JOT
JOURNAL OF
OBJECT TECHNOLOGY

Behavioral Analysis of a Digital Twin using Logging and Model Learning

Raghavendran Gunasekaran[*], **Boudewijn R. Haverkort**[†], **and Loes Kruger**[‡]
[*]Tilburg University, The Netherlands
[†]University of Twente, The Netherlands
[‡]Radboud University, The Netherlands

ABSTRACT Over the last few years, digital twins (DTs) have attracted increasing attention and uptake in both industry and academia. While several definitions exist for a DT, most of these focus on an exact virtual replica (often called the virtual entity (VE)) of a real-world object or process, which typically consists of several executable models interacting with each other. Furthermore, due to the connection and synchronization with their real-world physical counterpart, DTs evolve continuously across their lifecycle. Often, however, details of construction and internal structure of DTs are left un- or underspecified. Over time, both these factors (un(der)specification and real-time changes due to synchronization) might lead to misuse, undesirable behavior, or runtime issues, like errors, and performance problems. This hinders the (re)use of DTs and/or its components for the intended purpose or any other future purposes. In this paper, we propose a new approach that helps to overcome the above sketched issues. We do so, in a case-driven way, by addressing a DT of an autonomously driving truck, developed by several researchers over a longer period of time, and with input of several MSc and PhD students. As it turns out, this DT lacks overall complete documentation. We demonstrate how logging can be used to learn the actual runtime behavior of a DT and show how this behavior can differ from its intended behavior at design stage. We explore different passive model learning techniques, such as state merging and process mining, to automate the process of obtaining behavioral models of the DT. In addition, we showcase how the learned behavioral model of the DT can be analyzed further to detect underlying causes of perceived runtime issues in DTs.

KEYWORDS Digital Twin, Logging, Model Learning, Passive Learning, Reverse Engineering, Process Mining

1. Introduction

Recent advances in digital technology, such as cloud computing, big data, Internet of Things, combined with the oncoming of artificial intelligence have greatly impacted different sectors in industry. One such technology that has garnered widespread attention in recent times in both industry and academia are digital twins (DTs). DTs have been used for a wide range of applications, ranging from control, monitoring, predictive maintenance, optimization, to training, validation and others. Since DT is a new concept, there are a range of definitions provided to it, such as a consistent virtual representation of the physical counterpart (Zhuang et al. 2018), models of a physical entity that are interconnected to effect data exchange (Singh et al. 2021), integration of models representing the behavior of a real-world entity (Shafto et al. 2010), a digital replica of a physical entity (El Saddik 2018) to name just a few. Most of these definitions focus on having a high-fidelity virtual representation of the real-world entity (which is of interest) and a continuous synchronization of data between them. The DT undergoes continuous changes throughout its lifecycle due to this continuous synchronization with its real-world counterpart. This continuous evolution requires a technique that enables continuous verification of the DT.

JOT reference format:
Raghavendran Gunasekaran, Boudewijn R. Haverkort, and Loes Kruger. *Behavioral Analysis of a Digital Twin using Logging and Model Learning*. Journal of Object Technology. Vol. 24, No. 2, 2025.
http://dx.doi.org/10.5381/jot.2025.24.2.a7

Alongside its need for continuous verification, the lack of documentation for DTs is a growing concern. Lack of documentation affects the use of the DT for its intended purpose or reusing it for other purposes and leads to additional effort (Reim et al. 2023). Moreover, a lack of documentation can result in a lack of interoperability of components within a DT. Given that DTs are evolving systems, the absence of proper documentation introduces additional complexity in determining whether the observed behavior is an intended design decision, a consequence of continuous evolution, or other factors. Furthermore, it leads to reduced knowledge and a lack of clarity in understanding, which in turn leads to resource wastage. Moreover, considering the fact that DT development is a multi-domain endeavor, sufficient and effective documentation could help mitigate the loss of knowledge due to personnel changes within an organization (Muctadir et al. 2024). On the other hand, it is imperative to understand the structure and behavior of a DT since it provides possibilities for improvements, modifications, and updating of the DT and its components for other purposes. Moreover, knowledge of the structure and behavior of a DT provides possibilities for reusing a DT in its entirety or partially, for other applications. For observed issues in a DT, it provides possibilities to detect inconsistencies within its components or in the interactions between them. Moreover, at runtime, the interactions between the different components in a DT could be stochastic and time-critical which may lead to undesirable behavior, which needs to be detected. Logging that captures the dynamic aspects of a system can help with detecting the structure and runtime behavior of a DT when documentation and knowledge are insufficient. Moreover, logging that captures up-to-date measurements of runtime interactions enables continuous verification of DTs. As far as we know, there is currently no literature discussing how to deal with DTs that do not have proper documentation, nor literature that specifically addresses how to determine structure and runtime behavior (including runtime issues) for a DT that lacks documentation. In this paper, we discuss an exemplary DT (of an autonomously driving truck) for which we lack knowledge and documentation. We propose using logging and passive model learning techniques to determine run-time behavior and structure and understand the perceived run-time issues for this DT.

Contribution:

In our previous work (Gunasekaran & Haverkort 2024), we discussed how we used logging to detect runtime behavior for the same DT case study, for manual verification through statistical model checking. In addition to detecting the structure of the DT, we also cover these parts of detecting runtime behavior and uncover runtime behavioral patterns here, since it is essential to understand the importance of logging as a whole. Furthermore, we highlight the necessity of continuous verification of DTs as they evolve throughout their lifecycle. We explore and compare several passive model learning techniques to automate the process of obtaining behavioral models from the logs obtained from DTs; this comprises a first attempt towards (semi-) automatic verification of DTs. We use the behavioral model obtained through passive model learning to perform further analysis to detect cyclic dependencies (which may lead to runtime inconsistencies) and to perform a root-cause analysis to detect underlying reasons for such runtime issues.

Structure: The paper is structured as follows. Section 2 provides background concerning DTs, logging and passive learning, and discusses existing work using these techniques in the DT context. Section 3 describes the DT of an autonomously driving truck in a distribution center, which is used for our case study. Section 4 covers the detection of structure and runtime behavior of the DT through logging, elucidates the role of logging in continuous verification of DTs, and compares the observed and intended behavior of the DT. Section 5 explores different passive model learning techniques for learning the behavioral model from the logs generated from the DT and compares the results. Section 6 discusses the use of the learned models for root-cause analysis and for detecting cyclic dependencies in a DT. Section 7 describes several threats related to the performed case study. Section 8 discusses the lessons learned concerning logging and passive model learning and the challenges in applying these techniques in the DT context. Section 9 summarizes the results presented and the planned future work.

2. Background

In this section, we provide additional details on DTs and the problems associated with DTs that lack documentation. Next, we explain how reverse engineering can help uncover the runtime behavior of a DT that lacks documentation. Finally, we introduce the tools we use to reverse engineer a DT: logging and passive model learning.

2.1. Digital Twins

Since the concept of DT was introduced in 2003, DT has been provided with several definitions as mentioned in Section 1. Though several definitions exist for a DT, most of these definitions focus on the existence of a virtual entity that comprises several inter-connected models that interact with each other during execution to affect the purpose of the DT. These definitions also focus on the aspect of bi-directional synchronization of data between the real-world entity and its virtual entity, which contributes towards the continuous evolution of DTs. Hence, in this paper, we focus only on these two aspects of DTs. Moreover, DTs could exist for any real-world entity such as a physical or non-physical system, device, or process and at times it could be a non-existent real-world entity at the design stage (Grieves 2014; Tao & Zhang 2017; Muctadir et al. 2024). Some descriptions of DTs focus on DT being a high-fidelity virtual representation, while others describe that the level of fidelity depends on the DT's purpose and application (Muctadir et al. 2024). Though DTs are perceived to be like any other software system (such as distributed systems and others) which comprise several interacting components, the continuous evolution of DTs across their entire lifecycle sets them apart from other software systems. Currently, there is neither a clearly established nor a widely accepted standard methodology for the development, maintenance and operation of DTs. This makes it imperative to properly document information on the methodology used for the development of a DT. However, details on the development,

purpose and knowledge of DTs are often left unspecified and under- or even undocumented. This lack of documentation is a widespread issue in the software domain in general (Aghajani et al. 2020; Manai 2019; Yang et al. 2021). Lack of documentation leads to the diminution of knowledge and eventually the disappearance of knowledge, which wastes resources (time and cost) (Manai 2019). Another disadvantage of lack of documentation is that it leads to an unclear understanding of the software system, which again leads to increased resource usage (Matulevičius et al. 2009). Even with DTs, lack of documentation is a growing concern as mentioned in several works (Muctadir et al. 2024; Reim et al. 2023; Stadtmann et al. 2023; Guinea-Cabrera & Holgado-Terriza 2024). Lack of documentation in DT affects the use of a DT for its actual purpose and thus, leads to additional effort expending time and energy (Reim et al. 2023). Moreover, lack of documentation in DTs can drastically affect the possibilities of reuse of software artifacts and could lead to a perceived lack of interoperability of software components in a DT (Muctadir et al. 2024). Effectively, lack of documentation in DT affects both its use for the actual purpose and its reuse for other future purposes. To the best of our knowledge, there is currently no literature that discusses how to deal with DTs that lack documentation.

