Applying Passive Testing to an Industrial Internet of Things Plant

Marco Grochowski and Stefan Kowalewski
Embedded Software
RWTH Aachen University
Aachen, Germany
Email: {grochowski|kowalewski}@embedded.rwth-aachen.de

Melanie Buchsbaum and Christian Brecher
Laboratory for Machine Tools and Production Engineering
RWTH Aachen University
Aachen, Germany
Email: {m.buchsbaum|c.brecher}@wzl.rwth-aachen.de

***Abstract*—Safety and robustness play a crucial role in the context of the Industrial Internet of Things as autonomous and emergent behavior increase the complexity of Cyber-Physical Production Systems. Given the intractability of exhaustively verifying distributed production systems after modifications, testing and runtime monitoring seem to be two promising methods used to verify correctness in the digitally networked factory. Passive testing and external runtime monitoring are efficient and lightweight techniques that bridge the gap between testing and verification. This paper presents a framework for on-the-fly simulation of a specification relying on the Amazon Web Services Internet of Things architecture and the use of the digital shadow. The feasibility of the proposed architecture is evaluated using an industrial case study.**

Keywords–Passive testing, Industrial Internet of Things, Industrial Cyber-Physical Systems

I. INTRODUCTION

The growing demands for individual products and shorter product cycles caused a paradigm shift in manufacturing. The *Industrial Internet of Things* (IIoT) comes with many advancements, but also many challenges. While the technologies in use are well understood, the problem lies in translating applicable science and technology into engineering practice to meet future production needs [1]. The *Internet of Production* (IoP) [2] opens new possibilities for the interaction between different production systems by providing semantically adequate and context-aware data from development, production, and usage in real-time, on an adequate level of granularity. This is a blessing and a curse at the same time as insights gained from the emitted data during production are turned into data that controls the process. Consequently, this yields flexible value chains that are subject to a high degree of reconfigurability and experience an increasing complexity to meet their flexible demands. Beyond that, the data-driven IoP infrastructure, the highly iterative development, and agile manufacturing blur the distinction between design time and runtime, resulting in a lack of formal specifications in functionality, contexts, and constraints as the system is exposed to continuous changes in the environment, which take their tolls on the functional safety and reliability of software. Its validation must go beyond traditional validation using static methods, considering that not all scenarios are predictable during design time, due to the autonomous and emergent behavior.

Testing and runtime monitoring pose two potential approaches to tackle the emerging challenges for verifying the correctness in the digitally networked factory [3]. Based on this premise, this paper combines a specification-based, passive, black-box testing approach paired with runtime monitoring.

A. Approach

The techniques proposed in this paper are applied after the deployment of the system. Figure 1 shows the perception of the IoP and the shift of continuous quality assurance and testing to the operational phase. The systems require either the ability to test themselves while in operation or the existence of a monitoring component that is operated in parallel.
Currently, it is not possible to actively test the system during the operational phase as the components are interwoven, and each stimulus may trigger a response which cannot be intercepted. This leads to unwanted side effects and may disturb further process steps of the *System Under Test* (SUT). Furthermore, the system's functionality can not be interrupted arbitrarily during the operational phase (disregarding emergency stop), rendering a reset after each test case execution as infeasible. Because the system is heavily based on the exchange of asynchronous messages, non-deterministic behavior caused by, for instance, latency can complicate active testing and hinder the repeatability. The continuous monitoring and model-based passive testing of safety-critical properties during the operational phase may alleviate the risk of using the system in safety-relevant environments.

Nevertheless, the test cases generated with model-based testing [4] during earlier stages of development can be reused during the operational phase. The formal model from which test cases were derived can be used for passive testing, assuming that the specification used for generation is limited to the input and output behavior of the system.

As passive testing is a black-box testing approach, it relies on meaningful information exchanged between the industrial assets to claim properties about the internal behavior of the black-box. The passive tester runs on an external device, which listens to the communication to extract the relevant messages and therefore does not introduce any disturbances, slowing down the execution speed or interfering with the normal behavior of the system. Its purpose is to passively analyze the input and output behavior of the SUT to detect faults, and it is not intended for intervention.

The runtime monitor acts as a fail-safe, which triggers a safety response upon transitioning into an unsafe region to reduce the impact of the harm. External runtime monitoring is a good fit for closing the gap between testing and verification. It is a lightweight and scalable verification technique that does not necessarily rely on a specification per se but on individual requirements. The requirements and software components can evolve without repercussions on the external runtime monitor, and the physical separation, as with passive testing, guarantees no restrictions in the functionality of the monitored component.

