Verification of Digital Twins through Statistical Model Checking

Raghavendran Gunasekaran[1], Boudewijn R. Haverkort[1,2]

[1]Tilburg University, Tilburg School of Humanties and Digital Sciences, the Netherlands
[2]University of Twente, Faculty for Electrical Engineering, Mathematics and Computer Science, the Netherlands

Abstract

Since its inception as abstract notion in 2002, Digital Twins (DTs) have received increased attention over the last 5 to 10 years, in both industry and academia. Most definitions of DTs focus on the existence of a virtual entity (VE) which faithfully represents a real-world physical entity (PE), and typically consist of a number of inter-connected models, undergoing changes continuously owing to the synchronization with its PE. During cosimulation, interactions between the various components making up a VE can be stochastic and time-critical in nature, which could lead to undesired (and unexpected) behavior. The continuous evolution of VE might further affect the nature of these interactions and corresponding model execution times, which could possibly affect its overall functioning during run-time. This creates a need to perform (continuous) verification of the VE, to ensure that it behaves consistently during execution by adhering to desired properties of interest such as timeliness, functional correctness, deadlock freeness and others. In this paper we propose to use statistical model checking (SMC) as a technique to verify a VE at runtime. For that purpose, we propose to use a stochastic timed automata (STA) model of the VE and its interacting components, and verify it using UPPAAL SMC. We present our observations and findings from applying this technique on the cosimulation of a DT of an autonomously driving truck.

Keywords

Digital Twin, Statistical Model Checking, Verification

1. Introduction

The wide availability of digital techniques has given an enormous boost to what is now often referred to as smart manufacturing or (even more general) smart industry [1]. Only recently, the development and application of so-called digital twins (DTs) brought about new opportunities ranging from optimization, diagnosis and prognosis, validation, to monitoring for smart industrial systems; for that reason, DTs have been widely adopted by industries across varied domains. Being a relatively new concept, DTs have been defined in many different ways starting from high fidelity simulation [2], a digital representation of a real world object with focus on the object itself [3], an integrated model comprising geometrical, behavioral and contextual information [4], a digital representation with automated bi-directional data flow with the real world entity [5]. We adhere to the five dimensional model of a DT, cf. [6], which comprises (1) a Physical Entity (PE): the real world entity such as any object, process or concept; (2) a Virtual Entity (VE): a high fidelity virtual representation of the real world entity which comprises a collection of models; (3) Data: multi-temporal data from PE, VE, services, knowledge from domain experts and fusion of these data; (4) Services: application services such as simulation, optimization and others; and (5) Interconnections: Connections between the aforementioned four entities. Our view of DTs is highly influenced by this 5D model of DT. In the rest of the

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, Stockholm, Sweden, December 3-5, 2024

r.gunasekaran@tilburguniversity.edu (R. Gunasekaran); b.r.h.m.haverkort@utwente.nl (B. R. Haverkort)
https://research.tilburguniversity.edu/en/persons/boudewijn-haverkort (B. R. Haverkort)
0009-0002-5526-9130 (R. Gunasekaran); 0000-0002-0654-0740 (B. R. Haverkort)

paper, we adopt the terminology proposed in [6], i.e., VE to represent the virtual representation of the real world entity and PE to represent the real world entity. Though several definitions exist for DTs, most of these definitions focus on the fact that DTs include a virtual representation (often represented by a set of models) of a real world entity which synchronizes continuously with its physical counterpart. Within such a DT, multitude of interactions occur (between the different models in VE at runtime during cosimulation and also, between PE and VE owing to continuous synchronization for data exchange) which could possibly lead to undesirable behavior at runtime. This creates a need for verification of runtime behavior of DTs. The generally applied best practice for verification of systems is that they are verified at design stage, that is, before deployment. However, a DT is a system which evolves continuously over its entire lifecycle, hence, verification only at its design stage is not sufficient. Thus, it requires a verification technique that can potentially be applied at any stage of its lifecycle during its runtime as it evolves continuously.

Contribution of this paper. While existing literature primarily discusses verification of the conspicuous behavior of DTs, this paper, in contrast, focuses on verifying the runtime behavior of a DT. The runtime behavior of a DT is the emergent behavior arising from the compositional effect of interactions between the different components constituting the DT. We present a novel approach of applying the technique of statistical model checking to verify the runtime behavior of DTs, which can be applied at any stage of its lifecycle as it evolves continuously. We apply this technique to a DT case study undergoing cosimulation for control application, and perform what-if explorations using the UPPAAL SMC toolset [7].

2. Background

The runtime behavior of a DT is the emergent behavior arising from the compositional effect of interactions between the different components constituting the DT. Tao et al. [8] describe the interactions between the different components constituting the DT as one of the main pitfalls in a DT. They further discuss the need for focus on timeliness and comprehensiveness[1] of these interactions within a DT. Van den Brand et al. [9] discusses the stochastic nature of interactions between heterogeneous components within a DT which may lead to undesirable behavior. It further elucidates that the sequence of interactions within the VE could be time-critical, or of unexpected sequence, which when combined with variable model execution times may lead to functional and performance issues at runtime. Moreover, these runtime issues in a DT can only be observed during the execution of the overall DT, that is, when the different components in the DT interact with each other and it cannot be ascertained from the individual components. In the case of ad hoc creation of DTs, where modular DTs are combined (orchestration), also unforeseen runtime issues can arise. Bordeleau et al. [10] also presents research challenges in detecting and handling uncertainties in DTs and runtime environments, owing to variations in interactions with the PE, changing operational conditions and unpredictability in human behavior. Such uncertainties and issues at runtime in DTs, emanating from the comprehensiveness and variations of interactions between the different components constituting the DT, create a need for verification of runtime behavior of DTs.

