Automating Test Oracle Generation in DevOps for Industrial Elevators

Aitor Arrieta*, Maialen Otaegi*, Liping Han†, Goiuria Sagardui*, Shaukat Ali† and Maite Arratibel ‡
Mondragon University *, Simula Research Laboratory†, Orona‡
*{aarrieta, motaegi, gsagardui}@mondragon.edu, † {liping, shaukat}@simula.no, ‡marratibel@orona-group.com

***Abstract*—Orona is a world-renowned elevators developer. During elevators' lives, their software continues to evolve, e.g., due to hardware obsolescence, requirements changes, vulnerabilities, and bug corrections. Such continuous evolution demands the continuous testing of industrial elevators with the minimum manual effort possible. To this end, we present a tool, whose core component is a domain-specific language (DSL) with which a user can specify test oracles at a higher level of abstraction and independent of a testing level. The DSL also supports specifying uncertainty-aware test oracles to test elevators under various uncertainties inherent in them. Finally, the DSL is also equipped with test oracle generation that generates test oracle code automatically at the different DevOps testing levels (i.e., Software and Hardware-in-the-Loop test levels, and in operation) to enable reuse of test oracles across these levels. We evaluated this DSL with an industrial elevators case study at Orona's site to specify and generate test oracles. The evaluation showed that the high expressiveness of the DSL permits the high-level definition of test oracles in our industrial context. Based on the industrial application, we discuss our experiences and lessons learned.**

***Index Terms*—Domain Specific Language, Test Oracle Generation, Cyber-Physical Systems, Evolution**

I. Introduction

A Cyber-Physical System (CPS) consists of several communicating and coordinating systems that provide a specific service to a set of users [1]. This paper focuses on one such CPS, i.e., an industrial elevator installation of Orona that consists of a set of elevators with each having its dedicated controller, traffic master– a device to control and optimize traffic across all the elevators, and dedicated computers, e.g., for access control. Such an installation is devised to transport passengers safely, while at the same time considering certain Quality-of-Service (QoS) measures [2]. Examples of QoS measures include the energy consumption and the Average Waiting Time (AWT).

The software of industrial elevators continuously evolves due to, e.g., bug corrections, adaption to legislation changes, and including new functionality [3]. For instance, the COVID-19 pandemic required software updates of Orona's elevators installations that were in operation to better handle the social distance. Subsequently, the use of Design-Operation Continuum approaches (e.g., DevOps) are paramount to enhance the quality of the software development process for industrial elevator installations. Among others, these methods need automated software test solutions that require test oracles to deal with continuous evolution.

In the past, some works focused on the automated generation of test oracles with Domain-Specific Language (DSL) as a specification language. For instance, Menghi et al. generated test oracles for Simulink models [4]. Arrieta et al. generated both test inputs as well as test oracles using a DSL [5]. Nevertheless, the generated test oracles with these tools were prepared for design-time testing and do not support CPS uncertainties. In addition, most of the DSL-related testing work (e.g., [6], [7]) currently do not support the reuse of test oracles at different CPS DevOps testing levels.

Therefore, we developed a tool whose core component includes a DSL that enables a high-level specification of test oracles and abstraction independent of technical details of different testing levels, e.g., Software in the Loop (SiL), Hardware in the Loop (HiL), or operation. Given the inherent uncertainty in industrial elevators, the DSL also supports specifying uncertainty-wise test oracles. Moreover, the tool comes with a compiler that generates executable test oracles for different testing levels (e.g., SiL or HiL) as well as for operation. Consequently, we enable the reuse of test oracles at both the design and operation phases of industrial elevator installation supporting the reuse of test oracles across different testing levels. For the evaluation, we used an industrial case study from Orona. The tool we developed that can generate test oracles is necessary to automate test execution that up to now remained manual. Furthermore, we analyzed the uncertainties to which elevators are exposed and analyzed whether such uncertainties could be modeled in our tool, too. The evaluation of the DSL demonstrated sufficient expressiveness to model test oracles and uncertainties. Based on the evaluation, we also provide our experiences and lessons learned.

