Proceedings of the 2023 Winter Simulation Conference
C. G. Corlu, S. R. Hunter, H. Lam, B. S. Onggo, J. Shortle, and B. Biller, eds.

AUTOMATED SIMULATION AND VIRTUAL REALITY COUPLING FOR INTERACTIVE DIGITAL TWINS

Kai Franke
J. Marius Stürmer
Tobias Koch

Institute for the Protection of Terrestrial Infrastructures
German Aerospace Center
Rathausallee 12
Sankt Augustin, 53757, GERMANY

ABSTRACT

While there are many efforts to simulate technical systems in virtual environments and provide a visual interaction for applications such as training, authoring and analysis, the process of generating applications still requires a lot of manual work. This is particularly critical in the context of interactive Digital Twins for resilience, where uncertain events can occur and every malfunction or mistreatment of any part of the system needs to be modeled. This paper presents an approach to model such systems in a modular way by automating the generation of its components for a game engine and simulators based on a common specification. Component instances are then synchronized bidirectionally across applications to achieve interaction between the game engine and simulators. An example hydraulic system is implemented and tested to demonstrate our approach, which needs minimal manual work by using predefined components. The solution can be extended by integrating more components and simulations.

1 INTRODUCTION

The protection of *critical infrastructures* is an essential process to preserve a functioning society (Germany Federal Ministry of the Interior. 2009). An aspect that is of special importance for the protection of infrastructures are the concepts of *resilience* and *resilience management*. Resilience management aims to generally strengthen the system properties to better deal with the impact of unspecific and possibly unforeseen disruptions (Wied et al. 2020).

One measure to prepare for failures of infrastructure is the training and sensitization of people involved (Chowdhury and Gkioulos 2021). A particular technology that has established itself for training is *Virtual Reality (VR)*. This is mainly due to the high degree of immersion (ability to perceive and interact with a virtual environment in a natural manner) (Checa and Bustillo 2020; Psotka 1995), especially when dealing with environments in which spatial understanding is needed (North and North 2016). There already exist some applications that take advantage of these beneficial factors to create training applications for the protection of (critical) infrastructures (Grabowski and Wodzyński 2020).

Building and implementing these training applications in VR involves a lot of manual work. Therefore, training processes pursued in the application are often modeled in an idealized manner and rigidly scripted (Hsu et al. 2013) without all possible error scenarios and all sources of technical or human failure being included. Failure scenarios of the system need to be known in order to implement them as deviations (Schlüter 2021).

These restrictive factors can be neglected if the training is supposed to teach a fixed, predefined and known process, in which a false operation of the remodeled system is not possible in reality or does not result in any negative consequences. This may even extend to *Gray Swan scenarios* (scenarios that may be predictable but do not happen often), although an implementation would only be possible with a lot of effort (Hsu et al. 2013). In the context of resilience preparation training however, there is a need to deal with uncertain events which is not possible using the methods described above.

To overcome these limitations, a VR training environment capable of handling *what-if scenarios* (Hodicky et al. 2020) must be designed. To achieve this goal, the following framework conditions must be ensured for the virtual environment presented in this paper:

- Every part of a system may be prone to malfunctions;
- Every human-interactive part may be (mis-)handled;
- The system must react to every malfunction or maltreatment and display consequences to the user;
- The future state of the system can be forecasted to know if user interaction leads to negative behavior of the system.

The realization of such an environment poses different challenges. Since exploiting favorable factors of VR training is desirable, a VR-capable *game engine* should provide an interactive virtual environment (Bijl and Boer 2011). Game engines are also capable of mimicking real life behaviors to some degree (e.g., realistic rendering, gravity- and collision-based physics). However, they do have their limitations when dealing with realistic simulations of systems, especially when dealing with forecasting or systems of high complexity.

In order to cope with this challenge, *simulators* are particularly suitable. Technical systems often consist of complex, dynamic relationships. The behavior of these systems can be simulated with data-driven or first-principles models, for which suitable programs or simulators are used. Data-driven simulation models use recorded data of existing systems to approximate their behaviour from the relation of the systems inputs and outputs, for example with the usage of deep neural networks (Legaard et al. 2023). However, they often lack the ability to extrapolate on new, unseen behaviour which may not be included in the recorded data. First-principle models rely on classic chemical or physical descriptions. In simulation, the descriptions are expressed as sets of (differential) equations which are solved with numerical methods.

This paper presents a novel approach to generate and couple a VR environment with simulators. Based on related research on that matter (see Section 2), it describes a method to provide digital application-independent versions of individual components that make up real systems (see Section 3.1). In addition, a new method is proposed to automatically generate the digital version of a real system in a VR environment as well as in simulators using a single system specification (see Section 3.2). This eases the bidirectional communication of system changes between the VR environment and simulators caused by user interaction and changed simulation values, respectively (see Section 3.3). An example hydraulic system is then implemented (see Section 4) and evaluated (see Section 5) to test the solution. These results and some shortcomings are discussed in Section 6, before giving an concluding outlook on further improvements in Section 7.

2 RELATED WORK

The coupling of game engines and simulators to create an environment with enhanced functionality and visualization methods has been discussed in the literature.

Strassburger et al. provide a theoretical foundation on how to create a joint environment using discrete simulation systems and VR with the focus of synchronizing both parts (Strassburger et al. 2005). Given the dimensions of simulation systems provided, our approach can be categorized as a *temporally parallel* (simulation and visualization run next to each other) and *distributed* (with regard to hardware platforms) system with *bidirectional interaction* between an autonomous user interface and simulation. Lastly, *buffered* synchronization (application reads cached simulation values for certain time steps from a buffer, once it reaches the time to display the stored values) is deemed the most usable for VR applications.

