Using UML and OCL Models to Realize High-Level Digital Twins

Paula Muñoz, Javier Troya, and Antonio Vallecillo
ITIS Software. Universidad de Málaga, Spain
{paulam,jtroya,av}@uma.es

***Abstract*—Digital twins constitute virtual representations of physically existing systems. However, their inherent complexity makes them difficult to develop and prove correct. In this paper we explore the use of UML and OCL, complemented with an executable language, SOIL, to build and test digital twins at a high level of abstraction. We also show how to realize the bidirectional connection between the UML models of the digital twin in the USE tool with the physical twin, using an architectural framework centered on a data lake. We have built a prototype of the framework to demonstrate our ideas, and validated it by developing a digital twin of a Lego Mindstorms car. The results allow us to show some interesting advantages of using high-level UML models to specify virtual twins, such as simulation, property checking and some other types of tests.**

***Index Terms*—Model-based Software Engineering, Model-based Testing, Digital Twins, UML, OCL, USE.**

I. Introduction

In general, a Digital Twin (DT) is a digital replica of an object, process or service that exists in the physical world along with a bi-directional connection between the physical entity and its virtual representation. The ideas behind Digital Twins go back to the NASA/Apollo project, in which an identical space capsule on Earth was used to simulate the behavior of the capsule in space. Although both objects were physical entities, the evolution of digital technologies soon enabled the development of full-fledged digital replicas of the physical objects to be simulated, thus opening up new possibilities beyond simulation, such as monitoring or what-if analysis of the physical systems [1]. The advances in AI and Big Data allowed adding further features to DT, such as behavior prediction, preventive maintenance, or optimizations [2].

Engineering DT systems, and in particular testing them [3], is challenging for many reasons, one of them being their complexity: the digital replica has to faithfully emulate its physical counterpart. One way to address this challenge is by raising the level of abstraction, and this is where software models, and in particular model-based software engineering, can be of real help [4], [5]. Some authors have already claimed the potential benefits of using high-level models to specify and design the virtual replicas (i.e., the digital twins). These *lightweight* models reflect the simplified structure, physics and behavior of the physical twin "to reduce computational load especially in upfront engineering activities; they allow simulations of complex systems with fidelity in the appropriate dimensions to answer questions with minimal computational costs" [6]. The concept of *multi-fidelity* digital twins has also been introduced [7], describing the advantages of modeling the physical entity at different levels of abstraction, in this case, for testing purposes.

Still, the problem of integrating these high-level models into a DT system and connecting them to the physical entity is not easy. Some solutions propose dedicated component-based architectural frameworks that implement the basic infrastructure, and define domain-specific languages to describe the models [8]–[12], with MontiArc [13] being the most notable architectural language in this context.

In this paper, we explore the use of standard UML models, enriched with OCL constraints, for the specification of digital twins. We shall rely on USE [14] as modeling tool, since it provides a wide variety of analysis capabilities for UML models, including model validation, instance generation, or invariant checking [15]. The behavioral aspects of the systems will be specified in SOIL [16], an executable language available in USE. Incorporating these types of analyses into the engineering of DT systems will allow engineers to exploit the benefits of early analysis and testing of their DT, at a high level of abstraction, and with reasonable computational costs.

The problem we address in this paper is how to connect the high-level UML models of a digital twin with the actual system, i.e., its physical twin. Another goal is to achieve this in a modular way, so that, once validated, these lightweight models could be replaced by more detailed models of the system that provide finer-grained specifications.

We have defined a framework for the specification and deployment of DT systems, that uses UML models to specify the digital twins, and connects them through a *Data Lake* repository [17] implemented in Redis [18] that provides the bi-directional communication infrastructure between the digital and physical twins. We have built a prototype to demonstrate our ideas and validated it by developing a digital twin system of a Lego Mindstorms car.

The structure of this document is as follows. After this introduction, Sect. II briefly describes the background of our work and presents the example used to motivate our proposal. Next, Sect. III describes our proposed architecture for specifying and deploying digital twins, and Sect. IV illustrates how to use it applying it to the example case study. Finally, Sect. V relates our work to other similar approaches and Sect. VI concludes with an outline of future work.

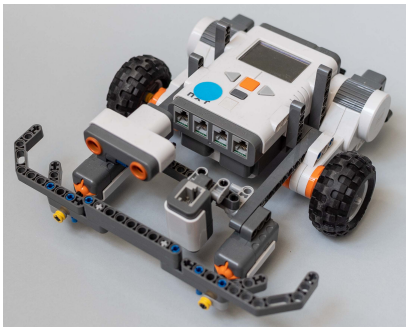

Figure 1. The Lego car used as physical twin.

II. Background

A. Digital Twins

A *Digital Twin* (DT) is a comprehensive digital representation of an actual system, service or product (the *Physical Twin*, PT), synchronized at a specified frequency and fidelity [19]. The digital twin includes the properties, condition and behavior of the physical entity through models and data, and is continuously updated with real-time system data about the PT performance, maintenance, and health status throughout its entire lifetime [4], [6], [20]. The exchange of data between the digital and the physical twins takes place through bi-directional data *connections*. This is why many authors have argued that a *DT system* contains three dimensions: physical, digital, and the connections between them [1].