2.2. Reverse Engineering

Reverse engineering has been defined by (Rugaber & Stirewalt 2004; García-Borgoñón et al. 2023) as the process of comprehending software systems and producing a model of it at a higher abstraction level, suitable for comprehension, documentation, maintenance, or reengineering. The application of techniques and principles of model-driven engineering (MDE) for reverse engineering is called Model-Driven Reverse Engineering (MDRE) (Favre 2005; García-Borgoñón et al. 2023; Rugaber & Stirewalt 2004). The general process of MDRE comprises two primary steps and they are: 1) Obtaining a view or a model of the system through analysis of source artifacts, and; 2) Exploiting the obtained model for a specific goal or objective, such as model-based verification, reengineering, documentation and others (García-Borgoñón et al. 2023; Raibulet et al. 2017). One of the five main characteristics of MDRE approaches described by Bruneliere et al. (Bruneliere et al. 2014) is the level of automation that can be applied to completely or partially automate the reverse engineering process. Driven by this automation-oriented MDRE approach, we explored existing techniques for (semi-)automating reverse engineering of software systems.

Model learning techniques (discussed in detail in Section 2.4) such as active or passive learning techniques have been used in several industrial cases for (semi-)automating reverse engineering. For example, active learning (Aslam et al. 2020) and process mining (Leemans 2018) have been used to reverse engineer legacy software systems in ASML. Process mining has also been used for automating the discovery of behavioral models from software execution data (Liu 2018). Though logging and passive model learning have been used in the context of software systems, to the best of our knowledge, we are not aware of any literature that discusses using these techniques for automating reverse engineering of DTs.

Considering the above, this paper presents a first attempt to automate reverse engineering of DTs in this paper. We use the DT of an autonomously driving truck in a distribution center that lacks documentation as our main case study. To understand the behavior and functioning of such a DT, it is imperative to study the system when it is "on the run". We apply logging and passive model learning for reverse engineering the structure and runtime behavior of the DT; (semi-)automate the process of obtaining behavioral models which can be used for model-based verification and further behavioral analysis. We define the runtime behavior of a DT as the emergent behavior arising from the compositional effect of interactions between the different components constituting the DT (Gunasekaran & Haverkort 2024). Though the behavior within an individual component in a DT could also be construed as runtime behavior, as long as this behavior within a component does not contribute to the overall observable emergent behavior of the DT, then we do not focus on this aspect of runtime behavior.

2.3. Logging

Logging is a technique of recording events and activities during the execution of a (software) system, which helps in understanding the runtime behavior of the system. Logging has been used for pattern extraction, behavior analysis, root-cause analysis, diagnosis, anomaly detection, and verification and improvement of software systems (Yang et al. 2021; Batoun et al. 2024). Moreover, in the context of DTs, several works, such as Bano et al. (Bano et al. 2022), Lugaresi et al. (Lugaresi & Matta 2021) and Park et al. (Park & Van Der Aalst 2021), propose to use event logs of the real-world system or process to generate a DT. However, to the best of our knowledge, no work discusses logging in existing DTs and the use of derived logs for further behavioral analysis of the DT. In Section 4, we discuss how we used logging to uncover the runtime behavior of DTs and its importance in facilitating continuous verification of DTs as they keep evolving across their entire lifecycle.

2.4. Passive Model Learning

Model learning infers formal models of systems and can be categorized into two main types: active and passive learning. In active learning, the model is learned by interacting with the system using a query-response mechanism (Steffen et al. 2011; Vaandrager 2017). On the other hand, passive learning processes and analyses the execution logs to learn the behavior of the system (Aslam 2021). There are several passive learning techniques including state merging algorithms, SAT-based approaches (Hammerschmidt 2017), process mining, etc.

Initially, passive learning was used to learn a language from a set of positive and negative examples (Gold 1967). In recent years, this technique has been used to infer state machines representing real-world systems from execution logs (Yang et al. 2019; Biermann & Feldman 1972; Van der Werf et al. 2008; Walkinshaw et al. 2016; Daniele et al. 2024). Several tools support passive learning algorithms such as AALpy (Muškardin et al. 2022), FlexFringe (Verwer & Hammerschmidt 2017), Jajapy (Reynouard et al. 2023), ProM (H. Verbeek et al. 2010),

Disco (Günther & Rozinat 2012) and others.

State merging algorithms build a prefix tree from the execution logs and iteratively merge states in the tree until no additional consistent merges are available. The merging procedure folds the logs into a more succinct representation of the behavior of the system. State merging algorithms vary in their definition of a 'consistent merge'. For example, the RPNI algorithm (Oncina et al. 1992) can merge almost every pair of states if the logs are sparse. Later algorithms, like EDSM (Lang 1998), prioritize merging states with high evidence. Initially, state merging algorithms were used to learn Deterministic Finite Automata (DFA) (Gold 1967). Recently, the scope of learnable models has broadened. For instance, Mealy machines, Markov chains and labeled Markov decision processes can now be learned (Carrasco & Oncina 1994; Mao et al. 2012). The automata learning library AALpy (Muškardin et al. 2022) implements several state merging algorithms to learn various types of state machines from logs.

Process mining is another passive learning technique that extracts knowledge based on the observed actual behavior of a system or process, which is recorded in event log data. Event logs form the starting point for process mining. There are three important techniques of process mining which include process discovery, conformance checking, and enhancement. *Process discovery* is a mining technique of automatically constructing models based on observed event log data. The constructed model helps in understanding the behavior of the system/process which is captured in the raw event logs. Moreover, this model could also be useful for conducting further analyses such as timing analysis, resource performance analysis and others.

Processes discovered through process mining can be represented in a variety of process models, e.g., Petri nets, Statecharts, UML activity diagrams, YAWL, EPCs, cf. (W. Van Der Aalst 2012; W. M. Van Der Aalst & Dustdar 2012; Leemans 2018). In Section 5, we discuss the usage of passive learning for learning behavioral models from DT execution logs.

3. A DT Case Study

We use a DT of an autonomously driving truck in a distribution center (see Figures 1 and 2) as our driving case study. This DT was developed at TU Eindhoven[1] over the past several years, by a large group of PhD and MSc students, to test the autonomous driving and docking in a distribution center with several obstacles.

This DT is a highly modified version of the DT developed by Barosan et al. (Barosan et al. 2020). As mentioned before, there is no complete or structured documentation on this DT, hence, only limited information is available for our analysis. Below, the initially known details about this DT are discussed. The following sections cover additional information uncovered about this DT and the methods used for discovery.

The complete description of this DT case study after discovery including its structure and behavior can be found in our previous work (Gunasekaran & Haverkort 2024). The DT of

[1] See https://www.tue.nl/en/research/research-labs/the-automotive-technology-at-lab/facilities/conference-table.

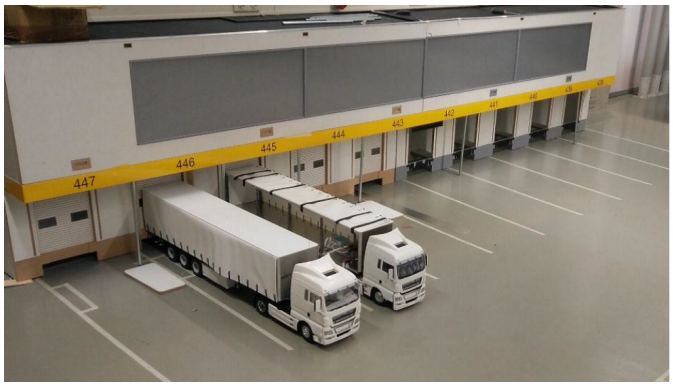

Figure 1 Autonomously driving truck in a truck distribution center - Real-world Entity (Barosan et al. 2020)

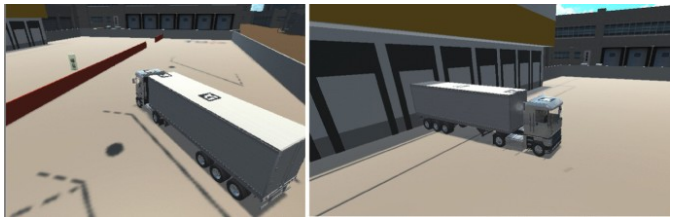

Figure 2 Digital twin of an autonomously driving truck in a truck distribution center - Virtual Entity (Barosan et al. 2020)

the autonomously driving truck has three main components as described:

- **Controller model**: The controller model in Simulink controls the motion of the truck in the simulation environment. This is achieved through a control feedback loop with tuned proportional gains to ensure optimal controller performance.
- **Path planner model**: To navigate the truck from point A to point B, a path must be planned from the starting position to the desired destination dock, while avoiding obstacles. The path planner model in Simulink determines the route the truck needs to follow.
- **3D simulation model**: The 3D simulation model in Unity Game Engine is a virtual simulation environment of the autonomously driving truck in a distribution center filled with static obstacles such as a pedestrian, walls and others.
- **Python Server**: A Python server was used for the communication between components in the DT. It is the only component through which the user interacts with the DT. The user provides two inputs for the DT execution, namely, the Loading Dock (destination information) and Kill Switch (a boolean value) which controls the initiation of autonomous driving.