[1] The term comprehensiveness has been used by Tao et al. and it represents the many components and links which are considered in the process of interaction within a DT.

In the case of the VE, not only the multitude of interactions between cross-domain models (possibly executing in different execution platforms) pose a challenge, the evolution of VE owing to its continuous synchronization of data with the PE poses additional complexity. Zhang et al. [11] discusses three types of evolution of VE, namely, self-to-self evolution, self-to-new evolution and VE reconstruction. Furthermore, other factors such as bug fixes made in VE and improvements or modifications made in PE and consequently in VE, could possibly contribute to the evolution of VE. All these changes that the models within a VE undergo as part of its evolution might possibly affect the interactions between them, creating more uncertainty which could lead to undesirable behavior. Hence, performing verification & validation (V&V) only at the design stage of DT wouldn't ensure proper functioning of a DT across its lifecycle and thus, its dynamic nature warrants its (continuous) verification at runtime across its lifecycle.

Current literature on V&V of DTs, such as Grieves et al. [12], discusses three validation tests for VE: visual tests, performance tests, and reflectivity tests. All three focus on validating the fidelity of the VE, i.e., how closely the VE imitates the behavior and characteristics of the PE. Dalibor et al. [13] discuss current methods for V&V of the VE, as found in their literature survey on DTs. These methodologies are consistency monitoring, simulation, testing, model checking and others. Most of these validation methods focus only on validating the VE at design time, *not* at run-time. Consistency monitoring, which is done during the operation of DT, only monitors the difference between predictions made through the VE and the data obtained from the PE. This is a very high level of validation which is similar to the reflectivity test discussed by Grieves et al. [12]. In the interview research paper [14], is is discussed that 13 out of the 19 practitioners from both industry and academia are currently validating their DT by comparing the behavior between VE and PE. Moreover, the aforementioned works do not take into account the fact that the VE evolves continuously.

To the best of our knowledge, there is currently no literature which explicitly proposes verification of runtime behavior of DTs. This paper presents a novel approach in applying SMC to verify the runtime behavior of DTs.

3. Statistical model checking

Testing is one of the most widely used technique for verification of systems. Nevertheless, testing does not ensure complete correctness of a system. On the contrary, formal verification techniques such as model checking are able to ensure complete correctness of a system. However, classical model checking suffers from state space explosion issues. Even for simple systems, at times the computed state spaces may become too large for realistic computational and memory resources. The state space explosion issue highly affects the adoption of this technique for verification of systems. On the contrary, approaches based on statistical analysis such as SMC do not face issues with state space explosion. SMC is a technique which focuses on running a large but finite number of system executions in order to obtain statistical evidence for the validity of the specified properties. Moreover, SMC provides the option to set the degree of statistical confidence with which the specified properties need to be checked. Since these techniques rely on samples of the system execution or simulations, they can be applied for a wider range of systems for verification with estimates on probability measures [15, 16, 17]. Furthermore, SMC can be applied to systems with complex dynamics, when their behavior is stochastic in nature

[18]. SMC can be seen as a forming compromise between testing and model checking [17]. Moreover, the executions needed for SMC can be distributed and run in parallel. SMC has been used in a wide range of domains such as software engineering, security, automotive, energy systems, systems biology and others [17, 18]. One of the popular tools for performing SMC is UPPAAL SMC. It is a feature rich tool which provides options for probability estimations, performing simulations, value estimations and hypothesis testing. Combining the results from these multiple techniques help in better analysis of systems. As mentioned in Section 2, to our knowledge, there is currently no literature addressing verification of runtime behavior of DTs using SMC. We propose to use SMC techniques for verification of DTs during actual execution, using the well-established tool UPPAAL.

4. Digital twin case study

We have taken a DT of an autonomously driving scaled-down truck in a distribution center as driving case study. This DT, as visualised in Figure 1, was developed in Trucklab at TU Eindhoven[2] over the past several years, by a large group of PhD and MSc students; it is a highly modified version of the DT developed by Barosan et al. [19]. The truck and distribution center (PE) at the Trucklab in TU/e are scaled down by a factor of 13.3, as compared to real life [19]. This DT was developed to test the autonomous driving and docking functionalities in a distribution center with several obstacles. The purpose of this DT was to help in improving the transportation efficiency and safety in distribution centers using autonomously driving trucks. The components in the DT interact with each other during cosimulation in order to effect the optimal control output for effective motion of the truck. There is no orchestrator here which ensures a proper schedule during cosimulation. There was no complete or structured documentation for this DT as well. Currently, in this DT, the truck in the virtual simulation environment always crashes before it reaches its destination dock. Clearly, this is not an acceptable behavior of the DT and moreover, there was no verification performed at design stage as well. Thus, there is a need for verification of this DT to understand this behavior further which would help in the betterment of its design. This DT comprises four components which are detailed below. The communication between all the components in DT occurs using the User Datagram Protocol (UDP). All the interactions occurring between the components of the DT during cosimulation and the variables exchanged during these interactions are depicted in Figure 2. The four components of the DT are:

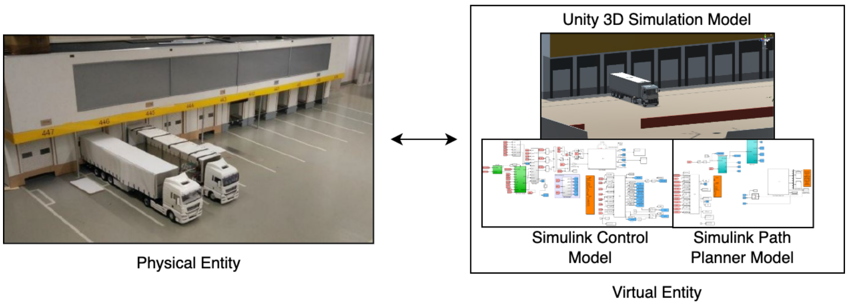

Figure 1: Digital Twin (PE and VE) of an autonomously driving truck in a distribution center