 ISBN: 978-1-61208-755-9

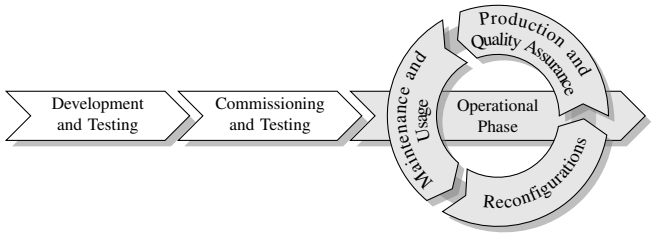

Figure 1. QA and Testing as perceived in the IoP following [3]

B. Contribution and Outline

The contribution of this paper is to evaluate the feasibility of a hybrid black-box testing approach for software quality assurance of an industrial case study in which the execution traces are observed, and the specification is assessed on-the-fly.

The remainder of this paper is structured as follows. Section II gives an overview of a related approach and delimits the contributions of this paper. Section III covers the preliminaries and introduces common definitions related to passive testing. Section IV introduces the architecture of the proposed system, shows how the system's behavior is supervised to give insights into the internal states and explains the interplay of the runtime monitor and the passive tester. Section V shows the application of the developed system using an industrial case study. Section VI draws a conclusion and presents future work.

II. Related Work

The contribution is heavily inspired by the work of Salva and Cao [5], yet it differs in many aspects. In their work, a combination of runtime verification and *ioco* passive testing is proposed. Instead of using a classical proxy or middleware to collect traces, they define a non-conformance relation using a formal model based on transition systems for testing a SUT and its specification with a so-called *proxy-monitor*. The proxy-monitor represents an intermediary between the environment and the SUT, which propagates the messages sent between those two entities, whereas the test monitor in this contribution only passively analyzes the traces. Given a specification modeled as an *input-output Symbolic Transition System* (ioSTS), Salva and Cao generate a proxy-monitor to check whether an implementation is ioco-conforming to its specification against a set of safety properties while analyzing the messages using the proxy-tester to detect failures. This contribution, as opposed to the work of Salva and Cao [5], abandons the idea of synthesizing one monitor from the specification and the safety requirement and instead keeps them separate - the focus is put on the specification in this contribution. This allows the specification and safety requirements to evolve and change during the operational phase without requiring a new synthesis of the monitor. For more information regarding the safeguarding of safety requirements during runtime and a brief overview on the related literature, the runtime monitoring algorithm based on requirements written in temporal logic is proposed in [6] by the authors. Further this contribution focuses on the application in an industrial context, whereas Salva and Cao applied their methodologies to the web service compositions deployed in *Platform as a Service* (PaaS) environments. Especially due to the high flexibility in the IIoT and the implications for the *Cyber-Physical Production Systems* (CPPS) this property is desired. Last but not least, the ioSTS model representing the functional behavior of the program is used to generate a monitor to check whether an implementation is ioco-conforming and meets safety properties in the work of Salva and Cao but this contribution directly executes the underlying model with the data from the observations. This leads to the work of Frantzen et al. [7] in which the state space explosion problem is avoided by lifting a test theory for *Labeled Transition Systems* (LTS) to their symbolic counterpart, where the data is treated symbolically. Instead of generating infinitely branching test cases offline as described in [7], the modeled specification in this contribution is unfolded on-the-fly, resulting in an efficient treatment of the possible infinite branching behavior.

Weiglhofer et al. [8] also build upon the *ioco* conformance relation and presented an approach for the selection of test cases using fault-based conformance testing. By mutating the specification syntactically, a fault is modeled at specification level such that the generated test cases fail if an implementation conforms to a faulty specification [8]. In this contribution deviating behavior from the specification is considered as faulty behavior and therefore sink states are introduced explicitly. It has yet to be shown, if the approach by Weiglhofer et al. [8] is a possible alternative to the ideas presented in this contribution regarding the test case selection outlined in the future work. Hierons et al. [9] proposed an algorithm for the construction of a monitor, which is able to handle asynchronous communication between the SUT and the monitor under certain conditions. Instead of operating on the constructed finite automaton the observed trace is used. The asynchronicity is of no concern for the passive testing in this contribution as the data is timestamped and utilized with regards to the event and not the processing time. This allows for more flexibility as delays in the communication are disregarded in the generation of the monitor.

Lima and Faria [10] provide an approach and an architecture that puts the testing of distributed and heterogeneous systems into a larger context. Of particular interest is the hybrid test monitoring approach, which was adopted from Hierons [11]. Hierons showed that multiple independent distributed testers that interact synchronously and a centralized tester that interacts asynchronously with the SUT are incomparable and result in different traces and faults [11]. Currently the techniques in this contribution were applied schematically to one specific processing station. Within this process station the communication was asynchronous and distributed, however the behavior was sequential and hence it was opted for a single tester that interacts synchronously with all the components in the SUT using the event time [11]. When scaling the use case, an approach similar to the one proposed by Lima and Faria [10] shall be considered. Hierons [11] mentions that it is possible to change the hybrid framework by making one of the local testers also act as the centralized tester. Lima and Faria [10] picked up this idea using a set of *Local Test Driving and Monitoring* (LTDM) components and a *Test Communication Manager* (TCM). The evaluation of this architecture is subject to future work of Lima and Faria, however, a similar approach shall be pursued for future work of this contribution.

