Software and Systems Modeling
https://doi.org/10.1007/s10270-025-01271-8

SPECIAL SECTION PAPER

Defining and generating multi-level and uncertainty-wise test oracles for cyber-physical systems

Pablo Valle[1] · Aitor Arrieta[1] · Liping Han[2] · Shaukat Ali[2] · Tao Yue[2]

Received: 22 March 2024 / Revised: 8 January 2025 / Accepted: 23 January 2025
© The Author(s) 2025

Abstract

Cyber-physical systems (CPSs) blend digital and physical processes. CPS software is the key to realizing their functionalities. This software needs to evolve to deal with different aspects, such as the implementation of new functionalities or bug fixes. Because of this, design–operation methods, colloquially known as "DevOps," are paramount to be adopted within these systems. During DevOps phases, automating test execution at design time is a key enabler of streamlined software development and software quality improvement. Likewise, monitoring whether a CPS is behaving as expected at operation is similarly important. In DevOps, test oracles play an important role in enabling automated testing, ensuring the reliability of software deployments, providing feedback to developers, etc. However, defining and generating test oracles in the context of DevOps practices in CPSs need to accommodate aspects specific to CPSs, such as their time-continuous behavior and inherent uncertainties. To this end, in this paper, we propose a domain-specific language (DSL) to ease the definition of test oracles and an automated solution for generating a microservice encapsulating the defined test oracles, which is compatible with a DevOps ecosystem for CPSs. We evaluated our DSL with two industrial case study systems and 9 open-source CPSs. Our evaluation results suggest that our DSL can model around 98% of the requirements of these systems through test oracles. Furthermore, it is possible to generate a microservice to be applicable at different test levels within less than 20 min, being fast enough to be adopted in practice.

Keywords Domain-specific language · Test oracle generation · Cyber-physical systems · Software testing

1 Introduction

Cyber-physical systems (CPSs) integrate digital technologies with physical processes [1–3]. The software of these systems provides their key functionalities. Usually, the software monitors the physical states of the system through sensors, executes its algorithms, and changes the system state through actuators. Same as for any other software system, the CPS software evolves throughout its life cycle to deal with hardware obsolescence, the addition of new functionalities, and fixing bugs [4]. Hence, design–operation methodologies, also colloquially known as "DevOps," have emerged to help software engineers accelerate software development while increasing its quality. However, adopting these methods in CPS context faces several challenges, especially due to their multifaceted and diversified nature. For instance, Zampetti et al. [5] identified challenges related to CPS simulators, test flakiness, test execution, automated oracle creation, and the need for specific fault models.

In this paper, we propose a tool supported by a DSL to create test oracles for CPSs which can be used across different testing levels and in operation. We aim at targeting the challenge of creating test oracles, which are the mechanisms that allow us to automatically determine whether a system is behaving as expected. As identified by Zampetti et

Communicated by Javier Troya and Alfonso Pierantonio.

✉ Aitor Arrieta
aarrieta@mondragon.edu

Pablo Valle
pvalle@mondragon.edu

Liping Han
hanliping93@gmail.com

Shaukat Ali
shaukat@simula.no

Tao Yue
taoyue@gmail.com

1 Mondragon University, Mondragon, Gipuzkoa, Spain

2 Simula Research Laboratory, Oslo, Norway

al. [5], the CPS execution environment significantly increases the complexity of defining test oracles. For instance, at different testing levels (e.g., Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL)), changes in the execution environment make test oracles designed for one level inapplicable to another. Furthermore, CPSs must deal with uncertainties in operation [6, 7], such as noise in signals and environmental uncertainty. Therefore, it is necessary to consider different uncertainty factors if we want to reuse test oracles as runtime monitors and supervisors in operation to take corrective actions in case the new software version provokes failures and recovery actions are needed to consider, closing this way, the loop between design and operation.

This complexity mandates considering multiple dimensions [5]: (i) the specific test scenario or requirement under evaluation; (ii) the degree of realism required in simulations; (iii) the development stage or maturity of the hardware proxies incorporated into the evaluation pipeline; and (iv) the nature of the output sources, which may derive from genuine sensor data or employ mocked/synthetic data. Furthermore, test oracles in the CPS context often encompass value ranges, such as time intervals, rather than discrete scalar values. They may involve signal processing to interpret complex outputs, as underscored by existing research in CPS testing [8, 9]. Including non-functional attributes, notably those related to timing, emerges as critical [10].

To deal with these challenges, we developed a domain-specific language (DSL)-based tool that allows the easy definition of test oracles for CPSs. The tool has a simple language with most of the properties that test oracles need when verifying the correct functioning of a system. After defining the test oracle using this DSL, the tool automates the generation of relevant source files and the generation of docker containers. These docker containers allow encapsulating test oracles as a single microservice compatible with a DevOps ecosystem for CPSs [11], permitting, this way, the interoperability across the different test levels as well as operation. Our tool encompasses the following key characteristics that, although some papers have addressed some of them individually (e.g., [12–15]), to the best of our knowledge, none has addressed them together:

- *Streamlined multi-level test oracles*: The generated test oracles are reusable across test levels as well as in operation, enhancing the software testing process's consistency and efficiency across both design time testing and operational monitoring of the system functionalities, colloquially known as DevOps. We enable this thanks for integrating containerized approaches that leverage dockers that can be reused across the different development levels.
- *Support for time-continuous behaviors*: CPSs provide data over time, requiring not only asserting that the system is behaving correctly at a specific instant but also considering evolving signals over time.
- *Need for considering uncertainty*: CPSs have several uncertainty sources, such as sensor noise, communication delays, and hardware delays, which need to be considered by test oracles, especially when the CPSs are deployed in operation. This aspect is vital when testing the CPS in latter development stages (e.g., communication loses at the HiL test level) as well as when monitoring the CPS in operation, such as uncertainty of the CPS environment (e.g., passengers' weights in elevators [16]). This feature is important to support and adapt DevOps methodologies to the context of CPSs.
- *Quantitative test verdicts:* CPS verdicts are not typically reported as Boolean: "Pass" or "Fail" values, but they typically provide a robustness degree (when the system is operating correctly) and severity degree (when the system fails). For instance, in the context of autonomous driving systems (ADS), a failure related to a collision is considered more severe if the collision occurs when the vehicle is driving at a higher speed. In the case of *passing* verdicts, quantitative verdicts can give information to test case generators to see how close they are to falsifying a requirement.
- *Support for synthetic data*: CPSs need often deal with the test oracle problem [17–20], i.e., given a test input, it is not evident to determine what the test output should be [17, 18]. Because of this, our DSL can integrate machine learning-based techniques to estimate CPS values based on previous operational data, similar to our previous work [17, 18].
- *Feedback from operation*: Test oracles are prone to false positives. We deal with this problem by considering operational data when defining test oracles. When we define a test oracle, our tool checks whether data from the operation of a CPS complies with the defined property, warning test engineers when defining test oracles that will likely lead to false positives.

This paper builds upon our previous work [21] from two main aspects: First, we give significantly further details of the tool implementation and its capabilities, addressing with concrete examples the particularities we are addressing. Second, we conduct a thorough evaluation of the tool with one more industrial CPS and 9 open-source CPSs.

The rest of the paper is structured as follows. In Sect. 2, we explain a motivating industrial case study system that we have used to study the key properties of CPSs for deriving our DSL for the test oracle definition. Afterward, we provide a high-level overview of the developed tool (Sect. 3). We evaluate our approach in Sect. 4. We position our work with closely related studies in Sect. 5. We conclude our paper in Sect. 6.

