`PSTMonitor`: Monitor Synthesis from Probabilistic Session Types

Christian Bartolo Burlò[1], Adrian Francalanza[2], Alceste Scalas[3], Catia Trubiani[1], Emilio Tuosto[1]

[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. □

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

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

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

2. Workflow and Tool Functionality

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

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:

$$
\begin{array}{rll}
\mathsf{S} & ::= & \&\left\{?\mathtt{l}_i(\mathsf{s}_i)[p_i].\mathsf{S}_i\right\}_{i\in I} \quad \textit{(external choice)} \\
& \mid & +\left\{!\mathtt{l}_i(\mathsf{s}_i)[p_i].\mathsf{S}_i\right\}_{i\in I} \quad \textit{(internal choice)} \\
& \mid & \mathsf{rec}\ X.\mathsf{S} \quad \textit{(recursion)} \\
& \mid & X \quad \textit{(recursion variable)} \\
& \mid & \mathsf{end} \quad \textit{(termination)}
\end{array}
$$

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

Example 1. We formalise the communication protocol outlined in Example 1 as the PST S_{game} in Figure 1, written from the perspective of the server. The type specifies that the server should wait 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, the server should reply with a `Hint` message including a string, and the session loops. The session should also loop after the outcome of a guess (`Correct` or `Incorrect`, at the internal choice +) is communicated to the client.

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

```
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`.

2.2. Monitor synthesis: behind the scenes

`PSTMonitor` generates executable session monitors in the Scala programming language. However, session type constructs (internal or external choices)

(a) Monitor and CPSPc synthesis.

(b) Type-checking implementations.

Figure 2: Synthesising and compiling the monitor.

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

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

```
1 sealed abstract class ExtChoice1
2 case class Guess(num: Int)(val cont: Out[IntChoice1]) extends ExtChoice1
3 sealed abstract class IntChoice1
4 case class Correct()(val cont: Out[ExtChoice1]) extends IntChoice1
5 case class Incorrect()(val cont: Out[ExtChoice1]) extends IntChoice1
6 // ... remaining classes representing the session type
```

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

Intuitively, every class represents a message specified within the session type (lines 2, 4 and 5), containing the *(i)* type of the payload; and *(ii)* the continuation type. Every choice point is represented by an abstract class (lines 1 and 3) that is inherited by all the choices within the choice point. The monitor uses these types to keep track of the current point of the interaction and verify that the components respect the session type. □

```
1  val guessR = """GUESS (.*)""".r
2  def receive(): Any = inBuf.readLine() match {
3    case guessR(num) => Guess(num.toInt)(null)
4    // case ...
5    case other => other
6  }
7  def send(msg: Any): Unit = msg match {
8    case Correct() => outB.write(f"CORRECT\n")
9    case Incorrect() => outB.write(f"INCORRECT\n")
10   // case ...
11   case _ => { close()
12               throw Exception("Invalid msg") }
13 }
```

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

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

(a) Setup for black-box components.

(b) Grey-box setup.

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

Example 3. Listing 2 shows a code snippet for a CM's `receive` and `send` methods. In this case, the CM is handling a TCP/IP socket, where the messages from the type S_{game} in Example 1 are serialised into a textual format, *e.g.,* `Guess`(n) into `"GUESS` n`"`.
When invoked by the monitor, the `receive` method checks the socket input buffer `inBuf`: if it finds a supported message (line 3, where the message matches the regex `guessR`), it returns an object of the corresponding CPSP class; otherwise, it returns the unaltered message. Therefore, when the client c in Figure 3 sends the message `"GUESS 23"`, the CM_c translates it to the CPSP class `Guess` (line 2 of Listing 1).

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

2.3. Synthesising and using a monitor

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

