University of Udine

Monitors that Learn from Failures

Machine Learning-based Monitoring for Runtime System Verification

Andrea Brunello - andrea.brunello@uniud.it

Introduction

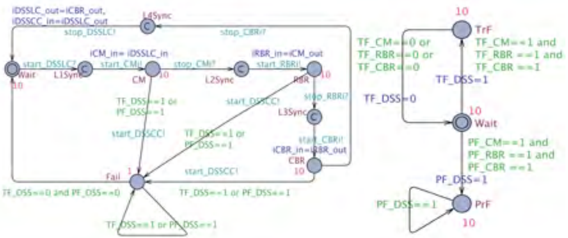
In several domains, systems generate **continuous streams of data** which may contain useful telemetry information

- They can be used for tasks such as predictive maintenance and preemptive failure detection (Industry 4.0)
- System behaviours can be convoluted, being the result of the interaction among several components and the environment
- Given the complexity of this setting, **deep learning** approaches are typically been considered. Problems:
    - resulting models are hardly interpretable
    - difficulty in providing guarantees on the obtained results

In critical contexts, formal methods have been recognized as an effective approach to ensure the correct behaviour of a system.

However, classical techniques, such as model checking, require a **complete specification** of the system and of the properties to be checked against it, and work in an **offline** fashion.

-> In some cases, their application can be very difficult!



A Novel Approach

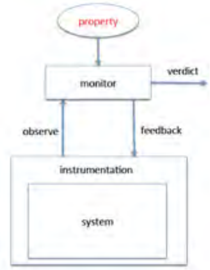
Framework that **combines machine learning and monitoring** to detect critical system behaviours in an on-line setting:

- system behaviour's complexity is dealt with by means of machine learning
- extracted formal properties are interpretable, so a domain expert can easily read and validate the generated model
- the framework is highly modular with respect to the logic used to encode the system properties

Monitoring

Monitoring is a **run-time** verification technique:

- it establishes satisfaction/violation of a property analyzing a finite prefix of **a single run** (trace) of the system
- lightweight technique compared to model checking
- naturally applicable to data streaming contexts



When the monitor reaches a verdict, the latter is definitive.

Positively monitorable properties:

- every system satisfying it features a finite trace witnessing the satisfaction
- $\Diamond(ack)$, at a certain point the system reaches an *ack* state

Negatively monitorable properties:

- every system violating it features a finite trace witnessing the violation
- $\Box(online)$, the system is always *online*

Not all properties are monitorable:

- $\Box(req \rightarrow \Diamond(ack))$, every request submitted to the system ultimately receives an answer

Linear Temporal Logic [Pnueli 1977]

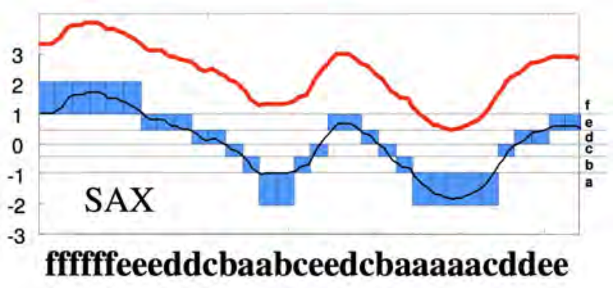
Linear Temporal Logic (LTL) allows one to express temporal properties over linear structures (single computation paths).

$$\varphi := \top \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 U \varphi_2 \mid X\varphi$$

$$\pi, s_i \models p \in P \Leftrightarrow V(p, s_i) = true$$
$$\pi, s_i \models \neg\alpha \iff \pi, s_i \not\models \alpha$$
$$\pi, s_i \models \alpha \wedge \beta \iff \pi, s_i \models \alpha \;\wedge\; \pi, s_i \models \beta$$
$$\pi, s_i \models X\alpha \iff \pi, s_{i+1} \models \alpha$$
$$\pi, s_i \models F\alpha \iff \exists j \geq i \;:\; \pi, s_j \models \alpha$$
$$\pi, s_i \models G\alpha \iff \forall j \geq i \;:\; \pi, s_j \models \alpha$$
$$\pi, s_i \models \alpha U \beta \iff \exists j \geq i \;:\; \pi, s_j \models \beta \text{ e } \forall k \;\; s.t. \;\; i \leq k < j \;:\; \pi, s_k \models \alpha$$

While being very intuitive, LTL cannot handle continuous time series data, but a preliminary discretization step is needed.