All communication within the DT happens through the connectionless User Datagram Protocol (UDP). While the communication protocol is known, there was no information on how these components were interconnected with each other (which component interacts with which other component in the DT or which components do not interact with each other) and how they interact with each other to achieve a coordinated overall

system simulation (whether it is with the use of an orchestrator or whether it is following any schedule or others).

4. Applying Logging in DTs

In this section, we first determine the internal structure of the DT and describe how we generate the event logs. Then, we describe how we determined behavioral patterns and compare the observed behavior with the intended behavior. Finally, we elucidate how logging enables continuous verification of DTs as they evolve across their entire lifecycle.

4.1. Determining the DT Structure

With limited information on the DT at hand, the first step was to determine the structure of the DT, i.e., which components are present and how are they connected. As mentioned in Section 3, UDP was the protocol used for the interaction between the components within the DT. While investigating the UDP communication in the Simulink models, we found that masking function blocks were used for UDP communication, using C programs. Similarly, the Unity model was developed in C# and among the 12,541 project files in Unity, we found a couple of C# programs responsible for handling the UDP communication in the Unity model. From matching the UDP port numbers, and the incoming and outgoing variables in the seven program files (C, C# and Python programs) in these models, we could establish the internal structure of the DT. Figure 3 shows the discovered internal structure of the DT, the interactions between the components and the data variables exchanged between these components. The interactions within the DT have been depicted through letters [*A*-*H*] listed in Figure 3, we use these letters to refer to the interactions in the remainder of the paper.

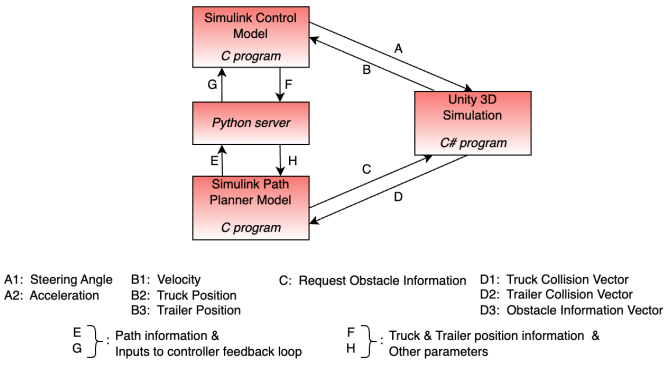

Figure 3 Internal structure of the DT (Gunasekaran & Haverkort 2024)

4.2. Generating Event Logs

As discussed in Section 2.3, event logs help in behavior analysis and pattern extraction, aiding in the comprehension of DT runtime behavior. To be consistent with our definition of runtime behavior as described in Section 2.2, we log the interactions between the components in our DT.

For generating logs in software, Leemans (Leemans 2018) discusses four techniques: external monitoring, tracing, log files provided by existing systems, and instrumentation (such as software profilers). While exploring the modeling tools that are part of this DT, we found that Simulink provides options for profiling. However, these profiling options focus more on the timing aspects of the individual components (such as Simulink blocks) rather than the inter-model communication. Since the UDP interactions between the three models have been configured through programs in C, C# and Python, this provided a feasible option to log these interactions by instrumenting these programs with logging commands. The logged interactions were directly written into a text file. The specific attributes that were logged include the direction of communication (whether it is an input or output for that specific tool), timestamp of that event (interaction) in microsecond precision and variables exchanged during the interaction. To prevent the generation of multiple logs for the same interaction in different locations, logs were meticulously planned to be generated only at two components that do not directly interact with each other, the Unity and Python components. The log files were exported into spreadsheet calculation software, where they were merged based on the timestamps (Gunasekaran & Haverkort 2024) and finally converted into CSV format with UTF-8 encoding.

4.3. Behavioral Pattern Detection

The expected behavior in the execution of a DT, would be a non-erratic and seamless interaction between the components in the DT. This coordinated interaction could be based on a schedule (time) or event or a combination of both. However, when analyzing the pattern of occurrence of interactions from the event logs, we detected a non-fixed, repetitive and atypical behavior in the interactions between Unity and Simulink. This interaction behavior was in the form of an on/off pattern and occurred only in the interactions between the two Simulink models and the Unity model. During the on-state, the Unity model communicates with the Simulink controller model for a certain period of time. During this period of communication, there is absolutely no interaction happening between the Simulink path planner model and Unity model. This behavior continues for a non-fixed period of time or a varying number of iterations. After a while, there is a switch in the interaction and the DT goes into the off-state and the Unity model starts communicating with the Simulink path planner model. Again, during this period of communication, there is no interaction occurring between the Simulink controller model and Unity model. After a non-fixed period of time, it goes into the on-state again. This unusual and unexpected pattern of alternation in communication within the DT between the on- and off-state occurs repeatedly during the entire period of the DT execution.

4.4. Comparing Observed and Intended Behavior

The intended behavior of DT at design stage is to have a seamless non-erratic timely interaction between the different models involved in the cosimulation. This design intention is to ensure that there is a smooth, continuous, collision-free movement of the truck in the Unity simulation environment. However, the observed behavior of the DT significantly deviates from this intended behavior. The detected on-off interaction pattern is clearly not an intended behavior, as the internal structure of the DT, illustrated in Figure 3, indicates that communication from

the Simulink controller model to the Unity model is critical. This is because steering angle and acceleration variables, which are communicated from the Simulink controller model to the Unity model are responsible for driving the motion of the truck in the simulation environment in Unity. However, due to the described on-off pattern, this critical communication may not occur for a non-fixed (variable) period of time, which possibly affects the motion of the truck in the simulation environment.

We performed multiple co-simulation runs to observe the motion of the truck in the Unity model. Surprisingly, we observed that in almost all of the runs, the truck collides with an obstacle, although at different points in time in relation to the starting point of the co-simulation. This kind of behavioral pattern is never desirable in a DT. Again, this observed behavior is clearly not the intended behavior of the DT and the DT would have been devoid of this behavior at design stage. The reason for the change in the observed behavior from the intended behavior at the design stage is unclear. However, we speculate that this could be a consequence of the continuous evolution of the DT across its lifecycle, where the DT could have been updated or modified somewhere along its lifecycle. We discuss this evolution of DTs in the following Section 4.5.

We understand that this observed runtime behavioral pattern in the DT cannot be ascertained from only its individual components. Rather, as discussed by Van den Brand et al. (Van Den Brand et al. 2021) and by Gunasekaran & Haverkort (Gunasekaran & Haverkort 2024), this is the emergent behavior arising from the compositional effect of interactions occurring between the components within a DT during runtime. In the following section, we describe verifying the runtime behavior of this DT to check for adherence to deadlock freeness, timeliness and functional correctness properties.

4.5. Continuous Verification of DTs Enabled by Logging

As described in Section 1, DTs are systems that undergo continuous synchronization with their physical counterpart across their entire lifecycle. This bi-directional synchronization with the real-world entity coerces the DT to evolve continuously to remain its faithful twin. Furthermore, the DT's evolution could be influenced by bug fixes, improvements, modifications and other changes made to both the entities that stay as faithful twins. Both Zhang et al. (Zhang et al. 2021) and Mertens et al. (Mertens et al. 2024) discuss three types of evolution that DTs undergo across their entire lifecycle. The continuous evolution of DTs indicates that verification and validation (V&V) only at the time of development is insufficient; they must be verified continuously throughout their entire lifecycle. Currently, no literature explicitly discusses techniques to enable the continuous verification of DTs.

One of the advantages of logging is that it provides up-to-date measurements of interactions occurring in a system at runtime. This means that even when DTs evolve continuously, logging would provide up-to-date measurements of the runtime interactions. The logs generated from DTs at any point of time in their lifecycle can thus be used to understand and model the runtime behavior of DTs. Behavioral models created through generated logs can be verified using techniques such as (statistical) model checking. Effectively, in this way, the DT can be verified at any point in their lifecycle to check whether they behave consistently at runtime. Thus, logging enables the possibility for continuous verification of DTs. In our recently concluded research, we used logging to generate event logs in this DT at runtime. These logs were then used to model its behavior as a network of stochastic timed automata (STA). The network of STAs was then verified using statistical model checking to check for timeliness and functional correctness properties (Gunasekaran & Haverkort 2024). The results indicate that this DT adheres only to properties of functional correctness and not to timeliness properties. However, since properties related to deadlock freeness and liveness cannot be verified using SMC and can only be verified using classical model checking, we also modeled the runtime behavior captured in the generated logs as a network of timed automata (TA). This network of TA was verified using classical model checking to check for deadlock freeness and liveness properties (Gunasekaran & Haverkort 2025). We found that there is no deadlock in this DT at runtime. Thus, the role of logging to enable continuous verification of DTs across their entire lifecycle is imperative.

The above two works on continuous verification of DTs require manual effort for modeling the behavior of DTs from the obtained logs at runtime. Moreover, manually modeling the behavior of DTs obtained from event logs may require some level of expertise and knowledge, which may affect the adoptability of these continuous verification techniques. To avoid this manual effort for modeling, we explore the possibility of applying model learning techniques which may help in (semi-)automating the process of obtaining behavioral models from generated logs. The following section explores various passive learning techniques for obtaining behavioral models from the generated event logs.

