Runtime Verification of Autonomous Systems utilizing Digital Twins as a Service

Morten Haahr Kristensen*, Alberto Bonizzi†, Cláudio Gomes*, Simon Thrane Hansen‡, Carlos Isasa*, Hannes Iven§, Eduard Kamburjan¶, Peter Gorm Larsen*, Martin Leucker§, Prasad Talasila*, Valdemar Trøjgård Tang*, Stefano Tonetta†, Lars B. Vosteen§, Thomas Wright*

*Department of Electrical and Computer Engineering
Aarhus University, Denmark
{mhk, claudio.gomes, cisasa, pgl, prasad.talasila, valdemar.tang, thomas.wright}@ece.au.dk

†Fondazione Bruno Kessler, Italy
{bonizzi,tonettas}@fbk.eu

‡Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
simon.hansen@uni.lu

§Institute for Software Engineering and Programming Languages
Universität zu Lübeck, Germany
hannes.iven@student.uni-luebeck.de, {leucker,vosteen}@isp.uni-luebeck.de

¶Department of Informatics
University of Oslo, Norway
eduard@ifi.uio.no

***Abstract*—Autonomous Systems (AS) enable systems to adapt to drastic and unprecedented environmental changes, a capability that can be enhanced through the utilization of Digital Twins (DTs). However, the additional capabilities of AS come at the cost of explainability, as the expanding adaptation space complicates the reasoning about the system's behavior. For certain types of systems, it is crucial to ensure that specific properties are upheld despite the system's autonomous behavior. To facilitate the monitoring of these properties, we propose the use of Runtime Verification (RV). This tutorial demonstrates the integration of RV tools into the Digital Twins as a Service (DTaaS) platform to monitor and verify the behavior of AS in real-time. By exploring various methods to incorporate RV tools within a DT context, the tutorial aims to advance the application of RV technologies in autonomic computing and self-adaptive system design. Specifically, we demonstrate how the behavior of a self-configuring DT can be verified utilizing RV. This is accomplished through the DTaaS platform, which supports seamless deployment of DT-based AS.**

***Index Terms*—Self-adaptivity, runtime verification, digital twins, monitor, TeSSLa, NuRV**

I. Introduction

The vision of autonomic computing inspired new approaches to designing flexible Autonomous Systems (AS) capable of adapting to dynamic environments [14]. Initially, research on AS primarily focused on reducing the complexity of large-scale software systems, which often comprise tens of millions of lines of code, particularly within purely software-based environments such as cloud computing. Since then, the field has advanced significantly, extending its concepts to new domains including dynamic software architectures [1], robotics [4], business process management (BPM) [16], and cyber-security [17]. However, the increasing level of adaptability makes the verification of such systems significantly more challenging. For instance, within the domain of robotics, there is a critical need for flexible self-adaptive robots that can operate reliably despite dynamic environmental changes. Nevertheless, the safety of human life must never be compromised, and ensuring that these robots adhere to safety standards despite their self-adaptivity remains an ongoing challenge. Research addressing these challenges, such as [4] and [13], involves formulating and validating the adherence to requirements at runtime. Similar requirements for continuous monitoring of system properties are present in other domains utilizing AS. Within this context, self-adaptivity can be considered as a component that increases the possible behaviors of the system, while Runtime Verification (RV) serves as the component that excludes unwanted behaviors.

The purpose of the tutorial is to demonstrate how researchers within the AS community can utilize RV tools within their work to ensure the correctness of their systems. In doing so, emphasis is placed on the different ways that RV tools can be integrated within a deployment platform and utilized by the existing system. Through this process, we aim that attendees will become familiar with how monitoring services are deployed, as well as gain practical insight into how to build them. The tutorial adopts a hands-on approach, utilizing the

The work presented here is partially supported by the RoboSAPIENS project funded by the European Commission's Horizon Europe programme under grant agreement number 101133807, the O5G-N-IoT project, funded by the German Federal Ministry for Economic Affairs and Climate Action, due to a resolution of the German Bundestag, and the SM4RTENANCE project under grant no. 101123423.