In the next section, a partial introduction to the theory

behind passive testing is given.

III. Theoretical Background

As transition systems are a well-known formalism to model reactive systems, they are considered as a formal representation for the specification. However, the choice of the semantic model can vary as shown in [12]. A *transition system* TS [13] is a tuple $(S, Act, \rightarrow, I, AP, L)$, where

- S is a set of states,
- Act is a set of actions,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- $L\colon S \rightarrow 2^{AP}$.

The semantic model as-is currently abstracts from the interactions with the environment. An explicit distinction between actions initiated by the environment and actions initiated by the system is made to account for the asymmetric communication between the system and the environment, following the definition of Tretmans [4]. For modeling the input and output behavior of a transition system, the set of observable actions Act is partitioned into two disjoint sets, an input set Act_I, which denotes the set of actions initiated by the environment, and an output set Act_O, which denotes the set of actions initiated by the system itself, where $Act = Act_I \cup Act_O$ and $Act_I \cap Act_O = \emptyset$. Internal actions, which are unobservable, are all commonly denoted with τ, where $\tau \notin Act$, as fairness is not explicitly considered. Since the components are part of a distributed control system, which uses the network to interact through asymmetric communication, states which have no outgoing output transition are forced to wait for an input from the outside. Therefore it is possible that even though the SUT is composed of determinstic components, the outputs interleave non-determinstically. In addition to the latency of the communication, the aforementioned leads to delays in the occurrence of observations. For formalizing the property of a state in which no output actions are enabled, a special symbol δ, where $\delta \notin Act \cup \{\tau\}$, which is called quiescence is introduced. A state $s \in S$ of a transition system TS is quiescent, if and only if no transition with an output action from s exists, that is, $\forall a \in Act_O\colon \left\{s' \in S \mid s \xrightarrow{a} s'\right\} = \emptyset$.

Quiescence is made explicit by introducing self loops with the symbol δ for all states $s \in S$, which do not have an outgoing transition enabled for output actions. It is often useful to consider transition systems where the observable behavior is determinstic. This means that the transition systems have at most one outgoing transition labeled with an action $a \in Act$ per state and hence only one initial state. The determinized transition system, which may serve as a canonical representation, is referred to as the suspension automaton [4].

In order to formally describe the possible behavior of a transition system, the notion of execution fragments is defined. Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A *finite execution fragment* ρ of TS is an alternating sequence of states and actions ending with a state $\rho = s_0 a_1 s_1 a_2 \ldots a_n s_n$ such that $s_i \xrightarrow{a_{i+1}} s_{i+1}$ for all $0 \leq i < n$, where $n \geq 0$.

The introduction of the execution fragment gives rise to the formalization of the passive tester. A *passive tester* is modeled as a program graph and is derived from the suspension automaton. In contrast to the proxy-testers from [5], which use symbolic transition systems [14], a slightly deviating definition for modelling the specification is used. A *program graph* PG [13] is a tuple $(Loc, Act, \mathit{Effect}, \rightarrow, Loc_0, g_0)$, where

- Loc is a set of locations and Act is a set of actions,
- $\mathit{Effect}\colon Act \times Eval(Var) \rightarrow Eval(Var)$ is the effect function,
- $\rightarrow \subseteq Loc \times Cond(Var) \times Act \times Loc$ is the conditional transition relation,
- $Loc_0 \subseteq Loc$ is a set of initial locations,
- $g_0 \in Cond(Var)$ is the initial condition.

In order to extract the execution fragments of the program graph, it is assumed to behave like a transition system. Hence, the transition semantics of a program graph $TS(PG)$ over the set Var are given by the tuple $(S, Act, \rightarrow, I, AP, L)$

- $S = Loc \times Eval(Var)$
- $\rightarrow \subseteq S \times Act \times S$

$$\frac{\ell \xrightarrow{g\colon a} \ell' \wedge \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{a} \langle \ell', \mathit{Effect}(a, \eta) \rangle} \tag{1}$$

- $I = \{\langle \ell, \eta \rangle \mid \ell \in Loc_0, \eta \models g_0\}$
- $AP = Loc \cup Cond(Var)$
- $L(\langle \ell, \eta \rangle) = \{\ell\} \cup \{g \in Cond(Var) \mid \eta \models g\}$.

To allow the system to make progress autonomously on the actions that it initiates, a formalism is needed in which the environment never refuses the outputs and the system never refuses the inputs by the system's environment. Therefore, the program graph is augmented with a sink state $\perp$, which can be reached from all locations $\ell \in Loc$ by taking a transition with a non-enabled input action

$$\frac{\forall a \in Act_I\colon \ell \xrightarrow{g\colon a}\!\!\!\!\!\!/\;\;}{\ell \xrightarrow{g\colon a} \perp} \; . \tag{2}$$

Once in the sink state $\perp$, any behavior is possible. This ensures that the program graph is always capable of accepting an action from the environment.