5. Exploring Passive Model Learning

Learning behavioral models of verifiable systems is a costly affair with several applicable techniques. Considering that logs have already been generated for the DT, the same set of logs was used to generate behavioral models using passive model learning. We explored two main classes of passive learning: state merging and process mining. We discuss these techniques, the tools used, the obtained results, and our observations from applying these techniques to the generated logs.

5.1. Exploring State Merging

State merging techniques can be used to infer several types of state machines and have frequently been used to infer software models (Walkinshaw et al. 2013). The nature of the interactions in the DT determines the ideal type of state machine. For example, if there is no randomness or variability in the system, a Deterministic Finite Automaton (DFA) can be used to model the system. Initially, the RPNI implementation of AALpy was used to infer a DFA from the DT logs. However, this led to unsatisfactory results: either a 1-state DFA where all possible sequences of interactions are accepted or an uninterpretable DFA with over 100 states that accepts exactly the logs and rejects all other sequences of interactions. In the DT scenario, 'accepted'

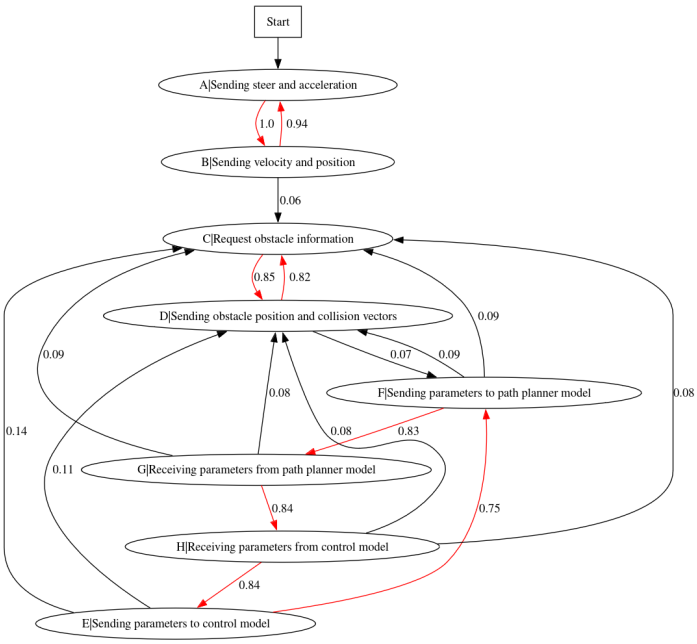

Figure 4 The Markov Chain obtained using Alergia. Transitions with probabilities under 0.05 are excluded for readability. The arrow entering the state labeled *A* indicates that *A* is the first interaction in the logs. Red transitions indicate the transition with the highest probability from a state and are added for readability.

traces may be produced by the DT while 'rejected' traces cannot be produced by the DT. For example, in the considered DT, a trace never contains two consequent *A* interactions because the Simulink Control Model will wait for a *B* interaction before sending the next *A*. Thus, all traces that contain *A A* cannot be produced by the DT and should be rejected by models inferred from the DT. Usually, both accepted and rejected traces are needed to have enough information to distinguish behavioral states and infer a sensible automaton. However, some techniques are designed for situations where only accepted traces are available (Avellaneda & Petrenko 2019). Initially, we believed the unsatisfactory results were caused by the absence of rejected traces. However, a closer examination of the logs indicates that the stochastic nature of the DT's interactions was the root of the unsatisfactory results. Section 4.3 describes how Unity communicates with the Simulink Control Model for some time and then switches to communication with the Simulink Path Planner Model. Since this switch does not occur after a fixed amount of steps but after an approximate number of milliseconds, the DT behavior is better explained by a stochastic state machine than a deterministic one. Due to these findings, we set out to learn a Markov Chain (MC) instead of a DFA. MCs can be used to model finite stochastic state machines where the states represent interactions that occur in the system. MCs can be learned using the Alergia algorithm (Carrasco & Oncina 1994) implemented in AALpy (Muškardin et al. 2022). Alergia can be viewed as the probabilistic version of RPNI and mainly differs in the use of a probabilistic prefix tree and statistical tests to determine whether two states in the prefix tree are consistent (Verwer & Hammerschmidt 2017).

However, for this DT, the logs cannot be used to directly infer an MC using Alergia because the logs are often too long which leads to hitting the maximal recursion depth due to the nature of the state-merging algorithm. Pre-processing the logs to have at most (around) 2000 interactions makes it possible to run Alergia. However, some logs start with over 1000 repetitions of *C D* (as shown in Figure 3) meaning that the shortened logs do not include all the interactions. Therefore, we pre-process the logs by removing highly repeating cycles and then shortening the logs to 2000 interactions. After pre-processing, the Alergia algorithm can be executed on the logs and produces an MC. Additionally, a post-processing step is needed to recalculate the probabilities for the inferred model on the original logs. The resulting MC is displayed in Figure 4. To create these pre- and postprocessing steps and obtain a state machine, visual inspection of the logs and expert knowledge of the DT were needed. Therefore, these processing steps are highly specific to this DT, and applying state merging techniques to other DTs might require different processing steps to make the logs suitable to infer a state machine. We briefly looked at Jajapy (Reynouard et al. 2023) with the hopes that this has less problems with the long logs and highly repeating cycles. Jajapy is a Python library that can infer MCs but does not use state merging but the statistical Baum-Welch algorithm. However, this approach ran into similar problems and the resulting models after post-processing missed interactions between several components while the interactions were present in the logs.

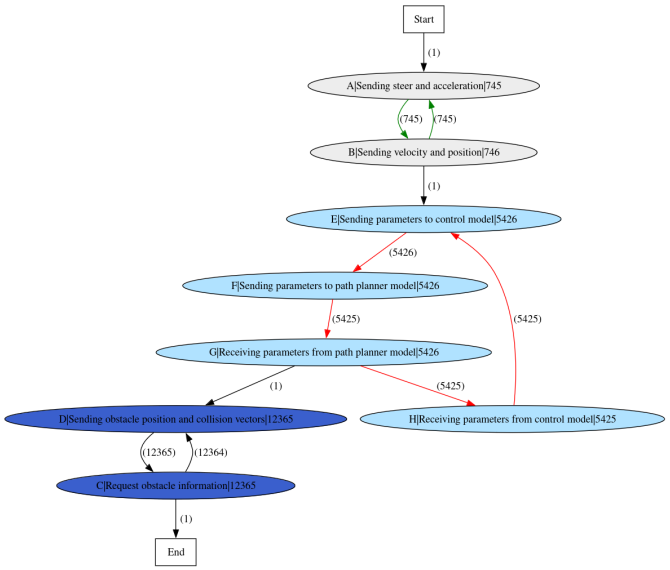

Figure 5 Statechart obtained from process mining using the statechart workbench. Equivalent to the statechart for case 2, depicting cyclic dependencies in a DT with four components.

5.2. Exploring Process Mining

We use the ProM (H. Verbeek et al. 2010) as our process mining tool of choice. One of the reasons for selecting this tool is its capability to customize existing plugins. The CSV file containing the event logs is imported into ProM, using a plugin to convert CSV files into XES[2] format (Gunther & Verbeek 2014). One of

[2] The XES format has been adopted by the IEEE Task Force on Process Mining as the standard to store and exchange event logs (Leemans 2018).

the challenges with process mining is to identify the mapping of event logs to so-called cases[3]. To analyze the complete runtime behavior of the DT, all event logs from a single execution were mapped to a single case, which is generally unusual in process mining due to the high number of logs.

ProM is a rich tool comprising a plethora of plugins for process discovery. We experimented with several process discovery plugins from ProM and their results were similar to those from RPNI, which were having only a single state. Due to this reason, we omit these results in this paper. One of the plugins that we had used was the statechart workbench (Leemans et al. 2018) for discovering a statechart of the interactions from the provided logs. While most plugins in ProM provide support for logs from business processes, this workbench is one of the plugins that can work with both software and business event logs. This was one of the main reasons for selecting this plugin. Since this plugin was developed for handling software logs, it accommodates hierarchical, recursive, error-handling or cancellation behavior in software. The hierarchical discovery is based on the traditional inductive miner framework's (IMF) divide and conquer approach where given a log L, it is divided into sub-logs which when combined together with a process tree operator can possibly reproduce the same log L again. The cancellation discovery is an extension of IMF which works with a cancellation trigger oracle. For complete details on the algorithms used for hierarchical, cancellation and recursive discovery, we refer the readers to this literature (Leemans 2018). Considering the feature-rich discovery algorithms in this plugin, we chose to use it to learn behavioral models from the generated logs in DTs, rather than reinventing the wheel. However, as this plugin is tailored for handling software event logs, its applicability for logs from DTs was experimentally evaluated. The results from the statechart workbench are shown in Figure 5.