Incubator case study [9], [10], which features a Digital Twin (DT) capable of self-configuring during anomalous situations. Although the tutorial is presented within the context of a DT, the majority of the concepts discussed extend beyond this specific application. Through the Incubator, we explore five different scenarios for integrating an RV tool within an existing AS, all of which are deployed on the Digital Twin as a Service (DTaaS) platform (detailed in Section II).

The tutorial is structured as follows: Section II introduces the main background concepts for following the tutorial. Then Section III presents five examples showcasing the implementation of monitoring using two different RV frameworks. Finally, Section IV concludes the tutorial.

II. Background

A. Runtime Verification

RV is a lightweight method of improving the integrity of deployed systems by extending a system with additional monitoring functionality to avoid unintended behavior at runtime. This is accomplished through a variety of monitoring techniques, which check whether a system conforms to a specification based on traces or streams of data from the running system.

A wide range of RV methods have been developed over the years, offering a variety of different specification languages for expressing the desired behavior of the system including temporal logics such as Linear Temporal Logic (LTL) [18] and Signal Temporal Logic (STL) [7] as well as domain-specific languages such as TeSSLa [6].

RV encompasses both passive monitoring techniques which focus on detecting errors without changing the behavior of the system, as well as more active techniques (also known as *runtime enforcement* [8]) which aim to block or correct bad behaviors.

B. NuRV

NuRV [5][1] is an extension of the nuXmv model checker for assumption-based LTL RV with partial observability and resets. Monitoring formulas are specified in LTL while assumptions are specified in SMV. Thanks to the assumption, the output of the monitor can be conclusive even in cases where the formula contains future operators or if not all variables are observable.

The tool provides commands for online/offline monitoring and code generation into standalone monitor code. Using the online/offline monitor, LTL properties can be verified incrementally on finite traces from the system under scrutiny. The code generation currently supports C, C++, Common Lisp, and Java, and is extensible. Furthermore, from the same internal monitor automaton, the monitor can be generated into SMV modules, whose characteristics can be verified by Model Checking using nuXmv.

[1]https://nurv.fbk.eu

C. TeSSLa

The Temporal Stream-based Specification Language (TeSSLa) [6] framework[2] combines a language and a suite of tools designed for real-time verification of systems through data stream analysis. TeSSLa allows the declaration of input data types and the transformation of this data into new, derived streams by applying a series of defined operations. This approach enables effective monitoring of complex systems, ensuring accurate tracking and analysis without overly complex processes.

TeSSLa provides extensive libraries and supports the creation of macros. These macros allow users to define custom operations, simplifying the specification of complex behaviors and increasing the accessibility of the language. TeSSLa also supports the generation of detailed output streams, including statistical data with precise event timestamps, and allows integration with monitoring tools developed in modern programming languages such as Rust and Scala. Its integration with the metrics collection agent Telegraf [21] contributes to its effectiveness in real-world applications. At its core, TeSSLa's strength lies in its ability to map input data to meaningful outputs, which is essential for real-time system monitoring and informed decision-making in areas such as DT technologies.

D. Digital Twins as a Service

The DTaaS[3] platform is a collaborative platform to build, use, and share DTs. It is based-off a microservices architecture with dedicated software containers[4] for DT assets, user workspaces, platform services, a front-end website, and service router.

One of the architectural principles used in the development of DTaaS is to conceive DTs as composed of reusable assets, which separate the functionality into their constituent parts. Within DTaaS, data, models [23], tools [19], services [20] and ready to use DTs [2] have been identified as reusable assets. The DT Assets software container provides an interface to perform create, reuse, update, and delete operations on the reusable stored within the DTaaS.

Users utilizing DTaaS have private workspaces in which they can build and use systems, from where they can access assets as a regular part of the filesystem. All workspaces have internet access thereby enabling the integration of DTs running inside workspaces with external software systems.

Out-of-the-box, DTaaS supports multiple commonly used services across DTs and users. The most commonly used are RabbitMQ and MQTT (communication), InfluxDB and MongoDB (data storage), and Grafana (data visualization). Additionally, it is possible to host private services accessible to a selected number of users. These services include the runtime services provided by TeSSLa and NuRV.