Additionally, other authors consider that a DT system may also comprise a set of *Services* that permit exploiting the data exchanged by the two twins in different ways [21], [22]. Examples of such services include, among others: Dashboards for visualizing and displaying the data in different formats; Machine Learning (ML) components to provide decision support and alerts to users, or to predict changes in the PT over time, e.g., to achieve preventive maintenance; or algorithms to analyze the available data to improve the PT performance or implement self-adaptive mechanisms.

This is the architecture that we shall adopt and use in our proposal, as described later in Sect. III, see Fig. 3.

B. Motivating example: a Lego Mindstorms vehicle

To demonstrate our proposal and to initially validate it, as physical entity we will use a Lego Mindstorms NXT car, which is a small car-like vehicle with sensors, see Fig. 1. The car is able to move, detect obstacles, and interact with its environment in different ways, e.g., by following a colored line on the floor. It uses a bluetooth connection to exchange data and commands with a computer, including the information about the current state of the car (position, speed) and the readings of its sensors.

In addition to a pose-provider, which determines its planar coordinates and the angle of its current direction, the car has an engine to move forward and rotate its direction at a certain angle. It also has three types of sensors: an ultrasonic sensor capable of detecting the distance to the object in front of it, a light sensor that can establish the color of the ground beneath it, and two touch sensors used as a bumper to determine whether the car has collided or not.

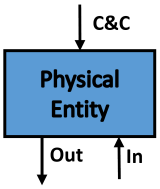

Figure 2. A simplified view of a physical entity.

The car downloads from the computer the software that drives the vehicle, which can define different behaviors. For example, in the LineFollowingBehavior, the car keeps moving forward as long as it detects a black line beneath it. If the color detected is not black, it means that the car is no longer above the line it was supposed to follow. Then, the car stops and starts turning left and right, increasing the angle each time, until the color is detected again. Once the line is found, the car continues moving. Further behaviors can be defined and incorporated to the car, such as turning a number of degrees when an obstacle hits the bumper, or when the distance to the object in front is below a given threshold, for example. Note that this software is integrated in the car, i.e., it is part of the physical twin.

A digital twin can be used to improve the existing physical entity (in this case, the car) in several ways, for instance:

- *Simulation analysis.* Mindstorms vehicles can execute user-defined software to perform specific tasks. The DT could specify the intended behavior of such applications and simulate their behavior. A testing component could compare the traces of both twins and check that the they both behave as expected [5].
- *Calibration.* Simulations would help deciding the best values for the configuration parameters of the car software to save energy or to perform its tasks faster.
- *Improve the system behavior.* The DT can learn from the obstacles encountered by the car and build a floormap of the area around which the car is moving. Such a map can be used to avoid future collisions. Note that the car simply knows how to obey orders or to follow a colored line, without any other intelligence. A DT would allow the implementation of additional self-adaptive features.

III. A framework for defining and deploying DTs

To realize our proposal, we have defined a reference architecture for the specification and deployment of DTs. In our work, we define the DT using high-level UML models, but this architecture can be applied to the final systems by replacing the UML models by lower-level implementations. Before describing its main components, let's start with how physical entities are represented in it. Figure 2 shows the relevant elements of a physical entity for our purposes.

In general, a physical entity interacts with its environment through a set of *inputs* (In) and *outputs* (Out). The values of

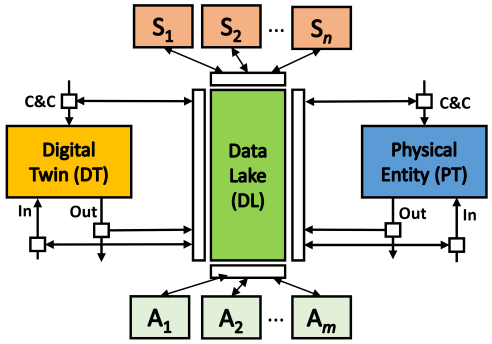

Figure 3. Our proposed Digital Twin System Framework.

inputs are normally generated by sensors that detect events or changes in its environment. For example, the distance to an obstacle, the presence of a particular color on the ground, or the fact that the car bumper has detected a collision. Outputs constitute the reactions of the physical entity to input events from its environment, to commands issued to it or to internal changes. Examples include changes in the speed of the car or its direction. They are usually captured by other sensors, which also allow knowing the state of the physical entity, e.g. the battery level or the exact position of the car.

Physical entities can also accept external orders in terms of *commands* that control their behavior (such as move or rotate) or changes in the values of their configuration parameters (e.g., the speed of the car at cruising mode). This is represented in Fig. 2 by the *Command and Control* (C&C) incoming arrow.

Figure 3 shows the architecture that we propose for specifying and deploying digital twins. It is inspired in the architecture proposed by several authors for realizing DTs, composed of the physical part, the digital part, the connection between them, and a set of services [21], [22]. Here, the physical entity that was depicted in Fig. 2 is shown to the right, acting as the physical twin (PT). The digital twin (DT) is a replica of the physical entity, i.e., its specular image.

The communication between them is achieved through a *Data Lake* (DL). As defined in [17], a data lake is "a flexible, scalable data storage and management system, which ingests and stores raw data from heterogeneous sources in their original format, and provides query processing and data analytics in an on-the-fly manner."

The rest of the components of the architecture use the DL to write data and obtain information, in a loosely-coupled and asynchronous manner. Basically, this implements a Blackboard architectural pattern [23] but with the property that the information is stored in raw format (i.e., as produced by the sources). The different components access the DL by means of *drivers* that transform the data into the formats that each component understands. The drivers are represented in Fig. 3 by the white rectangles surrounding the DL.