5.3. Comparing State Merging and Process Mining

When comparing the models obtained by state merging (Figure 4) and process mining (Figure 5), we notice that both models are high-level overviews of the interactions within the DT. Every state in the models contains the name and direction of the interaction (sending or receiving parameters) which is separated by 'l' symbol from the letters [A-H] denoting the interactions shown in Figure 3. Both models have a state for every interaction logged in the DT and the transitions are related to the frequency of the observed interactions. Furthermore, the order of interactions as observed from the logs and recursive behavior has been captured to some extent in both models. We observe that each of these models contains the cycles *A B*, *F G H E* and *C D*. However, we also observe that the interactions *A*, *B*, *C* and *D* which are part of the on-off behavioral pattern (as described in Section 4.3) are not completely interconnected in both the models. The models share many similarities, but there are also some differences. For example, the techniques capture the frequency of interactions differently. The process mining model depicts the load of interactions occurring in the DT by specifying the number of each interaction. While the state merging model captures the frequency in a state-dependent way, the transition probabilities represent the chance of seeing the interaction in the destination state label as the next interaction. For example, the transition from state *B* to *A* indicates that 94% of the time, interaction *B* is followed by interaction *A*. Additionally, the process mining tool automatically makes a readable figure by excluding outliers and adding colors based on how often an interaction occurs in the logs (more information on the state colors can be found in Section 6.2), while transitions with low probability were manually removed in the state merging result to make it readable. We conclude that both models are informative and can be used for visual analysis. However, there is a significant difference in the ease of use between the techniques. To obtain the state merging model from Alergia, pre- and post-processing steps specific to the DT needed to be added. Therefore, using state merging techniques may require a considerable amount of manual effort. This issue becomes particularly problematic if we want to automatically infer a model of the DT whenever changes are detected, as the necessary pre- and post-processing steps may need to be adjusted accordingly. Moreover, these processing steps require some level of knowledge about the specific DT. Considering the motivation of this paper is to evaluate passive learning techniques that can aid in the behavioral analysis of DTs in an automated manner, our conclusion is that process mining edges over state merging techniques as it is more easily applied in scenarios where little is known about the DT. Therefore, we use the model obtained by process mining for the analyses in the following sections.

6. Analyses of Learned Models

The experimentation of applying process mining on logs generated from DTs was primarily focused on obtained behavioral models which are verifiable through model-based verification techniques and for further behavioral analysis. The obtained models are not sufficiently mature for verification through model checking, as they do not encompass the complete runtime behavior of the DTs, such as the on-off interaction pattern discovered through logging. Furthermore, these models do not capture the timing information of the runtime interactions which makes it not suitable for verification of temporal properties which are critical. However, further behavioral analysis of obtained process models could possibly help with several applications. In the sections that follow, we describe how we used process mining to detect cyclic dependencies (Section 6.1) and for root cause analysis (Section 6.2) in the DT.

6.1. Detecting Cyclic Dependencies

Van den Brand et al. (Van Den Brand et al. 2021) discuss avoiding undesirable behavior in DTs at runtime and specify deadlock freeness as one of the properties to be ensured at runtime for this. Muctadir et al. (Muctadir et al. 2024) also found deadlock freeness as an important property to ensure consistent behavior at runtime, from an interview research conducted with 19 interviewees in the field of DTs from both industry and academia. Several papers explicitly indicate that cyclic dependencies may lead to a deadlock or live-lock situation, which is undesirable (F. Verbeek & Schmaltz 2012; Sánchez et al. 2006;

[3] A case in process mining refers to a process instance that comprises several activities or event logs

Duato 1995). Cyclic dependencies are highly likely in a DT, as during execution, multiple models interacting with each other may require data from another model to progress to the next step. Likewise, the latter model may also be dependent on the former model for the progress of its next step, which eventually leads to a lock situation. Consequently, to prevent deadlocks in DTs, it is crucial to detect cyclic dependencies during runtime interactions. The recursive aware discovery algorithm in the statechart workbench has been developed to detect such recursive behavior which includes circular/cyclic dependencies. The recursive aware discovery algorithm checks whether an event's activity label refers back to an existing named sub-tree during the divide and conquer approach discussed in Section 5.2. In that case, it detects that the event at the current level of the hierarchy represents a recursive reference. Moreover, this algorithm ticks the boxes of guarantees for process mining algorithms such as soundness, termination, fitness and polynomial runtime complexity and is scalable too (in terms of the number of execution logs that can be handled) (Leemans 2018). To check the effectiveness of this workbench in detecting cyclic dependencies and specifically, detecting changes to an already existing cyclic dependency, we took two separate cases which are discussed in the following two subsections.

6.1.1. Case 1: DT Comprising Three Components

In this case, the existing DT (discussed in Section 3) was modified by removing the Simulink path planner model from the DT. This modified DT could be used in the case where there are no obstacles in the path and the path to each of the destination docks is already specified. The effect of this modification is that all interactions from and to the Simulink path planner model also ceased to exist. We generated event logs for this modified DT and observed the obtained statechart with ProM, as shown in Figure 6. Comparing Figures 3 and 6, it can be observed that there are four interactions in this modified DT and the miner has extracted and represented all these four interactions as transitions. Moreover, it detects which interaction is dependent on the other and based on this, it shows the cyclic dependencies. As it can be observed, there are two cyclic dependencies in the modified DT, which corresponds to the interactions between the three components in the DT.

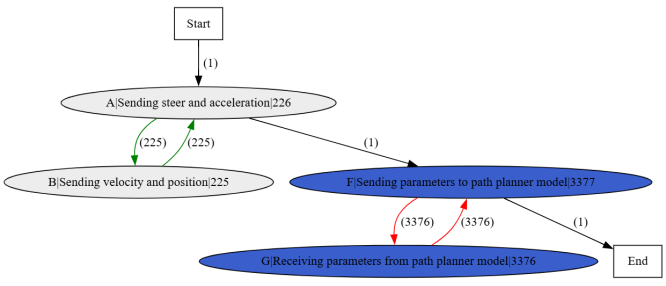

Figure 6 Statechart for Case 1, depicting cyclic dependencies in the DT with three components.

6.1.2. Case 2: DT Comprising Four Components

In the second case, we do not make any modifications to the DT as described in Section 3. We generated event logs for the DT to obtain the statechart as shown in Figure 5; we can observe the three cyclic dependencies in the DT corresponding to the interactions between the four components in the DT. The presence of the Simulink path planner model in case 2 alters the interactions within the DT compared to case 1. This change in interactions and the resulting alteration in cyclic dependency have been detected by the statechart workbench. In Figure 6, a cyclic dependency concerning interactions *F* and *G* can be observed (shown with red transitions). However, with the addition of Simulink path planner model in case 2, as can be observed in Figure 5, the same cyclic dependency now has four interactions (*E, F, G* and *H*) contributing to it (shown with red transitions). From this, we can understand that the statechart workbench is capable of not only detecting cyclic dependencies in a DT but also it can be used to detect changes in a specific cyclic dependency in a DT.

6.2. Root Cause Analysis in a DT

We encountered a consistent runtime issue in the DT related to restarting the simulation. We understand that this problem has been reported by several others (former students and researchers) and has been an unresolved issue for a long time. During execution of the DT, there is an option in the Unity model to reset the simulation, which puts the truck back in its original position. This reset simulation option can be used at any time, to reset the simulation when the truck collides with an obstacle and is not able to move any further. However, whenever the simulation is reset in the Unity model, the truck starts showing anomalous behavior, where it starts driving circles around its original position (See Figure 7). This is a recurring issue that happens whenever the simulation is reset in the Unity model.

As mentioned in Section 6.1, we used process mining to obtain statecharts representing the interaction behavior within the DT. The statechart workbench in Figure 5 provides details on the number of occurrences of each interaction within the DT on the transitions. For several runs, we observed the statecharts and we found a peculiar imbalance in the number of interactions. This load imbalance is visualized in Figure 5, where the colors of the states represent the number of occurrences of interactions: dark blue represents a high number of occurrences, light blue a moderate number of occurrences, and the greyish color represents a very low number of occurrences. We found consistent light grey colors for the interactions between the Simulink controller model and the Unity model (the number of occurrences of these interactions can be found in the green transitions). As discussed in Section 4.4, the interactions between the Simulink controller model and the Unity model are critical. Hence, it was remarkable to observe these critical interactions occurring significantly less often than the other interactions. After this observation, we started logging the action of resetting the simulation in Unity. We separated these newly generated logs into two sets: one set of logs comprising interactions that occurred before the simulation was reset, and another set of logs comprising interactions that occurred after the simulation was reset. The statechart obtained for the first set of interactions is shown in Figure 8; likewise, the statechart obtained for the second set of interactions is shown in Figure 9. When comparing these two statecharts, we can find a difference in the number

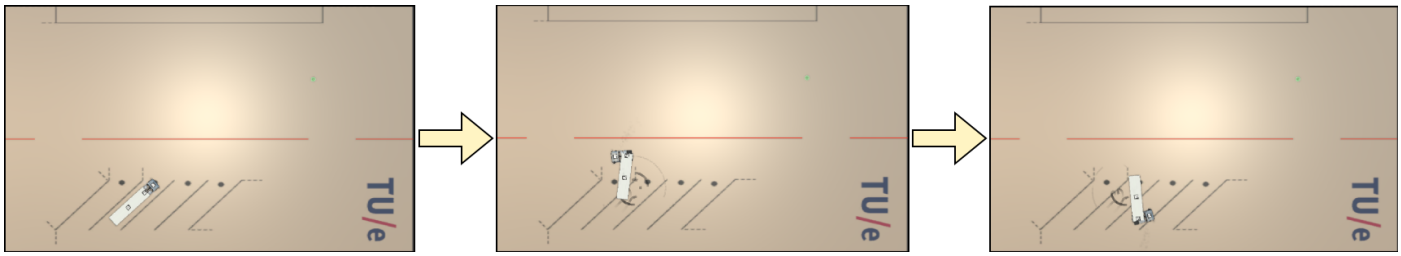

Figure 7 Aerial view of the anomalous behavior of truck where the truck is driving erratically in circles around its original position in Unity simulation environment, whenever the simulation is reset.

