Incrementally Predictive Runtime Verification*

Angelo Ferrando1[0000−0002−8711−4670] and Giorgio Delzanno1[0000−0001−7030−1050]

University of Genova, Italy
forename.surname@unige.it

Abstract. Runtime Verification is a lightweight formal verification technique used to verify the runtime behaviour of software (resp. hardware) systems. Given a formal property, one or more monitors are synthesised to verify the latter against a system execution. A monitor can only conclude the violation of a property when it observes such a violation. Unfortunately, in safety-critical scenarios, this might happen too late for the system to react properly. In such scenarios, it is advised to use Predictive Runtime Verification, where monitors are capable of anticipating (by using a model of the system) future events before actually observing them. In this work, instead of assuming such a model is given, we describe a runtime verification workflow where the model is learnt and incrementally refined by using process mining techniques. We present the approach and the resulting prototype tool.

1 Introduction

Runtime Verification (RV) [7] is a kind of formal verification technique that focuses on checking the behaviour of software/hardware systems. With respect to other formal verification techniques, such as Model Checking [11] and Theorem Provers [18], RV is considered more dynamic and lightweight. This is mainly due to its being completely focused on checking how the system behaves, while the latter is currently running. This is important from a complexity perspective. RV does not need to simulate the system in order to check all possible execution scenarios; but, it only analyses what the system produces (*i.e.*, everything that can be observed of the system). This is usually obtained through monitors, that are nothing more than validation engines which, given a trace of events generated by the system execution, conclude the satisfaction (resp. violation) of a formal property of interest. In turn, a formal property is the formal representation of how we expect the system should behave. The monitor's job is to verify at runtime whether such a property holds.

Since monitors are usually deployed together with the system under analysis, they are well suited for checking properties that require to be continuously monitored. This is especially true in safety-critical scenarios, where a system's

* Copyright © 2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

fault can cause injuries, loss of money and even deaths. A key example is autonomous and robotic systems, where *reliability* is vital [12], and the addition of monitors ensuring a correct behaviour is a valuable feature. Nonetheless, even though monitors are lightweight components, they are still an additional workload for the system. This is not a problem for large systems, but it might be for smaller ones, such embedded systems; where the amount of available resources can be limited. For both increasing reliability and reducing the impact of the monitors on the system, an extension of standard RV named Predictive Runtime Verification (PRV) has been proposed e.g. in [31]. As the name suggests, PRV differs from RV because it does not only consider the events observed by the system execution, but it also tries to *predict* future events. By predicting future events, the resulting monitors are capable of concluding the verification sooner. Indeed, since these monitors can predict how the system is going to behave in the future, they can reduce the search space of possible future continuations and, hypothetically, conclude the satisfaction (resp. violation) by analysing shorter traces of events (w.r.t. the standard counterpart). By concluding earlier, predictive monitors are a good choice: (i) for improving reliability in safety-critical scenarios, since the monitors can conclude a violation of a property before such a violation has even been observed; (ii) for reducing the workload introduced by the monitor, since the monitor can conclude the satisfaction (resp. violation) of the property sooner, it can be removed from the system and its resources can be reclaimed for other purposes.

The problem with PRV is that it requires additional knowledge on the system in order to predict future events. Usually, this is represented through an abstraction, the *model*, which is manually created by an expert of the system. The problem with this approach is that such a model is not always available, and even when it is, it might not be specified in a convenient way (e.g. wrong formalism). One possible way to avoid errors due to human intervention in the model generation step is to resort once again to observations collected at runtime. The guiding principle here is to learn the model behavior by observing real execution traces so as to create a sort of closed loop in which logs are used to adjust the candidate models, models are used to predict faults with certain confidence level, the confidence level increases with the log size, etc.

For this purpose, in this paper we present an initial study on how to use Process Mining (PM) to automate the model generation phase in practical applications of PRV. PM is a rich research area which consists in – but is not limited to – studying and developing automated techniques to synthesise models from log files. Specifically, we focus on the branch of PM called *Process Discovery*. We show how it can be applied in the context of PRV, and we present the updated verification workflow; where, starting from a set of log files generated by the system execution, we obtain a model that can be used to predict events for the monitor. We also show how the resulting approach can be used to obtain *incrementally* predictive monitors, where more the monitors are used to verify the system and more they improve at predicting future events, and thus, at concluding the satisfaction (resp. violation) of the analysed property.

It is important to remark that, to avoid human intervention in the model generation phase, the proposed approach has an empirical nature by construction since it always works with partial information on the entire set of possible system behaviours. In theory, a precise model of the system can be obtained only as the limit of a sequence of possible inaccurate candidate models. For this reason, our validation procedure returns truth values enriched with confidence scores that depend on the frequency of observed events applied during the PM pre-processing phase.

The remainder of this paper is structured as follows. Section 2 presents the preliminary notions of RV, PRV and PM. Section 3 shows our general verification workflow without selecting any specific formalism for properties and models. Section 3.1 instantiates our workflow with the most used formalisms in formal verification. Section 4 presents the prototype tool representing the implementation of our approach. Section 5 positions this contribution with respect to the state of the art. Finally, Section 6 summarises the results of this work and points out to future directions.