[2]https://tessla.io

[3]https://github.com/INTO-CPS-Association/DTaaS

[4]Container is a software component at level-2 of the C4 model.

E. FMI-based Co-simulation

Integrating verification methods early in development ensures system correctness from the start [22]. One approach is *co-simulation*, which combines multiple simulation tools into a single simulation [12], [15]. Co-simulation is crucial for modeling complex systems co-developed by multiple organizations and systems whose complexity transcends the capabilities of any single simulation tool.

Interoperability between heterogeneous simulation tools is achieved using Functional Mock-up Units (FMUs) defined by the Functional Mock-up Interface (FMI) standard [3]. An FMU encapsulates the behavior of a *dynamic system*, whose state evolves according to *evolution rules* and *external stimuli*, into a discrete trajectory. This allows complex behaviors to be represented modularly while protecting intellectual property.

Multiple FMUs are composed into a *scenario* by coupling their input and output ports to represent the behavior of a complex system. A *coupling* signifies that the state of one FMU (the output) directly influences the state of another (the input). A scenario is simulated using a co-simulation framework that interacts with the FMUs through their interface to advance them in lockstep and exchange values between the coupled ports.

III. Example integrations

Five examples showcasing the RV integration into the AS are presented below: three utilizing NuRV and two utilizing TeSSLa. For NuRV, the first example demonstrates a scenario where the components of a self-adaptive DT are validated before the system is deployed. This involves exporting the NuRV specification as an FMU and conducting co-simulations with the other components of the system. In the second example, the reusability of the FMU within a service-oriented architecture is demonstrated, enabling RV on the deployed system. It listens to real-time sensing data sent by the Physical Twin (PT), i.e., the physical counterpart of the system, through RabbitMQ to the DT, evaluating the truth value of LTL formulas. In the third example, the NuRV specification is deployed on a standalone server, with its services exposed to the DT. As a result, it is uncoupled from the DT instance. For TeSSLa, the two examples demonstrate passive and active monitoring. In passive monitoring, an alarm is raised in the event of a violation of the monitored conditions. In contrast, active monitoring entails altering the system's behavior if a monitored condition is falsified.

A. The Incubator

The different ways of integrating RV monitors are showcased using the Incubator system described in [11]. The objective of the Incubator is to keep the temperature inside a box close to a target temperature, a task that can be difficult to achieve when more sophisticated scenarios, such as the possibility of someone opening the lid or the object inside the box releasing heat, are considered. These considerations have led to the development of a DT [9], which consists of a dynamical model of the physical components of the PT and software components capturing its controller behavior. Additionally, the DT contains a self-adaptation service that reacts to possible changes in the environment and adjusts the Incubator's objective as necessary. In order to do this, a Kalman filter estimates the state of the system and compares it to the empirical data from the sensors. As soon as a deviation is detected, the DT looks at historical data to identify the anomaly and plan accordingly.

The self-adaptation service is divided into two different services: anomaly detection, which handles detecting the difference between the expected temperatures and the sensed temperatures, and energy saving, which changes the target temperature to a lower one in case the anomaly detection service has detected an opening of the lid. An overview of the interaction of these services and the PT can be seen in Fig. 1. The runtime monitoring property that is used throughout the examples, ensures the correct combined behavior of the anomaly detection and energy saver blocks. The STL property can be seen below:

$$\square(A \implies \Diamond_{[0,3]} S) \tag{1}$$

where A stands for the anomaly detection service detecting the opening of the lid and S stands for the energy saver service changing the target temperature. It can roughly be translated into: "It must always hold that if an anomaly is detected then energy saver is started within 3 seconds".

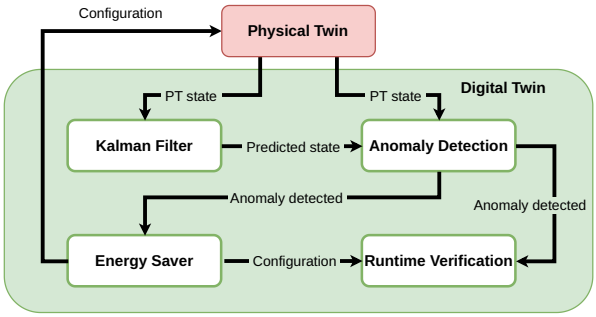

Fig. 1: High-level overview of the DT components relevant to the examples below. Arrows indicate RabbitMQ messages and associated data.