This concludes the introduction of the preliminaries behind passive testing. For details and further information, the reader is referred to the work of Salva and Cao [5] and Frantzen et al. [14].

IV. Adaptation to Industrial Internet of Things

Figure 2 gives a high-level overview of the architecture. The adapter can be seen as a semi-formal interface for transforming the messages passed between the adapter and the SUT into a suitable representation for the test and runtime monitor. In this contribution, the emphasis is put on the test monitor. However, a brief overview of the runtime monitor is given in the following.

The runtime monitor analyzes the execution traces provided via the adapter and concludes a certain property about the SUT. The property of interest is derived from the requirements, which usually originate from the design time and are given in natural language. Requirements describe, for instance, the relationship between two occurring events in which the second event must occur within a given time bound of the occurrence

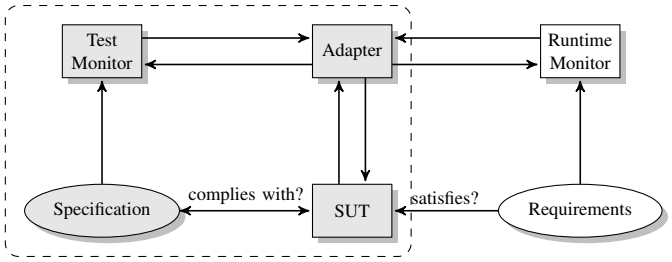

Figure 2. Overview of the architecture.

of the first event. As we only have a black-box view of the system, the possible monitorable requirements are limited to the observable properties. For the use in runtime monitoring, these requirements need to be transformed into formal logic, for instance, using *Metric Temporal Logic* (MTL).

Even though the runtime monitor is able to reason about the future time fragment of MTL, we limit ourselves only to the past fragment, because we can't set any fixed boundaries due to the inaccuracies caused by the asynchronous communication. The runtime monitor currently implements a rudimentary fail-safe, which issues the SUT to halt, neglecting any additional context information, in case of a violation. It was proposed in [6], and the reader is asked to consult the reference for details and further information.

During the execution of the SUT, the test and runtime monitor run in parallel. The runtime monitor is responsible for guaranteeing that the requirements are not violated, whereas the test monitor gives insights into whether the implementation deviates from the specification. The specification describes the behavior of the SUT and is used to derive the program graph for the test monitor, after determinization. The test monitor starts the simulation from the initial state in the transition system described by the program graph of the specification, that is, an initial location $\ell \in Loc_0$ and an initial evaluation η. If a new observation arrives, it is first preprocessed by the corresponding adapter before being passed to the test monitor. The test monitor receives either an input action with its parameters or an output action with the related digital shadow. The test monitor then proceeds with checking whether the program graph is able to make a transition from the current location ℓ_i to the next location ℓ_j with the received action taking the guard and the current evaluation of the variables into consideration. If the transition is possible, the test monitor continues with the simulation. In case ℓ_j is the sink state $\perp$, the test monitor stops the current simulation and saves the execution fragment up to and including ℓ_j for further analysis. It then backtracks to the last location, in which the specification and the SUT were conforming and continues the simulation from there on while logging arbitrary behavior until the initial location is reached again. This is justified by the fact that in case a severe violation occurred, it was hopefully already detected by the accompanying runtime monitor, which put the system into a safe state. Since the execution of a production line usually has a cyclical behavior, the test monitor and the SUT are synchronized in their initial location by (re-)setting the values of the variables. The execution fragment after the backtracking up to the reset is also kept for further analysis. If no observation arrives, the adapter passes a special symbol to the test monitor which is interpreted as quiescence. For practical reasons, a timeout for the observation of quiescence is introduced, such that if in any location a given time bound is exceeded, the program graph is transitioned into the sink state $\perp$. The given time bound may vary from location to location, taking into consideration the specified behavior. Currently, the adapter checks if an observation arrived in the past second, and if this is not the case, the special symbol is issued to the test monitor.

As mentioned earlier, the execution fragments and their simulation results are saved in order to guide the testing process during maintenance or for regression testing. They can be used to prioritize test cases by checking if the execution fragment matches a predefined test case. If that is the case, the test case should receive less importance during the testing process in maintenance as other test cases, which occurred less frequently with the same criticality. Furthermore, using the execution fragments obtained after backtracking, it is possible to investigate whether the test monitor was underspecified for a specific sequence of in- and output observations. The execution fragments that lead to the sink state $\perp$ can be used for debugging and aid the developer to validate the behavior of the SUT after modifying the software by fixing a bug, for instance.

In the next section, first, the case study is introduced. Following that, it is explained how the specifications modeled in a subset of UML and SysML are translated into a program graph used for passive testing, and a brief evaluation of the approach is given.