The white square boxes attached to the physical entity connections In, Out, and C&C, are able to intercept the information that flows through them and record it in the DL. Similarly, the drivers can query the DL for commands or inputs to the physical entity that have been stored by other components, and send them to the physical entity. This allows, e.g., emulating the environment of the physical entity. They can also be used to change parameters during run-time or implement self-adaptive behaviors. From the perspective of the DT, the drivers can be used, e.g., to emulate the inputs and commands received by the physical entity and to record its outputs in the DL.

The *Service* components (S_1,S_2,..., S_n) are in charge of implementing the additional functionality that a digital twin system can provide. Examples of such components include dashboards that dynamically visualize the data, or algorithms that learn from the past movements of the car and its collisions, draw floormaps, and avoid the obstacles. Note that in the Blackboard architectural pattern, any (permitted) component can write in the DL. This allows the services to issue commands to the physical entity, as if they were its external users.

Finally, given that our emphasis is on testing digital twins, the *Analysis* components (A_1,A_2,..., A_n) are in charge of implementing different types of tests on the physical entity, the digital twin, their connection, behaviors, or synchronization. For example, a monitoring component can check that the traces produced by both the DT and the PT are equivalent [5]. Another analysis component can exercise only one of the twins, sending commands to it and checking that the responses are as expected. The analysis components can also serve to test the Service components, validating that the algorithms they implement to provide self-adaptive behaviors or machine learning predictions do work.

Note as well that the framework's architecture allows having more than one digital twin, each one focusing on specific aspects of interest of the physical entity. Likewise, more than one physical twin can be integrated into the architecture. Service or analysis components can also be dynamically added to the system during its execution, given that all communications between the components happen through the DL.

IV. Implementation

This section describes how we have implemented the main components of our reference architecture, and how they are connected between them. As mentioned above, all connections are made through the DL. The description of all framework components is illustrated for the case of the Lego car.

A. High-level UML Digital Twins

In this work we are interested in the specification of digital twins using UML and OCL models. This allows high-level specification of relevant characteristics of the physical entity, providing *lightweight* models that represent its simplified structure and behavior. These models can be produced during the early engineering stages, with lower development costs than the full implementation of the digital replica, and allow simulations and different types of analysis to be performed with considerably reduced computational costs.

For example, in the case of the Lego car, Fig. 4 shows a UML model with its main components: the engine (Motor) to move forward and to rotate its direction a certain angle, and

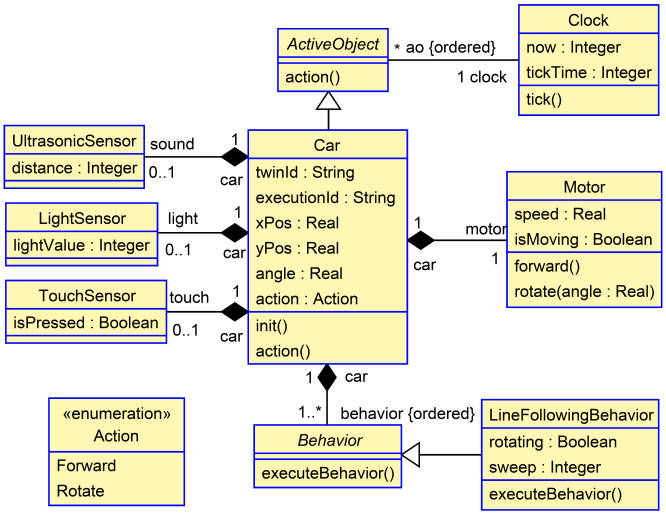

Figure 4. A UML model of the Lego car.

the three sensors mentioned in Sect. II-B. They were able to detect the distance to the object in front of the car, the color of the ground beneath it (using a light sensor) and the state of the bumper (pressed, not pressed).

In addition to the car planar coordinates and the angle of its current direction, the UML model of the car depicted in Fig. 4 shows other elements used in the simulations, such as the car identifier (twinId, to allow different cars in a simulation), the execution identifier (executionId, to distinguish between simulations), as well as a Clock object that is used to simulate the passage of time. With each tick of the clock, the action() method of all active objects is invoked. In case more than one behavior is defined for the car, it will execute them in order, using the executeBehavior() method.

To specify the behavior of the methods in the model we have used SOIL [16], a textual executable language defined for the USE modeling environment [14]. For example, the following piece of code shows the specification of the behavior of class Motor methods forward() and rotate() in SOIL.

```
forward()
  begin
    self.car.xPos := self.car.xPos + self.speed *
      self.car.clock.tickTime * self.car.angle.cos();
    self.car.yPos := self.car.yPos + self.speed *
      self.car.clock.tickTime * self.car.angle.sin();
  end
rotate(angle:Real)
  begin
    self.car.angle := angle;
  end
```

In some of our previous works we have extensively employed USE to analyze different types of physical systems using their UML and OCL specifications [24]–[27]. However, these specifications were defined and simulated independently and completely detached from the system they specified. In a Digital Twin context, our goal is to simulate them synchronized with the physical entity they represent, establishing a bidirectional connection between the UML models and the

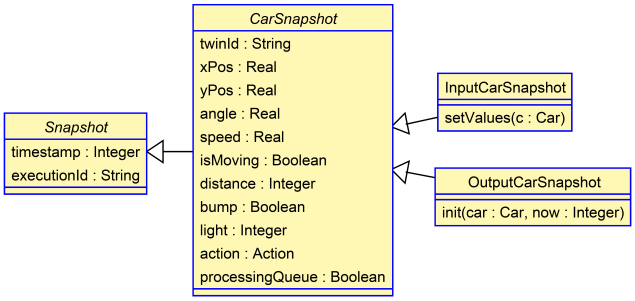

Figure 5. A UML model of the information exchanged with the system.

physical twin, allowing them to exchange data and control. This is precisely the objective of this work.