2 Motivating industrial case study system

We conducted four meetings with engineers from Orona and one meeting with engineers from Alstom to obtain as much information as possible regarding faulty behaviors of CPSs, key properties to check, tool capabilities needed to ease the test oracle definition, and specific requirements for the integration with existing testing frameworks. To generate these findings, we further analyzed commonalities and differences across several open-source benchmarks representing diverse domains such as automotive, aerospace, and health care. The goal was to ensure that the DSL could accommodate a broad spectrum of CPS testing needs, from simple actuator–sensor interactions to complex, distributed systems with intricate dependencies. The DSL is general to CPSs of various domains. Below we present an industrial elevator system to motivate the necessity of developing the DSL for test oracle definition.

Figure 1 shows a simplified overview of our industrial case study provided by Orona, a leading elevator company in Europe. In such a system, different devices communicate through different communication protocols to transport passengers safely and provide the best quality of service (QoS) possible. When a passenger calls a lift through the user interface, the traffic master receives information about the call. The traffic master system decides which elevator should attend each call. In conventional installations, the traffic master only receives information related to the call direction (i.e., going up or down), needing to deal with different uncertainties (e.g., number of passengers waiting on the floors, passengers' loading and unloading time). In destination selection installations, the traffic master receives the user's final destination. In addition, in more complex installations, certain restrictions are applied. For instance, in some hotel buildings, certain passengers can only go to certain floors. When such restrictions are applied, the traffic master communicates through Ethernet with an access control server to decide the most appropriate lift to assign a passenger. After the call has been assigned, the assignment is sent to the corresponding lift controller. The lift controller performs the low-level elevator control, including speed, acceleration, and door opening/closing.

The software of these systems is constantly changed to support (1) bug corrections, (2) adaptation to legislation changes, and (3) including new functionalities [4]. Every time the code is modified, Orona has a well-established software testing process, where simulation-based testing is the dominant technology. Different test levels are carried out. For instance, when testing the dispatching algorithm, the first test level is the SiL level. In this 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 HiL level. During the HiL test level, all infrastructures related to the controllers are real, whereas the physical part (i.e., mechanical elements like the elevator shaft, doors' models and engines) is emulated. After the HiL tests have been accomplished, the software module is compiled and manually deployed by the maintainer during the installation. When the software is deployed, the maintainer performs manual tests to ensure everything works 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 bugs before the code is in production, there is a need for higher test automation to establish a fully design–operation continuum software development method. This would allow for not requiring any human intervention to fully deploy a new software version in operation, which will lead to a drastic reduction of both economic and temporary resources and subsequently enhance the quality of software development for CPSs.

3 Tool-supported method

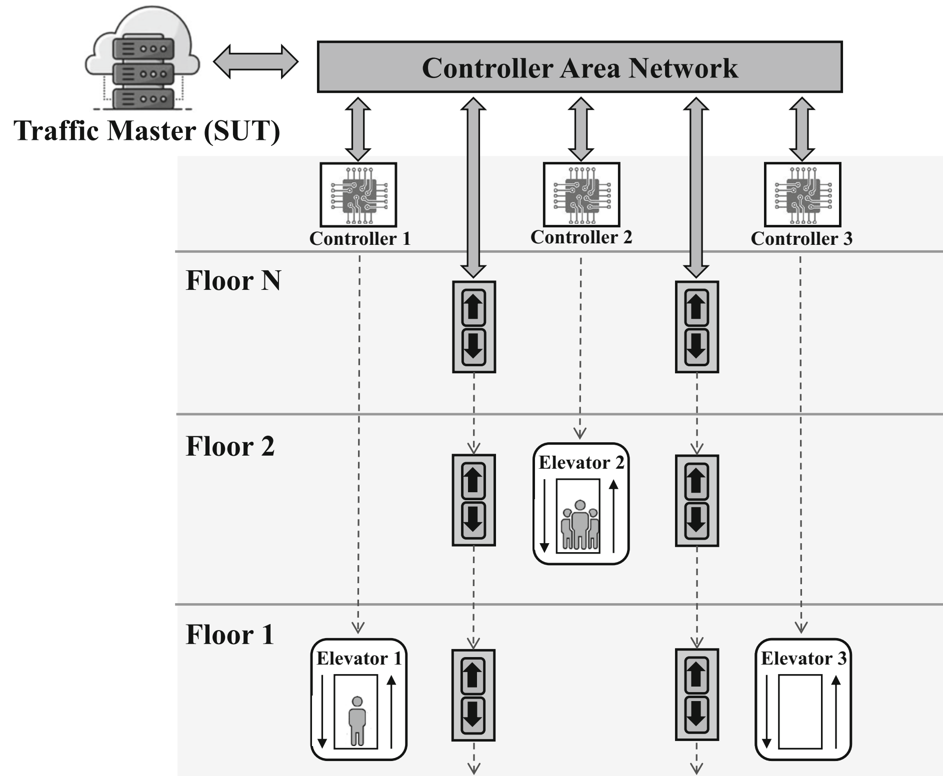
Figure 2 shows an overview of the framework and how it is integrated within the DevOps fashion within CPSs. The DSL is connected to *the oracle assessment module*, which obtains logging data from the CPS in operation and allows the framework to give feedback to users who implement test oracles with the DSL.

The microservice generation module automatically generates test oracles through genetic programming [19] (not tackled in this paper), and it automatically generates code from the specified test oracles. Specifically, it generates a dockerized microservice to ensure that the oracles can be reused across different levels (SiL, HiL) as well as in operation.

3.1 DSL for defining test oracles

A well-known framework was used to develop the DSL, Xtext [23]. This framework is aimed at developing programming languages as well as DSLs. A full infrastructure is provided for the language being developed, including a parser, a linker, a type checker, and a compiler. Xtext is prepared so it is relatively easy to customize the type checker and the compiler. Taking advantage of this feature, on the one hand, we have enhanced the type checker with semantic validations and a false positive detection module (described in Sect. 3.2). Notice that in this context a false positive is a failure being revealed without needing to have been revealed (i.e., the system is behaving correctly but the oracle raises an

Fig. 1 Overview of the industrial case study system [22]



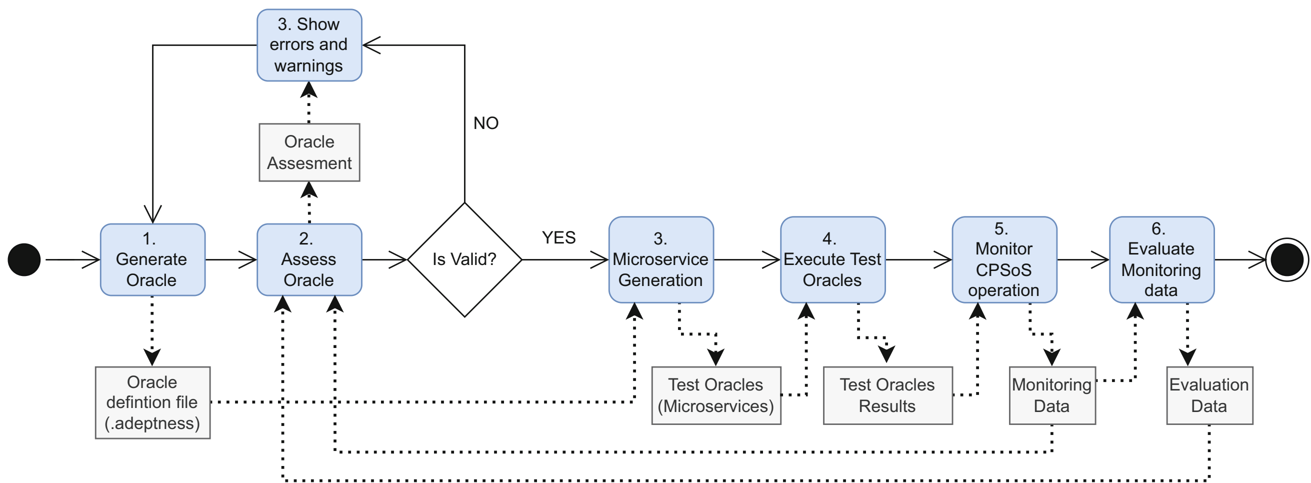


Fig. 2 Overview of the tool architecture for the automated generation of test oracles

issue). On the other hand, code for the test oracles microservices is generated through the compiler. This language is carefully designed so that any textual description of a concrete syntax is mapped to an in-memory representation of the derived semantic model. Therefore, after parsing the syntax of the DSL, which is conducted on the fly, a model can be used within other parts of the infrastructure, such as the type checker or the compiler.