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

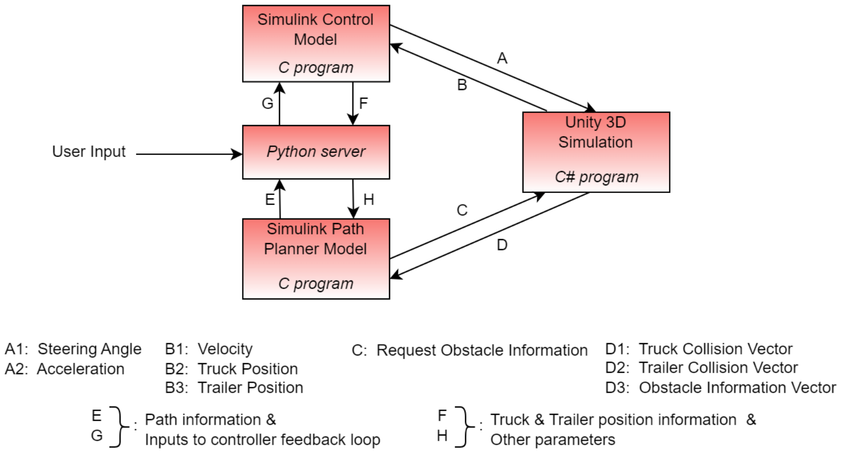
Simulink Control Model
C program
Python server
Simulink Path Planner Model
C program
Unity 3D Simulation
C# program
User Input
A
B
C
D
E
F
G
H
A1: Steering Angle
A2: Acceleration
B1: Velocity
B2: Truck Position
B3: Trailer Position
C: Request Obstacle Information
D1: Truck Collision Vector
D2: Trailer Collision Vector
D3: Obstacle Information Vector
E, G : Path information & Inputs to controller feedback loop
F, H : Truck & Trailer position information & Other parameters

Figure 2: Interactions within the Virtual Entity

- **Path planner model in Simulink:** In order for the truck to get from point A to B, a path needs to be planned from the starting position A of the truck to the desired destination dock B in the distribution center, thereby avoiding the obstacles on the path. The path planner model comprises a C program which is also responsible for enabling the necessary UDP communication. It requests for the obstacle position continuously in order to compute the path. A Unity model then sends the obstacle position, and truck and trailer collision vectors to this model.
- **Controller model in Simulink:** The controller model comprises a PID (Proportional Integral Derivative) controller, which controls the motion of the truck in the Unity simulation model. The controller model also comprises a C program which is responsible for enabling the UDP communication. From the Unity model, it receives the velocity, and the truck and trailer position as input. In addition, from the Path planner model it receives the path information as input. These two inputs help the Controller model in computing the optimal control output, which is sent to the Unity model. This control output, given in terms of the steering angle and acceleration, drives the motion of the truck, negotiating the obstacles in the Unity model.
- **3D virtual environment (simulation model) in Unity:** The 3D virtual environment in the Unity game engine is used for simulation of the autonomously driving truck in the distribution center. It is a virtual copy of the structure of the scaled down trucks in the distribution center. It comprises two C# programs which are responsible for handling the UDP communication with the Controller and Path planner models respectively. As mentioned earlier, it receives steering angle and acceleration as input from the Controller model in Simulink which drives the motion in the virtual environment. When the Unity model receives this input, it also sends the velocity and position of the truck and trailer to the Controller model. In addition, the Path planner model in Simulink requests the obstacle position from Unity. The Unity model then sends the obstacle position, and truck and trailer collision vectors as response to the Path planner model.
- **Python Server:** The python server is an additional component in the DT which is responsible for handling the UDP communication between the Controller and Path planner models. The Path planner model requires the velocity, position vectors of the

truck and other information from the Controller model, which is facilitated by this Python server. Similarly, the Controller model requires the inputs to the control feedback loop, path information, obstacle position and other information from the Path planner model which is again facilitated by the Python server. In addition, the Python server is the only component through which the user interacts with the DT. The user is required to provide two inputs for the DT execution, namely, the *Loading Dock* (destination information) and *Kill Switch* (a boolean value) which controls the initiation of the autonomous driving and effectively, for the overall cosimulation to begin.

5. Modeling in UPPAAL SMC

UPPAAL SMC is a statistical model checker which allows users to model the behavior of a system as a stochastic timed automata (STA). Understanding the runtime interactions of DTs and computing the time taken for every interaction within the DT, provides sufficient information for us to model the runtime behavior of DTs in UPPAAL SMC. We refer the reader to the literature for complete semantics, formal definition and modeling of STA.

5.1. Understanding the runtime behavior of DT

In order to model the runtime behavior of DT for SMC, it is necessary to log the interactions between the components constituting the DT. We are currently not aware of literature which explicitly focuses on generating event logs from DTs. 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 which 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 the variables exchanged during the interaction. Furthermore, caution had to be exercised to avoid log duplication, i.e., to avoid multiple logs generated for the same interaction (between two components) in two different places. In order to ensure this, we meticulously planned to generate logs only at two places (components) which do not directly interact with each other, thereby making sure to log all the interactions happening within the DT. Consequently, we decided to log interactions only in the Unity game engine and in the Python server. The log files were exported into spreadsheet calculation software, where they were merged based on order of timestamp. From a single run of this DT, we obtained a log file comprising some 50000 log items for 180 seconds of wall clock time (actual elapsed time). When we analyzed the event logs to understand the runtime behavior of DT, we detected an atypical behavior in the interactions between the Unity game engine and Simulink. This interaction behavior was in the form of an on/off-pattern (described in the following paragraph *STA of components of DT*) and occurred only in the interactions between the two Simulink models and the Unity model. This suggests the need to create a STA for every component of the DT involved in the interactions. In the following paragraphs, we elucidate on the STAs created for the components of the DT.