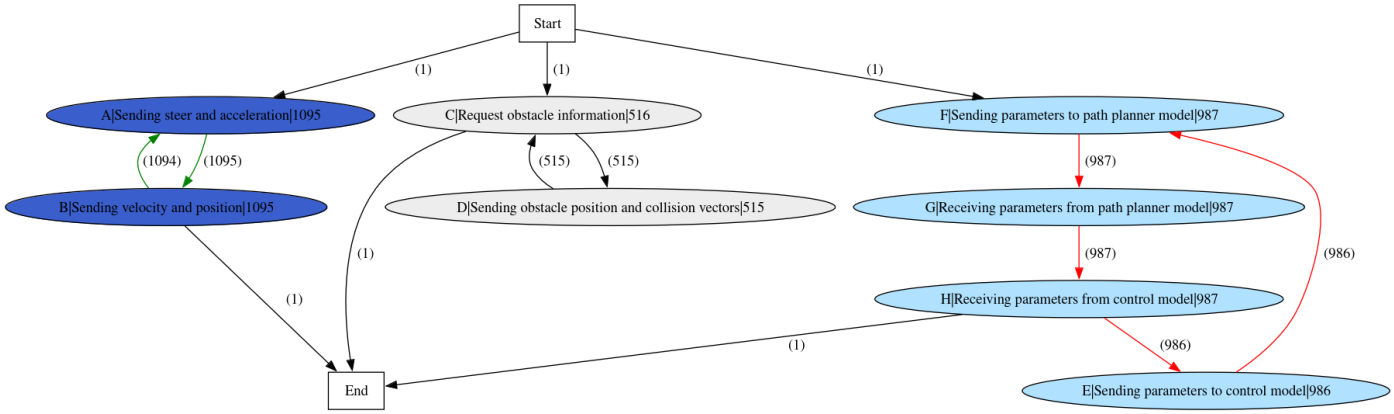

Figure 8 Statechart representing interactions within the DT *before* simulation reset.

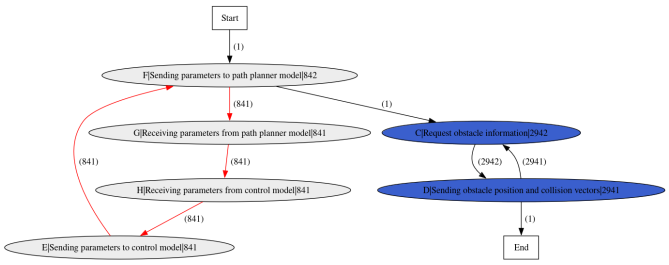

Figure 9 Statechart representing interactions within the DT *after* simulation reset.

of states in both the models, with Figure 8 depicting states for all interactions and Figure 9 depicting states for only six of the interactions in the DT. From this, it is clear that the interactions between the Simulink controller model and the Unity model (interactions *A* and *B*) cease to occur after the simulation is reset in Figure 9. From this observation, we conclude that the C# program for handling the UDP communication between the Simulink controller model and the Unity model does not execute anymore after the simulation is reset, somehow, it has been stopped. This is the root cause of the truck's abnormal behavior in the Unity simulation environment: after the reset function is executed, a crucial part of the communication in the DT is switched off. As a result, information about the steering angle and acceleration is no longer communicated, causing the truck to drive in circles endlessly. We would like to emphasize here again that, in this scenario as well, we were able to uncover the root cause of the runtime issue with the help of the logging and the subsequent process mining and visualization.

7. Threats to Validity

In this section, we discuss the threats to validity related to the application of logging and model learning performed on this digital twin case study and the measures to mitigate the threats.

Construct Validity This validity concerns the quality of the measurement of the constructs for the experiment and whether it may have unintended effects on the results. The logging that was used in the experiment could require a high level of comprehensiveness to ensure that all the interactions within the DT are logged. Moreover, logging can be invasive and introduce some level of overhead. This could lead to observing and consequently logging the DT interactions with an overhead (slightly modified behavior) than the actual behavior of the DT. Werner (Schutz 1991) describes this as an inevitable effect of observing the behavior of a system.

Choice of Learners. In this case study, we only explore state merging and process mining while other techniques such as SAT-based (Biermann & Feldman 1972; Verwer & Hammerschmidt 2017) approaches might also result in meaningful models. The used learners were chosen because they were used to infer state machines of software models in the past (Yang et al. 2019; Leemans 2018) and due to previous experience with the tools. We did briefly explore the statistical approach Jajapy (Reynouard et al. 2023) but this did not lead to promising results as stated in Section 5.1. **Internal Validity** To control the variables within our case study and ensure the techniques have access to the same information, we used the same CSV file containing the execution logs to infer the models obtained by state merging (Figure 4) and process mining (Figure 5).

External Validity This type of validity concerns the gener-

alization of our conclusion. The presented results in this case study, though encouraging, are from a single DT case study which could be a threat. In addition, the results from this case study only provide a single example of the root cause analysis of the runtime issue which occurs whenever the simulation is reset (not from the same simulation), which could also pose a threat. We plan to apply the learning techniques and analyses to a wider set of DTs from varied domains in the future such that more reliable and generalizable conclusions can be made. However, this paper describes preliminary research that can be used to facilitate further study of using logging and passive model learning of DTs.

8. Discussion

In this section, we discuss the lessons learned from applying logging and model learning techniques in a DT. Logging in DTs play a key role in enabling continuous verification, behavioral pattern discovery and detecting the evolution of DTs by comparing intended behavior with observed behavior. For this DT logging the interactions between the different models was possible because we could adjust the UDP protocol used for co-simulation. However, current practices of development of DTs do not provide the required level of importance for logging. Many existing modeling tools do not support the possibility of logging inter-model interactions in a DT. This is because DTs are often composed of interconnected cross-domain models in heterogeneous tools and considering that DT is being developed by experts from a wide range of domains who may lack knowledge and expertise in software, it may not be possible to always use/develop relevant software (such as additional software or servers) which could support logging. In such cases, the possibility of logging inter-model interactions by modeling and simulation tools would be a great addition. Currently, logging can support the continuous verification of DTs as they evolve throughout their entire lifecycle. However, one of the challenges with this application is that whenever a DT evolves, new components (models, databases, and others) may become part of the DT, leading to new interactions. This necessitates the manual logging of these new interactions within the DT.

Applying passive model learning techniques to this DT and the associated execution logs reveal that the obtained models do not capture the complete behavior of DTs. As described in Section 6, the learned models do not encompass the on-off interaction behavior pattern described in Section 4.3. A tailor-made algorithm could possibly detect such patterns in interactions, which could help in learning such patterns in the behavior of DTs. Moreover, the results indicate that state merging algorithms are more challenging to apply to DTs. Inferring the correct type of state machine is non-trivial as DTs are not necessarily expressed by Markov chains; several studies use Mealy machines to model real-world systems, see (Vaandrager 2017) for an overview. Additionally, the pre- and post-processing steps required to obtain Figure 4 are highly specific to this DT and setting up befitting pre- and post-processing steps require manual effort and expert knowledge which may affect its adoptability for other DTs but also for a continuously changing DT.

Even if a suitable model can be generated, there would still be several challenges that need to be overcome to use the model for V&V. For example, the model would need to be extended to include interaction timing information to fully verify the runtime behavior. This might be possible by computing mean values of time between interactions which are already stored in the logs. However, in our recent work for modeling STA for verification of runtime behavior of DTs, the obtained timing values were highly stochastic in nature with a wider range of values, and the arithmetic mean values do not necessarily express the actual observed stochastic temporal behavior (Gunasekaran & Haverkort 2024). Therefore, we speculate that to obtain such stochastic temporal behavior in behavioral models which can then be used for model-based verification, some level of human intervention may be required to assess this aspect of modeling and thus, a complete automation may not be feasible. Hence, one of the challenges with obtaining verifiable models through model learning is to capture the varying temporal behavior of interactions between components in the DT at runtime.

Though the (semi-)automatically derived models were not suitable for performing model-based verification, further behavioral analysis helped in detecting underlying causes of runtime issues and cyclic dependencies in the DT. We speculate that this behavioral analysis could also help with other applications with DT engineering, maintenance and operation. For example, behavioral analysis helps in understanding the issues in the runtime behavior of a DT possibly arising from bad design decisions. Finding such runtime issues could help in the redesign of the DT when developing DTs for similar product lines. Similarly, behavioral analysis can detect anomalies that may arise due to the evolution of DTs and therefore helps with the maintenance of DTs.

