UNIVERSITÀ DEGLI STUDI DI GENOVA

CILC 2021

Incrementally Predictive Runtime Verification

Angelo Ferrando and Giorgio Delzanno
University of Genova

angelo.ferrando@unige.it
Research Fellow

Formal Verification [in a nutshell]

In the context of hardware and software systems, formal verification is the act of **proving** or **disproving** the correctness of intended algorithms underlying a system with respect to a certain formal specification or property.

Application domains:

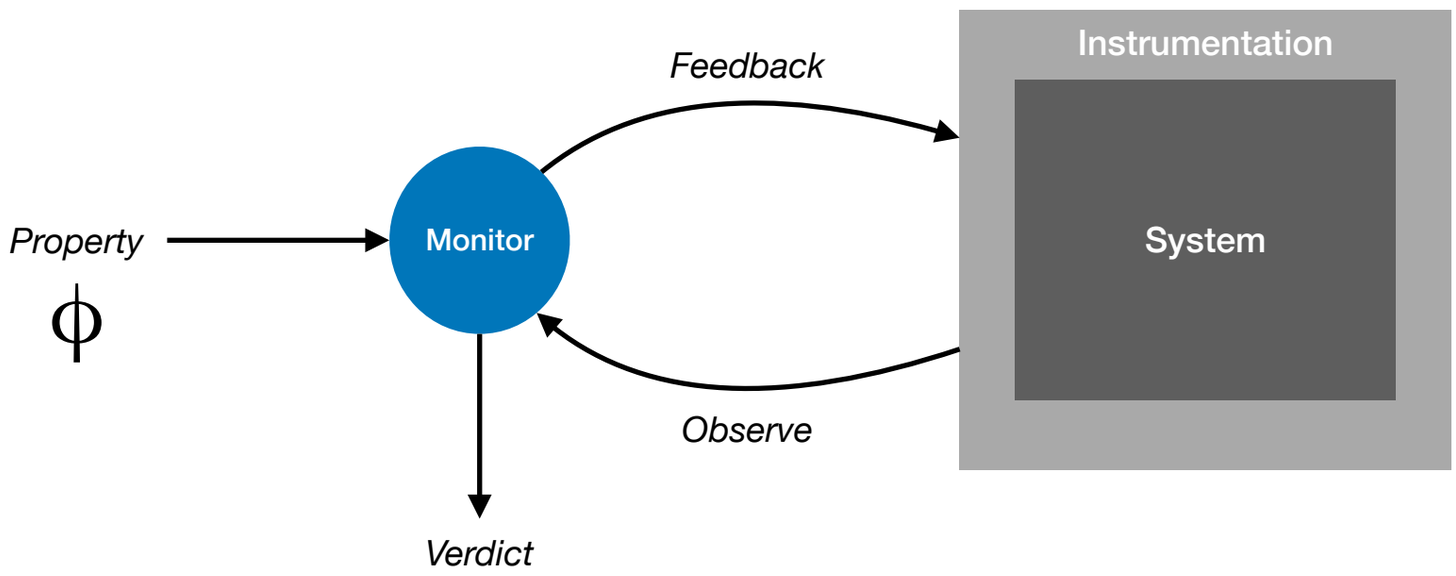
- Generally **safety-critical** systems: a system whose failure can cause death, injury, or big financial losses
- Embedded systems: often safety-critical and reasonably small (thus amenable to formal verification)

Runtime Verification

Complement of

- formal static verification (such as *Model Checking*)
  pros: formal, exhaustive **cons: suffers from scalability**
- testing.
  pros: scales well **cons: not formal, not exhaustive**

Dynamic checking of system behaviour using one (or multiple) **Monitor(s)**
pros: formal, scales well, can be done after deployment
cons: not exhaustive



Issues concerning Runtime Verification

Overhead

- 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**.

"Sorry, You crashed!"

- There might be scenarios where it is necessary to **anticipate** a violation (resp. satisfaction), because to report it only when it happens could be too late.

