

Figure 1: Probabilistic session type S_{game} .

73 2.2. Monitor synthesis: behind the scenes

74 `PSTMonitor` generates executable session monitors in the Scala program-
75 ming language. However, session type constructs (internal or external choices)



Figure 2: Synthesising and compiling the monitor.

76 are not natively supported by Scala (nor other mainstream programming lan-
 77 guages). For this reason, `PSTMonitor` leverages the library `lchannels` [5],
 78 which encodes (binary) session types constructs in the Scala type system
 79 through *Continuation-Passing Style Protocol classes* (CPSPc), capturing the
 80 order of the send and receive operations and choices in a session type.

81 **Example 2.** As depicted in Figure 2a, our tool `PSTMonitor` generates the
 82 CPSPc from a session type. Listing 1 contains some of the CPSPc from the
 83 autogenerated file representing the type S_{game} from Example 1.

```

84 sealed abstract class ExtChoice1
85 case class Guess(num: Int)(val cont: Out[IntChoice1]) extends ExtChoice1
86 sealed abstract class IntChoice1
87 case class Correct()(val cont: Out[ExtChoice1]) extends IntChoice1
88 case class Incorrect()(val cont: Out[ExtChoice1]) extends IntChoice1
89 // ... remaining classes representing the session type
90 
```

Listing 1: Continuation-Passing Style Protocol classes for S_{game}

92 Intuitively, every class represents a message specified within the session type
 93 (lines 2, 4 and 5), containing the (i) type of the payload; and (ii) the con-
 94 tinuation type. Every choice point is represented by an abstract class (lines
 95 1 and 3) that is inherited by all the choices within the choice point. The
 96 monitor uses these types to keep track of the current point of the interaction
 97 and verify that the components respect the session type. \square

```

98 val guessR = ""GUESS (.*)"".r
99 def receive(): Any = inBuf.readLine() match {
100   case guessR(num) => Guess(num.toInt)(null)
101   // case ...
102   case other => other
103 }
104 def send(msg: Any): Unit = msg match {
105   case Correct() => outB.write(f"CORRECT\n")
106   case Incorrect() => outB.write(f"INCORRECT\n")
107   // case ...
108   case _ => { close()
109                 throw Exception("Invalid msg") }
110 }
111 
```

Listing 2: receive and send methods of CM_c in Figure 3.

The monitors generated by `PSTMonitor` are agnostic to the transport protocol in use: they require user-supplied Connection Managers (CMs) that sit between them and the components under observation (as depicted in Figure 2b). A CM acts as a translator

109 and gatekeeper by transforming messages from the format used in the trans-
 110 port protocol to their respective CPSPc (and *vice versa*), while governing
 111 the interaction with the component. In order to do this, CMs must extend
 112 a provided abstract class (`ConnectionManager`) and implement the methods
 113 `setup`, `close`, `receive` and `send` (as in Listing 2). As the name implies, the
 114 first two manage the connection with the component; the monitor invokes
 115 them when it sets up the connection and before it terminates.

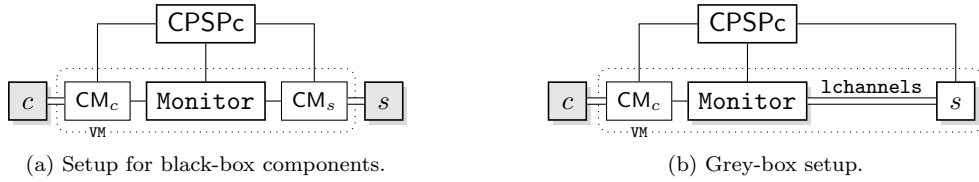


Figure 3: Alternative monitoring setups for a client c and a server s .

116 **Example 3.** Listing 2 shows a code snippet for a CM's `receive` and `send`
 117 methods. In this case, the CM is handling a TCP/IP socket, where the
 118 messages from the type S_{game} in Example 1 are serialised into a textual
 119 format, *e.g.*, `Guess(n)` into "GUESS n ".