There are several approaches of coupling Virtual (Reality) Environments to a single simulation. The approach closest to the goals described in this paper is carried out by Dorozhkin et al. (2012). The challenge of bidirectional interaction is solved by using buffers as described by Strassburger et al. and interrupting and restarting the simulation with updated values which may be useful for our approach as well. While bidirectional interactivity itself is achieved, the method of interaction with the virtual world is discouraged in our case, since the simulation is interrupted and restarted manually by using a menu and not by the interaction with individual components like desired. In addition, the approach by Dorizhkin et al. does not include how the visualization- and simulation environment is created to allow for synchronization between the two distributed instances.

In a work by Bijl and Boer (2011), they discuss the coupling of simulations based on the Discrete Event System Specification (DEVS) to a game engine. The simulation objects within DEVS are used as a common basis for the execution of the simulation as well as for the display of state changes in the game engine. Such a common data source is also required to achieve the goals of this paper. But since Bijl and Boer focus on the usefulness of advanced 3D visualization tools in the usage of simulation visualization, the method of mapping 3D visualization models and DEVS simulation objects is not further elaborated.

Another approach by Allard and Raffin (2006) focuses on the inter-simulation interactivity with integrated visualization using VRFlow. While this approach does establish a fully interactive environment consisting of simulators and a visualization frontend, it does not fulfill the requirements needed by this paper, since the goal is to model a real-life system consisting of multiple components that can be interacted with. In addition, even though the simulations are executed on a cluster of machines, the visualization framerate reached only 16 fps which is not sufficient when dealing with head mounted display VR applications (targeted 90 fps). This is mostly due to the computational demanding simulation methods used in the paper that should therefore not be considered in our approach. Instead, selecting expedient simulators that are just as complex as necessary for the technical system should be focused on when integrating them (see Section 4.2) .

3 METHOD

In general terms, the solution proposed in this paper consists of five methodical steps to connect simulators and a VR game engine environment (see Figure 1 for a schematic representation). First, it is described how a system can be modeled. A modular agent-based approach is chosen, i.e., the system is modeled by the behavior of and interplay between its components (Klügl 2016). Modeling the interaction with and simulation of components in a modular manner and consolidating them into one catalog makes them reusable for different systems and therefore reduces the amount of manual modeling work. In addition, this approach results in a more scalable system (Ponder 2004). This step is explained in more detail in Section 3.1. Next, the system itself needs to be modeled using the components provided. In a third step, the virtual environment is exported into a specification, containing a minimal but complete description of the contained components with their features and connections to other components. This specification is then used in the fourth step to automate the generation of components based on a specification for different applications. This results in the modeled system to be available to both simulators and game engines. Step three and four are explained in Section 3.2. Using the specification allows the generation process to achieve compatibility of system instances built up in simulators and the VR frontend. These instances are then synchronized with each other across applications in a last step to achieve an interactive VR environment with simulators providing simulated data for components (see Section 3.3).

3.1 System Modeling

To achieve a flexible and scalable description of a given system, a modular description of its components is carried out. Components should be functional on their own (even though they may depend on other components to work properly). This is achieved by defining *attributes* and *functions* for each component

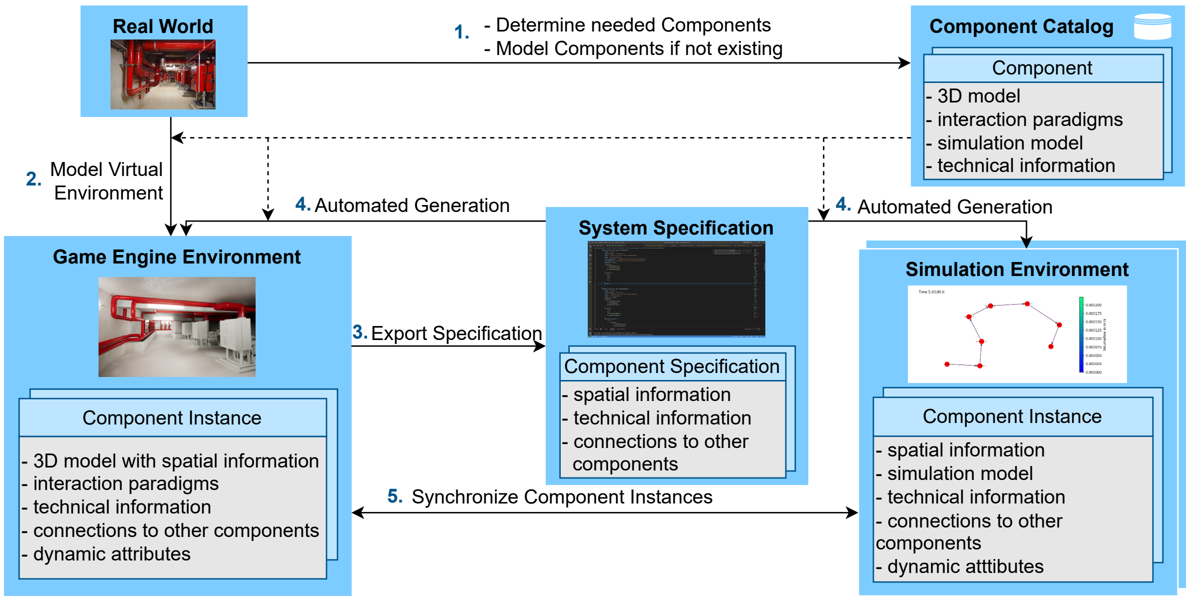

Figure 1: Proposed Method: High level concept of the generation steps to create coupled game engine- and simulation environments (arrows with labels denoting the order of steps). Components that are present in the component catalog are used to model the virtual environment and generate it as well as the simulation based on a system specification (dashed arrows).

that describe the components properties and behavior, respectively. Some *base component attributes* are needed to identify components and distinguish them from each other across applications in which they are used. These attributes include a *Universally Unique Identifier* (UUID) for identification and a *type identifier* in order to map further attributes and functions to the component. In addition, spatial information (position and rotation) are needed across all applications and are therefore persistent for each component.