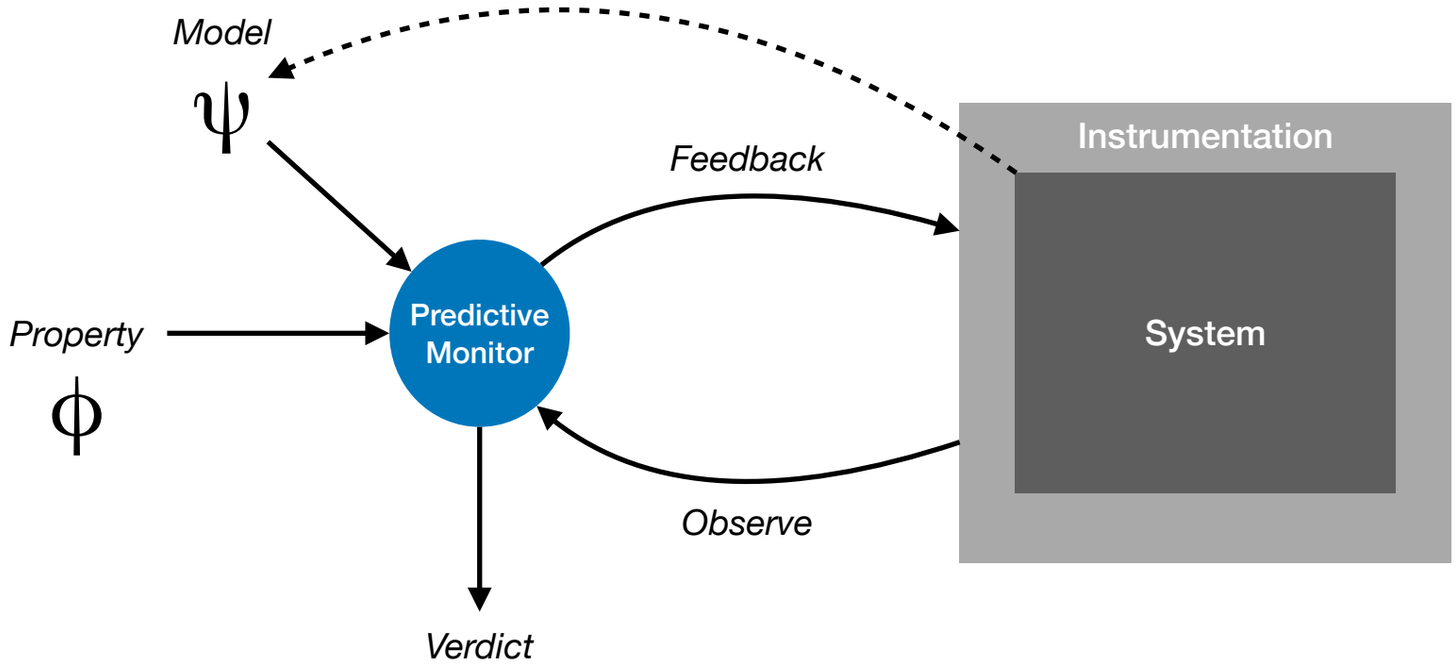
For both increasing **reliability** and **reducing the impact** of the monitors on the system, an extension of standard RV named **Predictive Runtime Verification** (PRV) has been proposed in the past.

Predictive Runtime Verification

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**.

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.



Issues concerning Predictive Runtime Verification

The problem with PRV is that a **model of the system 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 to observations collected at runtime.

The guiding principle here is to learn the model behaviour by observing real execution traces so as to create a sort of closed loop in which

1. **logs** are used to adjust the candidate models,
2. models are used to predict faults with certain **confidence level**,
3. the confidence level increases with the log size,
4. go back to 1.