2 Preliminaries

2.1 Runtime Verification and Predictive Runtime Verification

A system is denoted by S, and its *alphabet* (all of its observable events) is denoted by Σ_S (or Σ where there is no confusion). Given an alphabet, Σ, a *trace*, σ, is a sequence of events in Σ, and $tr(\Sigma)$ is the *set of all possible traces* (the language) over Σ. Properties are denoted by φ, potentially with subscripts, and $\overline{\varphi}$ denotes their negation. Given an alphabet Σ, a property φ is *satisfied by a trace* σ over Σ, written $\sigma \models \varphi$, if φ is true in σ. The set $[\![\varphi]\!] = \{\sigma \mid \sigma \models \varphi\}$ contains the *set of traces satisfying* φ, and we denote that a particular trace σ satisfies a property φ as $\sigma \in [\![\varphi]\!]$. A property φ can be specified in any formalism such that for a given alphabet Σ, and for any trace $\sigma \in tr(\Sigma)$, the following two conditions hold:

$$\sigma \in [\![\varphi]\!] \text{ is decidable} \tag{1}$$

$$\sigma \in [\![\varphi]\!] \iff \sigma \notin [\![\overline{\varphi}]\!] \tag{2}$$

Condition (1) states that, given a property φ specified with the formalism of choice, we can always test if a trace σ satisfies φ, i.e. σ belongs to the set of traces satisfying φ. This condition is mandatory since a monitor is defined upon the notion of trace acceptance. As we are going to show in Definition 1, a monitor requires to check if a trace satisfies the property under analysis, and this can be done only when condition (1) holds. Condition (2), instead, implicitly states that the formalism of choice has to be closed under negation and a trace σ satisfies a property φ, if and only if, σ does not satisfy its negation $\overline{\varphi}$. The negation of properties will be used in Definition 2 to define monitors with predictive flavour, where we will combine a model ψ with the negation of a property φ to check for traces satisfying ψ but not φ.

Definition 1 (Monitor). *Let S be a system with alphabet Σ, and φ be a property. Then, a* monitor *for φ is a function $Mon_\varphi : tr(\Sigma) \rightarrow \mathbb{B}_3$, where $\mathbb{B}_3 = \{\top, \bot, ?\}$:*

$$Mon_\varphi(\sigma) = \begin{cases} \top & \forall_{u \in tr(\Sigma)}.\sigma \bullet u \in [\![\varphi]\!] \\ \bot & \forall_{u \in tr(\Sigma)}.\sigma \bullet u \notin [\![\varphi]\!] \\ ? & otherwise \end{cases}$$

where $\bullet$ is the standard trace concatenation operator.

Intuitively, a monitor returns $\top$ if all continuations of σ satisfy φ; $\bot$ if all possible continuations of σ violate φ; ? otherwise.

Definition 1 describes a generic monitor that does not impose constraints on the formalism used. Consequently, we collapse the definition of $tr(\Sigma)$ for representing finite and infinite traces depending on what is supported by the formalism that is used to define φ. Thus, if φ accepts only traces of infinite length, then $tr(\Sigma) = \Sigma^\omega$; if φ accepts only traces of finite length, then $tr(\Sigma) = \Sigma^*$; otherwise, $tr(\Sigma) = \Sigma^* \cup \Sigma^\omega$.

Let S be a system with alphabet Σ. We denote its *model* by ψ, and use $[\![\psi]\!] \subseteq tr(\Sigma)$ to indicate the *set of traces recognised by* ψ (*i.e.*, ψ represents a formal abstraction of how S behaves). A model, ψ, can be specified in any formalism such that for a given alphabet, Σ, for any trace, $\sigma \in tr(\Sigma)$, and for any property, φ, the following holds:

$$\sigma \in [\![\psi]\!] \text{ is decidable} \tag{3}$$

$$[\![\varphi]\!] \cap [\![\psi]\!] \text{ is computable} \tag{4}$$

We indicate (4) via the use of a binary relation $\otimes$, that is, $[\![\varphi \otimes \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$.

Often, PRV frameworks express their properties in Linear-time Temporal Logic (LTL) [22] (for example [17,31]); however, we took our inspiration from a PRV framework where both the System Under Analysis (SUA) and property are defined using Timed Automata (TA) [21]. The reason is that in works such as [17,31], the predictive aspect is not formalised through a model of the system, but as a set of finite suffixes. These suffixes are then concatenated to the given trace σ allowing the monitor to predict the initial part of the possible continuations u. Instead, in our work, as in [21], we explicitly represent the model without focusing only on the first events after σ, but by applying the prediction to the entire possible continuation. This can be obtained using a model of the system as input to the monitor, alongside the property to be verified. Informally, the model generates the set of event traces that can be observed by executing the system. We follow the definition of a predictive monitor from [21], however, in this paper we remain formalism-agnostic.