B. NuRV FMU monitor

NuRV provides the capability to export runtime monitors as standalone FMU components. This feature enables users to easily interface monitors within custom applications using a variety of libraries that support the FMU standard. This section outlines how to utilize these FMU monitors to perform an early validation of the internal components of the Incubator before its deployment. Specifically, it is shown how to validate the energy saver and anomaly detection components using an FMU monitor.

1) Monitor definition and integration: The monitor is defined by the SMV model seen in Fig. 2. This model specifies a safety LTL property: Whenever an anomaly occurs, the system should reconfigure itself and enter energy-saving mode within a maximum of `3` time steps. The output of this monitor can be a final verdict of either *true* or *false*, or *unknown* if there is insufficient information to reach a definitive conclusion.

```
MODULE main
VAR
   anomaly : boolean;
   energy_saving : boolean;
LTLSPEC -- Safety
   G (anomaly -> F [ 0, 3 ] energy_saving)
```

Fig. 2: NuRV monitor model

2) Simulation environment: For the validation process, a simulation environment was established comprising several components (depicted in Fig. 3): the *energy saver* and *anomaly detection* components, each encapsulated within distinct FMUs, along with the NuRV monitor, exported by NuRV using the specification in Fig. 2. The input data for the simulation is generated by a purpose-built FMU component named *source*, which supplies testing data, simulating an anomaly occurring at time `t=60s`. A final component, *watcher*, is employed to verify whether the energy saver activates in response to an anomaly reported by the anomaly detector. The FMUs for the energy saver and anomaly detector were constructed packaging their Python code using `unifmu`[5], while the *source* and *watcher* components were generated using `OpenModelica`[6]. `maestro`[7] served as the co-simulation engine.

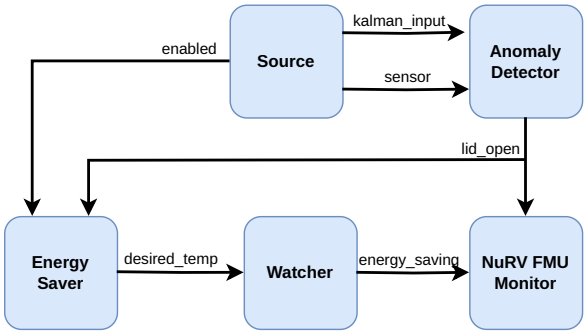

Fig. 3: Simulation architecture of components and their exchanged signals

3) Simulation creation and execution: The setup of the simulation is automatically performed through the `create` script, which installs the required dependencies and compiles the monitor from its specification model. The simulation is initiated using the DTaaS `execute` script, which also starts the *maestro* co-simulation engine to simulate the system. Given the characteristics of the FMU monitor exported by NuRV, each invocation of the *doStep* function corresponds to a logical heartbeat of the monitor. Consequently, this allows the monitor to assess the current values of its inputs and determine the appropriate outcome, thus providing validation of the system.

C. NuRV FMU service monitor

It is not possible to directly integrate the NuRV FMU monitor with the deployed Incubator. However, by implementing straightforward wrapper logic, it becomes viable to expose the FMU as an internal service, thereby enabling its utilization within the system. Theoretically, automating this process could be achieved through the development of a dedicated tool; however, no such tool currently exists to the authors' knowledge. For the purposes of this tutorial, a Python-based prototype has been developed to demonstrate the potential functionality of such a tool. It is important to underscore that this solution serves as a prototype only, and certain challenges, such as fault tolerance, remain unaddressed.