In reality, components of different types may be described in a fixed manner using *data sheets*. On the one hand, data sheets provide for a good source of attributes and functions which are therefore integrated into the description of the component. On the other hand, it is beneficial to describe a component using a data sheet directly instead of filling all of its attribute values into the component (e.g., to keep component specifications size small, see Section 3.2). To achieve this behavior, a *special type* base attribute is integrated which refers to a digitized version of a data sheet. The digitized data sheet should be able to store information using serializeable and unique key-value pairs in order to persist it and allow for deserialization respectively.

In order to make components accessible for usage in different contexts, *component wrappers* are defined. Component wrappers take attributes given by the component and convert them so that they can be used by the target application (e.g., unit conversions, formatting). Wrappers may also redefine existing or define new functions that are needed in the context (e.g., functions to simulate the component within a simulator or displaying it in a game engine). This prevents components from defining redundant attributes and functions for the sake of usability by different applications.

Component attributes can be categorized into *static* and *dynamic* attributes. While static component attributes are not altered by any application (such as the components UUID or its type), dynamic attributes will be changed by simulation or interaction. Dynamic attribute changes are communicated via events to allow applications, wrappers and/or components themselves to deal with the performed attribute change.

When dealing with components of different types, there might be more advanced ones that expand the functionality of another component type (e.g., a pump that uses a motor to pump materials in a hydraulic system) and/or consist of several other components in a functional dependent manner (e.g., a sink that combines an open tank with a drain, a valve and a pipe to direct water). To keep the modeling complexity low when defining new components that act similar to already existing ones, *component inheritance* is sufficient. Advanced components reuse its base attributes and functions and may define new ones. When dealing with complex components, consisting of several others, references to sub components (using their UUIDs) can be stored as attributes.

Many components in a system are connected with each other and interact in a functional manner (such as a straight pipe and its purpose to transfer material from one component to another). Such connections are stored analogously to complex components by storing the connected component's UUID as a reference.

3.2 Specification-Based Automated Generation

To automate the component instance generation across applications, a singular *specification* is used that contains all needed information for all applications to generate their component instances without providing redundant information. The specification should be able to store information in a hierarchical manner using key-value pairs, with the collection of stored components at the top level and its attributes names and values nested below (similar to digitized data sheets, see Section 3.1). This ensures that stored attributes are unique and can therefore be mapped to component attributes. Only static attributes that are not present in data sheets are integrated in the specification to keep its size as low as possible. Such a specification can be generated in an automated manner (Martínez et al. 2018) (see Section 7 for other sources to generate specifications). In this work the specification is generated using a game engine since it allows for the definition and spatial placement of components.

To generate the components based on the specification, an abstract factory software pattern is deployed. Factories for a variety of contexts produce components based on the given specification (especially the stored type) and wrap them into component wrappers for the context they are implemented for. If components have a certain special type, the factory fills attributes based on the given data sheet the special type refers to. Figure 2 shows a schematic representation of this concept.

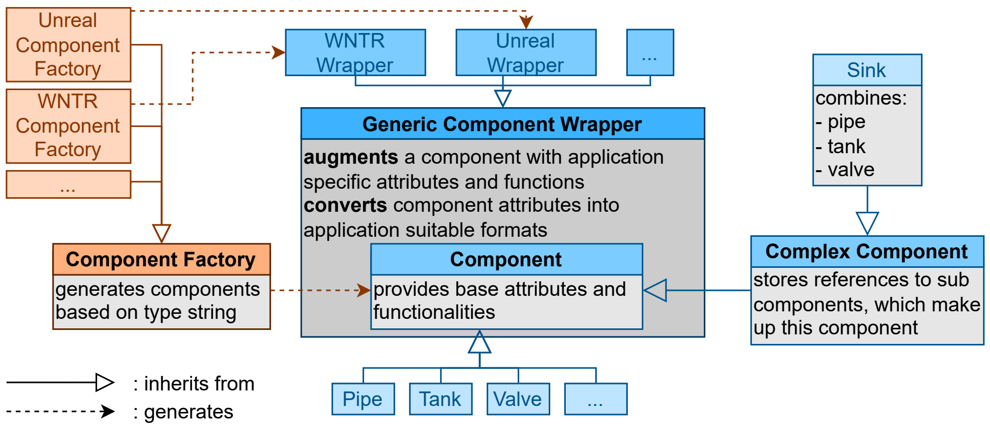
Unreal Component Factory
WNTR Component Factory
...
Component Factory
generates components based on type string
WNTR Wrapper
Unreal Wrapper
...
Generic Component Wrapper
augments a component with application specific attributes and functions
converts component attributes into application suitable formats
Component
provides base attributes and functionalities
Sink
combines:
- pipe
- tank
- valve
Complex Component
stores references to sub components, which make up this component
Pipe
Tank
Valve
...
: inherits from
: generates

Figure 2: High-level view on the component factory using an Abstract Factory pattern proposed in this paper. For our hydraulic demonstrator, *WNTR* (Klise et al. 2018) is used for hydraulic simulation and *Unreal Engine* is used for interactable VR visualization.

After generating a set of components (and optionally their wrappers) for a given system, some components need to have access to other ones they are connected to or are functionally dependent on (see complex components, Section 3.1). To achieve this, a *component pool* is introduced which collects components and allows accessing them via their UUID. Since component UUIDs are stored within interconnected or complex components, direct access to the specified component is possible via the pool. Events are used to communicate the addition or removal of components.