```
1 $ sbt "project monitor" "runMain monitor.Generate $DIR $ST $PRE"
```

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

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

`$PRE` (optional) is a file containing a preamble that will be added to the top of the generated files (*e.g.,* containing package declarations and imports).

Once completed, the auto-generated files `Monitor.scala` and `CPSPc.scala` will be saved in the directory `$DIR`.

2.4. Deploying the monitor

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

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

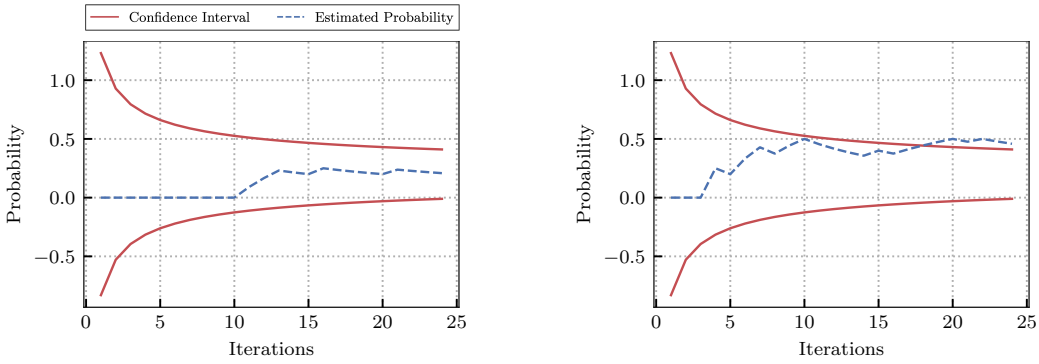
Grey-box monitoring setup (Figure 3b). This setup is possible when one of the components (*e.g.,* the server s) is implemented in Scala using `lchannels` and can be deployed together with the monitor. This allows for a direct `lchannels`-based connection between the monitor and the component, without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

2.5. Additional Features

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

Example 4. The plots depicted in Figure 4 are generated from monitor logs: they show the executions of two different clients for the guessing game PST S_{game} (Example 1). The two clients select the `Help` choice with different patterns and frequencies. The client in Figure 4a complies with the PST: it selects `Help` with a frequency that always remains within the confidence

[1] If 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.



(a) A compliant monitored client.

(b) A non-compliant monitored client.

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

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

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:

$$\mathsf{S}_{smtp} = !\texttt{M220}(\mathsf{Str}).\&\left\{\begin{array}{l} ?\texttt{Helo}(\mathsf{Str}).!\texttt{M250}(\mathsf{Str}).\mathsf{S}_{mail}, \\ ?\texttt{Quit}.!\texttt{M221}(\mathsf{Str}) \end{array}\right\}$$

$$\begin{aligned} \mathsf{S}_{mail} = \mathsf{rec}\ X.\big(\&\big\{&?\texttt{MailFrom}(\mathsf{Str})[0.5].!\texttt{M250}(\mathsf{Str}).\mathsf{rec}\ Y. \\ &\big(\&\left\{\begin{array}{l} ?\texttt{RcptTo}(\mathsf{Str})[0.6].!\texttt{M250}(\mathsf{Str}).Y, \\ ?\texttt{Data}.!\texttt{M354}(\mathsf{Str}).?\texttt{Content}(\mathsf{Str}).!\texttt{M250}(\mathsf{Str}).X, \\ ?\texttt{Quit}.!\texttt{M221}(\mathsf{Str}) \end{array}\right\}\big), \\ &?\texttt{Quit}.!\texttt{M221}(\mathsf{Str})\big\}\big) \end{aligned} \qquad (1)$$

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 (`MailFrom`) and recipient (`RcptTo`) address(es), followed by the mail contents (`Data`). The client can send multiple emails by repeating the loop on "X" starting from line (1). The purpose of the probability annotations in S_{smtp} and S_{mail} is to flag clients that appear to be sending spam, or using the server resources without a purpose. Setting a confidence level of 95%, such probability annotations result in a

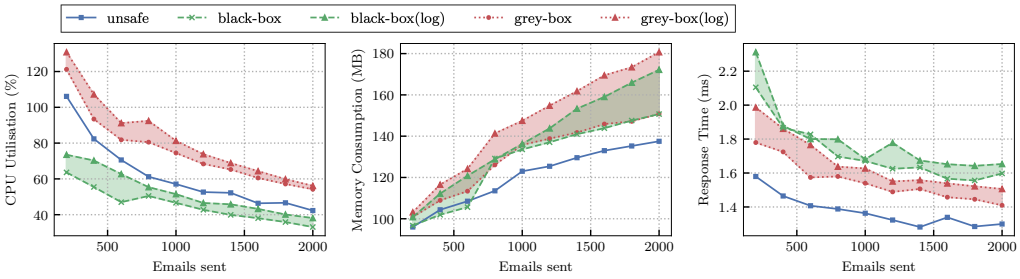


Figure 5: Benchmark results for SMTP monitoring: CPU usage, maximum memory consumption, and response time. (Averages of 20 repetitions; 2 Xeon E5-2687W 8-core CPUs @ 3.10GHz, 128 GB RAM, GNU/Linux 5.10.27)

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

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

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

We measure the monitor's response time, CPU utilisation, and maximum memory consumption, by running experiments where the client sends an increasing number of emails to the server. We measure the performance with logging enabled and disabled (as explained in Section 2.5). The results are shown in Figure 5[3].

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

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

[2] `http://www.postfix.org/`

[3] Ideally, 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.

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

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