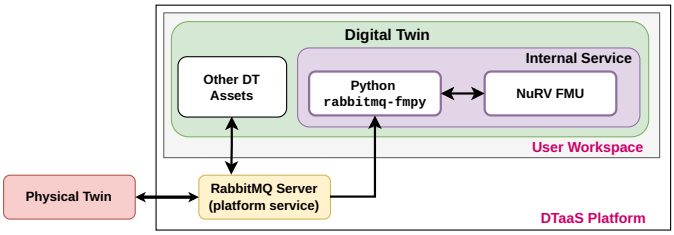

Fig. 4: Overview of components involved with the NuRV FMU service monitor. Notice that `rabbitmq` and `fmpy` are libraries.

1) Overview: As depicted in Fig. 4, the tool leverages the Python libraries `rabbitmq`[8] and `fmpy`[9], to realize its functionality. `rabbitmq` facilitates subscription to the RabbitMQ topics that are relevant for the monitor. `fmpy` enables the simulation of an FMU within Python, allowing the introduction of custom logic between each simulation step. In conjunction, this tool orchestrates its operations such that upon the occurrence of a new message on a RabbitMQ topic, the internal state of the Python program is updated, and the signals are subsequently forwarded to the FMU monitor, resulting in the generation of a new verdict.
Given the reuse of the FMU, the NuRV specification remains identical to the one outlined previously.

2) Creation, execution, and termination: As an extension of the configuration provided in Section III-B, the prerequisites are a superset of those previously outlined. Additionally, the Python libraries `fmpy` and `rabbitmq` must be installed. As a consequence, the `create` script fulfills the same function as

[5]https://github.com/INTO-CPS-Association/unifmu
[6]https://openmodelica.org/
[7]https://github.com/INTO-CPS-Association/maestro
[8]https://pypi.org/project/rabbitmq/
[9]https://pypi.org/project/FMPy/

described above, while also installing the requisite additional Python libraries.

In this configuration of the Incubator, an additional service in the form of the RV monitor is initiated concurrently with the DT. Given that the monitor is deployed as an internal service, it becomes the responsibility of the DT to manage the monitor, thereby intertwining their lifecycles. Consequently, the `execute` script commences the DT as usual but with the inclusion of starting the monitor. This enables the continuous monitoring of the anomaly detection and energy saving blocks, facilitating their verification at runtime.

D. NuRV ORBit2 monitor

Alternatively, NuRV can also be deployed as a standalone monitoring server service accessible to the DT. Consequently, the NuRV monitor and DT operate independently with their lifecycles entirely decoupled. This section delineates the steps to achieve this with the incubator.

1) NuRV monitor server: NuRV supports a network-based *monitoring server* mode: from the interactive shell mode, NuRV can enter with a command into a network listening state. This enables user code to remotely execute the heartbeat command for online monitoring. In server mode, NuRV can accommodate multiple clients connecting to multiple servers. In this context, each *monitor server* refers to a running NuRV process where numerous LTL properties are incorporated alongside their respective runtime monitors, established through the `build_monitor` command. It should be emphasized that a single NuRV process has the capacity to administer multiple monitors, each tailored to different LTL properties.

2) Monitor integration: The process of connecting the monitor server to the DT of the incubator is automated by the `execute` script. This script, in turn, employs a Python script file, that initially launches the `omniNames` CORBA Name Service utility from the *omniORB* toolset, followed by starting the `NuRV_orbit` version of NuRV. Subsequently, a connection is established with the monitor server using the *omniORB* Python library. Once this connection is established, the Python script starts the incubator DT and subscribes to relevant RabbitMQ topics such as energy saver status and lid open status. The lid open status is mapped to the anomaly for the NuRV monitor.

Figure 5 shows the architecture of the system comprising the incubator DT and the NuRV monitoring server. Upon receiving a message, the DT's status is relayed to NuRV via a heartbeat operation call through the CORBA interface. NuRV responds to this heartbeat by providing the monitor's output. If the monitor's output represents a final verdict, the monitor is reset to prepare for its utilisation for the subsequent execution of the DT.

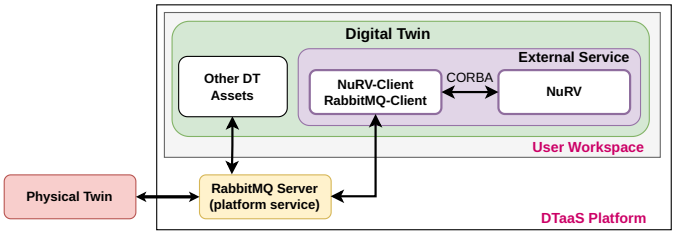

Fig. 5: Overview of components involved with the NuRV ORBit2 monitor.