The DSL aims to be expressible enough to characterize CPS requirements and properties. Figure 3 presents the DSL, which is divided into two main parts: the *MonitoringFile* and the *CPS*, both of which inherit component *Type*. Component *Type*, in turn, is at the same level as the *PackageDeclaration* and *Imports* components. Each part could be developed in a separate file. The CPS component is intended for the description of the oracles, and *MonitoringFile* is intended for the declaration of signals required by the oracles. *MonitoringFile* consists of a set of *MonitoringPlan*, which could be imported from an oracle definition file, and as can be seen in the figure, is referenced from the CPS component. There-

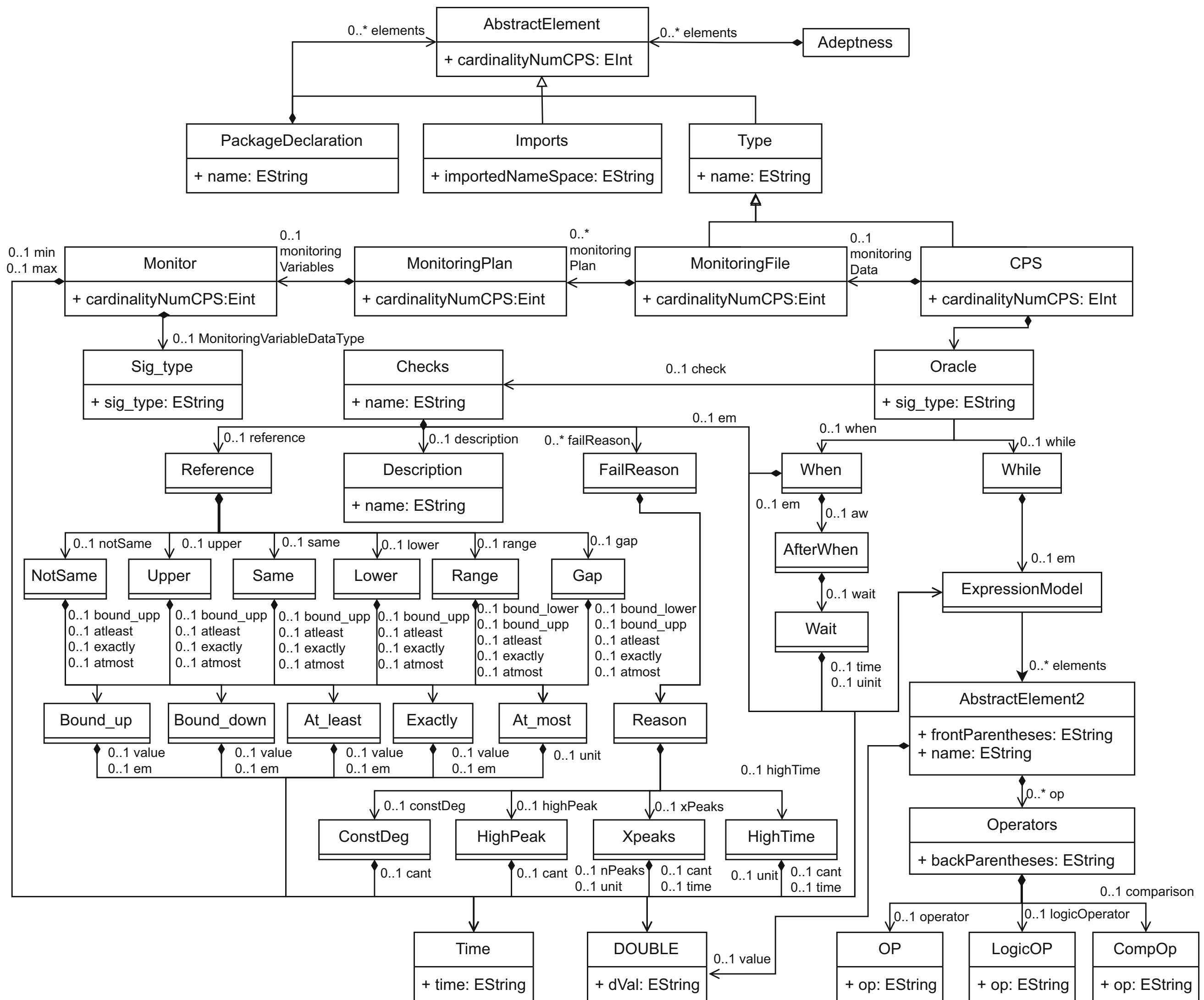


Fig. 3 DSL specification in UML class diagram notations

fore, if a monitoring plan is defined in a separate file, it could be used in several oracles' definition files. Further details are provided in the following sections.

3.1.1 Monitoring plan

As illustrated in Fig. 2, to perform validation, data from the CPS operation needs to be fed to the test oracle, via the *monitoring microservice* through an MQTT communication process. More details can be found in [11]. To define test oracles, it is necessary to identify available monitoring variables beforehand in the *monitoring plan file*. The file specifies the name of the monitoring plan, and the name, data type, and maximum and minimum values of each declared variable. The maximum and minimum values are only needed if the data type is numerical: i.e., double, as shown in Fig. 3.

For instance, as shown in Fig. 4a Line 1, the monitoring plan's name (i.e., *ORONA_TRAFFIC_MASTER*) is defined, followed by the specification of each variable being monitored (Lines 2-14), including its name (e.g., *AWT*), type (e.g., *int*), and maximum and minimum values (e.g., *500* and *0*). Having a monitoring file defined for a CPS gives the user a clear understanding of variables available for the test oracles and a unique source of trust. In addition, the oracle validation process uses this file to detect variable-related errors such as naming errors and errors related to signal boundaries.

The most common case is that the monitoring microservice directly provides the monitoring variables. However, variables can be derived such as synthetic monitoring variables, each of which predicts the outcome of an unobserved monitoring variable and can be used by the oracles in a way similar to the directly measured variables. Using state-of-the-art regression techniques and relying on previously gathered

```
1⊖ MONITORINGPLAN ORONA_TRAFFIC_MASTER:
2⊖     MONITOR ECOMODE:
3          type: boolean
4      ENDMONITOR
5⊖     MONITOR AWT:
6          type: int
7          max: 500
8          min: 0
9      ENDMONITOR
10⊖    MONITOR ATTD:
11         type: int
12         max: 1000
13         min: 0
14     ENDMONITOR
15 ENDMONITORINGPLAN
```

(a) Exemplifying the monitoring plan for a traffic master CPS of the elevator case study

```
11 MONITOR_INFERENCE Energy:
12     type: double
13     max: 1000
14     min: 0
15     model: 'dnn_model'
16     independent variables :[AWT ATTD]
17 ENDMONITOR_INFERENCE
```

(b) Exemplifying synthetic monitoring variables

Fig. 4 Exemplifying monitoring plan

data, synthetic monitors can provide accurate predictions [17, 18]. Figure 4b exemplifies the syntax of specifying synthetic monitoring variables in the monitoring plan. The syntax is similar to the one for variables measured by the monitoring microservice. But it also includes a list of the independent variables Fig. 4b Line 16 (e.g., *AWT* and *ATTD*) used to predict the synthetic monitoring variable, which is the dependent variable of this regression problem. It also requires a pre-trained machine learning model to be defined.