In our running example, the model of the information used by the digital twin to communicate with the physical twin is shown in Fig. 5. Instances of class InputCarSnapshot represent the data that is produced by the physical twin (i.e., the car). They contain snapshots of the state of the car at a given moment in time (timestamp, expressed in POSIX format [28]), namely its position, angle, and the values of the sensors' readings.

In general, the behavior of a digital twin depends on the aspects of the physical twin it is intended to emulate. In this example, we will show a behavior whereby the digital twin simulates the way the physical car reacts each time the sensors produce their readings. Thus, for every InputCarSnapshot object in the model, which describes the state of the physical car, the virtual twin will generate an OutputCarSnapshot object that represents the expected state of the physical car after a time elapse of "tickTime" time units (see class Clock).

This is shown in figures 6 and 7, respectively. The first one shows an object model with three input snapshots, each one at different moments in time, while Fig. 7 shows the corresponding output snapshots produced by the digital twin. This shows an example of the simulation of a reactive behavior.

The question now is how the input snapshots are created in the UML object model, and how the output objects produced by the model are stored in an external data-store for later analyses. This is precisely the goal of the next subsections.

B. The Data Lake (DL)

To implement the Data Lake repository that provides the bidirectional communication infrastructure between the digital and physical twins, we chose Redis [18]. This open-source in-memory data structure can be used as a database, cache, and message broker. It is also lightweight and optimized to deliver fast responses to a massive amount of petitions, which is one of the reasons why we chose this database since we want to ensure scalability for future versions of the tool.

Redis is a key-value database and supports various abstract data structures such as strings, lists, maps, or sets. We chose maps as the data structure to store the input and output snapshots as lists of key-value pairs. Maps are called Redis Hashes, and they have a primary key that identifies them. Inside the hash, each of the fields has its own key.

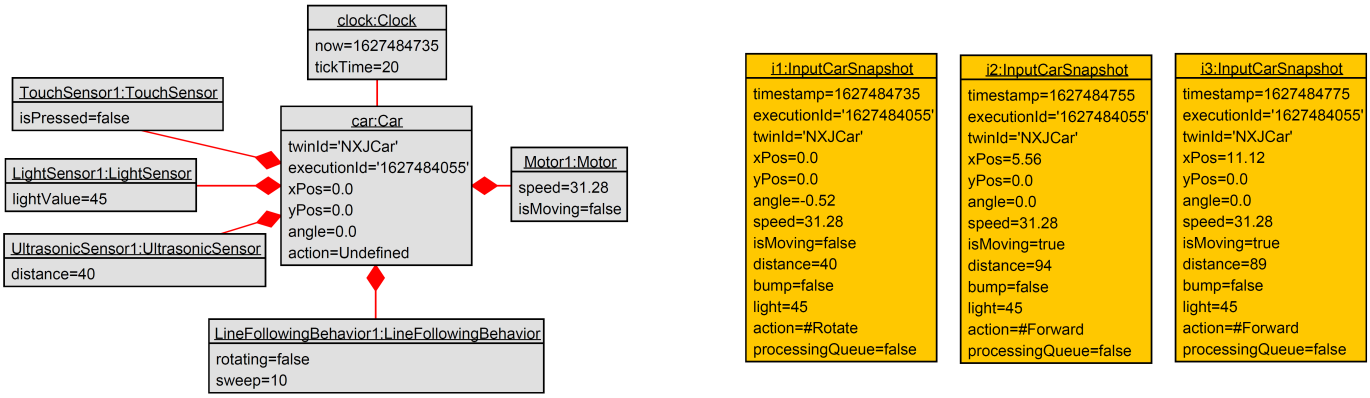

Figure 6. A UML object model of the Lego car with three input snapshots.

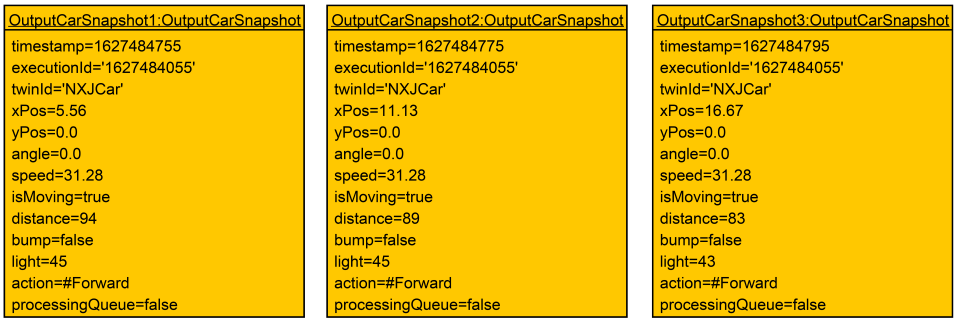

Figure 7. The three output snapshots produced by the digital twin.