The key contributions of this paper are: 1) A practical and expressive DSL to specify test oracles for industrial CPSs; 2) Explicit support to capture uncertainty-wise test oracles in CPSs and their environment; 3) Support to generate and compile executable test oracles for different testing levels, which is compatible with an architecture based on microservices explicitly designed for DevOps of CPSs [8]; 4) Validation of the DSL in the context of industrial elevators case study provided by our industrial partner.

II. Industrial Context and Problem

Figure 1 shows a simplified version of the architectural topology of a system of elevators from Orona. An installation

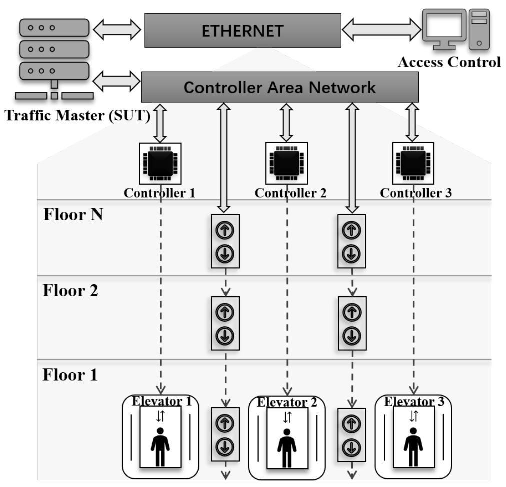


Fig. 1: Industrial Elevator Installation

may consist of a set of physical elevators (e.g., three in Figure 1), devices (e.g., traffic controllers, and lift controllers in Figure 1), and dedicated computers (e.g., for access control). Communication among different devices is enabled through different communication protocols (e.g., Controlled Area Network and Ethernet). The overall aim is to transport passengers safely and by providing the best QoS possible.

When a passenger calls a lift through the user interface, the traffic master receives information about the call. The traffic master is the system in charge of deciding which elevator should attend each call. In conventional installations, the traffic master only receives information related to the call direction (i.e., if it goes up or down), dealing with different uncertainties (e.g., number of people waiting on the floor, destination floor, etc). In the destination-selection installation, the traffic master receives the final destination of the user. In addition, each passenger may have certain restrictions (e.g., in hotel buildings, certain passengers can only go to particular floors). Such restrictions of controlling the accesses are programmed through a personal computer which communicates with the dispatcher through Ethernet. After the call has been assigned, the assignment is sent to the corresponding lift controller. The lift controllers are in charge of performing the low-level elevator control, including their speed, acceleration, doors opening/closing, etc.

The software of this system constantly evolves. Every time the code is modified, Company has a well-established software testing process, where simulation-based testing is the dominant technology. Testing at different levels is carried out. For instance, when testing the dispatching algorithm, the first test level is SiL. In such a case, a domain-specific simulator named Elevate is employed to carry out the initial tests. After the tests have been executed at this level, the software is integrated with the remaining software modules (e.g., real-time operating system), compiled, and deployed in the real-target processor for the tests to be executed at the next test level, i.e., HiL. During the HiL test level, all the infrastructure related to the controllers is real, whereas the physical part (e.g., electrical engines) is emulated. The tests are executed in real-time. After the HiL tests have been accomplished, the software module is compiled and manually deployed on the real installation through the maintainer. When the software is deployed, the maintainer performs some manual tests to ensure that everything is working correctly. As can be seen, as the test level maturity increases, the execution of tests becomes more expensive.

While the current testing process is robust and helps detect several bugs before the code is in production, there is a need for higher test automation to establish a DevOps software development method. This would not require human intervention to test and deploy a new software version in operation thoroughly. This paper focuses on capturing and generating test oracles to support testing at different test levels within this context. To achieve this, test oracles with the following characteristics are necessary:

- **Streamlined test oracles:** The test oracles shall be reuseable across the different test levels (i.e., SiL, HiL) as well as in operation. This requires for (1) an interoperable solution and (2) support for uncertainty-wise test oracles to deal with the inherent uncertainty that CPSs are exposed to at operation-time.
- **Support for asserting time-continuous behavior:** Elevators systems, as other CPSs, provide data over time, which requires asserting that the system is behaving correctly at a specific instant, and also by considering evolving signals over time. Furthermore, these systems are tested through simulation-based testing at development time, which is based on certain QoS metrics (e.g., energy consumption, AWT) [2].
- **Feedback from operation:** When a new dispatching algorithm is deployed in operation, problems that were undetectable at design-time might arise. One core requirement of the tool is to enable the compatibility of the generated test oracles to support run-time testing at operation. To this end, (1) the oracles incorporate uncertainty-wise functionalities, and (2) they are compatible with a microservice architecture targeting DevOps of CPSs [8].