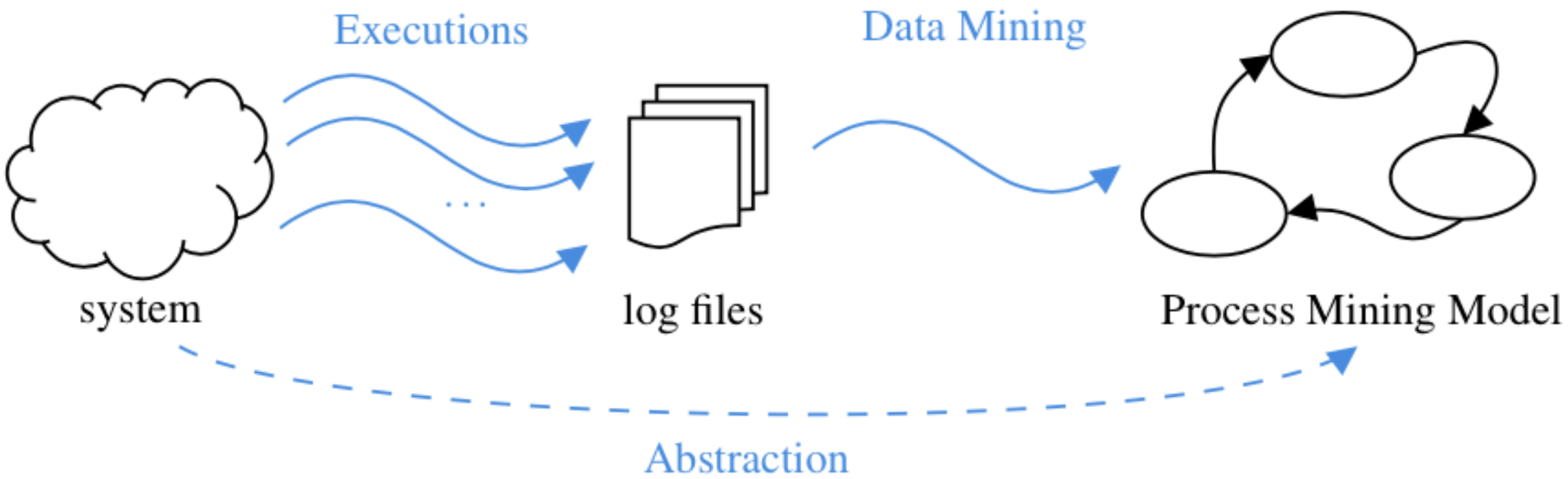
Specifically, we use **Process Mining** (PM) to automate the model generation phase in practical applications of PRV.