120 When invoked by the monitor, the `receive` method checks the socket input
 121 buffer `inBuf`: if it finds a supported message (line 3, where the message
 122 matches the regex `guessR`), it returns an object of the corresponding CPSP
 123 class; otherwise, it returns the unaltered message. Therefore, when the client
 124 c in Figure 3 sends the message "GUESS 23", the CM_c translates it to the
 125 CPSP class `Guess` (line 2 of Listing 1).

126 When the server sends a message to the monitor to be forwarded to the
 127 client, it invokes the method `send` in Listing 2: such a method translates
 128 messages from a CPSP class instance into the format accepted by the client,
 129 and sends them. In this case, if the server replies with a message `Correct`
 130 (line 4 of Listing 1), the server translates it into the textual format "CORRECT"
 131 and writes it on the TCP/IP socket output buffer (line 7 of Listing 2). The
 132 catch-all case (line 10) is used for debugging purposes. \square

133 2.3. Synthesising and using a monitor

134 Our tool `PSTMonitor` automatically generates the monitor code and the
 135 CPSPc from a session type description, by running the following command:

```
136 $ sbt "project monitor" "runMain monitor.Generate $DIR $ST $PRE"
137
```

139 `$DIR` is the directory where the source code of the monitor and classes will
 140 be generated;

141 `$ST` is the file containing the probabilistic session type (as in Figure 1); and

142 `$PRE` (optional) is a file containing a preamble that will be added to the top of
143 the generated files (*e.g.*, containing package declarations and imports).
144 Once completed, the auto-generated files `Monitor.scala` and `CPSPc.scala`
145 will be saved in the directory `$DIR`.

146 2.4. Deploying the monitor

147 The monitor generated in Section 2.3 can be used in a number of different
148 setups. We outline two such setups covering two common situations, depicted
149 in Figure 3:

150 **Black-box monitoring setup (Figure 3a).** This is the most general way
151 to deploy a monitor: both the client *c* and server *s* are treated as black
152 boxes. Here the monitor makes use of two connection managers, one for
153 each end of the interaction. To start the monitor itself, the user needs
154 to write a simple proxy that sits between the monitored client and
155 server, and starts the generated monitor whenever a new connection is
156 established.

157 **Grey-box monitoring setup (Figure 3b).** This setup is possible when
158 one of the components (*e.g.*, the server *s*) is implemented in Scala using
159 `lchannels` and can be deployed together with the monitor. This allows
160 for a direct `lchannels`-based connection between the monitor and the
161 component, without a connection manager. Moreover, the monitor and
162 component can be started on the same Java Virtual Machine (JVM),
163 thus improving the overall performance (as we illustrate in Section 3).¹

164 2.5. Additional Features

165 To further assist with the analysis of a monitored system, monitors gen-
166 erated by `PSTMonitor` can also log information about the current execution,
167 thus enabling, *e.g.*, the real-time visualisation of the monitor status. The logs
168 include (i) the estimated probability of how often a choice within a branch
169 was taken; and (ii) the boundary of the confidence interval.

170 **Example 4.** The plots depicted in Figure 4 are generated from monitor
171 logs: they show the executions of two different clients for the guessing game
172 `PST Sgame` (Example 1). The two clients select the `Help` choice with differ-
173 ent patterns and frequencies. The client in Figure 4a complies with the PST:
174 it selects `Help` with a frequency that always remains within the confidence

¹If a black-box component is supplied as a `.jar` file, it is also possible to deploy such a component and its monitor as different threads running on the same JVM; in this case, the monitor-component interaction would use a TCP/IP socket, which is less efficient than a direct `lchannels`-based connection. The resulting performance would be similar to the black-box scenario measured in Figure 3a.