III. DSL Implementation and Overview

Figure 2 shows an overview of the proposed framework for the definition and automated generation of test oracles in systems of elevators. The framework is divided into 1) the language for the definition of test oracles and 2) the test oracle generator.[1]

A. *Language for the definition of test oracles*

The language is implemented in Xtext. Its syntax has as objective to be expressive enough to characterize test oracles, but at the highest abstraction level possible. This abstraction

[1]The implementation of the language is available at https://github.com/maialenotaegi/AdeptnessDSL

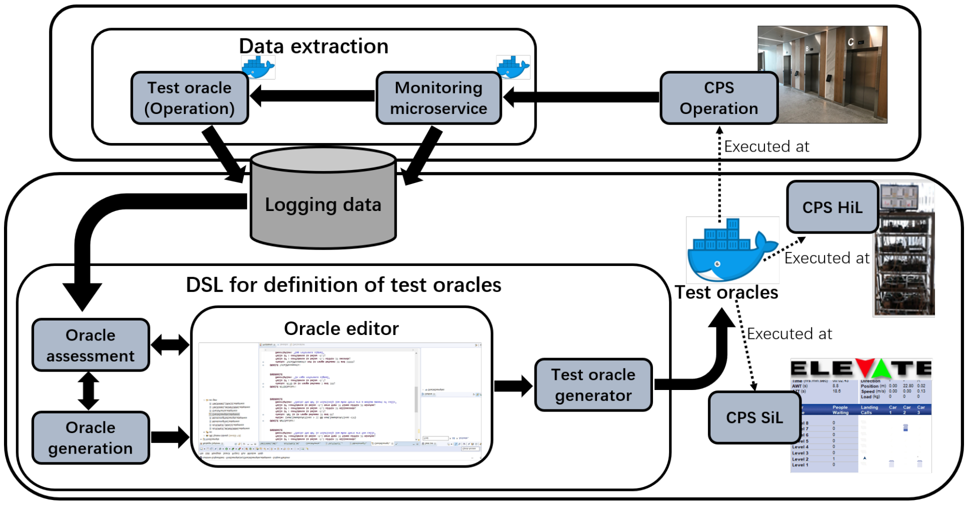


Fig. 2: Overview of the tool architecture for the automated generation of test oracles

should also include where the test oracles will be executed. The objective is to use these oracles in a streamlined way at different CPS DevOps testing levels (i.e., SiL, HiL test levels, or directly in operation).

1) Monitoring plan: In the upper side of Figure 2, the data extraction process for our test oracle is shown. This data extraction process is the same at all test levels (i.e., SiL, HiL and Operation), but in this case, we only show the operation side for simplicity. Test oracles need to monitor elevators' data. Such monitoring data is obtained through MQTT, and it is necessary to define test oracles. To do so, the monitoring plan file needs to be defined. Subsequently, we have developed a simple syntax in Xtext, where the monitoring plan's name is defined, and after that, each variable to be monitored. For each of these variables, their names, datatype and maximum and minimum values are specified.

2) Oracle definition: After defining the monitoring plan, oracles are defined for the CPS. Each CPS implements a monitoring plan, which means that it can access all data defined in the monitoring plan (previous section). Additionally, it is possible to include cardinality at this level, meaning that there is more than one CPS with the same information. For instance, for the installation in Figure 1, three lift controllers are available with the same information. The same oracles for the three elevators would be automatically triplicated by using this option.