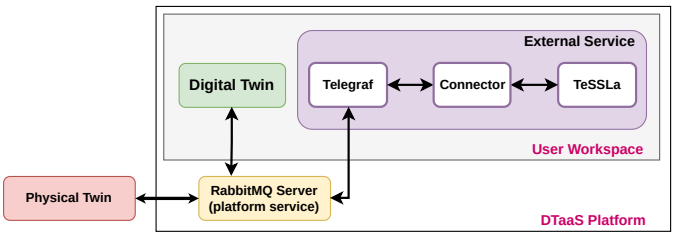

Fig. 6: Overview of components involved with the TeSSLA passive and active monitors.

E. TeSSLa passive monitor

As an alternative to NuRV, the RV tool TeSSLa can be utilized for monitoring the AS properties. Similar to the example presented in [21], the monitor consists of three parts (see Fig. 6). At its core, the TeSSLa monitor processes input streams and produces output streams, but is not itself capable of integrating them into a larger system context. A helper function (Connector) is compiled to handle the streams by connecting TeSSLa streams to sockets with which external tools can interact. Telegraf provides an additional layer of flexibility by adding:

- reconfigurability at runtime – the service can be configured to automatically adapt to a changing configuration file using the `--watch-config` flag, or simply restarted without losing the internal state of the monitor,
- data aggregation with basic statistical operations (such as count, mean, min or histograms),
- stream processing for filtering or transforming data streams, and
- by providing more than 200 integrations with different services and protocols to send or receive streams[10].

The `create` script prepares the system by ensuring that the necessary tools and software are installed and configured. It installs Java, Rust and Telegraf on the system and downloads the necessary files for the TeSSLa-Telegraf Connector[11]. Two files, a TeSSLa specification and a Telegraf configuration, must be provided by the user.

A TeSSLa specification suitable for this scenario (shown in Fig. 7) monitors two key states of the Incubator: whether the lid is open and whether the energy saving mode is used, which are passed to the TeSSLa monitor via different event streams. The helper function `raisingDelay` delays any change from `false` to `true` by three time steps without affecting changes from `true` to `false`. This function

[10]https://docs.influxdata.com/telegraf/v1/plugins/

[11]https://git.tessla.io/telegraf/tessla-telegraf-connector/-/blob/master/Release/tessla-telegraf-connector.zip

is used to define an internal data stream `critical` that represents when the energy saving mode is expected to be active. If it is not, an `alert` stream is set to `true`. This stream is sent back to the system. The `@TelegrafIn` and `@TelegrafOut` annotations allow the compiler to automatically create the Connector function and add to the Telegraf configuration. The Telegraf configuration consists of

```
include "./Telegraf.tessla"

@TelegrafIn("amqp_consumer","host=<hostname>",
↪  "lid_open")
in lid_open: Events[Bool]

@TelegrafIn("amqp_consumer","host=<hostname>",
↪  "energy_saver_on")
in energy_saver: Events[Bool]

def delayedOpen = raisingDelay(lid_open, 3)
def critical = lid_open && delayedOpen
def alert = critical && !energy_saver

@TelegrafOut("alert")
out alert

def raisingDelay(e: Events[Bool], d: Int):
Events[Bool] = merge3(false, const(true,
↪  delay(const(d, boolFilter(e)), e)),
↪  const(false, falling(e)))
```

Fig. 7: a TeSSLa specification for the passive monitor

two parts – where to connect to external data sources and sinks (RabbitMQ in this case), and how to connect to the TeSSLa monitor. The first has to be specified manually, as it depends on the specific case. Here it configures the AMQP plugin to connect to the RabbitMQ server, subscribe to the topics `incubator.diagnosis.plant.lidopen` as well as `incubator.energysaver.status` and publish the monitor verdict to the topic `incubator.energysaver.alert`.