3.1.2 Test oracle definition

After defining the monitoring plan, test oracles are defined. The CPS implements the monitoring plan, meaning that the CPS has the access to all data defined in the monitoring plan. Additionally, multiple CPSs can have the same oracles. For instance, for the installation in Fig. 1, three lift controllers have the same oracles, which would be automatically triplicated, if we want to independently model the oracles for each elevator. As shown in Fig. 3, the CPS component inherits the name attribute from *Type*, includes a *MonitoringFile*, and may be composed of several *Oracles*.

In terms of oracles, as shown in Fig. 3, there are components for specifying precondition, which could be *When* or *While*, and the *Check* component for gathering the assertion definition. The *Check* component is a conjunction of the signal to be asserted (modeled by the component *ExpressionModel*), an assertion pattern and a reference signal (modeled by component *Reference*), a failure reason (modeled by the *FailReason* component), and, finally, a *Description*. The following sections detail the preconditions, the assertion, and the reason for failures. Figure 5 shows an Oracle snippet modeled through our DSL, which defines two oracles for Orona's traffic master: StdbyECO1 and StdbyECO2.

Preconditions

A precondition is optional, as there can be oracles that constantly assert data. A precondition is a Boolean condition expressed through a while or when clause. The difference is that the while precondition specifies system states that must hold true continuously while the assertion is checked, requiring the assertion to be checked and asserted during the same cycle(s) the condition is evaluated. On the contrary, the when precondition specifies an event, once the event occurs, the assertion is checked without needing to continuously assert the condition. This difference becomes significant for temporal conditions, as when precondition also supports after temporal logic expressions, enabling the oracle to pause for a specified duration before the assertion is evaluated.

As shown in Fig. 3, both preconditions are composed by an *ExpressionsModel*, enabling the creation of an expression that can be evaluated to be true or false. Although the DSL allows for different types of expressions, this constraint is enforced within the syntax validation process (Sect. 3.2). The *When* clause specifies a temporal condition: *Wait*, composed of a time value and a time unit.

For instance, as shown in Fig. 6, the textual description of the first requirement *R1* (Line 22) could be formalized this way: G{0,T} (standby == 0 and apfail == 0 and supported == 1 and limits == 1 then pullup == 1). Thus, any time the precondition "standby == 0 and apfail == 0 and supported == 1 and limits == 1" is given the assertion, i.e., "pullup == 1," must be checked. In this case, either a when or a while clause could be used. However, the second requirement *R2* can be formalized as: G{0,T}(STATE== 0 and standby == 1 then STATE == 3), being STATE = 0 representing the STATE variable being in TRANSITION, and STATE=3 representing the STATE variable being STANDBY. This time, whenever the precondition is given, STATE == 0 and standby == 1, it is important to assert that the STATE variable is 3, not within the same cycle but the following one. This is modeled through a when precondition along with an after clause, which allows the oracle to wait a cycle when a precondition is given and, then, check the assertion (see Fig. 6 Lines 28-35).

Assertion

The assertion is the main condition of the oracle. It is decisive for a test to fail or succeed. However, if a precondition is set, the evaluation of the assertion completely depends

Fig. 5 Exemplifying oracles modeled with the DSL

```
3⊖CPS Orona2LiftsInstallations: implements ORONA_TRAFFIC_MASTER
4
5⊖   ORACLE StdbyECO1:
6⊖       when:(ECOMODE == 1 && Elevator1LogicPosition == Elevator2LogicPosition &&
7        Elevator1Standby == 1 && Elevator2Standby == 0 && NewFloorCall ==1)
8        after:500 milliseconds
9⊖       checks: Elevator2AttendingStatus shall be 1
10       fails if : assurance is below 0
11⊖      description: "When the ECO mode is active, the elevator that is not in
12                     standby shall attend the call"
13   ENDORACLE
14
15⊖  ORACLE StdbyECO2:
16⊖      when:(ECOMODE == 1 && Elevator1LogicPosition == Elevator2LogicPosition &&
17       Elevator1Standby == 1 && Elevator2Standby == 1 && NewFloorCall ==1)
18       after:500 milliseconds
19⊖      checks: Elevator1AttendingStatus shall be 1
20       fails if : assurance is below 0
21⊖      description: "When the ECO mode is active, if both elevators are in standby and
22                     there is a floor call, the first elevator shall attend the call"
23   ENDORACLE
```

Fig. 6 Snippet of oracles using preconditions

```
17   // R1 -> G{0,T} (not standby and not apfail and supported and limits ==> pullup)
18   ORACLE R1:
19       while: standby == 0 && appfail == 0 && supported == 1 && limits == 1
20       checks: pullup shall be 1
21       fails if: assurance is below 0
22       description: "Exceeding sensor Limits shall latch an autopilot pullup when
23                     the pilot is not in control (not standby) and the system is
24                     Supported without failures (not Apfail)."
25   ENDORACLE
26
27   // R2 -> G{0,T} (STATE = 0 and sntandby ==> STATE=3)
28   ORACLE R2:
29       while: STATE == 0 && standby == 0
30       after: 1 seconds // 1 cycle = 1 second
31       checks: STATE shall be 3
32       fails if: assurance is below 0
33       description: "The autopilot shall change states from TRANSITION to STANDBY
34                     when the pilot is in control (Standby)."
35   ENDORACLE
```

on the precondition, as previously mentioned. To define the assertion, CPS properties are analyzed. As a result, the main assertion of the oracle is composed of a signal to be evaluated, a comparison pattern, a signal to compare (we refer to this in the remainder of the paper as the reference signal), and, optionally, a temporal condition that determines the duration at which the assertion must be checked. The latter only makes sense if a precondition was specified for the oracle.

Figure 7 illustrates the six patterns we identified through our conversation with industrial CPS practitioners (including Orona's engineers and other CPS developers) and by reviewing the state-of-the-art CPS testing papers. The red lines stand for reference signals and denote the boundaries of the gray area. The signal being tested should be placed inside this area for the oracle to pass the test.

As shown in Fig. 3, the signal to be assessed is defined through *ExpressionModel*, which is constrained to express a signal by the syntax validation (Sect. 3.2) as has been done with the precondition. The DSL also captures comparison patterns and the reference signal. The six patterns (illustrated in Fig. 7) are modeled as *Same*, *NotSame*, *Upper*, *Lower*, *Range*, and *Gap*. The reference signal can be modeled as *Bound_up* or *Bound_down*. However, for the *Range* or Gap patterns, both types of the reference signals are necessary. Temporal conditions for the assertion could be established through *At_least*, *At_most*, and *Exactly* of the DSL.

Delving deeper into the patterns, on the one hand, there are the *Same* and *NotSame* patterns. The former asserts that a value is the same as the specified reference, while the latter asserts the opposite. In the DSL, they are expressed through shall *ref* and shall not be *ref* syntax, respectively, being *ref* as a reference signal. The above patterns fit perfectly to model two requirements of the "Autopilot" CPS from [24], as illustrated in Fig. 8a.