Figure 3 shows an example of the DSL implementing the definition of two oracles for Orona's traffic master. All oracles shall specify a name. Each oracle can optionally have a precondition satisfying a boolean condition, expressed through a **when** or a **while** clause. The difference is that a **when** precondition refers to events of the system, and it later allows for expressing temporal-logic expression, such as the **after** (which asserts the **checks** expression after the specified time). On the contrary, the **while** precondition models states of the system and it only permits later to assert the **checks** expression during the same cycle(s) that the precondition is given.

```
CPS Orona2LiftsInstallations: implements ORONA_TRAFFIC_MASTER

    ORACLE StdbyECO1:
        when:(ECOMODE == 1 && Elevator1LogicPosition == Elevator2LogicPosition &&
        Elevator1Standby == 1 && Elevator2Standby == 0 && NewFloorCall ==1)
        after(500,milliseconds)
        checks: Elevator2AttendingStatus should be 1;
        fails if : confidence is below 0;
        Description: "When the ECO mode is active, the elevator that is
                      not in standby shall attend the call"
    ENDORACLE

    ORACLE AWTCheckerNonECO:
        while: (ECOMODE == 0 && NumberOfActiveCalls<20)
        checks: AWT is below 50;
        fails if : confidence is below -0.5;
        fails if : confidence is below -0.2 within 30 minutes;
        fails if :confidence is below -0.3 more than 5 times within 20 minutes;
        Description: "Checks the AWT is relatively low when there are few calls"
    ENDORACLE

ENDCPS
```

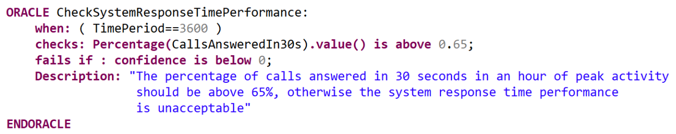
Fig. 3: DSL snippet example for our industrial case study

After the precondition, which is optional (i.e., there can be oracles that constantly assert data), the post-condition is asserted through the **checks** instruction. To define the assertion, we analysed the properties of the industrial case study and a total of six assertion patterns were defined. The **should be** pattern asserts that a signal is the same as the specified reference signal. The **should not be** pattern asserts that a signal is different from the specified reference signal. The **is below** pattern asserts that a signal is below the specified reference signal. The **is above** pattern asserts that a signal is above the specified reference signal. The **is in range between** ref_1 **and** ref_2 pattern asserts that a value is between the two specified references. Lastly, the **is not in range between** ref_1 **and** ref_2 pattern asserts that a value is above ref_1 or below ref_2.

Each of these data assertions are converted into a confidence level value every time the test oracle is triggered. The confidence level value ranges from -1 to 1. A positive value means that the property defined within the **checks** is asserted as “PASS”, whereas a negative value means that the property is being violated. However, by analysing the case study we noticed that certain violations could be accepted, especially those related to QoS measures. For instance, in the case of the $AWTCheckerNonECO$ oracle defined in Figure 3, having the AWT over 50 seconds for a small time window is not a reason for classifying a test as “FAIL”. To solve this issue, we used the confidence level and defined a total of three failing reasons, allowing this way some laxity in the confidence level in each failing reasons, expressed as **fails if** in the DSL. At least one failing reason shall be specified, but the three types could be used. Specifically, a test oracle can fail if: (1) its confidence value is below a certain threshold, (2) its confidence value is below certain threshold for long time, (3) its confidence value has been below certain threshold for N times or more in a given time window.

3) Uncertainty handling: To specify uncertainty related test oracles in industrial elevators, we implemented three uncertainty datatype libraries (i.e., *Probability library*, *Vagueness library* and *Ambiguity library*) based on an existing work [9]. The *Probability library* contains datatypes such as *Percentage* and various probability distributions (e.g., *Normal Distribution* and *Gamma Distribution*). In some cases, it is not possible to specify uncertainties as probabilities. For example, QoS-based passenger’s satisfaction defined in [10] is usually measured with fuzzy logic. Thus, we implemented data types related to fuzzy logic such as *Fuzzy set* and *Fuzzy Interval* in *Vagueness library*. Similarly, we implemented *Ambiguity library* for cases when probability and vagueness libraries are not adequate. For example, a specific traffic profile with a specific building configuration has an acceptable distribution of AWTs over a specific period (e.g., every 5 minutes) to achieve better user satisfaction. This means that AWTs per 5 minutes should not have too much uncertainty (e.g., AWTs per 5 minutes shouldn’t change frequently and drastically), which can be measured with *Shannon Entropy* implemented in *Ambiguity library*. Due to confidentiality reasons, we do not provide detailed information about uncertainties in the public repository.