Definition 2 (Predictive Monitor). *Let S be a system with alphabet Σ, model ψ and let φ be a property. A* predictive monitor *for φ given ψ is a function,*

$Mon_{\varphi,\psi} : tr(\Sigma) \to \mathbb{B}_3$*, where* $\mathbb{B}_3 = \{\top, \bot, ?\}$*:*

$$Mon_{\varphi,\psi}(\sigma) = \begin{cases} \top & \forall_{u \in tr(\Sigma)}.\sigma \bullet u \notin [\![\overline{\varphi} \otimes \psi]\!] \\ \bot & \forall_{u \in tr(\Sigma)}.\sigma \bullet u \notin [\![\varphi \otimes \psi]\!] \\ ? & otherwise \end{cases}$$

The intuitive meaning of the return values is the same as in the non-predictive case (Definition 1). Note the use $\otimes$ in the definition. For instance, the case for $\top$ requires all traces $\sigma \bullet u$ not to be in $[\![\overline{\varphi}]\!] \cap [\![\psi]\!]$ where $\overline{\varphi}$ represents the negation of φ.

2.2 Process Mining

Process Mining (PM) [26] is a technique used in software engineering to automatically synthesise a formal model which denotes the system behaviour. Such analysis is usually performed on event logs generated by multiple executions of the system.

In practice, by using data mining algorithms, knowledge is extracted by these logs and corresponding formal models are generated. These models represent an abstraction of the system, and can be used to understand how the system behaves. Usually, once such models are extracted, an expert of the system can use them to study and check whether the implemented system actually meets his/her expectations. This check can be manually performed by the expert, who inspects the extracted model and searches for inconsistencies, or it can be automatically performed through conformance checking algorithms.

The power of PM lies in its being flexible, customisable and automated. Thus, we decided to study and develop its integration in the context of PRV. Mostly because, one of the drawbacks of PRV is the assumption that an existing model of the system exists. Moreover, since PM completely depends on the event logs generated by the system execution, more logs are used, and better (*i.e.*, more precise) models are extracted. Because of this, PM does not only allow PRV to be applied when a model of the system does not exist, but it also makes PRV more robust and reliable. For instance, when the system under analysis presents dynamic aspects, to rely on a static model would eventually bring to wrong predictions. Instead, if the model is automatically obtained by observing how the system behaves, it may keep up with the change and be a more reliable representation of how the system is currently behaving.

3 Incrementally Predictive Runtime Verification

When applying RV with a predictive flavour, we need a model of the system to predict future events. Without such a knowledge, each event would be considered observable in the future, and the monitor would need to evaluate all possible continuations of the current analysed trace. Nonetheless, how the model of the system is generated is not defined a priori. In previous works [6,30,21,31,17,10], the model of the system is assumed to be manually created by an expert of the system. Here, we show how the standard predictive runtime verification workflow can be enhanced using process mining techniques to automatically synthesise the model of the system.

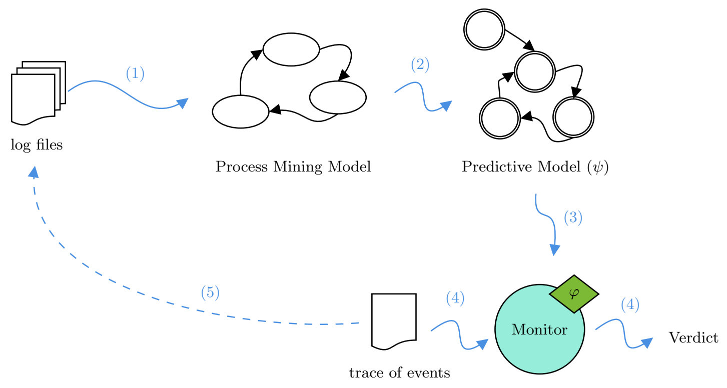

Fig. 1: Engineering steps to obtain incrementally predictive monitors.

Figure 1 shows a general overview of our approach. At this level, we do not pick any specific formalism for representing the properties and the models (as it is done in Section 2), but we focus on the necessary steps to define the workflow for obtaining an incrementally predictive RV approach.

Step 1: Model extraction As it is true for all process mining algorithms, everything starts from a set of log files. These log files contain information about previous executions of the system, and can be used to predict future executions as well. Given such a set of log files, a process mining algorithm of choice can be selected to parse the logs and generate a model representing the system behaviour (*Process Mining Model*). Depending on the algorithm, the resulting model may be different; even though usually the most common formalisms used to specify such models are Petri Net (PN) [20] and Directly-Follows Graph (DFG) [5].

Step 2: Predictive Model derivation The model extracted by the process mining algorithm (step (1)) may not be directly usable to synthesise a predictive monitor (e.g. formalism mismatch). If that is the case, one or more additional transformation steps are required[1]. These steps aim to transform the *Process Mining Model* into a more suitable *Predictive Model*, which can be used to synthesise predictive monitors as we show in Section 2. Again, these models can be specified using different formalisms which depend on the kind of predictive monitor we want to synthesise.