The following code fragment shows a record with information produced directly by the car, corresponding to an InputCarSnapshot instance (output snapshots are similarly stored). In the fragment, we can see how records are listed as sequences of strings, alternating the key of each attribute and its value. The displayed values correspond to the sensors' readings (distance, isMoving, bump, light), the car state (position, heading angle, action), and some auxiliary values to ease the later analysis process (twinId, executionId, timestamp, processingQueue).

```
HGETALL NXJCar:1627484055:1627484375
 1) "twinId"              2) "NXJCar"
 3) "bump"                4) "0"
 5) "processingQueue"     6) "0"
 7) "light"               8) "45"
 9) "angle"               10) "-0,52"
11) "executionId"         12) "1627484055"
13) "yPos"                14) "-0"
15) "speedFactor"         16) "31,28"
17) "isMoving"            18) "0"
19) "xPos"                20) "0"
21) "action"              22) "Rotate"
23) "distance"            24) "40"
25) "timestamp"           26) "1627484375"
```

The command to query a hash is HGETALL, followed by the key of the record we want to retrieve. As shown in the code fragment, we have created a key composed of three elements separated by semicolons. The first element is the identifier of the twin, in our case, the unique name we gave to the car. The second element is the executionId, which matches the timestamp of the first snapshot produced during the running execution. Finally, the third element is the timestamp of the current snapshot.

Since Redis is a simple, lightweight, NoSQL database, it does not allow complex queries. Therefore, to retrieve hashes by the values of their fields, it is necessary to store additional records that include the field's value and a reference to the hash key. For instance, the following fragment shows how we add a record to a list of executionIds to be able to filter every snapshot of a given execution.

```
ZADD executionId_list 1627484055
                      NXJCar:1627484055:1627484375
```

Following this approach, we store lists of any numerical value that we want to use for filtering. For example, the processingQueue attribute is used to determine if the DT has already processed the record or not, in order to identify the records that still need to be processed.

C. The Physical Twin

A connection between the PT and the data lake is required to store the raw information produced by the physical entity. The connection drivers are implemented using Java since both the Lego Mindstorms libraries and the USE API employ this language. The Redis client we use is Jedis [29], which is an open-source, small, and easy-to-use client that provides all the functionalities that we require to interact with the DL, including a publish-subscribe service.

Using the Jedis client, we take the raw data received through the car's Bluetooth connection and store it in the DL. However,

the Lego Mindstorms car is unable to execute external libraries such as Jedis. For this reason, we had to connect the process controlling and reporting the information from the car to another driver. This driver is in charge of processing and storing the raw information from the car in the DL. The connection between these two drivers uses a socket in order to ensure driver compatibility and reusability when the car is changed by any other device

D. Connecting the UML models to the Data Lake

Once we have a model of the physical entity that serves as a digital twin for it, the question is how the model can be integrated into the architecture described in Sect. III (Fig. 3).

To enable USE to interact with the DL, we have developed a plugin using its API, which allows creating model elements, i.e., classes, associations, and instances. Our plugin establishes two communication channels: from the DL to the DT and vice-versa. The first one sends queries about the raw data produced by the PT and creates an InputCarSnapshot instance for each unprocessed record found. The second one detects OutputCarSnapshot instances in the object model and processes them, creating a new record in the DL for each one, and deleting the corresponding snapshots once processed. This way, any of the Services or Analysis components can query the input and output snapshots and perform analysis without interacting with either the digital or the physical twin directly, i.e., using a decoupled approach through the DL.

Since both USE and the DL cannot send notifications once new data has arrived, the plugin also implements a publish-subscribe service. The structure for each direction is implemented as follows:

- *From USE to the DL*: There is a process listening to the object model that checks periodically if there are new OutputCarSnapshot instances. Once it finds one, it sends a message through the output channel that notifies all subscribed processes, which retrieve the new snapshots and store them in the DL.
- *From DL to USE:* A process is always listening to the DL, checking periodically for new records produced by the physical twin. Once a new record is found, the subscribed processes are notified through the input channel. The process retrieves the new records, processes them, and creates new InputCarSnapshot instances in the USE model. These instances will then be handled by the digital twin according to its specified behavior.

This plugin and the rest of the system implementations are all available on our GitHub repository [30].

E. Service and Analysis components

Essential parts of any DT system are also the *Service* and *Analysis* components, which are used to implement any smart behavior of the system, or to check that the twin system works as expected. Due to the loosely-coupled architecture of our framework, these components could be independently developed using any technology of choice, and then plugged into the framework via the data lake.

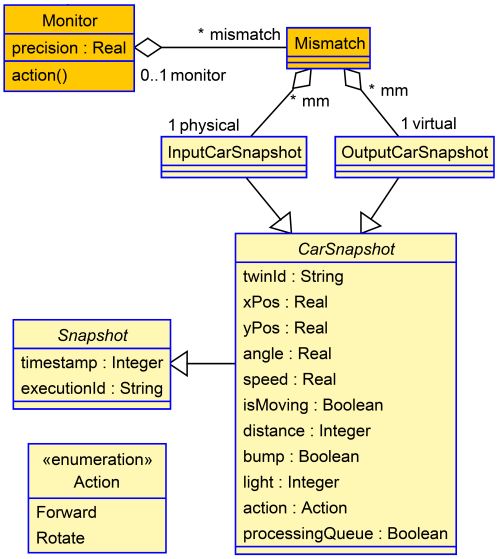

Figure 8. The UML model of the Monitor analysis component.