Figure 4 presents an example of a test oracle using the uncertainty datatype *Percentage* in *Probability library* to test the elevator system’s response time performance, which is one of the QoS indicators defined in CIBSE Guide D [10].



```
ORACLE CheckSystemResponseTimePerformance:
    when: ( TimePeriod==3600 )
    checks: Percentage(CallsAnsweredIn30s).value() is above 0.65;
    fails if : confidence is below 0;
    Description: "The percentage of calls answered in 30 seconds in an hour of peak activity
                  should be above 65%, otherwise the system response time performance
                  is unacceptable"
ENDORACLE
```

Fig. 4: Uncertainty test oracle snippet example

4) Syntax validation, error handling and oracle assessment: The rules in Xtext validate that the syntax provided by the user is correct (e.g., all monitoring variables are available, they are in specific ranges). Another problem with test oracles is that they are prone to false positives, which we handled by considering data obtained from monitors in operation. For instance, for an elevator installation, if the overall AWT is around 30 seconds, but the test engineer specifies that the AWT shall be below 20 seconds, the tool warns test engineers of the possibility of a false positive being introduced.

B. Automated generation of test oracles

After modeling the test oracles in the DSL tool, the oracles are generated and compiled to be used in a DevOps environment for CPSs. Such test oracle generation encompass four steps, which are automated and streamlined in a Continuous Integration (CI) pipeline running in GitLab, providing an easy to use and transparent workflow for engineers from Orona.

1) Step 1 - Generation of oracle artifacts: DSL files are compiled to obtain test oracle artifacts ready to integrate within any compatible infrastructure. Two files in C code language are generated for each of the oracles defined in the DSL, including all the necessary functions to evaluate the execution of tests. Furthermore, other generic C code language files are generated, encompassing specific functions related to uncertainty and other utility functions (e.g., compute averages, free arrays). In addition, this generator provides a *.json file with all the necessary information for integrating these test oracles in compatible infrastructures.

2) Step 2 - Integration of test oracles: In this step, the previously generated files are integrated on top of a microservice template defined in another work [8]. The .c files generated for each oracle contain a pre-agreed function (defined in the *.json file) that needs to be called to generate verdicts. This function needs to be able to receive the values published by monitors (implemented through other microservices [8]) and be able to publish the generated verdict through MQTT (which are later used by other microservices [8]). Additional functionality must also be integrated, such as configuring the oracle through REST API calls and managing the execution status.

3) Step 3 - Microservice compilation and docker containerization: After the required source code has been consolidated and modified to fit the oracles, the compilation and Docker image generation phase takes place. The main motivation for the generation of Docker images is the portability across platforms that it provides, being able to use the microservice in different systems where the sole requirement is having Docker installed. In the case of Orona, the required portability goes further, having to provide the ability to execute the microservices across multiple architectures, as several nodes in Orona’s installations are running on ARMv7 and ARM64 nodes, whereas other nodes are running on AMD64 architectures. The docker image generation stage runs as part of an Ansible task, generating the docker image compatible with these architectures.

4) Step 4 - Upload to registry: After the newly created microservices have been made available on the Docker Registry, they are ready to be used as validation components in a DevOps workflow. In order to make the other components in the DevOps ecosystem installed at Orona aware of the existence of the microservice images and Oracles, entities have to be uploaded to a context broker (i.e., in our case, Stellio). The model in Stellio contains entities for the Oracle definitions, the Oracle microservice image definitions, CPSs in the DSL, and the inputs that the oracles support. These entities allow the test engineers to directly select available oracles when creating test suites through a Test as a Service (TaaS) tool.

IV. Evaluation and Lessons Learned

A. Evaluation with Industrial Case Study

In the first step of our evaluation, we carefully analyzed the documentation of the tests carried out manually in Orona. Those documents indicate the expected output (e.g., elevator 1 should attend call 1) for each test case. For all these expected outputs in the documents, we first aimed at understanding what the systems should do. After that, we confirmed with engineers whether our understanding was correct. When confirming, we developed test oracles with our DSL. For all the analyzed test cases, our DSL had expressiveness enough to model the test oracles. In total, we defined 37 test oracles in the DSL, and their code was automatically generated.

In the second step, we analyzed the different uncertainties that elevator systems are exposed to. This was done by (1) studying the CIBSE standard and (2) interviewing domain experts. After we carried out this analysis, we assessed whether the DSL could express and consider such uncertainties. We concluded that all the uncertainty to which elevators are exposed to can be modeled with our DSL.