V. Case Study

The presented approach is validated using an industrial use case, which represents a part of the completion process from a windshield production. It has three processing stations: *Cleaning* (cleaning the windshield), *Priming* (application of a primer), and *Quality Assurance*. The *Cleaning* subprocess was used for evaluation, and it consists of a proximity sensor, a pneumatic suction cup including a valve, a camera, and a robot equipped with a cleaning tool. Each of these components, from now on, referred to as *industrial assets*, has a task-specific *digital shadow*. For further details, the reader is referred to [15].

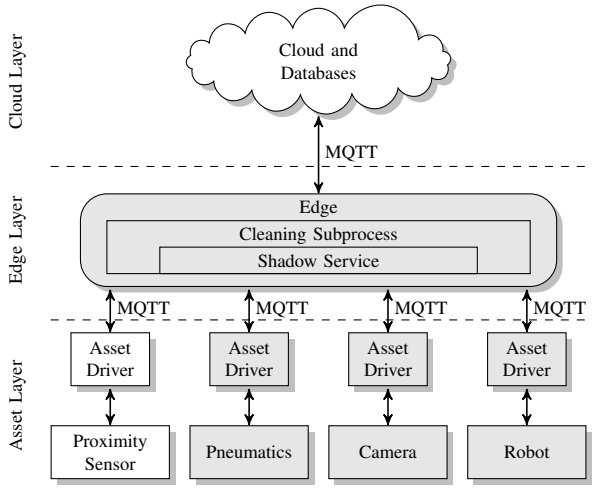

Figure 3. Overview of the *Cleaning* subprocess.

A. Digital Shadow

Even though the term *digital shadow* is ubiquitous, the notion of its concept still differs. The digital shadow comprises task-specific data of the processes, which allows for the reconstruction of the entire life-cycle of an industrial asset [16].

In this case study, Amazon Web Services was used to implement the control of the completion process. The digital shadow serves as a method of data aggregation and refinement for the control of the *Cleaning* subprocess, as shown in Figure 3 by the superordinate shadow service. Each industrial asset uses a digital shadow for its virtual representation and is controlled by an asset driver, which possesses a shadow service that is responsible for the communication with the superordinate shadow service of the entire *Cleaning* subprocess. The industrial assets can communicate locally with each other via the asset drivers using an edge device. All messages are exchanged and transmitted through the use of the *Message Queuing Telemetry Transport* (MQTT) protocol, as shown in Figure 3.

B. MQTT

MQTT is a lightweight and asynchronous *machine to machine* (M2M) protocol based on TCP/IP. It offers a 1-to-n connection and three *Quality of Service* (QoS) levels. Unlike request/response protocols such as HTTP, MQTT uses a publish/subscribe pattern of topics via a message broker, which reflect the hierarchical structures of the systems. Each industrial asset was assigned a $shadow/update$ topic, to which updates of its digital shadow can be sent. Similarly, other messages with additional information or describing certain actions were defined in [16].

C. Modeling the Use Case

The behavior of the SUT is specified using state machines in a subset of UML and SysML as depicted in Figures 4-7. In the following, the workflow of the *Cleaning* subprocess is described. The proximity sensor detects whether a windshield has been inserted into or removed from the workpiece carrier. This information is transferred to the asset driver via I^2C as a $24\,V$ signal if a windshield is in range of the sensor or a $0\,V$ signal if not. The *asset driver* passes this change via MQTT to the superordinate shadow service, as shown in Figure 3. The superordinate shadow service then proceeds with updating the digital shadow and issues a message, if the update was successful, with the corresponding content of the updated digital shadow via the respective $shadow/update/accepted$ topic. Figure 4 shows exemplarily the specification of the *Cleaning* subprocess. The focus in the subsequent section is put onto the control process after the proximity sensor has detected a workpiece in the workpiece carrier. All subsequent processes are triggered by the control logic of the *Cleaning* subprocess. As soon as the digital shadow of the proximity sensor indicates that a windshield has been placed in the workpiece carrier, the shadow service of the *Cleaning* subprocess sents a message to open the valve to the pneumatics asset driver using the topic $fpl/cleaner/cleaner\_pneumatics$, as illustrated in Figure 5. The asset driver of the pneumatics responds to this message by opening the valve and confirms the change afterward using its shadow service. As soon as the $shadow/update$ of the pneumatics is propagated in the shadow service of the superordinated shadow service, the control logic triggers the identification step (Figure 6) in which a camera detects the product identifier of the windshield and transfers it back to the control logic. Based on this information, the superordinated shadow service sends a message to the asset driver of the robot to start it (Figure 7). Once the robot has finished and updated its digital shadow, a message is sent to the pneumatic asset driver to close the valve. As soon as the asset driver of the pneumatics received the message and closed the valve, the digital shadow of the *Cleaning* subprocess is