5.2. STA of components of DT

The STA for the Unity model is shown in Figure 3 and the details of this model are elucidated in this section. From the *Begin* state, this STA can take the edges *start_simulation?* and *switch_on?* to synchronize with the edges in the STA of User. The STA of a User encompasses the behavior of the user providing the inputs discussed in Section 4 for manually initiating the autonomous driving and for the cosimulation to begin. These edges take the execution from *begin* state to the *S2*. The *S2* is an urgent state where there is no advancement of time. The edge *A1,A2?* represents the messages steering angle and acceleration communicated from the Simulink controller model to the Unity model, specified as A1 and A2 in figure 2, respectively. On receiving these values from the Simulink controller model, the Unity model moves to the *S3* state. The STA stays in this state for a time computed from the statistics (which is discussed in section 5.3), and specified in the invariants and guard conditions. The edge *B1,B2,B3!* represents the Unity model sending the velocity, and truck and trailer position to the Simulink controller model, specified as B1, B2 and B3 in Figure 2, respectively. On sending these variables, the STA moves to the state *S4*. From this state, there are two possibilities for the execution to move forward. The execution can either loop, i.e., interactions A1 and A2 can occur again or the STA can move to the *S5* state. The above states *S2*, *S3* and *S4* form a loop and this is the ON loop. From *S5*, the STA then goes into the OFF loop, i.e., the loop in which the Unity model receives the message C from the Simulink path planner model and responds by sending the messages D1, D2 and D3 as specified in Figure 2. From *S5*, the STA then goes to the *S6* which is an urgent state like *S2*. Whenever Unity model receives a request from the Simulink Path planner model, specified as C in Figure 2, then the edge *C?* is taken to *S7*. When the Unity model sends the messages D1, D2 and D3 to the Simulink path planner model, this STA takes the edge *D1,D2,D3!* and moves to *S8*. From this state, again there are two possibilities for the execution to move forward. The execution can either loop, i.e., the above interactions C, and D1, D2 and D3 can occur again, or it can move to *S9*. The above states *S6*, *S7* and *S8* form a loop and this is the so-called OFF loop From *S9*, the STA goes back to *S2*, and the entire execution as discussed above repeats. From observing the count of loop iterations from the logs (which is varying but roughly around 10 iterations), in this STA, it is assumed that each of the two (ON and OFF loops) occur for 10 consecutive iterations before switching between each other. This has been specified in the model using a counter variable, which is reset when switching to the other loop. Furthermore, there is one other state in this STA, which is the *End* state; the STA moves here whenever the *Simulation_time_end?* edge is taken, which can happen from all of the states in the STA, because the simulation time can end anytime, i.e., in any of the above states. Similar to the above description of STA of the Unity simulation environment, STAs of Simulink controller model, Simulink path planner model and Python server were created. We omit the details of these STAs here owing to space constraints. Delays were specified in locations whose values were obtained from the statistical analysis of the logs (see below). The simulation time was set to a wall clock time of 100 seconds and a separate clock variable was created to keep track of this time. The STAs created for every component of the DT were created as templates in UPPAAL SMC, which allows one to create multiple instances of each and every STA (if needed).

5.3. Statistical analysis & modeling exact behavior in UPPAAL SMC

As mentioned above, timestamps of interaction events in DT were logged in microsecond precision in order to grasp the duration of these interactions. For modeling the interactions during cosimulation, we computed the difference in time (delay) between each and every interaction in the DT. Considering that we have a multitude of delay values (between every two interactions) and the variations of these values, we had to perform statistical computations to arrive at values