SAX (Symbolic Aggregate approXimation) transforms a real-valued time series into a discrete sequence of states



$G(d \rightarrow F(e))$

LTL: Reduction to Finite Model Checking

- *BayesLTL* is a tool for LTL finite model checking and property extraction
- Nevertheless, the solution does not provide any support for monitoring
- Thus, we extended it with such a capability, and we devised a reduction from monitoring to finite model checking
- Since a finite model checking algorithm returns a Boolean answer ($\top/\bot$), while a monitor can also provide an undefined one (?), we first gave a transformation τ from an LTL formula φ to a pair of LTL formulas $\tau(\varphi) = \langle \varphi_1, \varphi_2 \rangle$
- Then we showed that monitoring φ against a given trace amounts to applying the finite model checking algorithm to φ_1 and φ_2, and suitably interpreting the outcomes

Reduction to Finite MC – Details

In what follows, given a pair of formulas $\tau(\varphi) = \langle \varphi_1, \varphi_2 \rangle$, we denote by $\tau(\varphi)_{|_1}$ (resp., $\tau(\varphi)_{|_2}$) the first (resp., second) formula of the pair, that is, $\tau(\varphi)_{|_i} = \varphi_i$ ($i \in \{1, 2\}$)

Definition

The mapping $\tau : LTL \to LTL \times LTL$ is inductively defined as:

1. $\tau(p) = \langle p, p \rangle$, for all $p \in \mathcal{AP}$;
2. $\tau(\neg\psi) = \langle \neg\tau(\psi)_{|_1} \wedge \neg\tau(\psi)_{|_2}, \neg\tau(\psi)_{|_1} \rangle$;
3. $\tau(\psi \vee \xi) = \langle \tau(\psi)_{|_1} \vee \tau(\xi)_{|_1}, \tau(\psi)_{|_2} \vee \tau(\xi)_{|_2} \rangle$;
4. $\tau(X\psi) = \langle X\tau(\psi)_{|_1}, X\tau(\psi)_{|_2} \rangle$;
5. $\tau(\psi U \xi) = \langle \tau(\psi)_{|_1} U \tau(\xi)_{|_1}, ((\tau(\psi)_{|_1} \vee \tau(\psi)_{|_2})\ U(\tau(\xi)_{|_1} \vee \tau(\xi)_{|_2})) \vee G(\tau(\psi)_{|_1} \vee \tau(\psi)_{|_2}) \rangle$.

Now, it is possible to reduce the monitoring problem to the finite model checking problem as follows:

Theorem

Let φ be an LTL formula and $\pi \in \Sigma^*$ be a finite trace. It holds:

1. *monitoring*(π, φ) *returns* $\top$ *iff* $\pi \models \tau(\varphi)_{|1}$,
2. *monitoring*(π, φ) *returns* $\bot$ *iff* $\pi \not\models \tau(\varphi)_{|1}$ *and* $\pi \not\models \tau(\varphi)_{|2}$, *and*
3. *monitoring*(π, φ) *returns ? iff* $\pi \not\models \tau(\varphi)_{|1}$ *and* $\pi \models \tau(\varphi)_{|2}$.

Given an LTL formula φ and a finite trace π, the monitoring algorithm transforms φ in $\tau(\varphi) = \langle \varphi_1, \varphi_2 \rangle$. Then, it runs the *BayesLTL* finite model checking procedure to check φ_1 against π: if $\pi \models \varphi_1$, then $\top$ is returned. Otherwise, *BayesLTL* is run again to check φ_2 against π: if $\pi \not\models \varphi_2$, then $\bot$ is returned; otherwise, the monitor returns an undefined verdict (?)