3.3 Cross-Application Synchronization

Each given application using the modeled system employs a factory that generates component instances and wrappers to make them usable. Since each factory is given the same specification, the resulting component pools will all have the same component instances, with equal component instances persisting the same UUID. This UUID-mapping and the event triggering caused by attribute changes (see Section 3.1) form the core functionality of cross application communication, i.e., whenever a component instance identified via a certain UUID fires a changed-event, the attribute change will be synchronized across applications.

The overarching platform is chosen to be a client-server architecture with simulation- and frontend environments being client applications to ensure a single point of truth on the server side. Since the ability to communicate local component attribute changes or apply server-side alterations is a functionality that can be provided by a component wrapper (see Section 3.1), we again use a component factory. This time, the factory only listens to the addition and removal of component instances to/from the pool of the application (see events of component pool, Section 3.2) and generates/removes only the wrappers for communication with the server. The server that fits the given platform can be arbitrary, as long as it is able to store the data model of the components (component serializability is ensured by the system specification, see Section 3.2) and change the data model of components, when given the UUID of the component and its attribute key and value. In addition, it should be able to communicate attribute values of components back (either via subscription or polling).

4 IMPLEMENTATION

A hydraulic system is chosen to test our methods explained in the last section (see Figure 3). The system employs a 300 L IBC water reservoir that is connected to a multi-turn gate valve to direct water to a second smaller tank with a capacity of 5 L. This tank is connected to a pump that directs water to a nearby sink The system has been chosen as a minimal example for various reasons. The multi-valve system allows

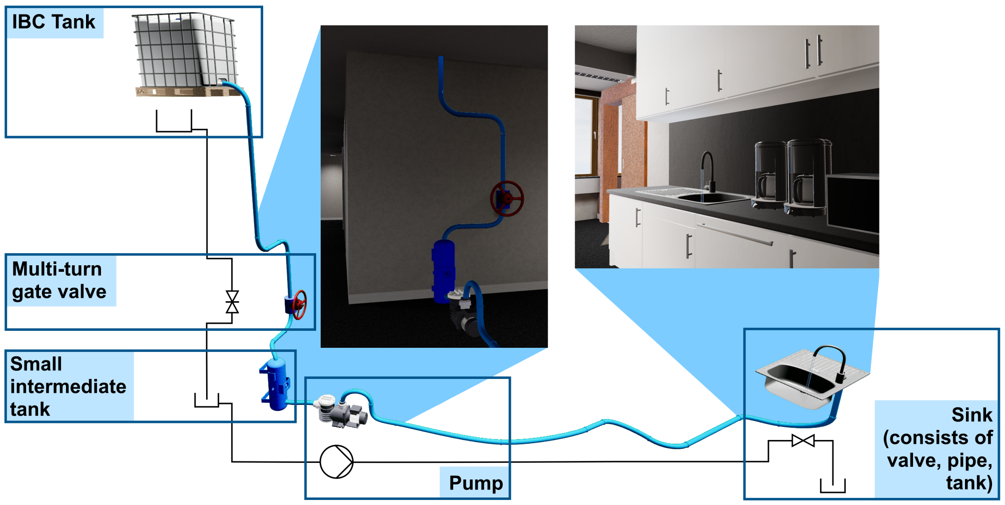

Figure 3: The technical system that is used to test the implementation. An IBC water tank is connected to a multi-turn gate valve, to which another much smaller tank, a pump and a sink are connected.

for arbitrary interaction with the system, while also having most of the parts that exist in safety-related on-site hydraulic systems such as sprinkler systems. Since the pump within the system should not run dry, there is also the possibility of mishandling the system by opening the tap valve of the sink and closing the supply from the reservoir using the gate valve. It also demonstrates the usage of all component modeling aspects described in Section 3.1. There exist interconnected (e.g., pipes) and complex components (sink) as well as component inheritance (pump that employs motor properties). Additionally, attribute values of tanks and the pump were taken from data sheets that are provided by the respective manufacturer.

The logic for component modeling, the component factory and cross application synchronization has been implemented in Python 3.9.7. To build up the component catalog, a simple folder structure is used. Each component type ⟨*Type*⟩ has its own folder *FtComponent*⟨*Type*⟩, in which all assets of the component are stored. One of these assets is a python script, describing attributes and functions for the type as a python class. The factory queries the folders for this script, when creating component instances. To ease the process of component (de-)serialization, each component class has a function to (de-)serialize its attributes into the JSON format. The water network tool *WNTR 0.4.4* by Klise et al. (2018) has been chosen to simulate the system, due to its application for resilient infrastructures. WNTR is a hydraulic simulator originally used to simulate water distribution systems. Systems can be modeled with EPANET 2.2 or Python directly. WNTR itself uses the types *tank*, *reservoir*, *pipe*, *pump* and *valve*. Therefore, WNTR attributes for these objects are integrated in the component definition, in addition to attributes specified in the found data sheets. The resulting attribute set per component is shown in Table 1. The resulting platform can be seen in Figure 4.

Table 1: Attributes for components used in the demonstration system. Base component attributes are excluded.