[1] Naturally, if the *Process Mining Model* is already defined in a suitable way for the predictive monitor (*i.e.*, the *Process Mining Model* and *Predictive Model* coincide), the entire step (2) can be removed.

Step 3: Monitor synthesis Once the model of the system ψ is obtained (steps (1-2)), it can be used to synthesise a predictive monitor. Such synthesis depends on which formalism is used for defining the property φ to verify, and which formalism has been used at step (2) to generate ψ. In principle, the synthesised monitor has to follow Definition 2; where the model is used to predict future events and to prune impossible continuations of the current analysed trace.

Step 4: Monitor execution The predictive monitor obtained at step (3) for the property φ and the *Predictive Model* ψ is used to analyse the system. More in detail, a trace representing the system execution is passed as input to the monitor. Such a trace can be retrieved incrementally while the system is running (events are passed to the monitor as soon as they are generated), or, it can be stored in a log file. In the former, we refer to online RV, while in the latter, we refer to offline RV. In both cases, the verification process ends with a verdict produced by the monitor. If the verdict is positive ($\top$), then the system execution has satisfied φ. If the verdict is negative ($\bot$), then the system execution has violated φ. Finally, if the system execution neither satisfies nor violates φ, the verdict is inconclusive (?).

Step 5: Log files update An interesting aspect of our approach is that log files can be used at two separate levels. They are used to synthesise the model of the system through process mining at step (1), and are used to verify the current system execution against a formal property φ at step (4). Nonetheless, in both cases they are nothing more than traces of events obtained by executing the system under analysis. When the monitor is done with a trace of events (it has completed the verification), in standard RV such a trace is discarded. Even though this is true for standard RV, it is not the case for PRV. Indeed, we can use the trace of events analysed by the monitor to increase the knowledge we have of the system. Specifically, this new trace of events can be added to the log files on which process mining is applied. In this way, more the predictive monitor is used to verify a system, and more it improves at predicting events on that system. Initially, the set of log files used at step (1) might be small, which would bring to a non informative model of the system; but, by using the predictive monitor to verify the system, additional event traces would be added to the set of log files. The addition of new traces enriches the knowledge the process mining algorithm has of the system, and consequently, it improves the quality of the synthesised model.

3.1 Incrementally Predictive Runtime Verification instantiation

In Section 3, we present the general workflow, where no specific formalism is fixed. In this way, depending on the domain, the approach can be customised for obtaining better results. Nonetheless, for a better understanding, we present an instantiation of our approach, where we use DFG as formalism for representing the *Process Mining Model*, Probabilistic Finite-State Machine (PFSM) [24] and

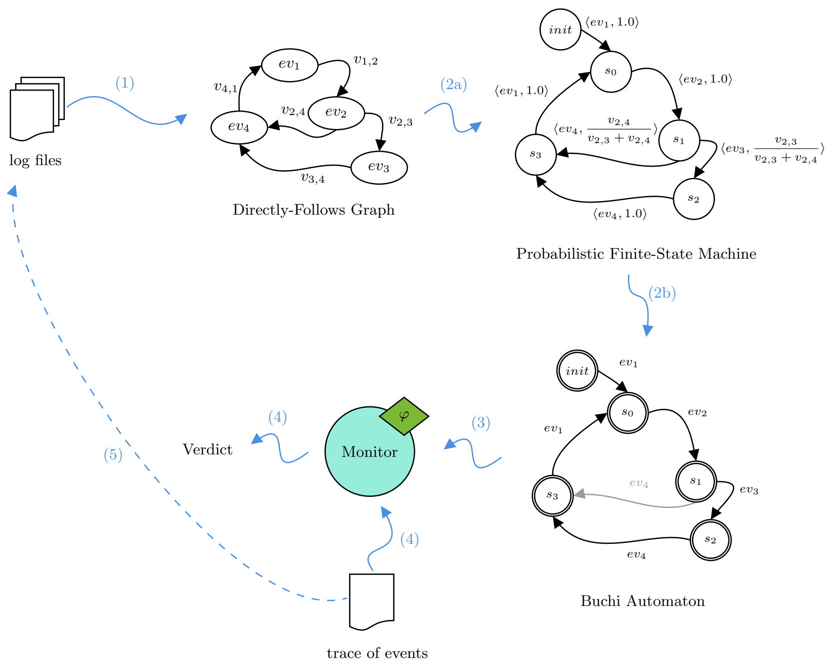

Fig. 2: Instantiated engineering steps to obtain predictive monitors.

Büchi Automata (BA) [8] for the *Predictive Model*, and finally, LTL for the properties to verify at runtime with the monitor.

Figure 2 shows an overview of the instantiated verification workflow; where, starting from a set of log files representing different system executions, and an LTL property φ to verify, we obtain a predictive monitor. The verification process consists in 6 steps (since step (2) of Figure 1 is split into (2a) and (2b)).