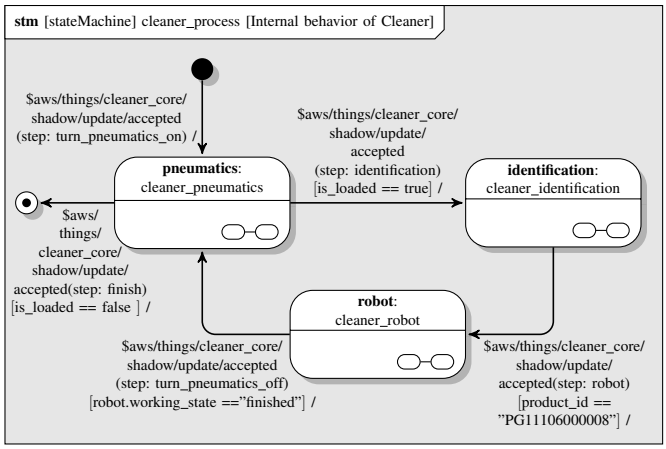

Figure 4. State machine of the *Cleaning* subprocess.

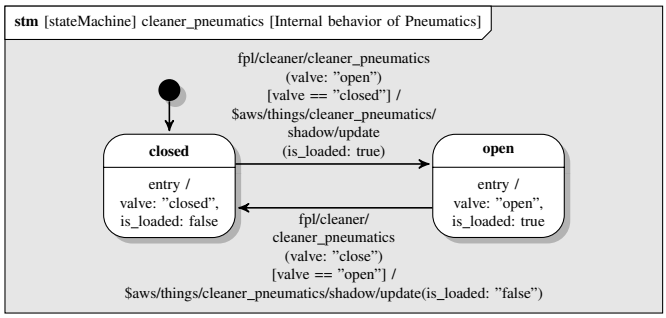

Figure 5. Sub-state machine cleaner_pneumatics of the *Cleaning* subprocess.

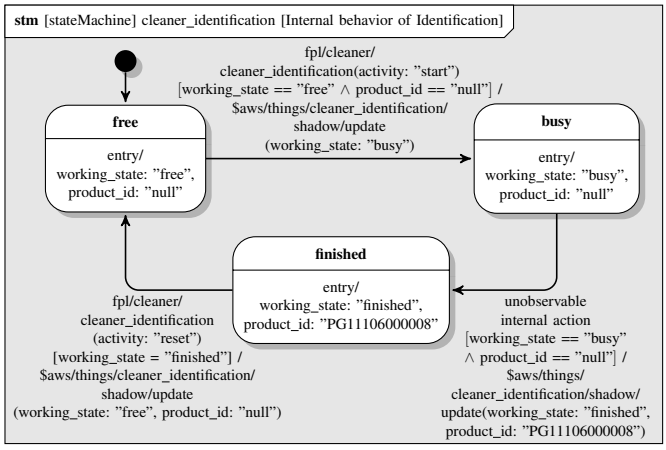

Figure 6. Sub-state machine cleaner_identification of the *Cleaning* subprocess.

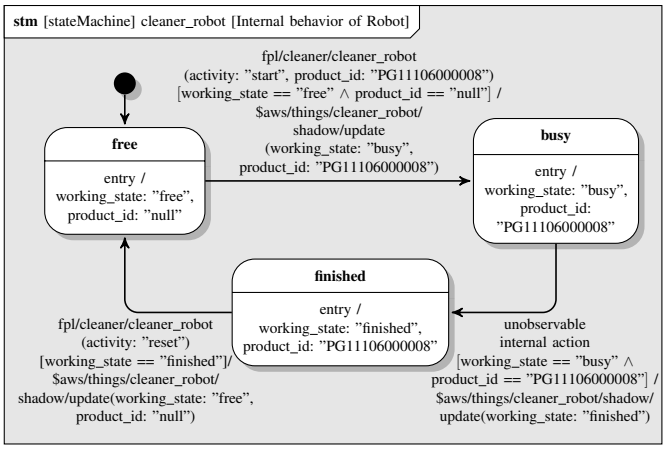

Figure 7. Sub-state machine cleaner_robot of the *Cleaning* subprocess.

updated to the *finished* state. Further, after the windshield has been removed and the proximity sensor no longer registers the windshield, the cell updates its state to *free*.

D. Transformation and Application

The transformation of the specification given in SysML to the program graph used by the test monitor is currently a manual task. The variables occurring in the digital shadow are modeled as atomic propositions $a \in AP$, which serve as invariants in the respective location. The topics of the MQTT messages are mapped to the actions Act of the program graph and are modeled as *signals* with their corresponding properties in SysML. The $shadow/update$ topics are always interpreted as outputs of the SUT, and the $shadow/update/accepted$ messages as input actions. Furthermore, all $fpl/cleaner$ messages are interpreted as input actions. Concretely, the transitions are interpreted as follows: the trigger of a transition is an input action in Act_I, the guard is a guard in $Cond(VAR)$, and the effect is an output action in Act_O. Before a guarded transition is taken, the associated guard is evaluated using the current evaluation of the variables in the source location of the transition.