Component	**Attribute**	**Type**	**Unit**
Motor	real_power	Float	W
	reactive_power	Float	VA
	power_consumption	Float	W h
Pump (derived from Motor)	material_inlets	List[Transform]	-
	material_outlets	List[Transform]	-
	pump_type	String	-
	pump_curve	Curve Coeffs.	-
	pump_speed_rel	Float	-
	current_flow_rate	Float	$m^3 s^{-1}$
	current_pressure_in	Float	Pa
	current_pressure_out	Float	Pa
	current_velocity	Float	$m s^{-1}$
Tank	outlets	Transform	-
	volume_curve	Curve Coeffs.	-
	diameter	Float	m
	current_level	Float	m
	current_pressure_out	Float	Pa
Reservoir	-	-	-
Pipe	from_components	List[UUID]	-
	to_components	List[UUID]	-
	diameter	Float	m
	route	List[Vector]	-
	roughness	Float	mm
	current_flow_rate	Float	$m^3 s^{-1}$
	current_velocity	Float	$m s^{-1}$
	current_headloss	Float	Pa
Valve	material_inlets	List[Transform]	-
	material_outlets	List[Transform]	-
	valve_type	String	-
	diameter	Float	m
	opening_norm_value	Float	-
	current_flow_rate	Float	$m^3 s^{-1}$
	current_pressure_in	Float	Pa
	current_pressure_out	Float	Pa
	current_velocity	Float	$m s^{-1}$

4.1 Game Engine Integration

As a frontend application, the game engine *Unreal Engine 5.0.3* is chosen. To make the component catalog accessible for the engine while still being able to handle each component type in a modular manner, each of the component folders *FtComponent*⟨*Type*⟩ is an Unreal plugin and the Python Editor Scripting plugin is used. The visual appearance of a component and its attributes are controlled using Unreal's Blueprint system, in which one base *Actor Blueprint* exists per component type, exposing its attributes and functions to the game engine. In addition, there exist child blueprints of the base Actor Blueprint for each component's special type. This allows for modular swapping of visuals within the engine, while still using the same attributes and functions. For each Actor, the mechanics to display component attributes and change them via interaction is modeled manually.

As explained in Section 3.1, Unreal component wrappers are used for various compatibility reasons. First, the wrappers are used to spawn Blueprint instances of the component of the specified (special) type, when they are generated by the factory. Second, attributes are integrated into the Actor instance. In most cases, this is done by simply applying attribute values of the component to Blueprint attributes of the Actor

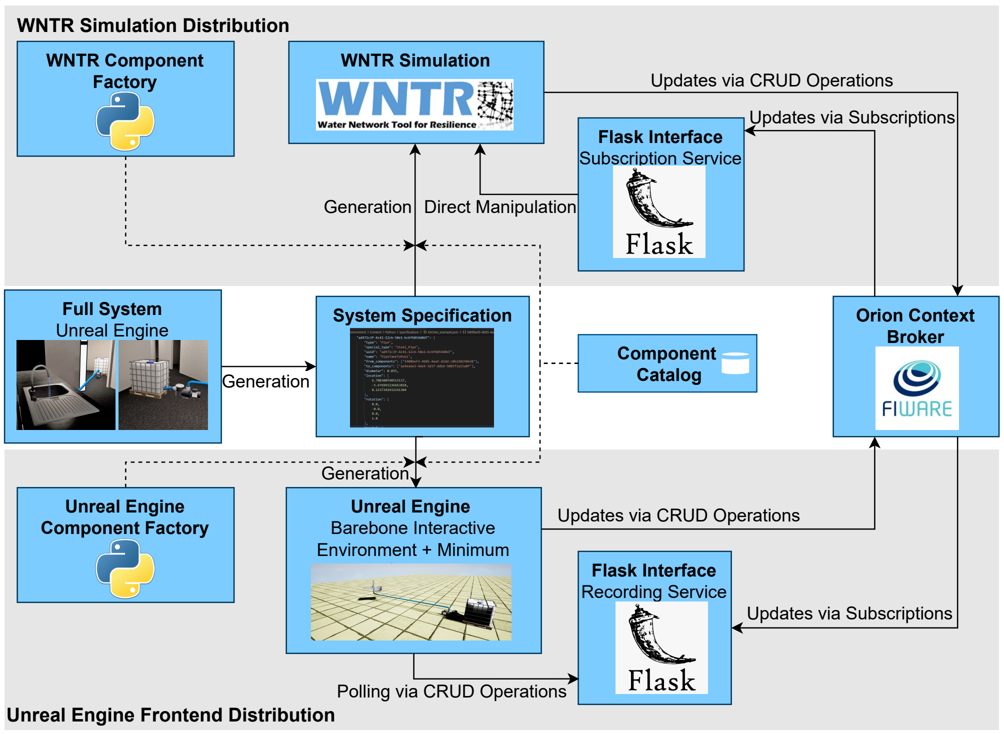
WNTR Simulation Distribution
WNTR Component Factory
WNTR Simulation
WNTR
Water Network Tool for Resilience
Updates via CRUD Operations
Updates via Subscriptions
Flask Interface Subscription Service
Flask
Generation
Direct Manipulation
Full System Unreal Engine
Generation
System Specification
Component Catalog
Orion Context Broker
FIWARE
Generation
Unreal Engine Component Factory
Unreal Engine Barebone Interactive Environment + Minimum
Updates via CRUD Operations
Flask Interface Recording Service
Updates via Subscriptions
Polling via CRUD Operations
Unreal Engine Frontend Distribution

Figure 4: Implementation of the coupling between a game engine (here: Unreal Engine) and a simulation environment. Component instances are generated based on a common specification by application specific factory implementations and synchronized using the FIWARE Orion Broker.

instance. Nevertheless, there are also cases where the appearance of the Actor is changed when integrating component attributes (e.g., when applying a pipe's route to the Unreal Actor). The wrappers are also used in the opposite direction to generate a specification based on a modeled system in the game engine. This means each Actor that is a component includes a function to serialize all its attributes into the JSON format. By mapping the component's UUID to this JSON and join these mappings for all components, a specification that fulfills the conditions specified in Section 3.1 is achieved.