The description of the requirement *R1_1* is formalized as: "G{0,T} (APEngPrev==0 AND APEng==1 then AilCmd!=0)," where *APEngPrev* is a variable that specifies whether the autopilot is engaged or not, and *AilCmd* represents the command to the roll actuator. The description for the R1_2 requirement is: "When the autopilot is engaged and no other lateral mode (HDG Mode, ALT Mode) is active, then roll hold mode shall be the active mode," which is formalized as follows [24]: "G{0,T} (APEng == 1 AND HDGMode == 0 AND ALTMode == 0 then isRoll == 1)" being APEng a vari-

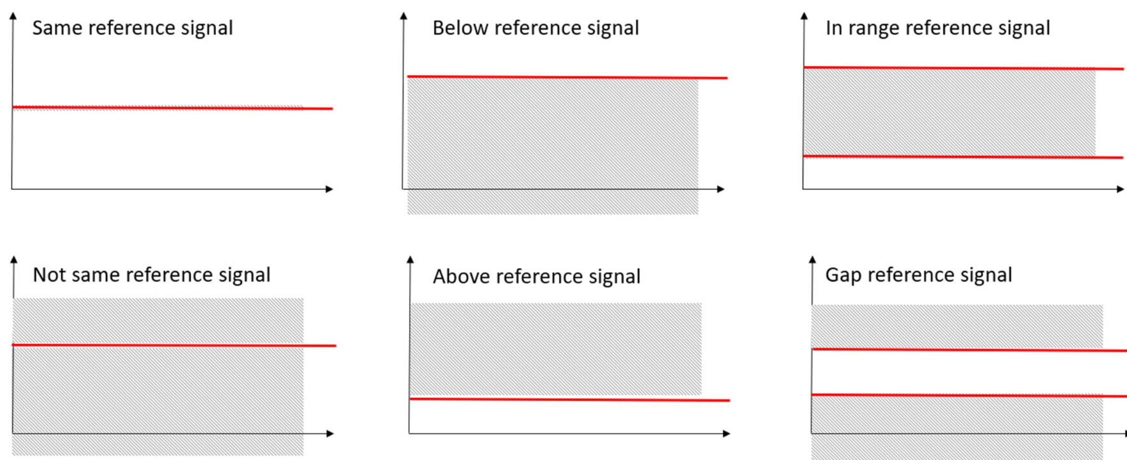


Fig. 7 Oracle assertion patterns

able that specifies whether the autopilot is engaged or not, and isRoll specifies if roll hold mode is active or not. Similar to the previous requirement, this requirement was also modeled with the DSL, as the equal assertion maps to a shall pattern.

There are also the *below* and *above* patterns, which assert that a value is *below* or above the specified reference, respectively. With the DSL, these patterns are expressed through is below *ref* and is above *ref* syntax, respectively, being *ref* as a reference signal. Figure 8b illustrates the modeling notations of these two patterns.

Finally, there are the *Range* and *Gap* patterns. The *Range* pattern asserts that a value is between two specified references, whereas the *Gap* pattern asserts that a value is not between two specified references. These two patterns are similar to the *Same* and *NotSame* patterns, but with two boundaries defined instead of one. With the DSL, the *Range* and *Gap* patterns are expressed through is in range between *ref1* and *ref2* and not in range between *ref1* and *ref2* syntax, respectively. Figure 8c illustrates the usage of the syntax.

Recall that if a precondition is specified, one of these three temporal conditions shall be defined, affecting the time at which the assertion pattern is evaluated: at least *time unit*, at most *time unit*, or exactly *time unit*. If an at least temporal condition is set, the assertion is checked during the specified duration. If an at most temporal condition is set, the pattern must assert at least one cycle, but not longer than the specified duration. If an exactly temporal condition is set, the pattern must assert during the specified duration and not any longer. Figure 9a (Line 15), b (Line 16), and c (Line 15) shows an example of each of the three temporal conditions, respectively.

Failure reason. Each of the data assertions is converted into a assurance level value each cycle an assertion is evaluated. The assurance level value ranges from -1 to 1 and 2. A value between 0 and 1 means that the property defined within the checks is asserted as "PASS," 1 meaning that the CPS is behaving as robust as possible. In contrast, a negative value means that the property is being violated, asserting the verdict as "FAIL," and -1 being the worst case scenario. The 2 assurance value refers to an oracle not fulfilling the specified precondition, so the property is not being checked (i.e., inconclusive). The assurance value is obtained differently for each of the type of data assertions. Equations 1, 2, 3, 4, 5, and 6 provide the assurance values for each of the six defined assertion types, i.e., same, not same, below, above, in range, and gap, respectively. *Sign* stands for the signal under check, *min* and *max* stand for the minimum and maximum values of the signal, respectively, and *ref*, *upperRef* and *lowerRef* stand for a reference signal of the pattern; in the cases where there is only one reference signal, *ref* is used; otherwise, *upperRef* is used for the upper bound reference signal and *lowerRef* for the lower bound reference signal.

$$conf(t) = \begin{cases} \frac{sign-ref}{ref-min} & \text{if}(sign < ref), \\ \frac{ref-sign}{max-ref} & \text{if}(sign > ref), \\ 1 & \text{if}(sign == ref) \end{cases} \quad (1)$$

$$conf(t) = \begin{cases} \frac{ref-sign}{ref-min} & \text{if}(sign < ref), \\ \frac{sign-ref}{max-ref} & \text{if}(sign > ref), \\ -1 & \text{if}(sign == ref) \end{cases} \quad (2)$$

$$conf(t) = \begin{cases} \frac{ref-sign}{ref-min} & \text{if}(sign < ref), \\ \frac{ref-sign}{max-ref} & \text{if}(sign \geq ref) \end{cases} \quad (3)$$

$$conf(t) = \begin{cases} \frac{sign-ref}{max-ref} & \text{if}(sign > ref), \\ \frac{sign-ref}{ref-min} & \text{if}(sign \leq ref) \end{cases} \quad (4)$$

Fig. 8 Snippets of oracles using various assertion patterns

```
7     // R1.1 -> G{0,T} ( APEngPrev=0 AND APEng=1 ==> AilCmd~=0 )
8     ORACLE R1_1:
9         while: (APEngPrev==0 && APEng==1)
10        checks: AilCmd shall not be 0;
11        fails if :  assurance is below 0;
12        description: "When the pilot selects the autopilot engage switch in the
13                      cockpit then roll autopilot shall engage and when the switch is
14                      deselected then the autopilot shall disengage and when is not
15                      engaged then te command to the roll actuator shall be zero."
16    ENDORACLE
17
18    // R1.2 -> G{0,T} ( APEng = 1 AND HDGMode = 0 AND ALTMode = 0 ==> isRoll == 1)
19    ORACLE R1_2:
20        while: (APEng==1 && HDGMode==0 && ALTMode==0)
21        checks: isRoll shall be 1;
22        fails if :  assurance is below 0;
23        description: "When the autopilot is engaged and no other lateral mode (HDG mode,
24                      ALT Mode) is active Then roll hold mode shall be the active mode."
25    ENDORACLE
```

(a) Oracles with the *shall* and *shall not be* assertion patterns

```
12    //R1 -> G{0,T} ( z <= 1.1 )
13    ORACLE R1:
14        checks: z is below 1.1
15        fails if: assurance is below 0
16        description: "The maximum value of the NN output, z, shall
17                      always be less than or equal to 1.1, regardless
18                      of the input values"
19    ENDORACLE
20
21    //R2 -> G{0,T} ( z >= -0.2 )
22    ORACLE R2:
23        checks: z is above -0.2
24        fails if: assurance is below 0
25        description: "The maximum value of the NN output, z, shall
26                      always be greater than or equal to -0.2, regardless
27                      of the input values"
28    ENDORACLE
```

(b) Oracles with the *below* and *above* assertion patterns

```
12    //DoorPositionChecker -> G{0,T} ( DoorAtElevatorFloor == 1 ==> 0 <= DoorPosition <= 3 )
13    ORACLE DoorPositionChecker:
14        while: (DoorAtlElevatorFloor == 1)
15        checks: DoorPosition is in range between 0 and 3
16        fails if: assurance is below 0
17        description: "If the elevator is on the floor of the door, the door could be
18                      closed, opened, closing or opening"
19    ENDORACLE
20
21    //PrivilegeChecker -> G{0,T} ( PrivilegedUser == 0 && ElevatorDestination != -3
22    //                       ==> ElevatorDestination < 2 and 4  < ElevatorDestination)
23    ORACLE PrivilegeChecker:
24        while: (PrivilegedUser ==0 && ElevatorDestination != (-3) )
25        checks: ElevatorDestination not in range between 2 and 4
26        fails if: assurance is below 0
27        description: "If the user does not have privileges, the elevator's destination
28                      could not be floors between 2 and 4."
29    ENDORACLE
```