The `execute` script uses the following command to add the configuration of how to communicate with the TeSSLa monitor to the supplied Telegraf configuration, create and run the Connector helper function, and compile as well as run the monitor.

```
./TesslaTelegrafConnector -i
↪  ./incubator.tessla -c ./telegraf.conf -r
```

Fig. 8: Command used within execute lifecycle script.

The script then starts the Telegraf service with `systemctl start telegraf`. This procedure allows data flow to and from the RabbitMQ broker, facilitating the collection, processing and monitoring of sensor data.

The `terminate` script stops the Telegraf service as well as the TeSSLa monitor and removes all temporary files.

F. TeSSLa active monitor

To use TeSSLa as a monitor for runtime enforcement, the TeSSLa specification (Fig. 7) and the Telegraf configuration must be changed.

To adapt the TeSSLa specification to control the energy saver mode instead of monitoring, the energy saver status is no longer needed as an input and the delayed signal can be provided as an output stream to switch on the energy saver if the lid is still open. Because only the rising edge is delayed, energy saving mode is switched off as soon as the lid closes.

The notable change in the Telegraf configuration is the line shown in Fig. 9 in the AMQP output plugin, which translates the boolean value for controlling the energy saving mode into the JSON format required by the Incubator. By adding this post-processing step to Telegraf[12], the user is able to change the formatting or desired temperature setting in the running system by changing the configuration without recompiling the monitor or losing its internal state.

```
transform = '{"temperature_desired":
↪  fields.value ? 21 : 35}'
```

Fig. 9: Telegraf JSON transformation

IV. Concluding remarks

This tutorial paper delineates the process of integrating RV within existing AS, by demonstrating different integration patterns through five use cases. Although the tool integrations has been demonstrated using NuRV and TeSSLa within a DT context utilizing DTaaS, the underlying concepts extend beyond these implementation details. Consequently, the learning outcomes can be generalized, enabling tutorial participants to apply RV tools in their research to provide stronger guarantees of the correctness of their AS. In the physical tutorial conducted at ACSOS 2024, the examples will be demonstrated sequentially, with a discussion of the deployment advantages and disadvantages of each approach. Participants will also have the opportunity to run the examples directly in the DTaaS platform.

References

[1] Albassam, E., Porter, J., Gomaa, H., Menasc, D.A.: DARE : A Distributed Adaptation and Failure Recovery Framework for Software Systems. In: IEEE International Conference on Autonomic Computing (2017), https://doi.org/10.1109/ICAC.2017.12

[2] Aziz, A., Chouhan, S.S., Schelén, O., Bodin, U.: Distributed Digital Twins as Proxies-Unlocking Composability and Flexibility for Purpose-Oriented Digital Twins. IEEE Access **11**, 137577–137593 (2023), https://doi.org/10.1109/ACCESS.2023.3340132

[3] Blockwitz, T., Otter, M., Åkesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., Viel, A.: Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In: Proc. 9th International Modelica Conference. Linköping University Electronic Press (2012)

[12]Telegraf first introduced the JSON transformation feature in version 1.24, which has not yet been widely distributed to package repositories.

[4] Cheng, B.H.C., Lansing, E., Clark, R.J., Lansing, E., Langford, M.A., Mckinley, P.K.: AC-ROS : Assurance Case Driven Adaptation for the Robot Operating System. In: 23rf ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 102–113. No. 1, ACM (2020), https://doi.org/10.1145/3365438.3410952

[5] Cimatti, A., Tian, C., Tonetta, S.: NuRV: A nuXmv Extension for Runtime Verification. 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. 382–392. Springer (2019), https://doi.org/10.1007/978-3-030-32079-9_23

[6] Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: temporal stream-based specification language. In: Formal Methods: Foundations and Applications: 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings 21. pp. 144–162. Springer (2018)

[7] Donzé, A.: On signal temporal logic. In: Runtime Verification: 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings 4. pp. 382–383. Springer (2013)

[8] Falcone, Y.: You should better enforce than verify. In: International Conference on Runtime Verification. pp. 89–105. Springer (2010)