4.2 Simulator Integration

As described above, WNTR is used to model and simulate the hydraulic system. WNTR has been chosen due to its simplicity, expedience and compatibility with Python. Thus, it can be directly used with the rest of the application. The step size to update the attributes is chosen to be 1 s, thus the simulation step size is also set to 1 s. However, WNTR by itself is not capable of real-time simulation.

Analogous to the game engine integration, WNTR wrappers are used to create and interact with the WNTR components and to update the component attributes with the values calculated by WNTR. Contrary to the game engine, where modeling complex components as the combination of various other components is needed, WNTR does not need this information and instead inspects only non-complex components.

The bidirectional coupling is realized with multi-threading. One thread continuously runs the WNTR simulation and inserts the new state of the system into a buffer. Thus, it calculates new states for the system as fast as possible for any length of time. A second thread transfers the system states from the buffer to the data model once per second in wall-clock time. A third thread is used to react to external input in an event-based manner. If there are external changes, this thread removes all system states that are dated newer than the current time from the buffer and restarts the simulating thread with the current system state.

4.3 Cross-Application Synchronization

To synchronize components and its attributes across simulators and the game engine, the *FIWARE Orion Context Broker* is used. This Broker is chosen because it fits the requirements stated in Section 3.3. It has already been used in applications similar to the targeted use case, like macroscopic water management (Kamienski et al. 2019) as well as data management in industry (Alonso et al. 2018). To implement a client, a component wrapper as explained in Section 3.3 is needed. In this case, the wrapper adds the functionality to (de-)serialize components from/to the NGSI-LD data model to communicate with the broker.

To upstream local changes according to change events of components, REST CRUD (Create, Read, Update, Delete) operations are used which change the state of the data model behind the FIWARE Orion Context Broker. Batch CRUD Operations are used to communicate altered component attributes that were modified in one operation (e.g., in one simulation step). To downstream remote changes of the data model, the subscription service provided by the broker is used which itself sends REST calls with updated components and attributes. To receive these messages, a web server is needed which may not be employable by every application natively in the same process (e.g., the Unreal Engine). To still be able to receive remote component updates, an interface web server using the python web framework *Flask* is implemented which is defined as the subscription target. If the interface is able to directly access components (this is the case if both the application and the interface can be run in the same process), the interface will directly update component attributes. If not, it will instead record the changes and return them, if the application asks about changes via polling.

5 RESULTS

The implementation of the last section has been tested on a *Razer Blade 15 Studio Edition* laptop (Quadro RTX 5000 Max Q, Intel Core i7 10875H, 64GB RAM). The example system described in Section 4 and all components included are modeled and exported into a system specification, which has the size of 20 kB. Both the Unreal Engine and the WNTR simulation environment are able to use this specification to automatically generate and synchronize components using the component catalog and FIWARE respectively.

When using the technical system explained in Section 4, it is possible to interact with the gate valve and the sink handle which triggers accurate responses from the WNTR simulation, running in the background. When opening the sink valve only, the small tank behind the gate valve empties and water needs to be supplied by opening the gate valve in order to keep a steady water flow.

In a first benchmark for the response time of the system, the gate valve is opened to allow for a water flow through all components up to the sink valve which is closed. In 50 iterations, the sink valve is then opened and the time is measured, until the changed valve value has reached WNTR and until the Unreal Engine itself receives new component attributes in response. The former happens in average 0.012 s, while the latter needed an average of 1.563 s. The process is repeated with an opened sink valve and opening the gate valve, which yielded the same response times.

To further elaborate on response time when scaling the system itself, a bench-marking component is introduced which consists of 50 attributes with a size of 16 B each. The second benchmark then similarly aims to measure the response time of the system when changing component states via interaction. It consists of several runs with different counts of components and changed attributes. In each run, attributes of the benchmark components are changed in the Unreal Engine. The time is then measured until all changes have reached the simulation environment. The synchronization times are between 0.01 s and 0.03 s when communicating attributes of a component. The times scale up to values between 0.26 s and 2.32 s when changing up to 100 components. When applying the attributes to the virtual environment in the Unreal Engine, the frame rate remained consistently above the target limit of 90 fps.

One challenge that occurred while testing the synchronization between the Unreal Frontend and WNTR is the unpredictability of user interaction. The user is able to manipulate the system at any given time and the interaction may happen over a longer period of time. Since attribute change detection is executed once a frame (once every 0.01 s) in the Unreal Engine, changed attributes are communicated in this frequency as well. This causes the WNTR simulator to react to these changes and rebuild the simulation environment, even though the interaction has not been finished yet, resulting in unnecessary work of the application.

6 DISCUSSION

The results presented in the last section show that the implementation is able to provide for a realistic environment, both regarding visual and technical aspects. The component-based approach resulted in less modeling work, since the tank, pipe and valve attributes and functions needed to be implemented only once. Using data sheets for specifying components resulted in a slim specification, while still being able to reconstruct all needed attributes. Component descriptions, data sheets and the system specification format are very modular and adaptable. This can be an benefit if the solution is to be adapted to support additional components, simulations and front-ends. Nevertheless, the modeling of singular component visuals and their interaction in the Unreal Engine is still a manual and time-intensive process which could be improved.

Compared to similar approaches to real-time coupling, the connection of the VR environment and simulator is achieved in a performant manner that allows both the desired frame rate for VR HMDs and acceptable simulator response times. To interpret the results of the interaction response time benchmark of the last section, one needs to differentiate the two cases of bidirectional attribute change communication. When dealing with component-based user interaction in the game engine, only one component is modified at a time and, in most cases, only one attribute is affected by the interaction (e.g., the opening percentage attribute of a valve). Therefore, synchronization times are relatively low. To synchronize changes that have been made in the simulator though, much more attributes of more components may need to be changed and communicated (e.g., flow rates of all pipes connected to a valve that has been opened). Given the simulation step interval of 1 s, communication times of more than 1 s might pose challenges for the system, since values are already outdated as soon as they arrive at the game engine.