9. Conclusion, Challenges & Future Work

In this paper, we proposed to using logging and model learning to (semi-)automate the reverse engineering of a DT for an autonomously driving truck in a distribution center that lacks structured documentation. This approach involved detecting the DT's structure, runtime behavior, runtime issues, and identifying the root causes of these issues. We showcased the importance of logging to understand the runtime behavior of DTs and detect the evolution of DTs by comparing the observed behavior and intended behavior of the DT. In addition, we elucidated on the key role that logging plays for enabling continuous verification of DTs as they keep evolving across their entire lifecycle. We investigated state merging and process mining techniques to (semi-)automate the process of obtaining models that encompass the runtime behavior of DTs, which can be used for further behavioral analysis or model based verification. Our conclusion was that process mining was a better suitable technique than state merging for this DT due to the high level of manual effort required with state merging. Moreover, we demonstrated how further behavioral analysis of the obtained models can be used for performing root cause analysis and detecting cyclic dependencies in the DT.

We believe there is currently a lack of awareness regarding the importance of documentation for DTs. Additionally, there is no standard for documenting DTs, given the myriad of def-

initions and descriptions that exist. We see documenting DTs as a crucial step towards improving their long-term usability, enabling their reuse for purposes beyond their initial intention, and facilitating their aggregation in the virtual space for complex applications, which has been predicted for the future in works such as Muctadir et al. (Muctadir et al. 2024). We plan to conduct a survey on DT documentation to understand what information academicians and industry practitioners from various domains believe should be included in DT documentation.

Moreover, given that DTs consist of various heterogeneous cross-domain models, one of the primary challenges we foresee with logging in DTs is obtaining modeling tool support for generating logs for inter-model interactions. Currently, only a handful of modeling tools support logging activities within a model. With the increasing interest in DTs across various domains and their expansive applications within the engineering spectrum, it is imperative for developers of modeling tools to start considering the perspective of DTs rather than simply modeling. This includes providing support for logging interactions between models within the same or different tools, which make up the DT, as well as providing options for processing collected logs to analyze runtime behavior of DTs. This could also help with the future trend of combining several DTs for a particular purpose to form federated (Vergara et al. 2023) and aggregated DTs (Redelinghuys et al. 2020).

Additionally, model learning to obtain verifiable models is in general a costly affair that is still gathering great interest among researchers in V&V of systems. We attempted to learn the runtime behavior of DTs from event logs using passive learning techniques. However, these models were not mature enough to be used for runtime verification yet. We plan to investigate methods to learn verifiable behavioral models by, for instance, modifying the statechart workbench in ProM. Additionally, we aim to explore the usefulness of the methods described in this paper on more DTs. The above-mentioned future work represents only the initial steps toward analyzing the dynamic behavior of DTs; the next steps will also involve exploring additional continuous validation and verification techniques to ensure runtime consistency for DTs, as well as to automate the detection of anomalous behavior of DTs based on learned models.

Acknowledgments

This research was funded by NWO, the Dutch national research council, under the NWO AES Perspectief Program on Digital Twins (Project P18-03/P3). We would like to thank Mark van den Brand and Loek Cleophas from Eindhoven University of Technology (TU/e) for their continued support in helping establish closer research collaboration between Tilburg University and TU Eindhoven (TU/e), and for helping to conduct our research in collaboration with the Automotive Engineering Science (AES) lab at TU/e. In addition, we would like to thank Eric Verbeek from TU/e for his support with process mining and the ProM tool.

References

Aghajani, E., Nagy, C., Linares-Vásquez, M., Moreno, L., Bavota, G., Lanza, M., & Shepherd, D. C. (2020). Software documentation: the practitioners' perspective. In *Proceedings of the acm/ieee 42nd international conference on software engineering* (pp. 590–601).

Aslam, K. (2021). Deriving behavioral specifications of industrial software components. *Doctoral Dissertation*.

Aslam, K., Cleophas, L., Schiffelers, R. R. H., & van den Brand, M. (2020). Interface protocol inference to aid understanding legacy software components. *Softw. Syst. Model.*, *19*(6), 1519–1540.

Avellaneda, F., & Petrenko, A. (2019). Inferring dfa without negative examples. In *International conference on grammatical inference* (pp. 17–29).

Bano, D., Michael, J., Rumpe, B., Varga, S., & Weske, M. (2022). Process-aware digital twin cockpit synthesis from event logs. *Journal of Computer Languages*, *70*, 101121.

Barosan, I., Basmenj, A. A., Chouhan, S. G., & Manrique, D. (2020). Development of a virtual simulation environment and a digital twin of an autonomous driving truck for a distribution center. In *Software architecture: 14th european conference, ecsa 2020 tracks and workshops, l'aquila, italy, september 14–18, 2020, proceedings 14* (pp. 542–557).

Batoun, M. A., Sayagh, M., Aghili, R., Ouni, A., & Li, H. (2024). A literature review and existing challenges on software logging practices: From the creation to the analysis of software logs. *Empirical Software Engineering*, *29*(4), 103.

Biermann, A. W., & Feldman, J. A. (1972). On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers*, *100*(6), 592–597.

Bruneliere, H., Cabot, J., Dupé, G., & Madiot, F. (2014). Modisco: A model driven reverse engineering framework. *Information and Software Technology*, *56*(8), 1012–1032.

Carrasco, R. C., & Oncina, J. (1994). Learning stochastic regular grammars by means of a state merging method. In *ICGI* (Vol. 862, pp. 139–152). Springer.

Daniele, C., Andarzian, S. B., & Poll, E. (2024). Uses of active and passive learning in stateful fuzzing. *CoRR*, *abs/2406.08077*.

Duato, J. (1995). A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, *6*(10), 1055–1067.

El Saddik, A. (2018). Digital twins: The convergence of multimedia technologies. *IEEE multimedia*, *25*(2), 87–92.

Favre, J.-M. (2005). Foundations of model (driven)(reverse) engineering: models–episode i: stories of the fidus papyrus and of the solarus. In *Dagstuhl seminar proceedings.*

García-Borgoñón, L., Barcelona, M. A., Egea, A. J., Reyes, G., Sainz-de-la maza, A., & González-Uzabal, A. (2023). Lessons learned in model-based reverse engineering of large legacy systems. In *International conference on advanced information systems engineering* (pp. 330–344).

Gold, E. (1967). Language identification in the limit. i nformation andcontrol, 10: 447-474, 1967.[11] s. *Jain. An infinite class of functions identifiable using minimal programs in all Kolmogorov numberings. I nternational J ournalofFoundationsofComputer Science*, *6*(1), 89–94.

Grieves, M. (2014). Digital twin: manufacturing excellence

through virtual factory replication. *White paper*, *1*(2014), 1–7.

Guinea-Cabrera, M. A., & Holgado-Terriza, J. A. (2024). Digital twins in software engineering—a systematic literature review and vision. *Applied Sciences*, *14*(3), 977.

Gunasekaran, R., & Haverkort, B. (2024). Verification of digital twins through statistical model checking. *Companion Proceedings of the 17th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling Forum, M4S,FACETE, AEM, Tools and Demos co-located with PoEM 2024*.

Gunasekaran, R., & Haverkort, B. (2025). Verification of digital twins through classical model checking. *International Workshop on Autonomous System Quality Assurance and Prediction with Digital Twins*. (Accepted.)

Günther, C. W., & Rozinat, A. (2012). Disco: Discover your processes. In *Demonstration track of the 10th international conference on business process management, bpm demos 2012* (pp. 40–44).

Gunther, C. W., & Verbeek, H. (2014). Xes-standard definition.

Hammerschmidt, C. (2017). Learning finite automata via flexible state-merging and applications in networking.

Lang, K. (1998). Evidence driven state merging with search. *Rapport technique TR98–139, NECI*, *31*.

Leemans, M. (2018). Hierarchical process mining for scalable software analysis. *Doctoral Dissertation*.

Leemans, M., Van Der Aalst, W. M., & Van Den Brand, M. G. (2018). The statechart workbench: Enabling scalable software event log analysis using process mining. In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)* (pp. 502–506).

Liu, C. (2018). Automatic discovery of behavioral models from software execution data. *IEEE Transactions on Automation Science and Engineering*, *15*(4), 1897–1908.

Lugaresi, G., & Matta, A. (2021). Automated manufacturing system discovery and digital twin generation. *Journal of Manufacturing Systems*, *59*, 51–66.

Manai, O. (2019). How software documentation helps communication in development teams: A case study on architecture and design documents.

Mao, H., Chen, Y., Jaeger, M., Nielsen, T. D., Larsen, K. G., & Nielsen, B. (2012). Learning markov decision processes for model checking. In *QFM* (Vol. 103, pp. 49–63).

Matuleviçius, R., Kamseu, F., & Habra, N. (2009). Measuring open source documentation availability. In *Proceedings of the international conference on quality engineering in software technology.[cited at p. 23, 193, 197]*.