Process Mining

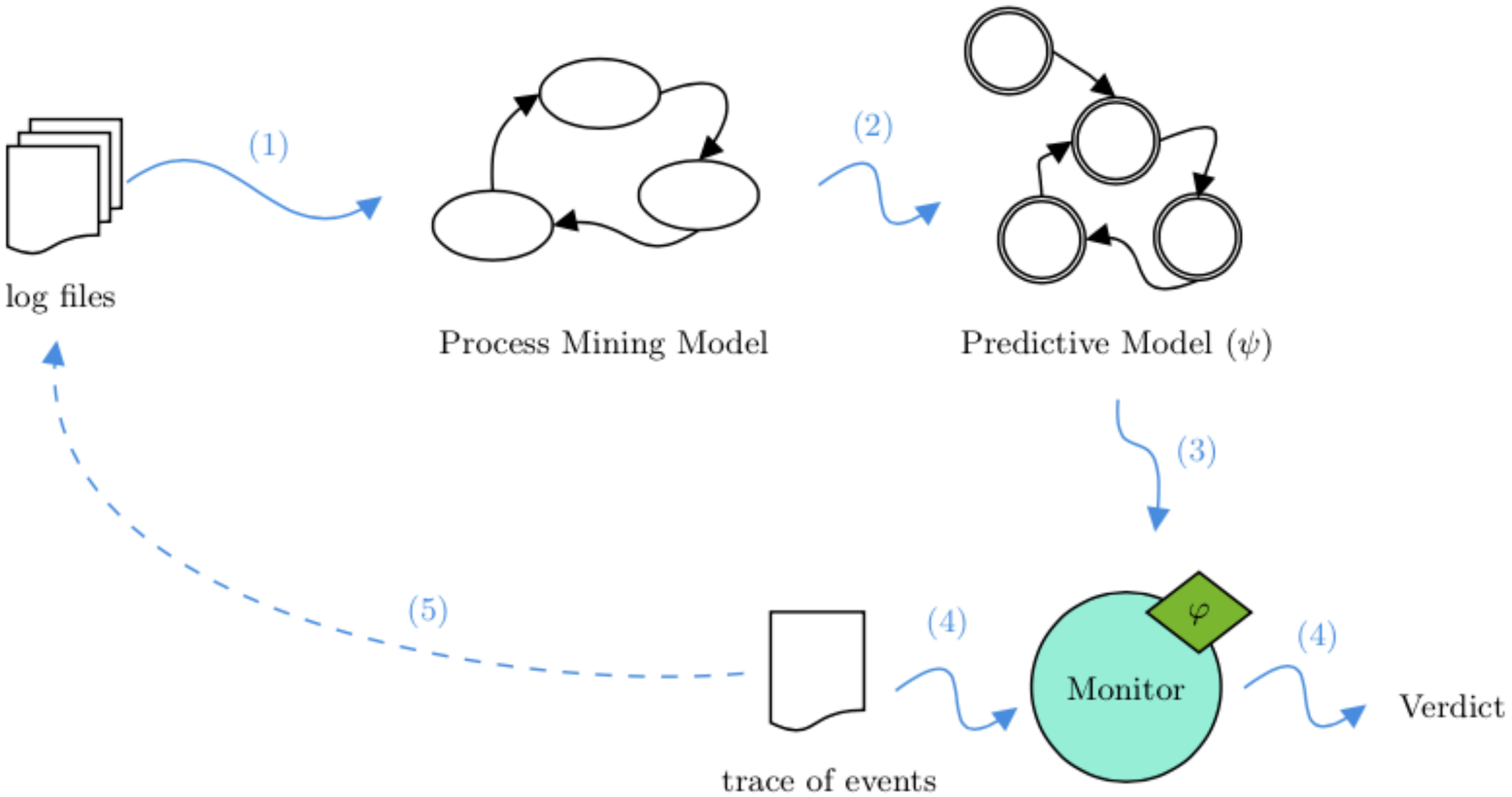
Process Mining is a technique used in software engineering to **automatically synthesise** a formal model which denotes the system behaviour. Such analysis is usually performed on event logs generated by multiple executions of the system.

In practice, by using data mining algorithms, knowledge is extracted by these logs and corresponding formal models are generated.