Another challenge that might arise when dealing with more complex technical systems is the simulator itself. In order to be used for a real-time application such as the presented setup, there needs to be a guarantee that the simulated time, in which the simulator produces values for the application, is greater than or equal to real-time. The modified WNTR simulation framework implemented in this paper can be considered *best effort real-time* and might not meet this condition, when using more components within a hydraulic system or when reducing the step size, in which simulation values are produced.

In addition, the reaction of the simulation framework to external component attribute changes proved to be a challenge. Loading the simulation from the moment of external change in real-time and restarting with changed input values results in unnecessary work of the application. Here, a real-time simulator that does not need to be restarted with changed values should be preferred. Optimal for the described system would be event-based real-time simulations that could react directly to change events of the components (see Section 3.1). Such simulators exist for other fields of application (Gosavi et al. 2019), but to the best of our knowledge no event-based hydraulic simulator similar to the quality of WNTR exists.

This additional application workload through restarting the simulation engine is particularly disadvantageous when it is caused by restarting multiple times because of continuous interaction at the frontend. One approach to overcome this issue would be including the start- and endpoint of a user interaction based on existing taxonomies (Bowman and Hodges 1999) with a component and communicating changed values after the interaction has been finished.

7 CONCLUSIONS AND FUTURE WORK

This paper provides an approach to create a multi-application environment that is able to simulate a technical system in a realistic and accurate manner and allows the display and interaction via a VR application based on a game engine. Both simulators and frontend are able to automatically build up the system using a common specification that consists of a modular description of its components. Attached simulators are able to react to component changes in real-time using bidirectional synchronization in a client-server side manner. Especially with respect to its flexibility in defining a system and connecting simulators, the presented approach is novel compared to other publications on that matter. In contrast to traditional rigor-scripted training applications, the resulting architecture is particularly useful in the application area of resilience management training, where uncertain incorrect behaviors need to be noticed and countered.

Currently, operating parameters of the system are the result of ongoing simulations. This is where the potential of integrating sensor data through IoT devices arises. The closeness of the approach of this paper to FIWARE and its decentralized architecture can potentially enable IoT devices to be integrated (Cirillo et al. 2019) and finally visualized in virtual environments (Wu et al. 2019).

While the presented approach automates the generation of virtual environments and simulations, the modeling of components and the inital modeling of the environment to export the system specification (see step one and two in Figure 1) still needs to be done manually. To automate the modeling of new components, CAD digital models provided by manufacturing companies could be converted and integrated into the component catalog. In addition, there are efforts to provide standardized attribute descriptions of components (e.g., https://schema.org/). Using these attributes instead of defining them manually could also improve the speed at which new components could be modeled. In addition, the modular approach enables the automatic generation of system specifications using measurement processing (Aldoma et al. 2012), technical plan detection (Kong et al. 2022) or artificial intelligence.

While the demonstration system described in Section 4 is less complex than the ones existing on sites of critical infrastructure buildings, it does contain most components found in industrial hydraulic systems. Nevertheless, more tests need to be carried out to ensure its usefulness for real world applications. This includes scaling up the system as well as introducing new simulators for more functional aspects (e.g., electricity, people movements). This scaling will increase the number of component (special) types and will lead to the extension/adaptation of attributes and functions of existing components to account for new contexts. Another interesting aspect that needs more testing are the different applications themselves which will have to deal with more components and attributes, when using a more complex system.

REFERENCES

Aldoma, A., Z.-C. Marton, F. Tombari, W. Wohlkinger, C. Potthast, B. Zeisl, R. B. Rusu, S. Gedikli, and M. Vincze. 2012. "Tutorial: Point Cloud Library: Three-Dimensional Object Recognition and 6 DOF Pose Estimation". *IEEE Robotics & Automation Magazine* 19(3):80–91.

Allard, J., and B. Raffin. 2006. "Distributed Physical Based Simulations for Large VR Applications". In *Proceedings of the 2006 IEEE Virtual Reality Conference*, edited by B. Fröhlich, D. Bowman, and H. Iwata, 89–96. Alexandria, Virginia: Institute of Electrical and Electronics Engineers, Inc.

Alonso, Á., A. Pozo, J. M. Cantera, F. De la Vega, and J. J. Hierro. 2018. "Industrial Data Space Architecture Implementation Using FIWARE". *Sensors* 18(7):2226.

Bijl, J. L., and C. A. Boer. 2011. "Advanced 3D Visualization for Simulation Using Game Technology". In *Proceedings of the 2011 Winter Simulation Conference*, edited by S. Jain, R. R. Creasey, J. Himmelspach, P. K. White, and M. C. Fu, 2810–2821. Institute of Electrical and Electronics Engineers, Inc.

Bowman, D. A., and L. F. Hodges. 1999. "Formalizing the Design, Evaluation, and Application of Interaction Techniques for Immersive Virtual Environments". *Journal of Visual Languages & Computing* 10(1):37–53.

Checa, D., and A. Bustillo. 2020. "A Review of Immersive Virtual Reality Serious Games to Enhance Learning and Training". *Multimedia Tools and Applications* 79(9):5501–5527.

Chowdhury, N., and V. Gkioulos. 2021. "Cyber Security Training for Critical Infrastructure Protection: A Literature Review". *Computer Science Review* 40(C):100361.