(c) Oracles with the *Range* and *Gap* assertion patterns

Fig. 9 Snippets of oracles using the three temporal conditions

```
13  ORACLE DoorOpeningStatus:
14      when: (Elevator1DoorStatus==1 && Elevator1DoorSensor ==1)
15      checks: Elevator1DoorStatus shall be 1 at least 2 seconds
16      fails if: assurance is below 0
17      description: "If a door is opened and an object has been detected in the door,
18                   the door shall keep being opened for the next 2 seconds."
19  ENDORACLE
```

(a) Oracle with the "at least" temporal condition

```
13  ORACLE TimeToActivate:
14      when: (Elevator1AttendingStatus==1 && Elevator2AttendingStatus ==1
15              && Elevator1Standby == 1 && NewFloorCall ==1 )
16      checks: Elevator1Standby shall be 0 at most 2 seconds
17      fails if: assurance is below 0
18      description: "Check the time needed by elevator 1 to activate from
19                   stand by mode, it cannot be higher than 2 seconds"
20  ENDORACLE
```

(b) Oracle with the "at most" temporal condition

```
13  ORACLE OpenDoorsIfObject:
14      when: (Elevator1DoorSensor==1 && Elevator1DoorStatus ==3 )
15      checks: OpenCommand shall be 1 exactly 5 seconds
16      fails if: assurance is below 0
17      description: "If elevator one's doors are closing and an object is
18                   detected, open command shall be activated for exactly
19                   5 seconds, so door closing commands are deactivated"
20  ENDORACLE
```

(c) Oracle with the "exactly" temporal condition

$$conf(t) = \begin{cases} \frac{upperRef - sign}{(upperRef - lowerRef)/2} & \text{if}(sign < upperRef \ \&\& \\ & sign > lowerRef + (upperRef - lowerRef)/2), \\ \frac{sign - lowerRef}{(upperRef - lowerRef)/2} & \text{if}(sign < lowerRef \ \&\& \\ & sign < lowerRef + (upperRef - lowerRef)/2), \\ \frac{sign - lowerRef}{lowerRef - min} & \text{if}(sign < lowerRef), \\ \frac{upperRef - sign}{max - upperRef} & \text{if}(sign > upperRef) \end{cases} \tag{5}$$

$$conf(t) = \begin{cases} \frac{sign - upperRef}{(upperRef - lowerRef)/2} & \text{if}(sign < upperRef \ \&\& \\ & sign > lowerRef + (upperRef - lowerRef)/2), \\ \frac{lowerRef - sign}{(upperRef - lowerRef)/2} & \text{if}(sign > lowerRef \ \&\& \\ & sign < lowerRef + (upperRef - lowerRef)/2), \\ \frac{lowerRef - sign}{lowerRef - min} & \text{if}(sign < lowerRef), \\ \frac{sign - upperRef}{max - upperRef} & \text{if}(sign > upperRef) \end{cases} \tag{6}$$

However, by analyzing the industrial case studies, we noticed that certain violations could be accepted, especially those related to QoS measures. For instance, having the AWT over 50 s for a small time window is not a reason for classifying a test as “FAIL.” To tackle this issue, two measures were undertaken, (1) the syntax to adjust the assurance level is provided, so that the user can specify a negative assurance value that is yet acceptable; and (2) the syntax for defining a total of four failing reasons is provided, so that different failure patterns can be specified, each accepting certain deviations from the property to be asserted. An example for each failing reason for a below assertion pattern is shown in Fig. 10.

When a below pattern is defined, values above the reference signal (drawn with a red line) result in a negative assurance value, i.e., the property defined in the oracle has been violated. This violation is then analyzed against the failure patterns to decide whether it is acceptable or not: (1) is the least flexible failure pattern, a single violation makes

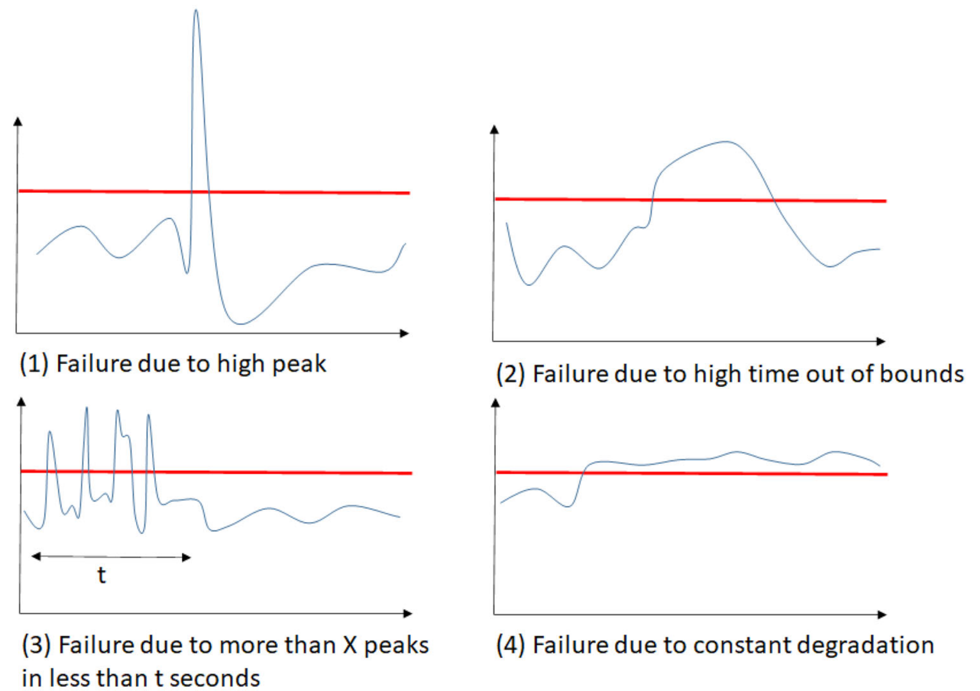


Fig. 10 Example for each failing reason for a below assertion pattern

Fig. 11 Snippets of oracles with different failure patterns

```
5    //R1.1 -> G{0,T} (( { Reset = 1 } and { IC <= tl } and { Ic >= bl } ==> { yout = Ic } )
6    ORACLE R1_1:
7        when: reset ==1 && ic <= tl && ic >= bl
8        checks: yout shall be ic
9        fails if: assurance is below 0
10   ENDORACLE
```

(a) Oracle with the "high peak" failure pattern

```
63   ORACLE AWTCheckerECO:
64       while: (ECOMODE == 1 && NumberOfActiveCalls < 20)
65       checks: AWT is below 0.4
66       fails if : assurance is below 0 within 5 minutes
67       description: "Checks the AWT is relatively low when there are few calls"
68   ENDORACLE
```

(b) Oracle with the "high time out of bounds" failure pattern

```
12   ORACLE ECOStandbyEnergyConsumption:
13       while: (ECOMODE == 1 && Elevator1Standby == 1 && Elevator2Standby == 1 && NewFloorCall == 0)
14       checks: Energy is below 100
15       fails if: assurance is below 0 more than 5 times within 1 minutes
16       description: "While ECOMODE is active and both elevators are in standby
17                     mode and there are not new floor calls, there shall not be
18                     more than 5 energy peaks within a minute."
19   ENDORACLE
```

(c) Oracle with the "X peaks in less than T" failure pattern

```
12   ORACLE ECOMODE:
13       while: (ECOMODE == 1)
14       checks: Energy is below 12000
15       fails if: assurance is in constant degradation 0
16       description: "If ECOMODE is active some energy peaks are accepted
17                     but the mean should remain below 12000Kwh"
18   ENDORACLE
```

(d) Oracle with the "constant degradation" failure pattern

the oracle fail; (2) accepts violations for a specific duration, but fails afterward; (3) accepts a determined number of violations, in a specified time window; and (4) aims to detect constant degradation, allowing some violations as long as the overall remains within the limits. At least one failing reason shall be specified, but the four types could be used. The concepts introduced above are modeled as illustrated in Fig. 3.