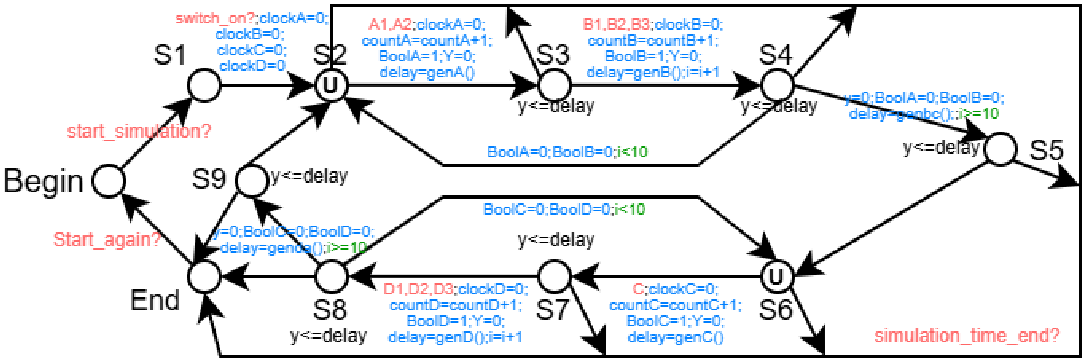

Figure 3: STA of Unity simulation environment

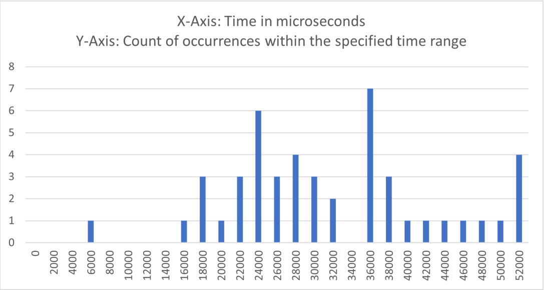

Figure 4: Time distribution obtained for an interaction within the DT

for a delay occurring between two interactions. We computed mean, median, mode, standard deviation, variance, minimum and maximum values for the ten delay variables. However, from these statistical computations we discovered that even within one single execution, comprising 50000 interactions, the timing values were highly varying. We created histograms of these timing values to better understand their variability. One such histogram is shown in Figure 4. The minima, mean and maxima values for the delay shown in this figure are 4256, 41,940.8 and 484,153 microseconds, respectively. It can be clearly understood that these 3 simple statistical values do not unequivocally represent the actual timing distribution shown in Figure 4. So, in order to verify the VE with its stochastic nature of interactions, we decided to use this exact varying timing data from histograms as it is for SMC.

Classical model checking does not provide the possibility to verify the DT with such stochastic timings. In contrast, UPPAAL SMC is more suitable to handle cases of stochastic temporal behavior. With UPPAAL SMC which supports C programs, it is possible to encode custom time distributions [17]. When custom time distributions are used, we can use a random generator to generate values from the specified time distribution.

6. Formal analysis and experimentation

The next step after modeling in UPPAAL SMC is to formally specify the properties of interest which are needed to be verified in the model. As mentioned earlier in Section 4, there is no documentation or requirements for this DT, which could be translated into properties. Thus, we had created a list of properties which we believe could possibly ensure a proper functioning and timeliness of the DT. These properties to be verified are listed in Table 1. The queries for UPPAAL SMC use the formal language of MITL (Metric Interval Temporal Logic) [20]. We refer the reader to UPPAAL SMC documentation[3] for query syntax and semantics. One of the disadvantages with SMC, as mentioned in Section 3, is that it does not perform a complete state space exploration. Thus, in cases where stochastic time distributions are specified using a random() function in UPPAAL, queries such as deadlock freeness cannot be verified. Hence, we used model checking to verify the property of deadlock freeness and other relevant properties in VE, which is planned to be discussed in another paper. The probability values derived by SMC will be between 0 and 1, where values closer to 0 denote smaller probability of a property holding, and values closer to 1 denote the higher probability of a property holding good.

All our experiments were performed on a an Intel(R) Core(TM) i7-10510U CPU, clocked at 1.80GHz, 2.30 GHz, which comprises 4 cores and 8 logical processors, 32GB RAM and running a 64bit Windows 10 OS. UPPAAL version 5.0.0 was used for the experiments for SMC, and the confidence parameters, i.e., the probability of false negatives(), and the probability of false positives() and uncertainty() were all set to 5% (Confidence Interval of 95%).

6.1. Properties and results from the UPPAAL SMC verifier

In addition to probabilistic results, SMC also helps in determining the worst case response time (WCRT) of each interaction, which may affect the overall cosimulation output. Moreover, the simulations also yield insights in the interaction behavior, by means of comparing the frequency and delay of occurrence of one interaction with that of other interactions.

The properties for verification and their corresponding results are listed in Table 1. The UPPAAL SMC queries in MITL used for the aforementioned properties are listed in [21] (due to space constraints). The results of simulation queries numbered 14 and 16 in Table 1, can be found in Figure 5. The results of simulation queries 13 and 15 are similar to the result shown in Figure 5a. In addition, the results of simulation queries 17 and 18 are similar to the result shown in Figure 5b. Considering this similarity and owing to space constraints, we do not include the results of queries 13, 15, 17 and 18. From the results in Table 1, one can observe that the probabilistic results of the properties of interest are self explanatory. In terms of timeliness, it can be observed from this table that all (queries 2, 3, 4 and 5) except one (query 6) of the timeliness properties are not satisfied. Furthermore, queries for WCRT also provide very high delay values which are clearly not suitable for an effective cosimulation for seamless control of motion of the truck. For instance, the WCRT of the control output sent by the simulink controller model is roughly 1387 milliseconds, which is a considerably larger delay for the truck to be in motion while not receiving any new control output. This clearly helps us understand why the truck in the virtual simulation environment always collides with the obstacle before reaching its destination dock. We speculate that this higher delay value could be attributed to either the communication protocol used in the DT or the lack of sufficient processing power or both. In terms of functional correctness, it can be observed that functional properties (queries 1 and 7) all hold. These properties ensure that a cosimulation once started will eventually end and also, during cosimulation, none of the components will terminate their execution while other components are still executing. This ensures some level of functional correctness of VE. However, analysis of these probabilistic results when combined with the results from

[3] See https://docs.uppaal.org/language-reference/query-semantics/smc_queries/.