In this paper we will show how several of these components, namely some analysis components, can be realized using high-level UML and OCL models too. Working at this level of abstraction introduces interesting benefits, mainly reducing the accidental complexity of using lower-level programming languages, and also achieving platform-independence.

For example, let us illustrate here the specification of one analysis component that checks that the behavior of the two twins is the same, by monitoring the traces produced by both twins and checking that they are equivalent [5].

The model of such Monitor component is shown in Fig. 8. Class Monitor is in charge of comparing the snapshots produced by both the digital and the physical car. Each time a mismatch is detected between two snapshots, they are recorded in a list of pairs, whose elements are of type Mismatch.

The behavior of the action() method simply has to check that the input and output snapshots of the same execution of a car with the same timestamps, have "similar" attribute values. Given that we are dealing with physical entities, it would be unrealistic to ask the values to be identical. Imprecision of measurements, tolerances and deviations cannot be neglected. This is why class Monitor provides an attribute, precision, that represents the allowed precision in the comparisons.

With this, the behavior of the Monitor can be specified in SOIL as the following code fragment shows. First, it selects the set of input and output snapshots that are not part of a mismatch already (lines 7–9). Then, a loop iterates over the set of input snapshots (coming from the physical car) and tries to select an output snapshot (produced by the virtual car) that corresponds to the same car execution, with the same timestamp (lines 11–14). In case such an output snapshot is found (line 15), it is compared against the input snapshot (line 16). Should they differ, a new mismatch is created and both snapshots are linked to it (lines 17–20). Otherwise, the snapshots are destroyed (lines 22–23).

```
Monitor::action()
  begin
    declare iSnap: Set(InputCarSnapshot),
            oSnap: Set(OutputCarSnapshot),
            os:OutputCarSnapshot, mm:Mismatch;
    iSnap := InputCarSnapshot.allInstances->
               select(i|i.mm->isEmpty());
    oSnap := OutputCarSnapshot.allInstances->
               select(o|o.mm->isEmpty());
    for is in iSnap do
      os:=oSnap->select(o|
          (o.twinId = is.twinId) and
          (o.executionId = is.executionId) and
          (o.timestamp=is.timestamp))->any(true);
      if os<>null then
        if not self.compare(is,os) then
          mm:=new Mismatch();
          insert(self,mm) into Mismatches;
          insert(mm,is) into SourceMismatch;
          insert(mm,os) into TargetMismatch;
        else
          destroy(is);
          destroy(os);
        end;
      end;
    end;
  end
```

The following method, compare(), is in charge of checking whether an input and output snapshots are "similar" or not (modulo the value of the precision attribute of class Monitor).

```
compare(i:InputCarSnapshot,o:OutputCarSnapshot): Boolean =
  (i.xPos-o.xPos).abs() <= self.precision and
  (i.yPos-o.yPos).abs() <= self.precision and
  (i.angle-o.angle).abs() <= self.precision
```

Figure 9 shows an object model that contains one mismatch. It is due to a deviation between the expected position of the PT, as computed by the DT, and its actual position. The car seems to move slower than expected for some unknown reason.

The connection between this model and the rest of the system is again via the data lake. The driver associated to this component gets the input snapshots from the DL, and creates the corresponding snapshot objects. When mismatch instances are created in the model, the driver is notified and moves these instances (and their corresponding references) to the associated snapshots to the DL, and then deletes them from the model.

Some other interesting examples of analysis components are those defined to test the digital twin, by issuing commands to it and checking that the outputs are correct. Further analysis include the verification of temporal or safety properties of the system, such as checking that certain states are reached, or in case of a system with more than one Lego car, ensuring that they maintain a certain distance. Note the benefits of using a high-level representation of the models, which allow a simpler specification of these kinds of properties. Furthermore, we can make use of all the USE toolkit for the analysis and validation of UML models, including test case generation, invariant checking, or even the analysis of emergent behaviors of the system [15], [31].

V. Related Work

Our contribution can be related to several kinds of works. First, we have those that show how USE can be employed for runtime monitoring of Java applications [32], [33]. We have

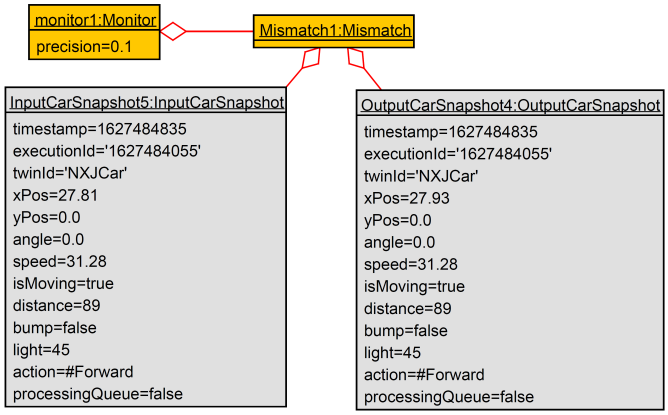

Figure 9. Object model showing one mismatch, as detected by the monitor.

used these ideas and the API that USE provides to connect the USE UML models with our data lake.