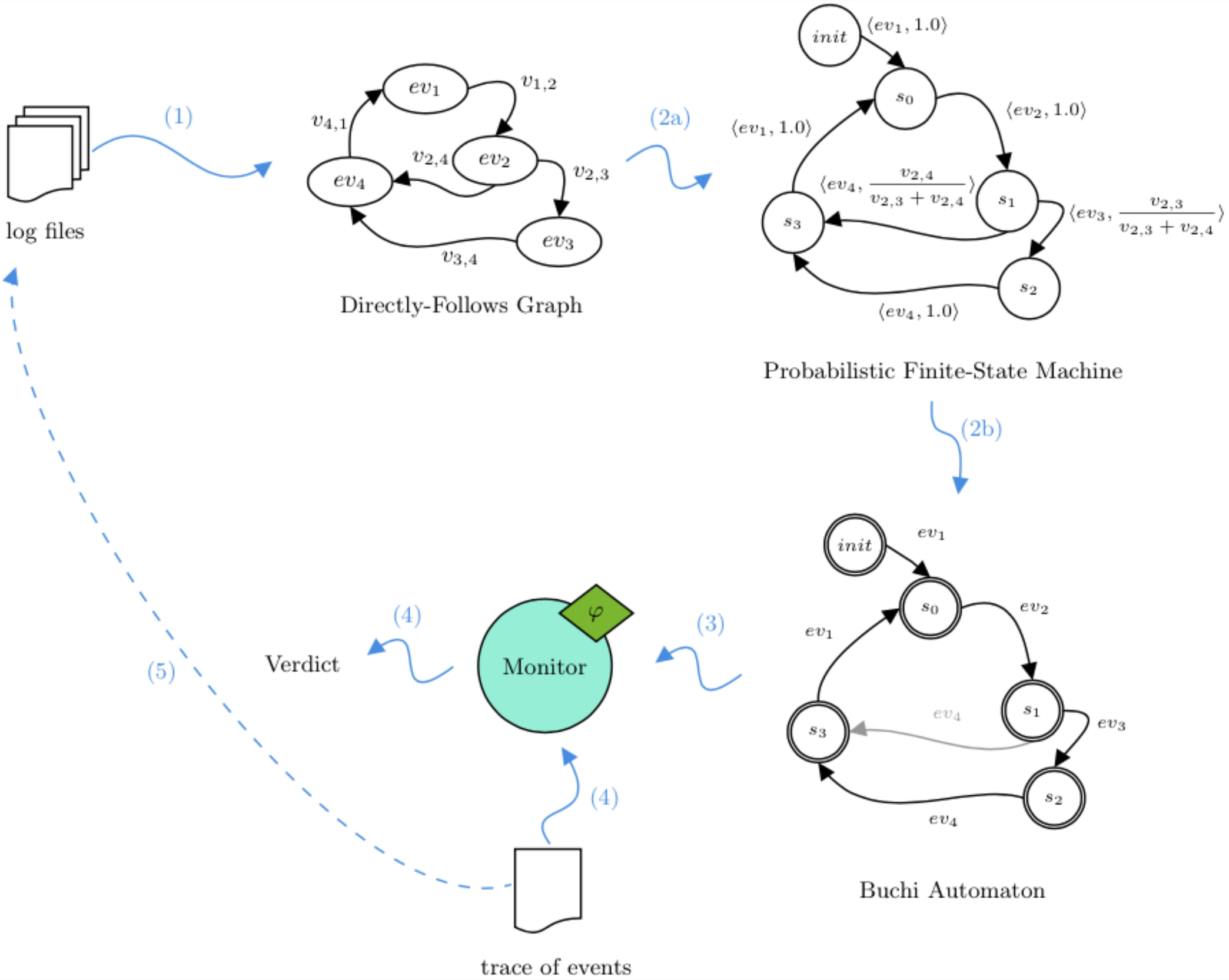
Since PM completely depends on the event logs generated by the system execution, **more logs** are used, and **more precise models** are extracted. Because of this, PM does not only allow PRV to be **applied when a model of the system does not exist**, but it also makes PRV **more robust and reliable**.



Incrementally Predictive Runtime Verification (iPRV)



iPRV instantiation



Implementation

A Python implementation of our approach is publicly available as a GitHub repository:

https://github.com/AngeloFerrando/IncrementallyPredictiveRV

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:

1. a set of log files (expressed as a single XES file), which is the standard format used in PM to represent event logs;
2. a threshold to guide the mapping from PFSM to BA;
3. an LTL property to verify;
4. a trace generated by the current system execution to analyse.

Conclusions and Future Work

Contribution

- A general verification workflow for integrating PM in the generation of predictive monitors.
- All the engineering steps that bring to the extraction of a model which can be used to predict future events and speed up the RV process.
- A general approach where no formalism is enforced. Nonetheless, to help better understanding the approach, we also show an instantiation with LTL properties and BA models.
- A Python prototype tool.

Future Directions

- 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.

UNIVERSITÀ DEGLI STUDI DI GENOVA

CILC 2021

Incrementally Predictive Runtime Verification

Angelo Ferrando and Giorgio Delzanno
University of Genova

angelo.ferrando@unige.it
Research Fellow

Thanks for your attention!