Step 1: Model extraction First of all, using the log files generated by multiple system executions, a DFG [5] is created. Such a graph represents the system behaviour and can be obtained by applying state of the art process mining algorithms, such as Alpha Miner [2,29], Heuristics Miner [4,28], Inductive Miner [16,13], Process Skeletonization [3], and Graph-Based Miner [27,25] (a thorough review can be found in [5]). The output's formalism of these algorithms can vary, but the most common representations are Petri Net [20], Causal Net [1], BPMN [9] and DFG. We decided to use DFG because their translation to automata is more direct. The DFG so generated is a graph where the states are labeled with events ev_i observed in the log files, and a directed edge goes from a state ev_i to a state ev_j if there is at least one trace in the log files where the event ev_i is followed by the event ev_j. Naturally, these events are domain dependent, e.g. messages, function calls, actions, etc. Moreover, each transition is labelled with the number of traces ($v_{i,j}$) where such causality relation is observed. For instance, in Figure 2, ev_1 is followed by ev_2 in $v_{1,2}$ traces in the log files, ev_2 is followed by ev_3 in $v_{2,3}$ traces and by ev_4 in $v_{2,4}$ traces, and so on.

Step 2a: Addition of probabilities After generating a DFG of the system, the next step is to explicitly represent which events can be observed and with which probability. This is achieved in step (2a), where starting from a DFG, a corresponding PFSM [24] is obtained. A PFSM is an extension of standard Finite-State Machine (FSM) where the transitions amongst states are labelled with tuples $\langle ev_i, p_i\rangle$, which denote that the event (ev_i) can be observed with probability (p_i) in that state. Since the PFSM is obtained by a DFG, the events and probabilities are extracted accordingly. If there is an edge from ev_i to ev_j labelled with $v_{i,j}$, and an edge from ev_i to ev_k labelled with $v_{i,k}$, then in the corresponding PFSM, a state s is generated, with two transitions labelled $\langle ev_j, \frac{v_{i,j}}{v_{i,j}+v_{i,k}}\rangle$ and $\langle ev_k, \frac{v_{i,k}}{v_{i,j}+v_{i,k}}\rangle$. These two transitions specify the probability of observing events ev_j and ev_k in s. For instance, in Figure 2, in the DFG, the state ev_2 is followed by the states ev_3 and ev_4. This is mapped to state s_1 in the PFSM, in which the two events ev_3, and ev_4 can be observed. The probability attached to ev_3 is the probability of observing ev_3 after ev_1, which can be computed as the number of times ev_3 has followed ev_1 in the log files $(v_{1,3})$ divided by the total number of traces where ev_1 has been observed (similar reasoning for ev_4). Let us assume $v_{1,3} = 7$ and $v_{1,4} = 3$, then the probability of observing ev_3 after ev_1 is $\frac{7}{7+3} = 0.7$ (*i.e.*, 70%); while the probability of ev_4 is $\frac{3}{7+3} = 0.3$ (*i.e.*, 30%). Thus, the transitions from state s_1 to states s_2 and s_3 are $\langle ev_3, 0.7\rangle$ and $\langle ev_4, 0.3\rangle$, respectively. It is important to note that, each state in PFSM has a transition for each possible event of the system. The events that have never been observed in a state have zero probability (such transitions are omitted in Figure 2 for readability). For instance, in state s_1, the event ev_1 has zero probability to be observed, since no trace has ever been observed with an event ev_2 followed by an event ev_1. Nonetheless, we need to explicitly represent the events with zero probability as well because initially, when the model extracted at step (1) is not trustworthy, any event has to be considered possible. How much trust we put in the PFSM is determined by the following step (2b), where a probability threshold is used to prune (resp. keep) transitions on the basis of their probability values.

Step 2b: Approximation with a threshold In a standard predictive RV approach, the model of the system does not contain probabilities. Instead, it simply specifies which traces can be generated by the system execution. Probabilities, even though relevant, are not usually taken in consideration when the model is used for predicting future events [30,21,31,17,10]. Nonetheless, probabilities can be used to select which events are observable in which states. One possible way to give importance to probabilities without explicitly reporting them in the model is to set a probability threshold. Given a probability threshold, all transitions with a probability less than the threshold can be removed (grey transition in Figure 2), while the transitions with a probability greater than the threshold can be preserved (but deprived of the probability information). In this way, given a