Table 1
Properties & results from UPPAAL SMC verifier for actual case and what-if exploration (case 1 and 2)

No.	Properties	Actual case	Case 1	Case 2
1	What is the probability of a situation occurring when the interactions between the two simulink models and python server will not stop while the interactions between the two simulink models and unity model continues to progress?	Pr >= 0.999631	Pr >= 0.999631	Pr >= 0.999631
2	For the constant motion of the truck in the Unity simulation engine, what is the probability of Simulink controller model sending control output to Unity repeatedly within a specific time?	Pr <= 0.00036882	Pr <= 0.00036882	Pr >= 0.999631
3	For continuous computation of path information, what is the probability of the Unity simulation engine sending the obstacle position information repeatedly within a specific time?	Pr <= 0.00036882	Pr <= 0.00036882	Pr >= 0.833902
4	What is the probability of the Simulink controller model receiving the path information every time before it sends the control output to Unity model?	Pr <= 0.00036882	Pr <= 0.00036882	Pr <= 0.00036882
5	What is the probability of the Simulink path planner model receiving the obstacle position information every time before it sends the path information to the Python server?	Pr <= 0.000279784	Pr <= 0.00036882	Pr <= 0.00764802
6	What is the probability that interactions between the Simulink controller model and Python server happens every time after the interaction between Simulink controller model and Unity model occurs?	Pr >= 0.997945	Pr <= 0.00036882	Pr >= 0.999631
7	What is the probability that the cosimulation will eventually end?	Pr >= 0.999631	Pr >= 0.999631	Pr >= 0.999631
8	What is the maximum value of delay of sending control output by the Simulink controller model repeatedly?	1387.94 +/- 1.35 ms	1301.09 +/- 1.31 ms	77.99 +/- 0.02 ms
9	What is the maximum value of delay of sending velocity information by the Unity model repeatedly?	1387.38 +/- 1.33 ms	1301.88 +/- 1.34 ms	77.99 +/- 0.03 ms
10	What is the maximum value of delay of sending the obstacle information by the Unity model repeatedly?	696.998 +/- 0.88 ms	634.845 +/- 1.01 ms	93.49 +/- 0.46 ms
11	What is the maximum value of delay of sending path information to the Simulink controller model repeatedly?	976.239 +/- 2.03 ms	24.78 +/- 0.03 ms	803.012 +/- 2.25 ms
12	What is the maximum value of delay of sending truck position to the Simulink path planner model repeatedly?	1009.05 +/- 2.28 ms	28.26 +/- 0.04 ms	819.57 +/- 2.41 ms
13	Simulate the occurrences of the interactions Simulink controller model output (A1,A2) and the python server sending it the path information (G).	-	Figure 6	-
14	Simulate the occurrences of the interactions Simulink pathplanner model output (E) to the python server and the Unity model sending the obstacle information (D1,D2,D3).	Figure 5a	-	-
15	Simulate the occurrences of the interactions Simulink controller model output (F) and the Unity model sending the velocity, and truck and trailer positions (B1,B2,B3).	-	-	-
16	Simulate the delay values of Simulink controller model output(A1,A2) and the python server sending it the path information(G).	Figure 5b	-	-
17	Simulate the delay values of Simulink pathplanner model output (E) and the Unity model sending the obstacle information (D1,D2,D3).	-	-	-
18	Simulate the delay values of Simulink controller model output (F) and the Unity model sending the velocity, and truck and trailer positions (B1,B2,B3).	-	-	-

the simulations in queries (13-18), provides better clarity on the runtime behavior of VE. For example, the results of property 5 (the probability of the Simulink path planner model receiving the obstacle information always before it sends the path information to the python server) evaluates to a probability of only 0.000279784 (doesn't hold good always). However, the reason why property 5 may have evaluated to a very low probability could be attributed to the initiation phase of the cosimulation which is marked in green circle in Figure 5a. Within the green circle, it can be observed that before the cosimulation goes into a steady state, for a brief period of time, the number of occurrences of the interactions of Unity model sending the obstacle information to the pathplanner model is lower than than that of pathplanner model sending the computed path information to the python server. Hence, this is the reason why property 5 doesn't hold good *always* in the DT.

Moreover, observing the results of queries 13, 14 and 15 in Figure 5a, one can comprehend that the occurrences of interactions between the Unity model and the two Simulink models are eventually much higher than the occurrences of interactions between the two Simulink models through the python server. A hypothetical speculation made from this would be that the delays in interactions between the Unity model and the two Simulink models would always be much lesser (owing to the higher number of occurrences of these interactions) than the delays in interactions between the two Simulink models and the python server. However, in reality this is not the case, as can be observed from the results of queries 16, 17 and 18 in Figure 5b, that delays in interactions between the Unity model and the two Simulink models are higher than that of the interactions between the two Simulink models and the python server. This contrast in results between figures 5a and 5b can be better understood by observing the increase in the number of occurrences of interactions between the Unity model and the two Simulink models in figure 5a. It can be observed that there is a staircase-kind-of (stairs of red lines in figure 5a) increase for the number of occurrences of interactions between the Unity model and the two Simulink models. This staircase-kind-of increase suggests that there are bursts of occurrences of interactions between the Unity model and the two Simulink models and then, there is a short period of non-occurrence of these interactions. This represents the on-off interaction behavior pattern between the two Simulink models and the Unity model, which was discussed in section 5.

6.2. What-if explorations based on time

In what follows, we investigate what the impact is of speeding up some interactions in VE.

6.2.1. Case I: Speeding up interactions pertaining to the Python server

The time distribution used for the speed up was to make all the interactions involving Python server faster than the interactions involving the Unity model. So, the difference in time of interactions E, F, G and H as shown in figure 2 have been decreased. The same queries used and their corresponding results from the verification of updated model are listed in Table 1. The result of simulation query 13 is shown in figure 6. The results of simulation queries 14-18 have been intentionally excluded from this paper owing to space constraints. From the results of case 1, we can observe differences in both the probabilistic results and WCRT of case 1 and those of the actual runtime behavior of VE. This helps in understanding that speeding up interactions involving one component (Python server), eventually leads to the speed up of interactions involving the other components (Unity model), even though of lower magnitudes.