We have previously employed USE for simulating the behavior of different types of physical systems specified in UML [24]–[27]. These simulations have proved to be very useful for the specification of digital twins in our present work. However, they were not synchronized with the physical systems they emulated or specified.

Other authors have also proposed different architectures for DT systems, with different number of components types (also called *dimensions* in some papers). The initial architectures defined three main components: the digital twin, the physical twin and the connections between them [1], [3]. Further proposals define five [21], [22], or even more dimensions [34].

The potential usages and advantages of MBSE for leveraging DT technologies are discussed in [4], [6] in general terms. Concrete proposals include, e.g., [9] that introduces a reference framework for leveraging model repositories for digital twins, connecting models at design and runtime. The closest works to ours are [8], [11], [12], [35], which propose variations of a model-based reference framework for developing digital twins of manufacturing machines (e.g., injection molding machines), a hot rolling machine on a steel production chain, or a fire extinguisher system. These proposals use a domain-specific language [13] to specify in the digital twin the events that occur in its physical counterpart. Our proposal differs from these works in three main aspects. First, our framework is defined to be independent from the system to be modeled. Second, it is architected in a loosely-coupled manner. Finally, we have shown how the digital twin can be effectively specified in UML within the context of the USE tool, and how the UML models can be connected to the physical twin.

Despite its relevance [3], up to our knowledge testing and validating digital twins has not received much attention, with just a few works that explicitly consider it. For example, [5] proposes a model-based testing approach for validating behavioral equivalence between the two twins. Further types of specific tests for digital twins are described in generic terms in [3]. Finally, [7] introduces the idea of defining digital twins

of the same physical twin at different levels of abstraction, in order to obtain the most cost-effective tests depending on the property we want to validate. However, no concrete proposal is given in that paper. This is precisely where our work provides its main contribution, showing how high-level UML models could be perfectly connected to a DT reference framework, and be used not only to specify digital twins but also to define tests for them at the appropriate level of abstraction.

VI. Conclusions

In this paper, we have described a framework for engineering digital twins systems using a loosely-coupled and modular architecture. This framework is suitable not only for high-level models but also for lower-level implementations, since the models could be later replaced by their corresponding implementations. With our approach, the UML model of the digital twin can be specified at the right level of abstraction, depending on the specific type of analysis required. This also allows any part of the software, such as the behavioral logic or performance of a specific element, to be independently analyzed and validated.

We also described a realization of this framework to demonstrate our proposal using a Lego Mindstorms NXT car. We include the complete communication system between the twins and the data lake and an analysis module to test the behavior logic of the physical twin against its digital counterpart.

Of course, we are aware that full-fledged digital twins require much more complex implementations than those described here. However, our claim is that digital twins are software artifacts and, as such, require appropriate software engineering methods and practices, including rigorous specification, validation and verification processes. Our contribution has shown how it is possible to use high-level UML and OCL models for the specification of digital twins to verify their expected behavior in the early stages of their construction, and with very low computational costs and development efforts.

There are several lines of work that we plan to address in the near future. First, we want to further validate this proposal through a more detailed analysis of other physical systems to assess its applicability. Second, we plan to create other analysis and services modules for monitoring and testing different properties in the system. For example, we want to analyze systems with more than one digital twin to evaluate synchronization with our proposal. Third, we intend to evaluate the framework's performance under stressful conditions to determine its scalability and applicability to larger systems, as well as the effects of the synchronization latency in the overall behavior and fidelity of the digital twin system. Finally, we plan to improve the proposed implementation to configure any connection and any physical twin easily.

Acknowledgment

We would like to thank the reviewers for their insightful comments and constructive suggestions, which helped us to improve the paper. This work was partially funded by Research Projects PGC2018-094905-B-100 and P20-00067-FR.

References

[1] M. Grieves, "Digital twin: Manufacturing excellence through virtual factory replication," Florida Institute of Technology, White Paper, 2014.

[2] M. M. Rathore, S. A. Shah, D. Shukla, E. Bentafat, and S. Bakiras, "The Role of AI, Machine Learning, and Big Data in Digital Twinning: A Systematic Literature Review, Challenges, and Opportunities," *IEEE Access*, vol. 9, pp. 32 030–32 052, 2021.

[3] M. Grieves and J. Vickers, *Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems*. Springer, 2017, pp. 85–113.

[4] F. Bordeleau, B. Combemale, R. Eramo, M. van den Brand, and M. Wimmer, "Towards model-driven digital twin engineering: Current opportunities and future challenges," in *Proc. of ICSMM'20*, ser. CCIS, vol. 1262. Springer, 2020, pp. 43–54.

[5] A. Khan, M. Dahl, P. Falkman, and M. Fabian, "Digital twin for legacy systems: Simulation model testing and validation," in *Proc. of CASE'18*. IEEE, 2018, pp. 421–426.

[6] A. M. Madni, C. C. Madni, and S. D. Lucero, "Leveraging digital twin technology in model-based systems engineering," *Systems*, vol. 7, no. 1, p. 7, 2019.

[7] A. Arrieta, "Multi-fidelity digital twins: a means for better cyber-physical systems testing?" *CoRR*, vol. abs/2101.05697, 2021.

[8] J. C. Kirchhof, J. Michael, B. Rumpe, S. Varga, and A. Wortmann, "Model-driven digital twin construction: synthesizing the integration of cyber-physical systems with their information systems," in *Proc. of MoDELS'20*. ACM, 2020, pp. 90–101.