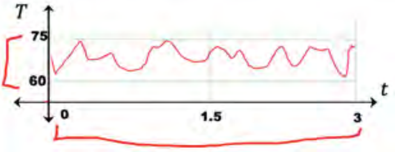
Signal Temporal Logic [Maler et al. 2004]

Signal Temporal Logic (STL) is an extension of LTL with *real-time* and *real-valued* constraints

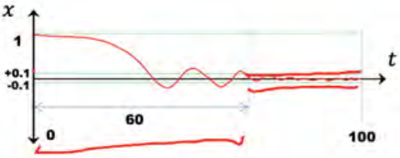
φ ::=	$f(\mathbf{x}) \sim 0$	|	$f: \mathbb{D} \rightarrow \mathbb{R}$ is a function over the signal $\mathbf{x}$: $\mathbb{T} \rightarrow \mathbb{D}$, $\sim \in \{\leq, <, >, \geq, =, \neq\}$
	$\neg\varphi$	|	Negation
	$\varphi \wedge \varphi$	|	Conjunction
	$\mathbf{F}_{[a,b]}\varphi$	|	At some Future step in the interval $[a, b]$
	$\mathbf{G}_{[a,b]}\varphi$	|	Globally in all times in the interval $[a, b]$
	$\varphi\ \mathbf{U}_{[a,b]}\ \varphi$	|	In all steps Until in interval $[a, b]$
	$\varphi\ \boldsymbol{S}_{[a,b]}\ \varphi$	|	In all steps Since in interval $[a, b]$

STL: Examples

$\mathbf{G}_{[0,3]}((T > 60) \wedge (T < 75))$
always, between time 0 and time 3, $60 < T < 75$



$\mathbf{F}_{[0,60]}(\mathbf{G}(|x| < 0.1))$
eventually, at some time t between 0 and 60, from t on, $|x| < 0.1$



The satisfaction of a formula φ by a (multivariate) signal $x = (x_1, \ldots, x_n)$ at time t is given by:

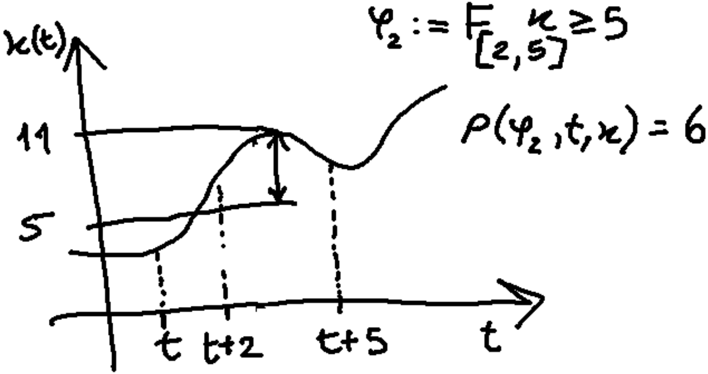
$$
\begin{array}{lcl}
(\mathbf{x}, t) \models \mu & \Leftrightarrow & f(x_1[t], \ldots, x_n[t]) > 0 \\
(\mathbf{x}, t) \models \varphi \wedge \psi & \Leftrightarrow & (x, t) \models \varphi \wedge (x, t) \models \psi \\
(\mathbf{x}, t) \models \neg\varphi & \Leftrightarrow & \neg((x, t) \models \varphi) \\
(\mathbf{x}, t) \models \varphi \, \mathcal{U}_{[a,b]} \, \psi & \Leftrightarrow & \exists t' \in [t+a, t+b] \text{ such that } (x, t') \models \psi \wedge \\
 & & \qquad \forall t'' \in [t, t'], \ (x, t'') \models \varphi\}
\end{array}
$$