4. Conclusions and Discussion

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

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

Notably, the proposed methodology and the tool instantiating it are independent of each other. While the methodology can serve as the basis for other runtime analysis techniques for quantitative-based specifications, `PSTMonitor` is not limited, nor bound, to the current statistical technique. Rather, the synthesis permits for interchangeable statistical inference paradigms. For instance, we can improve our CI estimation by utilising the Wilson score interval [12] which is more costly but also more reliable 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.

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

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

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

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

References

[1] C. Bartolo Burlò, A. Francalanza, A. Scalas, C. Trubiani, E. Tuosto, Towards probabilistic session-type monitoring, in: COORDINATION, Vol. 12717 of Lecture Notes in Computer Science, Springer, 2021, pp. 106–120.

[2] Network Working Group, RFC 5321: Simple Mail Transfer Protocol, `https://tools.ietf.org/html/rfc5321` (2008).

[3] A. Das, D. Wang, J. Hoffmann, Probabilistic resource-aware session types, CoRR abs/2011.09037 (2020).

[4] O. Inverso, H. C. Melgratti, L. Padovani, C. Trubiani, E. Tuosto, Probabilistic analysis of binary sessions, in: CONCUR, Vol. 171 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 14:1–14:21.

[5] A. Scalas, N. Yoshida, Lightweight session programming in scala, in: ECOOP, Vol. 56 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 21:1–21:28.

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

[7] E. Bartocci, Y. Falcone, A. Francalanza, G. Reger, Introduction to runtime verification, in: Lectures on Runtime Verification, Vol. 10457 of Lecture Notes in Computer Science, Springer, 2018, pp. 1–33.

[8] E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker, K. Havelund, Y. Joshi, F. Klaedtke, R. Milewicz, G. Reger, G. Rosu, J. Signoles, D. Thoma, E. Zalinescu, Y. Zhang, First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014, Int. J. Softw. Tools Technol. Transf. 21 (1) (2019) 31–70.

[9] L. Aceto, D. P. Attard, A. Francalanza, A. Ingólfsdóttir, On benchmarking for concurrent runtime verification, in: FASE, Vol. 12649 of Lecture Notes in Computer Science, Springer, 2021, pp. 3–23.

[10] C. B. Burlò, A. Francalanza, A. Scalas, On the monitorability of session types, in theory and practice, in: ECOOP, Vol. 194 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 20:1–20:30.

[11] C. B. Burlò, A. Francalanza, A. Scalas, On the monitorability of session types, in theory and practice (artifact), Dagstuhl Artifacts Ser. 7 (2) (2021) 02:1–02:3.

[12] E. B. Wilson, Probable inference, the law of succession, and statistical inference, Journal of the American Statistical Association 22 (158) (1927) 209–212. arXiv:https://www.tandfonline.com/doi/pdf/10.1080/01621459.1927.10502953, doi:10.1080/01621459.1927.10502953.
URL https://www.tandfonline.com/doi/abs/10.1080/01621459.1927.10502953

[13] U. Sammapun, I. Lee, O. Sokolsky, RT-MaC: Runtime Monitoring and Checking of Quantitative and Probabilistic Properties, in: Proceedings of the International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2005, pp. 147–153.

[14] R. Calinescu, C. Ghezzi, K. Johnson, M. Pezzé, Y. Rafiq, G. Tamburrelli, Formal verification with confidence intervals to establish quality of service properties of software systems, IEEE Transactions on Reliability 65 (1) (2015) 107–125.

[15] B. Debnath, M. Solaimani, M. A. G. Gulzar, N. Arora, C. Lumezanu, J. Xu, B. Zong, H. Zhang, G. Jiang, L. Khan, Loglens: A real-time log analysis system, in: Proceedings of the International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 1052–1062.

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)