It is important to note that the signals should not be modeled as in- and outputs at the same time. The message $fpl/cleaner/cleaner\_identification$, for instance, should not be modeled as an output from pneumatics to identification in the *Cleaning* subprocess and as an input in the state machine of cleaner_identification from *free* to *busy*, because that would not reflect the way the messages are passed using AWS. The change of state must take place beforehand, and this can be done, e.g., by the message $\$aws/things/cleaner\_core/shadow/update/accepted$.
The $entry/$ keyword of a state in the SysML state machine implicitly models a $shadow/update/accepted$ message. For example, in Figure 5, the transition from $closed$ to $open$ with the trigger $fpl/cleaner/cleaner\_pneumatics$ has an output action $shadow/update$ as an effect, which triggers a $shadow/update/accepted$ message in the $entry/$ method of the state *busy* and sets the values of the variables in VAR implicitly on the evaluation derived from the digital shadow.

The experimental evaluation of the hybrid-approach was done on a Raspberry Pi 3 Model B+, which was added as an additional industrial asset beneath the *Cleaning* subprocess. The industrial asset was subscribed to all occurring topics in the specification using the adapter.

E. Results and Insights

The test monitor was able to detect deviations from the specification from the behavior of the SUT at runtime. There were no severe errors, only a few implementation inaccuracies. For instance, cleaner_identification was exposed to a faulty $shadow/update$. The digital shadow was set to *busy* even though cleaner_identification was in the state *finished*, and cleaner_robot was started already. Furthermore, cleaner_robot immediately switches its state from *free* to *finished*. Consequently, the state *busy* was never set in the $shadow/update$. Last but not least, cleaner_pneumatics receives an $activity$: "$start$" message but isn't implemented with the *free*, *busy* or *finished* concept in mind. The cleaner_process updates its own digital shadow using an internal function of AWS and hence does not send any $shadow/update$ messages. This restricts the set of observable messages to the $shadow/update/accepted$ messages.

VI. Conclusion

It was shown that a specification-based, passive, black-box testing approach paired with runtime monitoring is an appropriate way for improving the quality assurance during operation. While the model-based testing theory describes how to derive test cases, it does not state how to prioritize or select test cases. Therefore, the execution fragments can be used to guide testing during maintenance by prioritizing test cases, which were not observed during runtime or by focusing on the test cases that failed during machine operation. Another benefit of bookkeeping the execution fragments is the possibility to recheck them against a variety of system properties, which have not yet been considered. In the case of machine modification and reconfigurations, the execution fragments can be used in regression testing. Another possibility for the test case selection poses the work of Weiglhofer et al. [8], which uses a fault-based testing technique and can also be applied to the use case from this contribution.

A. Outlook

Currently, the techniques were applied schematically to the *Cleaning* subprocess. Future work shall extend the methodologies to the other subprocesses and also consider their distributed communication. Here, the approaches by Hierons [11] and the concept of Lima and Faria [10] shall be examined.

The runtime monitor currently halts the execution of the system by sending a stop message in case a violation is detected. In some situations, this may lead to damage of the product or the machine. Improved routines could be developed by considering context information such that no harm is caused to the product or machine.
Currently, it is not possible to apply a stimulus to the SUT without affecting other industrial assets. It is expected that testing in idle phases of the process increases reliability. Future work shall enable testing during runtime.

If a deviation from the specified behavior is detected, but the system remains in a state that is not violating, the model of the specification might be underspecified. In this case, appropriate suggestions for updating the model of the specification

 ISBN: 978-1-61208-755-9

to improve the quality could be proposed. Furthermore, the derivation of the test monitor from the specification modeled in SysML is currently a manual task, and error-prone, which shall be automated in future work. Last but not least, an evaluation of how well this approach aids in regression testing is pending.

ACKNOWLEDGMENT

The presented research is a work-in-progress in the Cluster of Excellence (CoE) on "Internet of Production" funded by the German Research Foundation DFG. The CoE "Internet of Production" advocates the vision of enabling a new level of cross-domain collaboration between several institutes at the RWTH Aachen University by providing semantically adequate and context-aware data from production, development and usage in real-time, on an adequate level of granularity. The authors would like to thank the German Research Foundation DFG for the kind support within the Cluster of Excellence "Internet of Production" (Project-ID: 390621612).

REFERENCES

[1] S. Jeschke, C. Brecher, H. Song, and D. B. Rawat, Eds., Industrial Internet of Things - Cybermanufacturing Systems, ser. Springer Series in Wireless Technology. Cham: Springer International Publishing, 2017. [Online]. Available: https://doi.org/10.1007/978-3-319-42559-7 [accessed: 2019-10-07]

[2] J. Pennekamp et al., "Towards an Infrastructure Enabling the Internet of Production," in Proceedings of the 2nd IEEE International Conference on Industrial Cyber-Physical Systems (ICPS '19), May 6-9, 2019, Taipei, TW. IEEE, 5 2019, pp. 31–37. [Online]. Available: https://www.comsys.rwth-aachen.de/fileadmin/papers/2019/2019-pennekamp-iop-infrastructure.pdf [accessed: 2019-10-07]