Note that:

- $\mathbf{F}_{[a,b]}\varphi = \top \, \mathcal{U}_{[a,b]} \, \varphi$
- $\mathbf{G}_{[a,b]}\varphi = \neg(\mathbf{F}_{[a,b]}\neg\varphi)$

STL also quantifies the *robustness degree* of satisfaction of a formula by a given trace x at time t

$$
\begin{aligned}
\rho(\top, x, t) &= +\infty \\
\rho(x_i \geq c, x, t) &= x_i(t) - c \\
\rho(\neg\phi, x, t) &= -\rho(\phi, x, t) \\
\rho(\phi_1 \wedge \phi_2, x, t) &= \min\{\rho(\phi_1, x, t), \rho(\phi_2, x, t)\} \\
\rho(\phi_1 U_I \phi_2, x, t) &= \sup_{t_1 \in t+I} \min\{\rho(\phi_2, x, t_1), \inf_{t_2 \in [t, t_1)} \rho(\phi_1, x, t_2)\}
\end{aligned}
$$

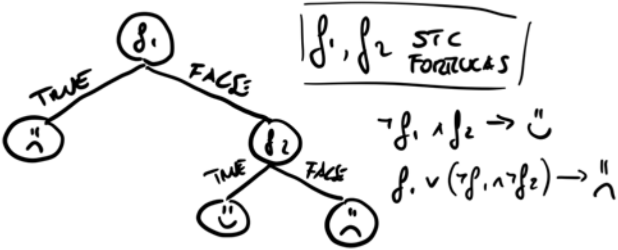

$\varphi_2 := F_{[2,5]}\, x \geq 5$
$\rho(\varphi_2, t, x) = 6$
$x(t)$
11
5
t
t+2
t+5
t

Framework: General Idea

- We consider a pool of monitored properties, that ought to detect or predict system failures in an online fashion
- At each time instant, such properties are checked against the incoming execution trace of the system
- If a failure is detected, the trace is divided into a *good* and a *bad* part, and we look for new properties capable of discerning between such sub-traces
- The new properties are added to the monitoring pool and the monitoring process is resumed
- Intuitively, the framework can be initialized with a very small pool of simple properties
- Then, over time, the pool will be automatically extended with new properties capable of increasing the completeness and preemptiveness of failure detection

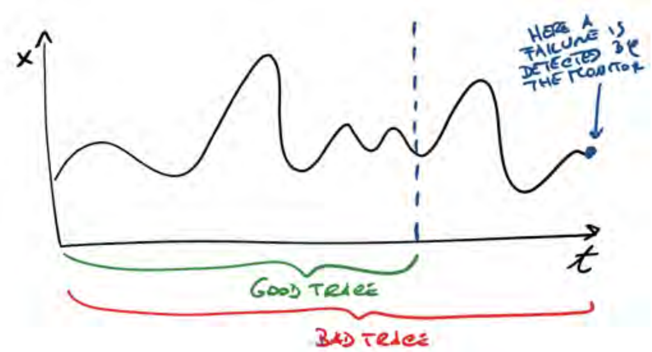
Framework: Monitoring Pool

- The properties in the pool are expressed by means of a suitable temporal logic (in the remainder we focus on STL)
- Actually, in a more general sense, properties can be encoded by a combination of STL formulas, relying on decision tree models (given their interpretability)