B. Experiences and Lessons Learned

Reusability of Test Oracles with Abstraction and Automation. With our DSL test oracles are described at a higher level of abstraction independent of low-level technical details (*Abstraction*). Moreover, with test oracle generation, the code of specified test oracles is automatically (*Automation*) generated for any testing level. Thus, our DSL with the tool support provides an expressive and easy-to-use tool for our industrial partner to test oracle generation.

Continuous Evolution of Dispatcher based on Test Results. Based on the validation results from test oracles, the dispatcher can be continuously improved to ensure the desired QoS. Such an approach of continuously validating the dispatcher against specified test oracles even in operation is essential for continuous improvement of the quality of the dispatcher since industrial elevators remain operational for years. Thus, with such an approach, new behaviors of dispatchers in real operational conditions are learned and validated that are not possible during the design time.

Uncertainty-wise Testing. Given that uncertainty is present everywhere in industrial elevators, our DSL provides a tool to the developers to capture and assert uncertainties within the design-operation continuum of industrial elevators and at various testing levels. Consequently, our DSL offers a tool for elevator developers to consider uncertainty explicitly during testing, which is not common practice, thus enabling them to handle uncertainties systematically in DevOps.

V. Conclusion and Future Work

We presented the industrial application of a domain-specific language precisely designed to specify test oracles and automatically generate their code in the context of industrial elevators development in Company. The evaluation results with the industrial case study, including interviews with the domain experts at Orona, suggest that our DSL is expressive enough to specify test oracles for this domain. Thanks to this tool, several tasks that were manual before can be automated at Orona. Based on our acquired knowledge, we reported our experiences and lessons learned that are beneficial for researchers and practitioners. Our next step is to investigate the technology transfer of the DSL to Orona such that the DSL can be integrated in their day-to-day work. Moreover, in the future, we plan to evaluate to the tool in other CPS domains and get their feedback to further improve our tool.

Acknowledgment

This publication is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871319.

References

[1] P. Derler, E. A. Lee, and A. S. Vincentelli, "Modeling cyber–physical systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 13–28, 2011.

[2] J. Ayerdi, S. Segura, A. Arrieta, G. S. Arratibel, and M. Arratibel, "Qos-aware metamorphic testing: An elevation case study," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 104–114.

[3] J. Ayerdi, A. Garciandia, A. Arrieta, W. Afzal, E. Enoiu, A. Agirre, G. Sagardui, M. Arratibel, and O. Sellin, "Towards a taxonomy for eliciting design-operation continuum requirements of cyber-physical systems," in *2020 IEEE 28th International Requirements Engineering Conference (RE)*. IEEE, 2020, pp. 280–290.

[4] C. Menghi, S. Nejati, K. Gaaloul, and L. C. Briand, "Generating automated and online test oracles for simulink models with continuous and uncertain behaviors," in *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 27–38.

[5] A. Arrieta, J. A. Agirre, and G. Sagardui, "A tool for the automatic generation of test cases and oracles for simulation models based on functional requirements," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2020, pp. 1–5.

[6] A. Cretin, B. Legeard, F. Peureux, and A. Vernotte, "Increasing the resilience of atc systems against false data injection attacks using dsl-based testing," in *International Conference on Research in Air Transportation*, 2018.

[7] A. Vernotte, A. Cretin, B. Legeard, and F. Peureux, "A domain-specific language to design false data injection tests for air traffic control systems," *International Journal on Software Tools for Technology Transfer*, pp. 1–32, 2021.

[8] A. Gartziandia, J. Ayerdi, A. Arrieta, S. Ali, T. Yue, A. Agirre, G. Sagardui, and M. Arratibel, "Microservices for continuous deployment, monitoring and validation in cyber-physical systems: an industrial case study for elevators systems," in *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2021, pp. 46–53.

[9] M. Zhang, S. Ali, T. Yue, R. Norgren, and O. Okariz, "Uncertainty-wise cyber-physical system test modeling," *Software & Systems Modeling*, vol. 18, no. 2, pp. 1379–1418, 2019.

[10] G. Barney, *Transportation systems in buildings : CIBSE Guide D: 2010*. London: Chartered Institution of Building Services Engineers, 2010.