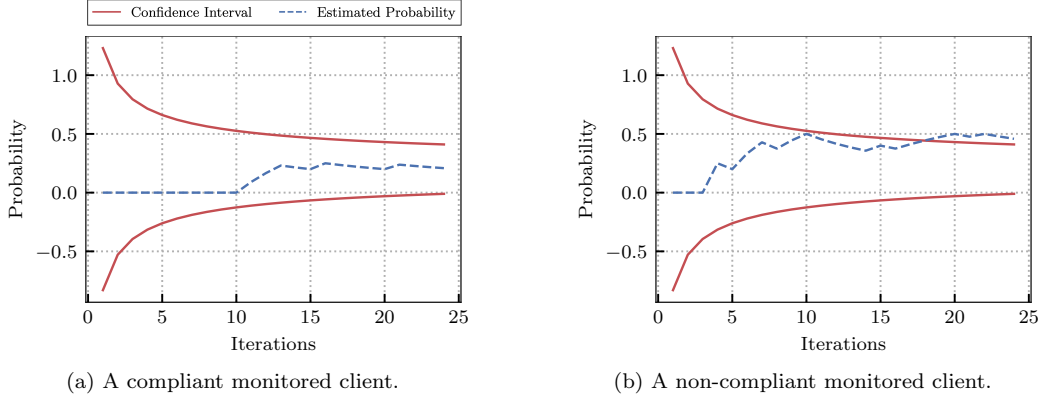


Figure 4: Evolution of the monitoring status for the `Help` branch of S_{game} (Example 1).

175 interval of the probability specified in S_{game} . Instead, the client in Figure 4b
 176 diverges from the PST: it selects `Help` too often, hence the estimated proba-
 177 bility moves outside the confidence interval after a few iterations; when this
 178 happens, the monitor issues a warning. \square

179 3. Evaluation

We evaluate `PSTMonitor` by measuring the overheads induced by the monitors it synthesises. As a benchmark, we consider a probabilistic fragment of the Simple Mail Transfer Protocol (SMTP) [2] (server-side) formalised as the session type S_{smtp} below:

$$\begin{aligned}
 S_{smtp} &= !M220(\text{Str}).\& \left\{ \begin{array}{l} ?\text{Hello}(\text{Str}).!M250(\text{Str}).S_{mail}, \\ ?\text{Quit}.!M221(\text{Str}) \end{array} \right\} \\
 S_{mail} &= \text{rec } X.(\&\{ ?\text{MailFrom}(\text{Str})[0.5].!M250(\text{Str}).\text{rec } Y. \\
 &\quad (\&\left\{ \begin{array}{l} ?\text{RcptTo}(\text{Str})[0.6].!M250(\text{Str}).Y, \\ ?\text{Data}.!M354(\text{Str}).?\text{Content}(\text{Str}).!M250(\text{Str}).X, \\ ?\text{Quit}.!M221(\text{Str}) \end{array} \right\}), \\
 &\quad ?\text{Quit}.!M221(\text{Str}) \})
 \end{aligned} \tag{1}$$

180 When a client establishes a connection, the server sends a welcome message
 181 (`M220`), and waits for the client to identify itself (`Hello`). Then, the client can
 182 recursively send emails by specifying the sender (`MailFrom`) and recipient
 183 (`RcptTo`) address(es), followed by the mail contents (`Data`). The client can
 184 send multiple emails by repeating the loop on “`X`” starting from line (1). The
 185 purpose of the probability annotations in S_{smtp} and S_{mail} is to flag clients that
 186 appear to be sending spam, or using the server resources without a purpose.
 187 Setting a confidence level of 95%, such probability annotations result in a