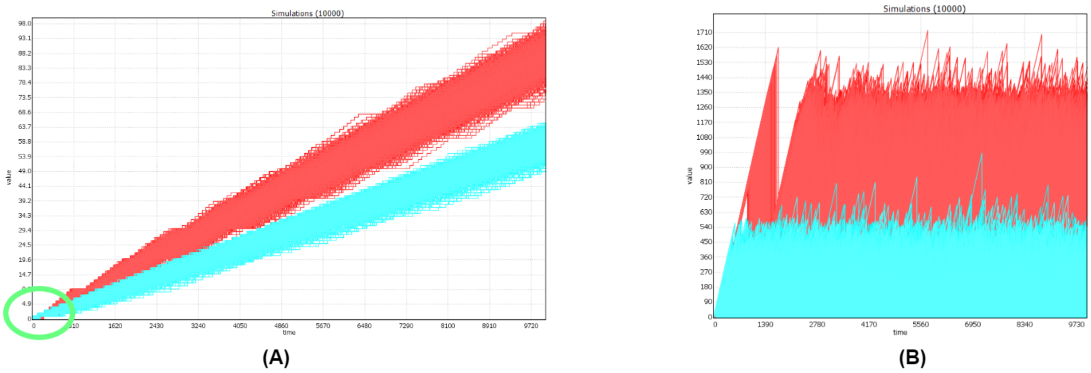

Figure 5: A: Simulation result of Query 14: Number of occurrences of the interactions Unity model sending obstacle information (D1,D2,D3; represented in red) and the path planner model sending the computed path information to python server(E; represented in blue); B: Simulation result of Query 16: Amounts of delay of the interactions of Simulink controller model output (A1,A2; represented in red) and the Python server sending to it the computed path information (G; represented in blue)

Moreover, in this case 1 of speeding up interactions involving python server, one would have expected that probabilistic query 4 to have a higher probability. This is because the speeding up of interactions involving python server would possibly ensure that the path information sent from the python server to the Simulink controller model would occur more frequently than the control output sent from the Simulink controller model to the Unity model. However, from the green circle in figure 6, it can be observed that during the initiation phase of cosimulation, the interaction of Simulink controller model sending control output to Unity model has higher number of occurrences than the interaction of python server sending the path information to the Simulink controller model. Eventually, the effect of speed up of the interaction of python server sending path information to the Simulink controller model can be observed by finding the higher number of occurrences of this interaction once the cosimulation reaches steady state.

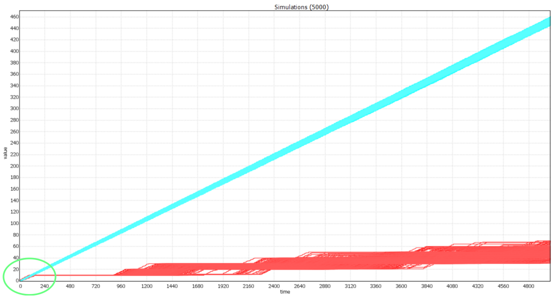

Figure 6: Simulation result of Query 13 for case I: Number of occurrences of the interactions Simulink controller model output (A1,A2; represented in red) and the Python server sending it the computed path information (G; represented in blue)

6.2.2. Case II: Speeding up interactions pertaining to Unity simulation engine

The time distribution used for the speed up in this case was to make all the interactions involving Unity model faster than the interactions involving the python server. The same queries used and their corresponding results from the verification of updated model are listed in Table 1.

The results of simulation queries 13-18 have been intentionally excluded from this paper owing to space constraints. From the results of case 2, we can observe differences between the probabilistic queries of case 2 and those of the actual runtime behavior of VE. Alike in case 1, we can also observe differences in WCRT of all the interactions in case 2, with that of the actual runtime behavior. The speed of interactions involving Unity model has sped up interactions involving the python server. In addition, we can also observe differences in the probabilistic values for queries 2 and 3 with that of the actual behavior of DT. The higher probabilistic values for queries 2 and 3, suggest that speeding up the interactions involving Unity model eventually leads to a quicker response time for critical interactions such as the control output from Simulink controller model and obstacle information sent from the Unity model to the path planner model. This can also be corroborated with queries 8 and 10, which yield a lower WCRT compared to the actual runtime behavior. The quicker response time of these critical interactions possibly helps in speeding up the overall cosimulation, which may effectively lead to a smooth continuous motion of the truck in Unity without any collisions (which is currently not the case).

7. Discussion

In this section, we discuss the learnings from the verification of runtime behavior of DTs, based on the executed case study. SMC helped in verifying timeliness and functional correctness of VE. From the results in Section 6, one clearly understands that overall runtime behavior cannot be concluded by simply observing a parameter and extrapolating or predicting the related behavior based on this. Moreover, the results from what-if analysis helped us understand how a change in one part of the system affects the runtime behavior, both locally and globally. Furthermore, we found that SMC can be used for exploring design alternatives for DTs by iterative modeling and verification. In addition, SMC could also be helpful in exploring alternate communication channels (based on required communication latency) for interaction between the components in a DT. In our DT case study, we observe that the motion of the truck in the Unity simulation environment is not seamless and it even crashes before reaching its destination dock (in several cases). We found that the actual reason why the truck in the virtual simulation environment is not having a seamless motion is related only to the runtime behavior of the VE. On verifying the timeliness of the VE, we understand that certain critical timeliness properties for effective control of the motion of the truck do not hold and this is the reason for the truck crashing in the virtual environment. More generally, in our case study, we found that the use of SMC for runtime behavior analysis of the VE provides a fair amount of clarity on how the non-seamless motion of the truck in Unity is connected to aspects of the runtime behavior of VE. Furthermore, verification of functional and timeliness properties using SMC by performing what-if analysis, helps in improvement of the design of a DT along its lifecycle, e.g., in terms of adopting alternate communication protocols, improving processor capabilities, setting up the schedule for cosimulation by using an orchestrator and others.