[9] Feng, H., Gomes, C., Gil, S., Mikkelsen, P.H., Tola, D., Larsen, P.G., Sandberg, M.: Integration Of The Mape-K Loop In Digital Twins. In: 2022 Annual Modeling and Simulation Conference (ANNSIM). IEEE (Jul 2022), https://doi.org/10.23919/annsim55834.2022.9859489

[10] Feng, H., Gomes, C., Thule, C., Lausdahl, K., Iosifidis, A., Larsen, P.G.: Introduction to Digital Twin Engineering. In: 2021 Annual Modeling and Simulation Conference (ANNSIM). IEEE (Jul 2021), https://doi.org/10.23919/annsim52504.2021.9552135

[11] Feng, H., Gomes, C., Thule, C., Lausdahl, K., Sandberg, M., Larsen, P.G.: The Incubator Case Study for Digital Twin Engineering. arXiv:2102.10390 (2021)

[12] Gomes, C., Broman, D., Vangheluwe, H., Thule, C., Larsen, P.G.: Co-Simulation: A Survey. ACM Computing Surveys **51**(3) (2018)

[13] Jahan, S., Riley, I., Walter, C., Gamble, R.F., Pasco, M., McKinley, P.K., Cheng, B.H.C.: MAPE-K/MAPE-SAC: An interaction framework for adaptive systems with security assurance cases. Future Generation Computer Systems **109**, 197–209 (2020), https://doi.org/10.1016/j.future.2020.03.031

[14] Kephart, J., Chess, D.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003), https://doi.org/10.1109/MC.2003.1160055

[15] Kübler, R., Schiehlen, W.: Two Methods of Simulator Coupling. Mathematical and Computer Modelling of Dynamical Systems **6**(2) (2000)

[16] Malburg, L., Hoffmann, M., Bergmann, R.: Applying MAPE-K control loops for adaptive workflow management in smart factories. Journal of Intelligent Information Systems **61**(1), 83–111 (2023), https://doi.org/10.1007/s10844-022-00766-w

[17] Papamartzivanos, D., Gómez Mármol, F., Kambourakis, G.: Introducing deep learning self-adaptive misuse network intrusion detection systems. IEEE Access **7**, 13546–13560 (2019), https://doi.org/10.1109/ACCESS.2019.2893871

[18] Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. IEEE (1977)

[19] Qi, Q., Tao, F., Hu, T., Anwer, N., Liu, A., Wei, Y., Wang, L., Nee, A.: Enabling technologies and tools for digital twin. Journal of Manufacturing Systems **58**, 3–21 (2021)

[20] Robles, J., Martín, C., Díaz, M.: OpenTwins: An open-source framework for the development of next-gen compositional digital twins. Computers in Industry **152**, 104007 (2023)

[21] Streichhahn Hendrick and Duodaki, Abdul Rahman and Hyttrek, Christian and Wolf, Jakob and Kreth, Yannick: TeSSLa Telegraf Connector. https://tessla.io/blog/telegrafConnector/ (2024)

[22] Talasila, P., Craciunean, D.C., Bogdan-Constantin, P., Larsen, P.G., Zamfirescu, C., Scovill, A.: Comparison between the HUBCAP and DIGITBrain Platforms for Model-Based Design and Evaluation of Digital Twins. In: Proceedings of the 5th Workshop on Formal Co-Simulation of Cyber-Physical Systems. CoSim CPS (2021)

[23] Zambrano, V., Mueller-Roemer, J., Sandberg, M., Talasila, P., Zanin, D., Larsen, P.G., Loeschner, E., Thronicke, W., Pietraroia, D., Landolfi, G., Fontana, A., Laspalas, M., Antony, J., Poser, V., Kiss, T., Bergweiler, S., Serna, S.P., Izquierdo, S., Viejo, I., Juan, A., Serrano, F., Stork, A.: Industrial Digitalization in the Industry 4.0 Era: Classification, Reuse and Authoring of Digital Models on Digital Twin platforms. Array p. 100176 (2022). https://doi.org/https://doi.org/10.1016/j.array.2022.100176, https://www.sciencedirect.com/science/article/pii/S2590005622000352