PFSM, we can generate a corresponding BA [8], for a certain probability threshold. Such a BA explicitly describes the language recognised by the model of the system. We decided to use BA because is one of the most used formalisms in formal verification. By generating a BA, we assure a wider use of our solution, since existing predictive RV tools already support BA. Depending on the threshold, the BA can be more or less reliable. For instance, if the threshold is 0.01 (*i.e.*, 1%), then all transitions with probabilities less than 0.01 are pruned. This means that, the BA so generated recognises all traces generated by the log files, except for the traces that are unlikely. Naturally, higher is the threshold, and smaller is the language recognised by the BA; since more transitions are pruned in the process from PFSM to BA. Because of this, the threshold needs to be chosen carefully. In principle, more we trust the model generated at step (1), and higher we can set the threshold. Indeed, if we trust the model extracted by the process mining algorithm, then we can trust that the transitions with low probability represent outliers, and by pruning them we can have a faster[2] verification. On the other hand, if we do not trust the model generated at step (1), we can select a lower threshold and prune less transitions in the transformation from PFSM to BA. For instance, this can happen when not many log files are available, and the process mining is not accurate enough. While the threshold can be increased later on, when more log files are available and the resulting process mining is more trustworthy. Note that, the BA corresponds to an over-approximation of the DFG obtained at step (1). Since the DFG only specifies the cause-effect relation amongst the system's events, in presence of loops in the DFG (resp. in the BA), we cannot infer the number of times the system execution passes on such loops. Because of this, in order to not prune possible system executions from the BA model, each state in the BA is set to be final. In this way, the BA recognises a superset of the execution traces generated by the system.

Step 3: Monitor synthesis Once we obtain a model of the system, a predictive monitor can be synthesised (following Definition 2). In [31], the authors present the approach for synthesising predictive monitors for LTL properties, where the model of the system is assumed being a BA. Following the same approach, at step (3), we synthesise a predictive monitor by combining an LTL property φ given in input, and the BA obtained at step (2b). Naturally, depending on the formalism used at step (2b), the monitor synthesis may vary. For instance, instead of BA, we could use timed automata to denote the model of the system, as it is done in [17]. Note that, this would cause a modification of only step (2b) and (3); the remaining steps of the workflow would remain unchanged. In fact, we selected LTL and BA as formalisms for representing properties and models because are widely used and well-known. Nonetheless, the workflow presented in this work is not constrained to any specific formalism. With few modifications it can be adapted to other formalisms as well.

Steps 4-5: Monitor execution and Log files update These steps do not depend on the specific instantiation and are the same as the steps in Section 3.

[2] Since less future continuations have to be considered.

4 Implementation

The Python implementation of our approach is publicly available as a GitHub repository[3]. We implemented all the engineering steps presented in this paper, when instantiated to the case with LTL properties and BA models. More in detail, the resulting tool takes in input: (i) a set of log files (expressed as a single XES file[4]), which is the standard format used in PM to represent event logs; (ii) a threshold to guide the mapping from PFSM to BA; (iii) an LTL property to verify; (iv) a trace generated by the current system execution to analyse.

To implement step (1), *i.e.* the application of process mining to the log files, we used the PM4Py[5] Python library. PM4Py is the leading open source process mining platform written in Python, and is developed by the process mining group of Fraunhofer Institute for Applied Information Technology[6]. Such a library allows to develop process mining applications very quickly. In fact, the step (1) of our approach has been completely implemented by calling PM4Py API. Starting from the XES file denoting the knowledge we have of the system, PM4Py generates a DFG as a Python object. The rest of the code has been fully implemented in Python from scratch. First, the Python object representing the DFG is translated into another Python object denoting the corresponding PFSM (step (2a)). Such translation is straightforward, because it is enough to unroll the transitions amongst states in the DFG; similarly to what it is usually done when translating Kripke structures [15] to BA. Once the Python object representing the PFSM is obtained, the tool goes on with its translation to BA (step (2b)). Again, this translation is direct because it simply requires to prune the transitions with a probability lower than the chosen threshold, and make all states final. With the BA Python object, a predictive monitor is then generated. This, in particular, is obtained by using a Python library supporting the generation of predictive monitors for LTL properties and BA models[7]. This library has been developed by one of the authors of this paper as well. Using this library, the step (3) is completed, and a predictive monitor ready to be used is synthesised. Finally, the tool terminates by using such a monitor to verify a trace of events given in input, and reporting the final verdict to the user (step (4)). The trace so analysed can then be integrated (step (5)) inside the XES file given in input to enhance future process mining phases (*i.e.*, next times the approach is used).

5 Related Work

RV is a rich research area, but PRV is still rising. One of the oldest works on PRV is presented in [31]. In this work, the authors present an extension of the standard RV approach when applied to LTL properties. Differently from the

[3] `https://github.com/AngeloFerrando/IncrementallyPredictiveRV`
[4] `http://www.xes-standard.org/`
[5] `https://pm4py.fit.fraunhofer.de/`
[6] `http://fit.fraunhofer.de/process-mining`
[7] `https://github.com/AngeloFerrando/MultiModelPredictiveRuntimeVerification`