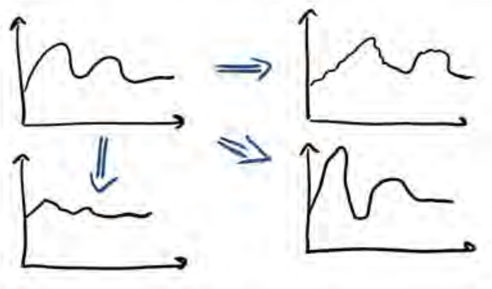


Framework: Failure Detection

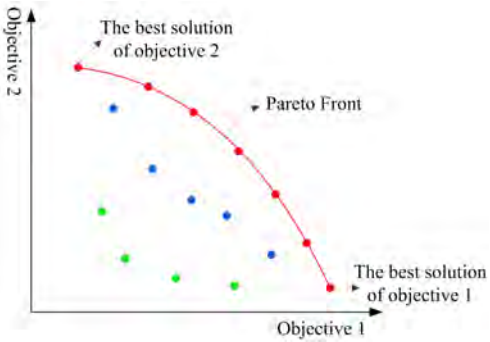
- When a failure is predicted by a property in the pool, the incoming trace is divided into a *good* and a *bad* part, according to a windowing approach
- The length of the window is a fixed hyperparameter of the framework



- At this point, new properties are extracted that are capable of discerning between the *good* and the *bad* trace
- A genetic algorithm (GA) is employed that tries to generate highly discriminative and robust STL formulas (2 objs)
- In order to avoid overfitting, starting from the good and bad traces, new traces are generated by applying different kinds of transformation. This is the training set of the GA.



- Since the genetic algorithm follows two (maximization) objectives, a set of optimal solutions (formulas) is produced at the end of its execution (Pareto front)
- Some of them won't be very useful
- Some would be more effective if combined with others



- A dataset is built to support the supervised learning of a decision tree model, were each instance corresponds to a subtrace used during the GA operation
- Each instance is represented by a set of Boolean predictors, one for each extracted formula
- Each predictor is true if and only if the corresponding formula is satisfied by the instance

	δ_1	δ_2	δ_3	…	δ_n	LABEL
GOOD	T	F	F		T	☺
GOOD'	T	F	T	…	T	☺
GOOD''	T	F	F		F	☺
⋮			⋮			⋮
BAD	F	F	T		T	☹
BAD'	F	T	F	…	F	☹
BAD''	F	T	T		T	☹
⋮						

Framework: Monitoring Pool Update

- The decision tree model is added to the monitoring pool
- Intuitively, such a model will be capable of predicting a forthcoming failure earlier than the property that initially triggered the process that led to its generation
- To each property in the pool, a validity score is attached, that tracks its performance in the detection of failures (F1 score, jointly considering precision and recall measures)
- In this way, the pool is constantly updated: redundant or under-performing properties are removed

Algorithm 1 Framework execution

Input: initial pool of properties $\mathcal{P}$, incoming system trace t

while True **do**
  if $m \in \mathcal{P}$ predicts a failure in t at time i or FAILURE(i) **then**
    UPDATEPOOLINFORMATION($\mathcal{P}, t, i$)
    $T \leftarrow$ GENERATETRAINDATA(t, i)
    $F \leftarrow$ EXTRACTDISCRFORMULAS(T)
    $m' \leftarrow$ BUILDCLASSIFIER(T, F)
    CHECKANDADD($m', \mathcal{P}$)
    SYSTEMFIXANDRESTART()

Algorithm 2 UPDATEPOOLINFORMATION

Input: pool of properties $\mathcal{P}$, trace t, failure timestep i

Global: forget rate α, minimum goodness g_{min}

$\mathcal{M} \leftarrow$ GETTRIGGEREDCLASSIFIERS($\mathcal{P}, t, i$)
for $m \in \mathcal{M}$ **do**
  $good_m \leftarrow (1-\alpha) *$ NEWF1SCORE(m, t) +
      $\alpha * good_m$
  if $good_m < g_{min}$ **then**
    REMOVE($m, \mathcal{P}$)
HANDLEREDUNDANCY($\mathcal{P}$)

Algorithm 3 BUILDCLASSIFIER

Input: training data T, list of extracted formulas F

$X \leftarrow$ new empty (length(T) $\times$ length(F)) matrix
y $\leftarrow$ new empty array of length(T) elements
for t from 1 to length(T) **do**
  for f from 1 to length(F) **do**
    $X[t][f] \leftarrow$ MONITORING($T[t], F[f]$)
  y[t] $\leftarrow T[t]$.*label*
return TRAINCLASSIFIER(X, y)