If a "high peak" failure pattern (the HighPeak component in the model) is defined, a single violation makes the oracle fail. It allows a single value: the assurance level. This enables the assertion to be more flexible as if a negative assurance level is set, states whose assurance value is in range from the specified level to zero would also be considered valid. The syntax of this pattern is expressed as follows: fails if: assurance is below *confValue*. Figure 11a is taken from the modeling of one of the "Tustin" CPS [25] requirements:

If a "high time out of bounds" failure pattern (the HighTime component in the model) is defined, negative assurance values are allowed for a given duration, failing after the specified limit is reached. It allows three attributes: (1) the assurance level, (2) the duration expressed by a numeric value, and (3) a time unit. It is expressed as follows: fails if: assurance is below *confValue* within *duration timeUnits*.

Among the example properties modeled for the industrial case study of the elevation domain, there is one that fits in this pattern. It is described as follows: "Checks the AWT is relatively low when there are few calls." As stated before, AWT is a QoS metric; therefore, it does not indicate that the CPS is in failure mode, but the quality of the service has been compromised. In an ECO mode and few active calls, it is allowed to have some high AWT, but cannot last more than 5 min; it would be considered unacceptable. Figure 11b shows the oracle modeling this property.

A high time out of bounds failure pattern fits perfectly in this case as negative assurance values are accepted until 5 min is reached, but not afterward. If a "x peaks in less than t" failure pattern (the XPeaks component in the model) is set, the assurance value is checked for a time window, where certain negative values are accepted, but fails if the specified number of peaks is reached. It allows 4 attributes: (1) the assurance level, (2) the number of peaks for a failure, (3) the duration expressed by a numeric value, and (4) the time unit. The syntax for this failure pattern is as follows: fails if: assurance is below *confValue* more than *nPeaks* times within *duration timeUnits*. Figure 11c shows a snippet of a requirement modeled using "x peaks in less than t" failure pattern. The description of the requirement is as follows: "While ECOMODE is activated and both elevators are in standby mode and there are no new floor calls, there shall not be more than 5 energy peaks within a minute." Elevators perform maintenance processes from time to time that should not involve more than 5 energy peaks within a minute.

If a "constant degradation" failure pattern (the *ConstDeg* component in the model) is defined, it aims to detect a constant degradation within the CPS. To this end, a determined period of time is taken into account, and the values are analyzed to detect whether there is any tendency to violate the assertion. It allows a single attribute: the assurance level. The syntax for this failure pattern is expressed as follows: fails if: assurance is in constant degradation *confValue*. Figure 11d shows an example of a "constant degradation" failure pattern. The oracle modeled stands for a requirement that is described as follows: "If ECOMODE is activated, some energy peaks are accepted but the mean should remain below 12000KWh." In addition, a small mathematical library is also provided along with the grammar, so the most common mathematical operations are applicable to the variables monitored.

Custom oracle

Although the first approach of the DSL turned out to have a high expressiveness, during the generalization of the grammar some particularly complex properties were encountered. To facilitate the modeling of these kinds of properties, a special type of oracle was introduced: Custom Oracle. The main purpose is to provide means to the user to be able to implement custom evaluations through the DSL.

The user only needs to specify the input parameters for the evaluation, previously defined in the monitoring plan and, if a precondition is needed, the type of the precondition, i.e., when or while, as well as the input parameters for the precondition. A description is also optional.

The developed infrastructure has been exploited so that with the information gathered through the syntax, a full structure of an oracle is generated in a C programming language, including the headers of the evaluation function and, if it corresponds, the headers of the precondition function. By completing the functions in C, the user will be able to model the complex CPS' property.

3.1.3 Inclusion of uncertainty functions

The uncertainty-related part of the DSL builds on previous work on uncertainty modeling and testing for CPSs, and in particular, UncerTum [6]. This modeling solution was built on an uncertainty conceptual model called U-Model [7]. U-Model defined three high-level uncertainty-related measures, i.e., Probability, Vagueness, and Ambiguity. These measures were implemented in UncerTum as UML data types. Consequently, we used these data types in the DSL to support modeling test oracles related to uncertainty with the ultimate goal of handling a variety of uncertainties that exist in different CPS types. These three types of uncertainty libraries are the Probability library, Vagueness library, and Ambiguity library, respectively, which can facilitate the specification of most uncertainties in CPSs. We create uncertainty-related test oracles based on these uncertainty data types.

Fig. 12 Snippets of test oracles employing uncertainty functions

```
20      // From Table 3.8 in [CIBSE. Transportation system in buildings.Cibse Pub.. londo, 2010]
21      ORACLE CheckSystem30SPerformance:
22          while: ( Time==3600)
23          checks: Percentage(CallsAnsweredIn30).value() is above 0.65
24          fails if: assurance is below 0
25          description: "The percentage of calls answered in 30 seconds in an hour of peak activity
26                       shall be above 65%, otherwise the system response time performance
27                       is unacceptable."
28      ENDORACLE
```

(a) Test oracle of elevator system response time performance

```
20      // From Table 3.7 in [CIBSE. Transportation system in buildings.Cibse Pub.. londo, 2010]
21      ORACLE CheckSystemAWTPerformance:
22          while: ( Time==3600)
23          checks: FuzzySet(PATQ).MembershipDegree(AWT).value() is below 0.5
24          fails if: assurance is below 0
25          description: "To achieve satisfactory passenger experiences,
26                        the passenger average waiting time (AWT) shall not be Poor time defined
27                        in CIBSE Guide D. If the belongingness to the Poor is above 0.5,
28                        the AWT performance is unacceptable."
29      ENDORACLE
```

(b) Test oracle for checking elevator system AWT performance

```
21      ORACLE CheckSystemATWStability:
22          while: ( Time==3600)
23          checks:  ShannonEntropy(AWTPer5Min).h() is below MaximumallowedEntropy
24          fails if: assurance is below 0
25          description: "The observed AWT distribution (collected every 5 mins) shall have low
26                        entropy (MaximumAllowedEntropy) as compared to the expected distribution
27                        of a specific traffic profile."
28      ENDORACLE
```

(c) Test oracle of system stability in terms of AWT

```
6     ORACLE AverageWaitingTimeUnder10Seconds:
7         checks: AWT is below 10
8         fails if : confidence is below -0
9         description: "..."
10    ENDORACLE
```

```
6     ORACLE AverageWaitingTimeUnder10Seconds:
7         checks: AWT is below 10
8     There is operational data out of bounds. is below -0
9         description: "..."
10    ENDORACLE
```

Fig. 13 False positive detection warning

Probability library

Probability is one of the most common ways of measuring uncertainties. There are various probability types, ranging from simple probability values to a wide range of probability distributions. To increase the universality of the DSL to support different types of probability applicable to different CPS types, we implemented the Probability library from UncerTum.

The Probability library facilitates specifying uncertainties in CPSs, e.g., due to sensor errors and other uncertain environmental conditions. At a high level, following UncerTum, we implemented the three types of probability-related data types. First, we implemented simple types (e.g., percentages, intervals, probability values). Second, data types capture probabilities as qualitative values (e.g., likely, certain). Finally, we implemented common probability distributions such as normal and uniform distributions.

Probabilities are commonly used in CPSs. For example, self-driving cars (a representative CPS) are equipped with sensors to detect obstacles in their paths, e.g., proximity sensors. Since sensors have measurement errors, calculating the distance to an obstacle multiple times may result in different readings. However, such non-deterministic measurement errors may correspond to a specific known probability distribution.

Similarly, probabilistic test oracles can also be modeled with our DSL in the context of the Elevator use case. In particular, we show an example in Fig. 12a of a test oracle for testing the elevator system's time to respond to calls that is an indicator of the quality of service of elevators based on the CIBSE Guide D [26]. The guide provides recommended response times of elevators installed in different types of buildings to determine different levels of services. The response time is calculated over the hourly activity of an elevator. The service levels are classified into four classes, i.e., Excellent, Good, Fair, and Unacceptable. The example test oracle in Fig. 12b is modeled to check whether the response time of an elevator installed in an office building is of an acceptable level, i.e., greater than 65% for calls answered in 30 s for an hourly peak activity in an office building.