rest of works in literature, the prediction of future events is not obtained using a model of the system, but is limited to a single sequence of events (called *finite predictive word*) that is concatenated to the analysed trace. With respect to our solution, no explicit – nor complete – representation of the system is used. The same year, one of the authors proposed an extended version [17], where instead of considering one single predictive word, a BA model is integrated inside the standard generation of LTL monitors. The resulting approach is at the basis of the predictive library we use in our tool to synthesise predictive LTL monitors. The main difference is in the generation of the BA, which in [17] is given, while in our approach is automatically extracted through PM. Moreover, our approach is intrinsically general. In fact, LTL and BA are just a possible instantiation. A more recent work where prediction is obtained through assumptions is presented in [10]. The authors propose a different way to synthesise predictive monitors for LTL properties, where the model used to predict the future events, instead of being a BA, is a Fair Kripke Structure (FKS) [14]. Similarly, in [21], another PRV approach is presented, where the model is instead defined as a Timed Automaton. In both cases, the main difference with our approach is in the generality and the fact that the model is not given but automatically synthesised. On a completely different line of research, we may find [6], where probabilistic models, like Markov Models, are used. This work is very similar to ours because, differently from the others mentioned before, it does not assume the model of the system is given. In fact, the probabilistic model used in the resulting RV framework is trained on samples generated by the system execution using the Baum-Welch algorithm [23]. Even though similar in principle, this work differs from our solution in three aspects: (i) it uses machine learning techniques to extract the model, while we rely on PM techniques; (ii) it does not define any notion of incrementality; (iii) it considers only finite extensions. To the best of our knowledge, the work that is closest to ours is presented in [19]; where the authors use PM to extract the model used for synthesising predictive monitors as well. With respect to our approach, in [19] the objective of the predictive monitor is slightly different. Instead of being purely focused on the verification of the system under analysis, they are more interested in using the monitors to recommend the system on how to proceed. To do so, they focus on the probabilistic aspects and present a framework to suggest the best actions to perform in order to increase the probability of satisfying a given LTL formula. Moreover, their solution is less general than ours. Their approach is hard-coded in the Business Process scenario, while our verification workflow is general and tackles all the engineering steps necessary to integrate PM inside the creation of predictive monitors.

6 Conclusions and Future Work

In this paper, we present a general verification workflow for integrating PM in the generation of predictive monitors. We show all the engineering steps that, from a set of log files, bring to the extraction of a model which can be used to predict future events and speed up the RV process. We present an overview of the verification workload where no formalism is enforced. This choice increases the impact of the work in the verification community, since these engineering steps can be customised w.r.t. the user's needs. Nonetheless, to help better understanding the approach, we also show an instantiation with LTL properties and BA models. The choice of these two formalisms is due to their being largely used in the verification community.

Finally, a Python prototype tool is presented. We briefly show its features and how it implements the engineering steps presented in the paper.

With respect to future directions; this is an initial work on the topic and there are many different aspects that still need to be tackled. For instance, at the current level, the probability is not considered in the monitor and it is lost in the translation from PFSM to BA. Nonetheless, this is an interesting aspect to explore further. Indeed, the notion of threshold could be used to add more information to the monitor's outcome. This could also bring to the generation of multiple BA, each corresponding to a different threshold. This would be interesting to analyse, and it could be a starting point for a more thorough comparison between our approach, and the one presented in [19].

References

1. van der Aalst, W.M.P.: Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer (2011). https://doi.org/10.1007/978-3-642-19345-3, `https://doi.org/10.1007/978-3-642-19345-3`
2. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Trans. Knowl. Data Eng. **16**(9), 1128–1142 (2004). https://doi.org/10.1109/TKDE.2004.47, `https://doi.org/10.1109/TKDE.2004.47`
3. Abe, M., Kudo, M.: Business monitoring framework for process discovery with real-life logs. In: Sadiq, S.W., Soffer, P., Völzer, H. (eds.) Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8659, pp. 416–423. Springer (2014). https://doi.org/10.1007/978-3-319-10172-9_30, `https://doi.org/10.1007/978-3-319-10172-9_30`
4. Augusto, A., Conforti, R., Dumas, M., Rosa, M.L., Bruno, G.: Automated discovery of structured process models from event logs: The discover-and-structure approach. Data Knowl. Eng. **117**, 373–392 (2018). https://doi.org/10.1016/j.datak.2018.04.007, `https://doi.org/10.1016/j.datak.2018.04.007`
5. Augusto, A., Conforti, R., Dumas, M., Rosa, M.L., Maggi, F.M., Marrella, A., Mecella, M., Soo, A.: Automated discovery of process models from event logs: Review and benchmark. IEEE Trans. Knowl. Data Eng. **31**(4), 686–705 (2019). https://doi.org/10.1109/TKDE.2018.2841877, `https://doi.org/10.1109/TKDE.2018.2841877`
6. Babaee, R., Gurfinkel, A., Fischmeister, S.: $\mathcal{P}$revent : A predictive run-time verification framework using statistical learning. In: Johnsen, E.B., Schaefer, I.