Table 1 Framework hyperparameters

	Description	Value	Search range
α	*forget rate* for properties goodness measure update	0.9	{0.7, 0.8, 0.9}
g_{min}	*minimum goodness*, properties goodness threshold	0.9	{0.7, 0.8, 0.9}
h_{max}	*maximum height* for property tree representations	3	{2, 3, 4, 5}
$n_{(F)}$	*number of formulas* obtained in the extraction phase	10	{5, 10, 15, 20}
n_{aug}	*number of augmentations* for each failure trace	100	{50, 100, 150}
l	*failure window length* for generating train data	–	domain specific

Framework: Operating Scenarios

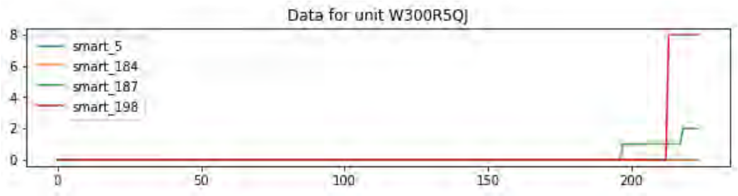
Execution Modes:

- *warmup*: mimic the continual arrival of the available traces from data pertaining to past system failures or generated by means of simulations
- *online*: incoming traces of the currently monitored system are considered

Execution Strategies:

- *semi-supervised*: domain experts specify an initial set of properties to be monitored
- *unsupervised*: monitor initialized with just a single “the machinery is in operation” property

- Information regarding the health status of ST4000DM000 hard drive model in the Backblaze data center
- Data recorded daily from 2015 to 2017
- 21 SMART parameters including both discrete and real values
- Label which indicates a drive failure



Application: Experiment Setup

- Initial *unsupervised learning warmup* phase performed concatenating a series of training set execution traces
- Two evaluation modes:
  - *offline*, for SOTA comparison purposes
  - *online*, in which the framework continues to learn properties from the execution traces of the test set
- Counter-overfitting measures (trees):
  - maximum height of 3
  - minimum F1 score of 0.9

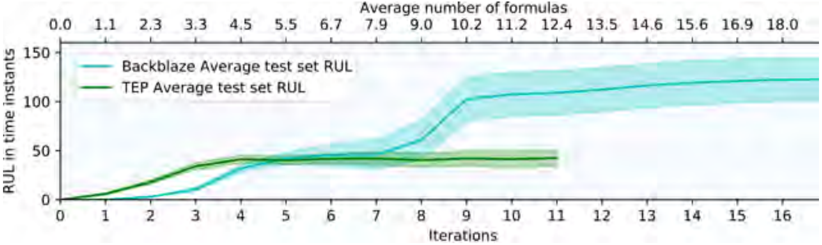
Application: Offline Results

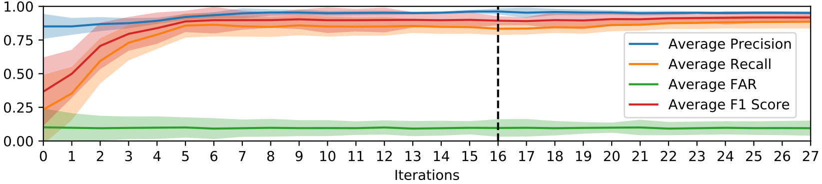
	S1 LTL	S1 STL	S1 (Huang, 2017, NN)	S2 LTL	S2 STL	S2 (Su, 2019, LSTM)
Precision	0.71	0.73	0.50	0.91	0.97	0.91
Recall	0.43	0.42	0.53	0.85	0.83	0.94
FAR	0.02	0.03	0.01	0.07	0.08	0.05
F_1 score	0.53	0.53	0.52	0.88	0.89	0.93

$$\text{Precision} = \frac{TP}{TP+FP}, \quad \text{Recall} = \frac{TP}{TP+FN},$$

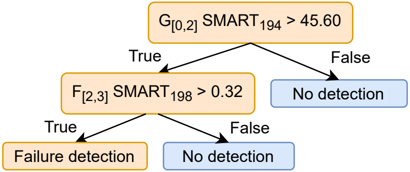
$$\text{FAR} = \frac{FP}{FP+TN}, \quad \text{F1} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}.$$