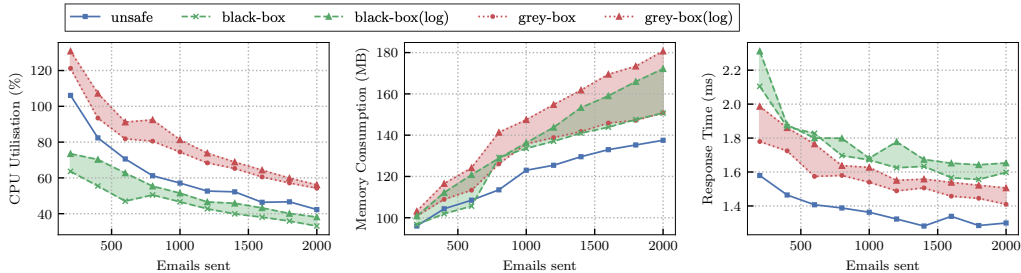


Figure 5: Benchmark results for SMTP monitoring: CPU usage, maximum memory consumption, and response time. (Averages of 20 repetitions; 2 Xeon E5-2687W 8-core CPUs @ 3.10GHz, 128 GB RAM, GNU/Linux 5.10.27)

188 warning if a client (a) sends three or more emails in a single connection
 189 (`MailFrom`), or (b) includes six or more recipients (`RcptTo`).

190 The type S_{smtp} above and the synthesised monitors are not tied to any
 191 specific message transport protocol; we implement them using TCP/IP (as
 192 per [2]) by providing suitable connection managers to the synthesised monitor
 193 (as explained in Section 2). To conduct our experiments, we implement a
 194 client that sends emails to an SMTP server and measures the response time.
 195 As a server, we use a default instance of Postfix² (one of the most used
 196 SMTP servers [6]) configured to receive emails and discard them. To study
 197 the monitoring overheads, we compare:

- 198 - *an unsafe setup*: the SMTP client and server interact directly;
- 199 - *a black-box monitored setup* where communication is mediated by a
 200 monitor instantiated separately, as in Figure 3a; and
- 201 - *a grey-box monitored setup* with the client (written in `Scala+1channels`)
 202 and monitor running on the same virtual machine, as in Figure 3b.

203 We measure the monitor’s response time, CPU utilisation, and maximum
 204 memory consumption, by running experiments where the client sends an
 205 increasing number of emails to the server. We measure the performance with
 206 logging enabled and disabled (as explained in Section 2.5). The results are
 207 shown in Figure 5³.

208 Overall, the plots in Figure 5 fall within the range of typical overheads
 209 introduced by run-time verification [7, 8, 9]. They have two main origins:

- 210 (1) the calculation and bookkeeping of probability estimates at choice points,
 211 and their checking against S_{smtp} ; and

²<http://www.postfix.org/>

³Ideally, the results are also compared with other similar approaches. However, unfortunately, it is very hard to reliably compare overhead figures across different frameworks due to the discrepancies and peculiarities of each setup.

212 (2) the translation and duplication of messages being forwarded between
213 the client and server (which is mitigated in the grey-box setup).

214 The response time is arguably the most important measurement, since slower
215 response times can be immediately perceived when interacting with a moni-
216 tored system: in the black-box monitored setup we measured an overhead of
217 25% (or 30% with logging enabled). Such overhead can be reduced to 13%
218 (or 20% with logging enabled) by adopting a grey-box setup which minimises
219 network communication.

220 4. Conclusions and Discussion

221 We have presented `PSTMonitor`, a tool aimed to analyse quantitative
222 aspects of a system’s interactions at runtime.

223 `PSTMonitor` is implemented as an extension of the monitoring framework
224 `STMonitor` from [10, 11]. Originally, `STMonitor` synthesises monitors from
225 binary session types augmented with assertions (*i.e.*, predicates over commu-
226 nicated values). In order to support probabilistic session types (PSTs), we
227 have replaced the type assertions with probabilities, and we have refactored
228 the synthesis to generate monitors that conduct probabilistic analysis with
229 the methodology introduced in [1] (and here summarised in Section 2). We
230 have also implemented the logging functionality outlined in Section 2.5, thus
231 enabling further insight into the behaviour of a monitored system (as shown
232 in Example 4).

233 Notably, the proposed methodology and the tool instantiating it are
234 independent of each other. While the methodology can serve as the ba-
235 sis for other runtime analysis techniques for quantitative-based specifica-
236 tions, `PSTMonitor` is not limited, nor bound, to the current statistical tech-
237 nique. Rather, the synthesis permits for interchangeable statistical inference
238 paradigms. For instance, we can improve our CI estimation by utilising the
239 Wilson score interval [12] which is more costly but also more reliable when
240 the sample size (observed messages up to the current point of execution) is
241 small or the specified probability is close to 0 or 1.