Cirillo, F., G. Solmaz, E. L. Berz, M. Bauer, B. Cheng, and E. Kovacs. 2019. "A Standard-Based Open Source IoT Platform: FIWARE". *IEEE Internet of Things Magazine* 2(3):12–18.

Dorozhkin, D., J. Vance, G. Rehn, and M. Lemessi. 2012. "Coupling of Interactive Manufacturing Operations Simulation and Immersive Virtual Reality". *Virtual Reality* 16(1):15–23.

Germany Federal Ministry of the Interior. 2009. "National Strategy for Critical Infrastructure Protection (CIP Strategy)".[736048], Federal Republic of Germany, Federal Ministry of the Interior.

Gosavi, A., G. Fraioli, L. H. Sneed, and N. Tasker. 2019. "Discrete-Event-Based Simulation Model for Performance Evaluation of Post-Earthquake Restoration in a Smart City". *IEEE Transactions on Engineering Management* 67(3):582–592.

Grabowski, A., and M. Wodzyński. 2020. "The Use of Virtual Reality for Training in Securing the Functioning of Critical Infrastructure". *Problems of Mechatronics. Armament, Aviation, Safety Engineering* 11(4):73–82.

Hodicky, J., G. Özkan, H. Özdemir, P. Stodola, J. Drozd, and W. Buck. 2020. "Dynamic Modeling for Resilience Measurement: NATO Resilience Decision Support model". *Applied Sciences* 10(8):2639.

Hsu, E. B., Y. Li, J. D. Bayram, D. Levinson, S. Yang, and C. Monahan. 2013. "State of Virtual Reality Based Disaster Preparedness and Response Training". *PLoS Currents* 5.

Kamienski, C., J.-P. Soininen, M. Taumberger, R. Dantas, A. Toscano, T. Salmon Cinotti, R. Filev Maia, and A. Torre Neto. 2019. "Smart Water Management Platform: IoT-based Precision Irrigation for Agriculture". *Sensors* 19(2):276.

Klise, K., R. Murray, and T. Haxton. 2018. "An Overview of the Water Network Tool for Resilience (WNTR)". *Proceedings of the 1st International WDSA/CCWI Joint Conference* 1(75).

Klügl, F. 2016. *Agent-based Simulation Engineering*. Ph. D. thesis, Örebro University, Örebro, Sweden.

Kong, M.-C., M.-I. Roh, K.-S. Kim, J. Lee, J. Kim, and G. Lee. 2022. "Object Detection Method for Ship Safety Plans Using Deep Learning". *Ocean Engineering* 246:110587.

Legaard, C., T. Schranz, G. Schweiger, J. Drgoňa, B. Falay, C. Gomes, A. Iosifidis, M. Abkar, and P. Larsen. 2023. "Constructing Neural Network Based Models for Simulating Dynamical Systems". *ACM Computing Surveys* 55(11):1–34.

Martínez, G. S., S. Sierla, T. Karhela, and V. Vyatkin. 2018. "Automatic Generation of a Simulation-Based Digital Twin of an Industrial Process Plant". In *44th Annual Conference of the IEEE Industrial Electronics Society*. October 20th - 23th, Washington D.C.

North, M., and S. North. 2016, 02. "A Comparative Study of Sense of Presence of Virtual Reality and Immersive Environments". *Australasian Journal of Information Systems* 20:1–15.

Ponder, M. 2004. *Component-based Methodology and Development Framework for Virtual and Augmented Reality Systems*. Ph. D. thesis, École polytechnique fédérale de Lausanne, Lausanne, Switzerland.

Psotka, J. 1995. "Immersive Training Systems: Virtual Reality and Education and Training". *Instructional science* 23(5):405–431.

Schlüter, C. 2021. "LiftNick (VR) - Spielerisches Lernen für Mehr Motivation". Technical report, Fraunhofer Institute for Material Flow and Logistics, Department Information Logistics and Decision Support Systems, Dortmund, Germany.

Strassburger, S., T. Schulze, M. Lemessi, and G. D. Rehn. 2005. "Temporally Parallel Coupling of Discrete Simulation Systems with Virtual Reality Systems". In *Proceedings of the 2005 Winter Simulation Conference*, edited by M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 1949–1957. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Wied, M., J. Oehmen, and T. Welo. 2020. "Conceptualizing Resilience in Engineering Systems: An Analysis of the Literature". *Systems Engineering* 23(1):3–13.

Wu, P., M. Qi, L. Gao, W. Zou, Q. Miao, and L.-L. Liu. 2019. "Research on the Virtual Reality Synchronization of Workshop Digital Twin". In *2019 IEEE 8th Joint International Information Technology and Artificial Intelligence Conference*, 875–879. Institute of Electrical and Electronics Engineers, Inc.

AUTHOR BIOGRAPHIES

KAI FRANKE is a research associate in the Institute for the Protection of Terrestrial Infrastructures at the German Aerospace Center. His research interests focus on using innovative technologies such as Virtual- and Augmented Reality. Specifically, he is interested in embedding this equipment into Digital Twins to allow for spatial information display as well as teaching and training. His email address is kai.franke@dlr.de.

J. MARIUS STÜRMER is a research associate in the Institute for the Protection of Terrestrial Infrastructures at the German Aerospace Center. His research interests include modelling and simulation of various technical systems with a special focus on automating the generation and controlling these systems using deep learning. His email address is jan.stuermer@dlr.de.

TOBIAS KOCH is a research group leader at the Department for Digital Twins for Infrastructures in the Institute for the Protection of Terrestrial Infrastructures at the German Aerospace Center. His research focus is the generation and operation of Digital Twins for Infrastructures as a decision support and crisis management tool. His email address is tobias.koch@dlr.de.