Mertens, J., Klikovits, S., Bordeleau, F., Denil, J., & Haugen, Ø. (2024). Continuous evolution of digital twins using the dartwin notation. *Software and Systems Modeling*, 1–22.

Muctadir, H. M., Manrique Negrin, D. A., Gunasekaran, R., Cleophas, L., van den Brand, M., & Haverkort, B. R. (2024). Current trends in digital twin development, maintenance, and operation: An interview study. *Software and Systems Modeling*, 1–31.

Muškardin, E., Aichernig, B. K., Pill, I., Pferscher, A., & Tappler, M. (2022). Aalpy: an active automata learning library. *Innovations in Systems and Software Engineering*, *18*(3), 417–426.

Oncina, J., Garcia, P., et al. (1992). Inferring regular languages in polynomial update time. *Pattern recognition and image analysis*, *1*(49-61), 10–1142.

Park, G., & Van Der Aalst, W. M. (2021). Realizing a digital twin of an organization using action-oriented process mining. In *2021 3rd international conference on process mining (icpm)* (pp. 104–111).

Raibulet, C., Fontana, F. A., & Zanoni, M. (2017). Model-driven reverse engineering approaches: A systematic literature review. *Ieee Access*, *5*, 14516–14542.

Redelinghuys, A., Kruger, K., & Basson, A. (2020). A six-layer architecture for digital twins with aggregation. In *Service oriented, holonic and multi-agent manufacturing systems for industry of the future: Proceedings of sohoma 2019 9* (pp. 171–182).

Reim, W., Andersson, E., & Eckerwall, K. (2023). Enabling collaboration on digital platforms: a study of digital twins. *International Journal of Production Research*, *61*(12), 3926–3942.

Reynouard, R., Ingólfsdóttir, A., & Bacci, G. (2023). Jajapy: A learning library for stochastic models. In *QEST* (Vol. 14287, pp. 30–46). Springer.

Rugaber, S., & Stirewalt, K. (2004). Model-driven reverse engineering. *IEEE software*, *21*(4), 45–53.

Sánchez, C., Sipma, H. B., Manna, Z., Subramonian, V., & Gill, C. (2006). On efficient distributed deadlock avoidance for real-time and embedded systems. In *Proceedings 20th ieee international parallel & distributed processing symposium* (pp. 10–pp).

Schutz, W. (1991). On the testability of distributed real-time systems. In *Proceedings tenth symposium on reliable distributed systems* (pp. 52–53).

Shafto, M., Conroy, M., Doyle, R., Glaessgen, E., Kemp, C., LeMoigne, J., & Wang, L. (2010). Draft modeling, simulation, information technology & processing roadmap. *Technology area*, *11*, 1–32.

Singh, M., Fuenmayor, E., Hinchy, E. P., Qiao, Y., Murray, N., & Devine, D. (2021). Digital twin: Origin to future. *Applied System Innovation*, *4*(2), 36.

Stadtmann, F., Wassertheurer, H. A. G., & Rasheed, A. (2023). Demonstration of a standalone, descriptive, and predictive digital twin of a floating offshore wind turbine. In *International conference on offshore mechanics and arctic engineering* (Vol. 86908, p. V008T09A039).

Steffen, B., Howar, F., & Merten, M. (2011). Introduction to active automata learning from a practical perspective. *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures 11*, 256–296.

Tao, F., & Zhang, M. (2017). Digital twin shop-floor: a new shop-floor paradigm towards smart manufacturing. *IEEE access*, *5*, 20418–20427.

Vaandrager, F. W. (2017). Model learning. *Commun. ACM*, *60*(2), 86–95.

Van Den Brand, M., Cleophas, L., Gunasekaran, R., Haverkort, B., Negrin, D. A. M., & Muctadir, H. M. (2021). Models meet data: Challenges to create virtual entities for digital twins. In *2021 acm/ieee international conference on model driven engineering languages and systems companion (models-c)* (pp. 225–228).

Van Der Aalst, W. (2012). Process mining: Overview and opportunities. *ACM Transactions on Management Information Systems (TMIS)*, *3*(2), 1–17.

Van Der Aalst, W. M., & Dustdar, S. (2012). Process mining put into context. *IEEE Internet Computing*, *16*(1), 82–86.

Van der Werf, J. M. E., van Dongen, B. F., Hurkens, C. A., & Serebrenik, A. (2008). Process discovery using integer linear programming. In *International conference on applications and theory of petri nets* (pp. 368–387).

Verbeek, F., & Schmaltz, J. (2012). Proof pearl: A formal proof of dally and seitz'necessary and sufficient condition for deadlock-free routing in interconnection networks. *Journal of Automated Reasoning*, *48*(4), 419–439.

Verbeek, H., Buijs, J., Van Dongen, B., & van der Aalst, W. M. (2010). Prom 6: The process mining toolkit. *Proc. of BPM Demonstration Track*, *615*, 34–39.

Vergara, C., Bahsoon, R., Theodoropoulos, G., Yanez, W., & Tziritas, N. (2023). Federated digital twin. In *2023 ieee/acm 27th international symposium on distributed simulation and real time applications (ds-rt)* (pp. 115–116).

Verwer, S., & Hammerschmidt, C. A. (2017). flexfringe: A passive automaton learning package. In *ICSME* (pp. 638–642). IEEE Computer Society.

Walkinshaw, N., Lambeau, B., Damas, C., Bogdanov, K., & Dupont, P. (2013). STAMINA: a competition to encourage the development and assessment of software model inference techniques. *Empir. Softw. Eng.*, *18*(4), 791–824.

Walkinshaw, N., Taylor, R., & Derrick, J. (2016). Inferring extended finite state machine models from software executions. *Empirical software engineering*, *21*, 811–853.

Yang, N., Aslam, K., Schiffelers, R., Lensink, L., Hendriks, D., Cleophas, L., & Serebrenik, A. (2019). Improving model inference in industry by combining active and passive learning. In *2019 ieee 26th international conference on software analysis, evolution and reengineering (saner)* (pp. 253–263).

Yang, N., Cuijpers, P., Schiffelers, R., Lukkien, J., & Serebrenik, A. (2021). An interview study of how developers use execution logs in embedded software engineering. In *2021 ieee/acm 43rd international conference on software engineering: Software engineering in practice (icse-seip)* (pp. 61–70).

Zhang, L., Zhou, L., & Horn, B. K. (2021). Building a right digital twin with model engineering. *Journal of Manufacturing Systems*, *59*, 151–164.

Zhuang, C., Liu, J., & Xiong, H. (2018). Digital twin-based smart production management and control framework for the complex product assembly shop-floor. *The international journal of advanced manufacturing technology*, *96*, 1149–1163.

About the authors

Raghavendran Gunasekaran is currently pursuing his PhD. research in the topic of dynamic consistency in digital twins, in the Tilburg School of Humanities and Digital Sciences at Tilburg University. He is also a guest researcher in Software Engineering and Technology group at Eindhoven University of Technology. Prior to this, he obtained his Masters degree in Embedded Systems from Eindhoven University of Technology. He completed his bachelors in Electronics & Instrumentation Engineering at Shanmugha Arts, Science, Technology and Research Academy (SASTRA university). Furthermore, he has 3 years of experience working in the software industry. His research interests include model-based system engineering (MBSE), digital twin engineering, verification and validation (V&V) of software systems including digital twins, cyber-physical systems (CPS), process mining, domain specific languages (DSLs) and internet-of-things (IoT). You can contact the author at r.gunasekaran@tilburguniversity.edu.

Boudewijn R. Haverkort is Dean of the Faculty of Electrical Engineering, Mathematics and Computer Science at the University of Twente in the Netherlands, since 2024. He received an engineering and a Ph.D. degree from the University of Twente in 1986 and 1991, respectively. Before moving to University of Twente in 2024, he was the Dean of School of Humanities and Digital Sciences at Tilburg University, the Netherlands from 2019 to 2024. He has been a full professor at the University of Twente from 2003 to 2018 and was a professor at RWTH Aachen University, Germany, from 1995 to 2002. From 2009 to 2013, he also served as Scientific Director of the public–private Embedded Systems Institute, Eindhoven, the Netherlands, an applied research institute focusing on high-tech cyber-physical systems design. His field of interest encompasses internet technology, cyber–physical systems, smart energy systems, energy management in data centers, computer performance and reliability evaluation, stochastic model checking, AI and data science, and digital twins. He has published 250+ papers, as well as two textbooks (on computer-system performance and dependability evaluation; both with Wiley) and chaired many international conferences. He has been a Fellow of IEEE since 2007, and received together with co-authors the *Jean-Claude Laprie Award for Dependable Computing* in 2023, for the 2003-paper "Model-checking algorithms for continuous-time Markov chains". You can contact the author at b.r.h.m.haverkort@utwente.nl.

Loes Kruger is a PhD. researcher at Radboud University, Nijmegen, The Netherlands. She obtained a Bachelors degree in Artificial Intelligence and a Masters degree in Computing Science, both at Radboud University. Her research interests lie in automata learning with a specific focus on adaptive automata learning and improving the approximation of the equivalence query during active automata learning. You can contact the author at loes.kruger@ru.nl.