[3] M. Weyrich and A. Zeller, "Testing industry 4.0 systems - how networked systems and industry 4.0 change our understanding of system testing and quality assurance," 4. VDI-Fachtagung mit Fachausstellung, Düsseldorf, Jan. 2016, oral Presentation with Slides. [Online]. Available: https://www.ias.uni-stuttgart.de/en/research/presentations/ [accessed: 2019-10-07]

[4] J. Tretmans, "Model based testing with labelled transition systems," in Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers, ser. Lecture Notes in Computer Science, R. M. Hierons, J. P. Bowen, and M. Harman, Eds., vol. 4949. Springer, 2008, pp. 1–38. [Online]. Available: https://doi.org/10.1007/978-3-540-78917-8_1 [accessed: 2019-10-07]

[5] S. Salva and T. Cao, "Proxy-monitor: An integration of runtime verification with passive conformance testing," IJSI, vol. 2, no. 2, 2014, pp. 20–42. [Online]. Available: https://doi.org/10.4018/ijsi.2014040102 [accessed: 2019-10-07]

[6] M. Grochowski, S. Kowalewski, M. Buchsbaum, and C. Brecher, "Applying runtime monitoring to the industrial internet of things," 2019, to appear in IEEE Xplore. [Online]. Available: https://tinyurl.com/etfa2019-preprint [accessed: 2019-10-07]

[7] L. Frantzen, J. Tretmans, and T. A. C. Willemse, "Test generation based on symbolic specifications," in Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers, ser. Lecture Notes in Computer Science, J. Grabowski and B. Nielsen, Eds., vol. 3395. Springer, 2004, pp. 1–15. [Online]. Available: https://doi.org/10.1007/978-3-540-31848-4_1 [accessed: 2019-10-07]

[8] M. Weiglhofer, B. K. Aichernig, and F. Wotawa, "Fault-based conformance testing in practice," Int. J. Software and Informatics, vol. 3, no. 2-3, 2009, pp. 375–411. [Online]. Available: http://www.ist.tugraz.at/aichernig/publications/papers/ijsi2009.pdf [accessed: 2019-10-07]

[9] R. M. Hierons, M. G. Merayo, and M. Núñez, "An extended framework for passive asynchronous testing," J. Log. Algebr. Meth. Program., vol. 86, no. 1, 2017, pp. 408–424. [Online]. Available: https://doi.org/10.1016/j.jlamp.2016.02.004 [accessed: 2019-10-07]

[10] B. Lima and J. P. Faria, "An approach for automated scenario-based testing of distributed and heterogeneous systems," in ICSOFT-EA 2015 - Proceedings of the 10th International Conference on Software Engineering and Applications, Colmar, Alsace, France, 20-22 July, 2015., P. Lorenz and L. A. Maciaszek, Eds. SciTePress, 2015, pp. 241–250. [Online]. Available: https://doi.org/10.5220/0005558602410250 [accessed: 2019-10-07]

[11] R. M. Hierons, "Combining centralised and distributed testing," ACM Trans. Softw. Eng. Methodol., vol. 24, no. 1, 2014, pp. 5:1–5:29. [Online]. Available: https://doi.org/10.1145/2661296 [accessed: 2019-10-07]

[12] J. Peleska, "Industrial-strength model-based testing - state of the art and current challenges," in Proceedings Eighth Workshop on Model-Based Testing, MBT 2013, Rome, Italy, 17th March 2013., ser. EPTCS, A. K. Petrenko and H. Schlingloff, Eds., vol. 111, 2013, pp. 3–28. [Online]. Available: https://doi.org/10.4204/EPTCS.111.1 [accessed: 2019-10-07]

[13] C. Baier and J. Katoen, Principles of model checking. MIT Press, 2008.

[14] L. Frantzen, J. Tretmans, and T. A. C. Willemse, "A symbolic framework for model-based testing," in Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers, ser. Lecture Notes in Computer Science, K. Havelund, M. Núñez, G. Rosu, and B. Wolff, Eds., vol. 4262. Springer, 2006, pp. 40–54. [Online]. Available: https://doi.org/10.1007/11940197_3 [accessed: 2019-10-07]

[15] C. Brecher, M. Buchsbaum, and S. Storms, "Control from the cloud: Edge computing, services and digital shadow for automation technologies*," in 2019 International Conference on Robotics and Automation (ICRA), May 2019, pp. 9327–9333. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8793488 [accessed: 2019-10-07]

[16] C. Brecher, M. Obdenbusch, M. Buchsbaum, T. Buchner, and J. Waltl, "Edge computing and digital shadow: Key technologies for the automation of the future," wt Werkstattstechnik online, vol. 108, no. 5, 2018, pp. 313–318. [Online]. Available: https://publications.rwth-aachen.de/record/726028 [accessed: 2019-10-07]

 ISBN: 978-1-61208-755-9