242 *Related work.* Other publications and tools (also discussed in [1]) propose
243 related techniques or have objectives similar to our work. Albeit addressing
244 the problem from a different angle with different specification languages,
245 these tools and their respective methodologies can complement the analysis
246 conducted by `PSTMonitor`.

247 The work closest to ours is RT-MaC [13], a tool that generates monitors
248 to determine whether a system satisfies a probabilistic property by analysing

249 its behaviour at runtime and performing statistical hypothesis testing. Sim-
250 ilarly to us, they make use of confidence intervals to gauge the accuracy of
251 the observed behaviour. Unlike us, their tool sets up hypotheses from the
252 specified (logic-based) properties and once the monitor has observed enough
253 information to accept or reject the hypotheses, it decides whether the prop-
254 erty is satisfied or not. By adopting the technique from [13], our monitors
255 can be made to reach *irrevocable verdicts* on the observed behaviour, rather
256 than issue retractable warnings.

257 Confidence intervals are also used in [14] for analysing quality of service
258 properties (such as reliability, performance or cost) of a system. Unlike our
259 work, their tool-supported approach is applied in the post-deployment phase
260 on models obtained from system logs or via runtime monitoring. This is done
261 with the aim of establishing unknown behavioural aspects of the system (*e.g.*,
262 how often information is requested from a cache), and since such properties
263 cannot be definitively confirmed with the available information (*i.e.*, a set
264 of observations), confidence intervals are used to give an approximation of
265 the system’s behaviour. Similarly, LogLens [15] analyses system logs to de-
266 tect anomalies; with no (or minimal) knowledge about the system, LogLens
267 uses machine learning based techniques to discover patterns in its behaviour
268 from previous system logs and then compare those in real-time logs to find
269 inconsistencies. On the contrary, our technique does not rely on *any* pre-
270 vious information about the system; moreover, the focus of [14, 15] is not
271 on the probabilistic properties of the system. These tools can complement
272 the methodology enabled by PSTMonitor: given their capability of logging
273 information on the system executions (Section 2.5), our monitors can be used
274 to extract information about the system itself (even when it is a black-box)
275 which can be passed on to tools such as [14, 15] for further analysis.

276 References

- 277 [1] C. Bartolo Burlò, A. Francalanza, A. Scalas, C. Trubiani, E. Tuosto,
278 Towards probabilistic session-type monitoring, in: COORDINATION,
279 Vol. 12717 of Lecture Notes in Computer Science, Springer, 2021, pp.
280 106–120.
- 281 [2] Network Working Group, RFC 5321: Simple Mail Transfer Protocol,
282 <https://tools.ietf.org/html/rfc5321> (2008).
- 283 [3] A. Das, D. Wang, J. Hoffmann, Probabilistic resource-aware session
284 types, CoRR abs/2011.09037 (2020).