[9] D. Lehner, S. Wolny, A. Mazak-Huemer, and M. Wimmer, "Towards a reference architecture for leveraging model repositories for digital twins," in *Proc. of ETFA'20*. IEEE, 2020, pp. 1077–1080.

[10] T. Bolender, G. Bürvenich, M. Dalibor, B. Rumpe, and A. Wortmann, "Self-adaptive manufacturing with digital twins," *CoRR*, vol. abs/2103.11941, 2021.

[11] M. Dalibor, J. Michael, B. Rumpe, S. Varga, and A. Wortmann, "Towards a model-driven architecture for interactive digital twin cockpits," in *Proc. of ER'20*, ser. LNCS, vol. 12400. Springer, 2020, pp. 377–387.

[12] P. Bibow, M. Dalibor, C. Hopmann, B. Mainz, B. Rumpe, D. Schmalzing, M. Schmitz, and A. Wortmann, "Model-driven development of a digital twin for injection molding," in *Proc. of CAiSE'20*, ser. LNCS, vol. 12127. Springer, 2020, pp. 85–100.

[13] A. Haber, J. O. Ringert, and B. Rumpe, "Montiarc - architectural modeling of interactive distributed and cyber-physical systems," *CoRR*, vol. abs/1409.6578, 2014.

[14] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 27–34, 2007.

[15] M. Gogolla, F. Hilken, and K.-H. Doan, "Achieving model quality through model validation, verification and exploration," *Computer Languages, Systems & Structures*, vol. 54, pp. 474–511, Dec. 2018.

[16] F. Büttner and M. Gogolla, "On OCL-based imperative languages," *Sci. Comput. Program.*, vol. 92, pp. 162–178, 2014.

[17] R. Hai, C. Quix, and M. Jarke, "Data lake concept and systems: a survey," *CoRR*, vol. abs/2106.09592, 2021.

[18] RedisLabs, "Redis," 2021. [Online]. Available: https://redis.io/

[19] Digital Twin Consortium, "Glossary of digital twins," 2021. [Online]. Available: https://www.digitaltwinconsortium.org/glossary/index.htm

[20] S. Haag and R. Anderl, "Digital twin –- proof of concept," *Manufacturing Letters*, vol. 15, pp. 64–66, 2018.

[21] F. Tao, M. Zhang, Y. Liu, and A. Nee, "Digital twin driven prognostics and health management for complex equipment," *CIRP Annals*, vol. 67, no. 1, pp. 169–172, 2018.

[22] F. Tao, H. Zhang, A. Liu, and A. Y. C. Nee, "Digital twin in industry: State-of-the-art," *IEEE Trans. Ind. Informatics*, vol. 15, no. 4, pp. 2405–2415, 2019.

[23] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern–Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[24] P. Muñoz, P. Karkhanis, M. van den Brand, and A. Vallecillo, "Modeling objects with uncertain behaviors," *Proc. of ECMFA'21. Journal of Object Technology*, vol. 20, no. 3, pp. 8:1–24, Jun. 2020.

[25] M. F. Bertoa, L. Burgueño, N. Moreno, and A. Vallecillo, "Incorporating measurement uncertainty into OCL/UML primitive datatypes," *Softw. Syst. Model.*, vol. 19, no. 5, pp. 1163–1189, 2020.

[26] A. Vallecillo and M. Gogolla, "Modeling behavioral deontic constraints using UML and OCL," in *Proc. of ER'20*, ser. LNCS, vol. 12400. Springer, 2020, pp. 134–148.
[27] M. Gogolla and A. Vallecillo, "(An Example for) Formally Modeling Robot Behavior with UML and OCL," in *Proc. of MORSE@STAF'17*, ser. LNCS, vol. 10748. Springer, 2017, pp. 232–246.
[28] IEEE Std 1003.1-2008, *The Open Group Base Specifications. Issue 7, Sect. 4.16, Seconds Since the Epoch*, 2016.
[29] "Jedis," 2021. [Online]. Available: https://github.com/redis/jedis
[30] P. Muñoz, J. Troya, and A. Vallecillo, "Digital Twins Modeling Framework – Git repository," 2021. [Online]. Available: https://github.com/atenearesearchgroup/digitalTwinModelingFramework
[31] A. Vallecillo and M. Gogolla, "Modeling behavioral deontic constraints using UML and OCL," in *Proc. of ER'20*, ser. LNCS, vol. 12400. Springer, 2020, pp. 134–148.
[32] L. Hamann, O. Hofrichter, and M. Gogolla, "Ocl-based runtime monitoring of applications with protocol state machines," in *Proc. of ECMFA'12*, ser. LNCS, vol. 7349. Springer, 2012, pp. 384–399.
[33] L. Hamann, M. Gogolla, and M. Kuhlmann, "Ocl-based runtime monitoring of JVM hosted applications," *Electron. Commun. Eur. Assoc. Softw. Sci. Technol. (ECEASST)*, vol. 44, 2011.
[34] A. Sharma, E. Kosasih, J. Zhang, A. Brintrup, and A. Calinescu, "Digital twins: State of the art theory and practice, challenges, and open research questions," *CoRR*, vol. abs/2011.02833, 2020.
[35] M. Liebenberg and M. Jarke, "Information systems engineering with digital shadows: Concept and case studies," in *Advanced Information Systems Engineering*. Springer, 2020, pp. 70–84.