Application: RUL and Online Results





Application: Interpretability



The decision tree issues a failure prediction for a hard disk if the latter:

- in the first three days, maintains a temperature exceeding 45.6 °C
- then, its *uncorrectable sector count* value becomes greater than 0.32

Pattern witnessed during the *warmup* phase:

1. Formula $f_1 = \mathbf{F}_{[25,45]} SMART_{198} > 2.59$ is extracted at iteration i
   - critical sensor regarding *sector read/write errors*
2. Formula f_1 triggers a failure prediction at iteration $j > i$
3. As a consequence, $f_2 = \mathbf{F}_{[11,36]} SMART_{189} > 8.28$ is extracted at iteration j
   - non-critical sensor regarding *unsafe fly height conditions*

The disk head is operating at an unsafe height, ultimately damaging a disk sector and consequently causing read and write errors (link between a non-critical and a critical sensor).

Future Work

- Formula-dependent failure windows
- How to estimate RUL for the formulas extracted during the online phase?
- Experimentation with different logic formalism and case studies

Bibliography I

Giuseppe Bombara et al. (2016). "A decision tree approach to data classification using signal temporal logic". In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, pp. 1–10.

Andrea Brunello, Dario Della Monica, and Angelo Montanari (2019). "Pairing Monitoring with Machine Learning for Smart System Verification and Predictive Maintenance". In: *Proceedings of the 1st Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis*. Vol. 2509. CEUR Workshop Proceedings, pp. 71–76.

Andrea Brunello, Dario Della Monica, et al. (2020). "Learning How to Monitor: Pairing Monitoring and Learning for Online System Verification". In.

Ian Cassar et al. (2017). "A Survey of Runtime Monitoring Instrumentation Techniques". In: *Proceedings of the 2nd International Workshop on Pre- Post-Deployment Verification Techniques*, pp. 15–28.

Alexandre Donzé (2013). "On signal temporal logic". In: *International Conference on Runtime Verification*. Springer, pp. 382–383.

Marcus Gerhold, Arnd Hartmanns, and Mariëlle Stoelinga (2019). "Model-Based Testing of Stochastically Timed Systems". In: *Innovations in Systems and Software Engineering* 15.3-4, pp. 207–233. DOI: 10.1007/s11334-019-00349-z. URL: https://doi.org/10.1007/s11334-019-00349-z.

N. Jansen et al. (2018). "Machine Learning and Model Checking Join Forces". In: *Dagstuhl Reports* 8.3, pp. 74–93.

Bibliography III

Martin Leucker and Christian Schallhart (2009). “A brief account of Runtime Verification”. In: *Journal of Logical and Algebraic Methods in Programming* 78.5, pp. 293–303. ISSN: 1567-8326. DOI: http://dx.doi.org/10.1016/j.jlap.2008.08.004. URL: http://www.sciencedirect.com/science/article/pii/S1567832608000775.

Oded Maler and Dejan Nickovic (2004). “Monitoring temporal properties of continuous signals”. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, pp. 152–166.

Amir Pnueli (1977). “The temporal logic of programs”. In: *Proceedins of the 18th Annual Symposium on Foundations of Computer Science*. IEEE, pp. 46–57.

Bibliography IV

Property Pattern Mappings for LTL (n.d.). `https://matthewbdwyer.github.io/psp/`. Kansas State University CIS Department, Laboratory for Specification, Analysis, and Transformation of Software (SAnToS Laboratory) – accessed online on 17 July 2020.