8. Conclusion & Future work

As mentioned before, to our knowledge, there currently exists no literature which explicitly focuses on verification of runtime behavior of DTs. The possibility of issues arising during the execution of VE and its nature to evolve continuously across its lifecycle, however, creates a need for such verification at runtime. In this paper, we propose to use statistical model checking for verification of runtime behavior of DTs. We verified the runtime behavior of the VE against several properties of interest and performed what-if exploration based on changes in timings pertaining to specific components in the VE. Owing to lack of documentation and requirements for the DT, we verified the VE against a list of properties we created, which ensures some level of functional correctness and timeliness of VE. This provided us with several important insights in the actual behaviour of the VE, which helped in understanding the crashing of the truck in the virtual simulation environment. The extensive logging and required instrumentation of DT components for this technique adds complexity. In general, building verifiable models is expensive and researchers in V&V also aim at automatic model extraction and others. This could possibly be done by combining SMC with techniques such as active and passive learning, which could help in automating the learning of behavior of the DTs. While we had applied this technique on a simple DT case study, there could be a possibility that practical limitations such as computational overhead and others could affect its scalability. Moreover, it can be comprehended that logs generated during execution only cover the actual "observed run" of the system and may not contain the "unobserved run" of the system at times. This deficiency could possibly be overcome by performing what-if explorations to explore boundary conditions and others, i.e., by making changes in the delay values in the various interactions involving the different components in VE, as we had performed. Furthermore, we found that performing what-if analysis using SMC can also be used for exploring design alternatives for a DT through iterative modeling and verification. In addition, we are planning to explore other verification techniques, which could be used for schedulability analysis of DTs. Moreover, what-if exploration based on addition or removal of other components in a DT such as including PE and its synchronization with VE, other DTs which could interact with this DT to form a federated DT and others is planned for future work.

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/e, and for helping to conduct our research in collaboration with the Automotive Engineering Science (AES) lab at TU/e. We would also like to thank professor Marius Mikučionis from the department of computer science at Aalborg University for his assistance with UPPAAL SMC.

References

[1] B. R. Haverkort, A. Zimmermann, Smart industry: How ICT will change the game!, IEEE internet computing 21 (2017) 8–10.

[2] T. Gabor, L. Belzner, M. Kiermeier, M. T. Beck, A. Neitz, A simulation-based architecture for smart cyber-physical systems, in: 2016 IEEE international conference on autonomic computing (ICAC), IEEE, 2016, pp. 374–379.

[3] A. Canedo, Industrial iot lifecycle via digital twins, in: Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2016, pp. 1–1.

[4] S. Liu, J. Bao, Y. Lu, J. Li, S. Lu, X. Sun, Digital twin modeling method based on biomimicry for machining aerospace components, Journal of manufacturing systems 58 (2021) 180–195.

[5] W. Kritzinger, M. Karner, G. Traar, J. Henjes, W. Sihn, Digital twin in manufacturing: A categorical literature review and classification, IFAC-PapersOnLine 51 (2018) 1016–1022.

[6] F. Tao, M. Zhang, Digital twin shop-floor: a new shop-floor paradigm towards smart manufacturing, Ieee Access 5 (2017) 20418–20427.

[7] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, M. Hendriks, Uppaal 4.0 (2006).

[8] F. Tao, H. Zhang, C. Zhang, Advancements and challenges of digital twins in industry, Nature Computational Science 4 (2024) 169–177.

[9] M. Van Den Brand, L. Cleophas, R. Gunasekaran, B. Haverkort, D. A. M. Negrin, H. M. Muctadir, 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), IEEE, 2021, pp. 225–228.

[10] F. Bordeleau, B. Combemale, R. Eramo, M. v. d. Brand, M. Wimmer, Towards model-driven digital twin engineering: Current opportunities and future challenges, in: International Conference on Systems Modelling and Management, Springer, 2020, pp. 43–54.

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

[12] M. Grieves, J. Vickers, Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems, in: Transdisciplinary perspectives on complex systems, Springer, 2017, pp. 85–113.

[13] M. Dalibor, N. Jansen, B. Rumpe, D. Schmalzing, L. Wachtmeister, M. Wimmer, A. Wortmann, A cross-domain systematic mapping study on software engineering for digital twins, Journal of Systems and Software (2022) 111361.

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

[15] D. Basile, A. Fantechi, L. Rucher, G. Mandò, Analysing an autonomous tramway positioning system with the uppaal statistical model checker, Formal Aspects of Computing 33 (2021).

[16] A. Legay, M. Viswanathan, Statistical model checking: challenges and perspectives, International Journal on Software Tools for Technology Transfer 17 (2015) 369–376.

[17] A. David, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, Uppaal smc tutorial, International journal on software tools for technology transfer 17 (2015) 397–415.

[18] G. Agha, K. Palmskog, A survey of statistical model checking, ACM Transactions on Modeling and Computer Simulation (TOMACS) 28 (2018) 1–39.

[19] I. Barosan, A. A. Basmenj, S. G. Chouhan, D. Manrique, Development of a virtual simulation environment and a digital twin of an autonomous driving truck for a distribution center, in: European Conference on Software Architecture, Springer, 2020, pp. 542–557.

[20] A. Dokhanchi, B. Hoxha, G. Fainekos, Metric interval temporal logic specification elicitation and debugging, in: 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), IEEE, 2015, pp. 70–79.

[21] Weblink to access the list of queries used for smc, 2024-10-06. URL: https://surfdrive.surf.nl/files/index.php/s/fhFHmat6olP7IbI, (SMC queries in MITL).