(eds.) Software Engineering and Formal Methods - 16th International Conference, SEFM 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10886, pp. 205–220. Springer (2018). https://doi.org/10.1007/978-3-319-92970-5_13, `https://doi.org/10.1007/978-3-319-92970-5_13`

7. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Lectures on Runtime Verification, pp. 1–33. Springer (2018)
8. Büchi, J.R.: On a Decision Method in Restricted Second Order Arithmetic, pp. 425–435. Springer New York, New York, NY (1990). https://doi.org/10.1007/978-1-4613-8928-6_23, `https://doi.org/10.1007/978-1-4613-8928-6_23`
9. Business process model and notation. `https://www.bpmn.org/`, accessed: 2021-06-24
10. Cimatti, A., Tian, C., Tonetta, S.: Assumption-based runtime verification with partial observability and resets. In: Finkbeiner, B., Mariani, L. (eds.) Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11757, pp. 165–184. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_10, `https://doi.org/10.1007/978-3-030-32079-9_10`
11. Clarke, E.M.: Model checking. In: International Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 54–56. Springer (1997)
12. Fisher, M., Mascardi, V., Rozier, K.Y., Schlingloff, B., Winikoff, M., Yorke-Smith, N.: Towards a framework for certification of reliable autonomous systems. Auton. Agents Multi Agent Syst. **35**(1), 8 (2021). https://doi.org/10.1007/s10458-020-09487-2, `https://doi.org/10.1007/s10458-020-09487-2`
13. Ghawi, R.: Process discovery using inductive miner and decomposition. CoRR **abs/1610.07989** (2016), `http://arxiv.org/abs/1610.07989`
14. Kesten, Y., Pnueli, A., Raviv, L.: Algorithmic verification of linear temporal logic specifications. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1443, pp. 1–16. Springer (1998). https://doi.org/10.1007/BFb0055036, `https://doi.org/10.1007/BFb0055036`
15. Kripke, S.A.: Semantical considerations on modal logic. Acta Philosophica Fennica **16**, 83–94 (1963)
16. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - A constructive approach. In: Colom, J.M., Desel, J. (eds.) Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7927, pp. 311–329. Springer (2013). https://doi.org/10.1007/978-3-642-38697-8_17, `https://doi.org/10.1007/978-3-642-38697-8_17`
17. Leucker, M.: Sliding between Model Checking and Runtime Verification. In: Runtime Verification. LNCS, vol. 7687, pp. 82–87. Springer (2012)
18. Loveland, D.W.: Automated theorem proving: a logical basis, Fundamental studies in computer science, vol. 6. North-Holland (1978), `https://www.worldcat.org/oclc/252520243`
19. Maggi, F.M., Francescomarino, C.D., Dumas, M., Ghidini, C.: Predictive monitoring of business processes. In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki,

Greece, June 16-20, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8484, pp. 457–472. Springer (2014). https://doi.org/10.1007/978-3-319-07881-6_31, `https://doi.org/10.1007/978-3-319-07881-6_31`

20. Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, Universität Hamburg (1962)
21. Pinisetty, S., Jéron, T., Tripakis, S., Falcone, Y., Marchand, H., Preoteasa, V.: Predictive Runtime Verification of Timed Properties. Journal of Systems and Software **132**, 353–365 (2017)
22. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32, `https://doi.org/10.1109/SFCS.1977.32`
23. Rabiner, L.: A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE **77**(2), 257–286 (1989). https://doi.org/10.1109/5.18626
24. SALOMAA, A.: Chapter ii - finite non-deterministic and probabilistic automata. In: SALOMAA, A. (ed.) Theory of Automata, International Series of Monographs on Pure and Applied Mathematics, vol. 100, pp. 71–113. Pergamon (1969). https://doi.org/https://doi.org/10.1016/B978-0-08-013376-8.50008-3, `https://www.sciencedirect.com/science/article/pii/B9780080133768500083`
25. Sarno, R., Sungkono, K.R., Johanes, R., Sunaryono, D.: Graph-based algorithms for discovering a process model containing invisible tasks. International Journal of Intelligent Engineering and Systems **12**, 85–94 (2019)
26. Van Der Aalst, W.: Data science in action. In: Process mining, pp. 3–23. Springer (2016)
27. Waspada, I., Sarno, R., Sungkono, K.: An improved method of parallel model detection for graph-based process model discovery. International Journal of Intelligent Engineering and Systems **13**, 127–139 (04 2020). https://doi.org/10.22266/ijies2020.0430.13
28. Weijters, A.J.M.M., Ribeiro, J.T.S.: Flexible heuristics miner (FHM). In: Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2011, part of the IEEE Symposium Series on Computational Intelligence 2011, April 11-15, 2011, Paris, France. pp. 310–317. IEEE (2011). https://doi.org/10.1109/CIDM.2011.5949453, `https://doi.org/10.1109/CIDM.2011.5949453`
29. Wen, L., Wang, J., van der Aalst, W.M.P., Huang, B., Sun, J.: Mining process models with prime invisible tasks. Data Knowl. Eng. **69**(10), 999–1021 (2010). https://doi.org/10.1016/j.datak.2010.06.001, `https://doi.org/10.1016/j.datak.2010.06.001`
30. Yu, K., Chen, Z., Dong, W.: A Predictive Runtime Verification Framework for Cyber-Physical Systems. In: Software Security and Reliability-Companion. pp. 223–227. IEEE (2014)
31. Zhang, X., Leucker, M., Dong, W.: Runtime Verification with Predictive Semantics. In: NASA Formal Methods. LNCS, vol. 7226, pp. 418–432. Springer (2012)