- 285 [4] O. Inverso, H. C. Melgratti, L. Padovani, C. Trubiani, E. Tuosto, Prob-
286 abilistic analysis of binary sessions, in: CONCUR, Vol. 171 of LIPIcs,
287 Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 14:1–14:21.
- 288 [5] A. Scalas, N. Yoshida, Lightweight session programming in scala, in:
289 ECOOP, Vol. 56 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für In-
290 formatik, 2016, pp. 21:1–21:28.
- 291 [6] SecuritySpace, Mail (MX) server survey (2021).
292 URL [http://www.securityspace.com/s_survey/data/man.202103/](http://www.securityspace.com/s_survey/data/man.202103/mxsurvey.html)
293 [mxsurvey.html](http://www.securityspace.com/s_survey/data/man.202103/mxsurvey.html)
- 294 [7] E. Bartocci, Y. Falcone, A. Francalanza, G. Reger, Introduction to run-
295 time verification, in: Lectures on Runtime Verification, Vol. 10457 of
296 Lecture Notes in Computer Science, Springer, 2018, pp. 1–33.
- 297 [8] E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker,
298 K. Havelund, Y. Joshi, F. Klaedtke, R. Milewicz, G. Reger, G. Rosu,
299 J. Signoles, D. Thoma, E. Zalinescu, Y. Zhang, First international com-
300 petition on runtime verification: rules, benchmarks, tools, and final re-
301 sults of CRV 2014, *Int. J. Softw. Tools Technol. Transf.* 21 (1) (2019)
302 31–70.
- 303 [9] L. Aceto, D. P. Attard, A. Francalanza, A. Ingólfssdóttir, On benchmark-
304 ing for concurrent runtime verification, in: FASE, Vol. 12649 of Lecture
305 Notes in Computer Science, Springer, 2021, pp. 3–23.
- 306 [10] C. B. Burlò, A. Francalanza, A. Scalas, On the monitorability of session
307 types, in theory and practice, in: ECOOP, Vol. 194 of LIPIcs, Schloss
308 Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 20:1–20:30.
- 309 [11] C. B. Burlò, A. Francalanza, A. Scalas, On the monitorability of session
310 types, in theory and practice (artifact), *Dagstuhl Artifacts Ser.* 7 (2)
311 (2021) 02:1–02:3.
- 312 [12] E. B. Wilson, Probable inference, the law of succession, and
313 statistical inference, *Journal of the American Statistical As-*
314 *sociation* 22 (158) (1927) 209–212. arXiv:[https://www.](https://www.tandfonline.com/doi/pdf/10.1080/01621459.1927.10502953)
315 [tandfonline.com/doi/pdf/10.1080/01621459.1927.10502953,](https://www.tandfonline.com/doi/pdf/10.1080/01621459.1927.10502953)
316 [doi:10.1080/01621459.1927.10502953.](https://www.tandfonline.com/doi/pdf/10.1080/01621459.1927.10502953)
317 URL [https://www.tandfonline.com/doi/abs/10.1080/01621459.](https://www.tandfonline.com/doi/abs/10.1080/01621459.1927.10502953)
318 [1927.10502953](https://www.tandfonline.com/doi/abs/10.1080/01621459.1927.10502953)

- 319 [13] U. Sammapun, I. Lee, O. Sokolsky, RT-MaC: Runtime Monitoring and
320 Checking of Quantitative and Probabilistic Properties, in: Proceedings
321 of the International Conference on Embedded and Real-Time Comput-
322 ing Systems and Applications (RTCSA), 2005, pp. 147–153.
- 323 [14] R. Calinescu, C. Ghezzi, K. Johnson, M. Pezzé, Y. Rafiq, G. Tambur-
324 relli, Formal verification with confidence intervals to establish quality of
325 service properties of software systems, *IEEE Transactions on Reliability*
326 65 (1) (2015) 107–125.
- 327 [15] B. Debnath, M. Solaimani, M. A. G. Gulzar, N. Arora, C. Lumezanu,
328 J. Xu, B. Zong, H. Zhang, G. Jiang, L. Khan, Loglens: A real-time
329 log analysis system, in: Proceedings of the International Conference on
330 Distributed Computing Systems (ICDCS), 2018, pp. 1052–1062.

331 **Current code version**

Nr.	Code metadata description	Please fill in this column
C1	Current code version	v0.0.1
C2	Permanent link to code/repository used for this code version	https://github.com/chrisbartoloburlo/stmonitor/tree/pstmonitor
C3	Permanent link to Reproducible Capsule	
C4	Legal Code License	BSD-2-Clause License
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	Scala, Python
C7	Compilation requirements, operating environments & dependencies	sbt
C8	If available Link to developer documentation/manual	https://github.com/chrisbartoloburlo/stmonitor/blob/pstmonitor/README.md
C9	Support email for questions	christian.bartolo@gssi.it

Table 1: Code metadata (mandatory)