Vagueness library

In certain CPSs, it is difficult to precisely specify uncertainty as probabilities, such as in the context of steering control systems of autonomous vehicles, navigation systems,

and obstacle avoidance systems of mobile robots. For this reason, concepts related to fuzziness are relevant. To this end, UncerTum defined the Vagueness library, which considers the concepts of fuzzy theory. Some examples of data types we implemented from UncerTum's vagueness library are various types of fuzzy sets and fuzzy logic.

Figure 12b shows an example of a test oracle related to fuzzy sets. The example is based on the experience of passengers using the elevator based on QoS parameters, e.g., average waiting time (AWT). According to the CIBSE Guide D, the target AWT shall be less than 25 s, whereas greater than 30 s is considered Poor. However, the boundary between Target and Poor is fuzzy. Thus, we define a test oracle using fuzzy sets to check whether the passenger experience based on AWT is satisfactory and not Poor. Assuming a membership function that takes input an AWT value and returns a value (membership degree) telling how much the AWT value belongs to Poor performance. If it is greater than or equal to 50%, then the test oracle fails and the elevator's performance is poor.

Ambiguity library

The Ambiguity library mainly considers the data types corresponding to relevant ambiguity measures, including those in belief theory (BeliefInterval, Belief, Plausibility) and Shannon entropy. To demonstrate the applicability of the Ambiguity library on the elevator case study, we will use ShannonEntropy as a measurement.

The ShannonEntropy data type has an h attribute, where a lower value means low entropy. A specific traffic profile with a specific building configuration has an expected distribution of AWTs over a specific period of time (e.g., every 5 min). Shannon entropy can be calculated to determine the degree of uncertainty (i.e., h) by comparing the expected distribution with the observed distribution (e.g., after a simulation). Depending on the specific configuration of the building and traffic profile, different degrees of Shannon entropy will be acceptable. This test oracle is captured in Fig. 12c, where MaximumAllowedEntropy determines the acceptable level of entropy.

3.2 Syntax validation and oracle assessment

Below we present syntax validation and oracle assessment. It is worth mentioning that if an error is encountered within an oracle's syntax, the oracle is excluded from the oracle assessment process.

3.2.1 General syntax validation

Similar to grammar, syntax validation is divided into two main parts: the monitoring plan and the oracle definition. For the monitoring plan, the errors checked and reported if encountered are: (1) the ones related to empty or duplicated names, (2) issues with Boolean-type variables and value ranges, (3) missing model specifications for inference variables, and (4) problems related to the independent variable attributes in declarations.

Regarding the oracles definition, the errors covered are the ones related to: (1) empty elements, duplicated names and undeclared variables; (2) incorrect expressions; (3) improper comparison expressions and duration specifications; (4) issues with signal representation and reference values; (5) logical operations and temporal conditions with assertions; and (6) problems with failure patterns and assurance values. Moreover, for the custom oracles there are another two errors covered: (1) non-empty inputs for precondition and checks, and (2) ensuring input variables are declared in the monitoring file.

Finally, for the Mathematical library we implemented three different constraints related to its syntax, which must be like *Math.libraryName(expression)*. The errors covered related to the expression are the following: (1) ensuring correct syntax for math expressions, (2) requirements for parenthesis placement and signal names, and (3) prohibiting logical or comparison operators and recursive mathematical functions.

XTEXT provides some syntax validations automatically, e.g., those related to the structure of the syntax, i.e., "precondition" tag shall be proceeded by an "input" tag or ":;" etc. Finally, for the uncertainty-related libraries, the value of the properties in some uncertainty data types shall be between 0 and 1. This is the case of the value property in MembershipDegree or the lambda property in FuzzySetCut, for example. The type checker has been enhanced so that an error is thrown if this constraint is violated.

3.2.2 Oracle assessment based on operational data

Test oracles are meant to define properties that cannot be violated. If a property defined in a test oracle has been violated, a failure should be detected. However, test oracles are prone to false positives, meaning they can catalog a test as a fail when they should not. In this context, a false positive refers to a failure that is exposed without it needing to be (i.e., the system functions properly, but the oracle indicates a problem). To reduce the number of false positives, our DSL implements an oracle assessment module. This module prompts the user with a warning if a false positive has been detected within the test oracles being developed.

The assessment module is provided with operational data to detect false positives within the test oracles. A failure-free state of the operational data is crucial; otherwise, if oracles are defined correctly, the oracle assessment module will detect those failures and users will be warned of false positives. This will lead to misunderstandings, and poten-

tially, users will change the oracle to get rid of the warning, accepting the failure as a correct state of the CPS.

In our implemented DSL, the data is provided via a CSV file (to assess the viability of this approach). It is structured as follows: Each column in the file represents a variable, where the variable's name is specified within the first row. Each of the other rows represents the value of the CPSs' variable at a specific time stamp. A time stamp variable should also be provided. It is important to reiterate that this version of the DSL, in general, and the assessment module in particular, is an early prototype whose feasibility is assessed. The following versions of the DSL will consider a way to specify the location of the file/files and a dynamic data loading to integrate the DSL with the rest of the DevOps ecosystem. Operational data could also be obtained from Stellio. Additionally, different sources of operational data should be envisaged.

Operational data provided to the oracle assessment process is considered to represent the correct CPS performance. With this in mind, if a failure is detected in any given oracle, it is assumed that it is throwing an error when (probably) it should not; that is, a false positive has been detected. The user is warned about the incident and should probably correct the ambiguity. For example, Fig. 13 is the description of an oracle that asserts that the Average Waiting Time (AWT) shall below 10 s. Figure 13 (Line 8) shows that the AWT variable takes higher values. Because operational data represents a failure-free state of the CPS, the user is notified of the detection of a false positive.

For the assessment of an oracle to take place, there are two mandatory prerequisites. First, operational data for each declared variable within the oracle must be provided. Second, the oracle cannot contain any structural error. After checking that these prerequisites are met, the assessment starts. For every value of the time stamp in the global variable, a verdict is calculated for the specified oracle using the values of the variables at that given time stamp. If any error is detected, the process is interrupted as a false positive has been detected, and the user is warned.

As for the developed grammar, the process of giving a verdict is divided into the failure patterns defined, i.e., each failure pattern is assessed independently. The assessment process is stopped when a failure is detected and, consequently, a warning is thrown. High peak, high time out of bounds, and x peak failure patterns allow a precondition to be defined. This is the first condition to check before proceeding with the Oracle assertion process.

3.3 Test oracle generator

The test oracle generation is composed of two main processes: (1) compilation of the DSL artifacts and (2) microservice generation. The former compiles all the artifacts specified by the user to generate the oracle. The latter relates to the generation of a microservices that makes use of the generated oracles. Moreover, to facilitate this generation process, a streamlined process of microservice generation on GitLab has been implemented.

3.3.1 Compilation of the DSL artifacts

The compilation of DSL artifacts is divided into three main parts: (1) compilation of organic oracles, (2) compilation of pre-specified oracles, and (3) compilation of Uncertainty libraries.

Compilation of organic oracles

Organic oracles do not contain any machine learning component for inferring monitoring data. After specifying these kinds of test oracles with the DSL, these need to be compiled afterward to be executed in a real environment. The framework used to develop the DSL, Xtext, provides a useful module to develop our own compiler for the developed DSL, and the language to compile the DSL can be any language of our choice. Furthermore, this module is compatible with Xtend, which facilitates compiler development. Xtend is a programming language for code generation, which provides useful mechanisms for writing compilers, like multi-line template expressions. It also provides features that make visiting and traversing the model obtained by parsing the DSL easy.

From the test oracles modeled using the DSL, we extract (1) the test oracles' executable code, (2) some common libraries needed by the oracles, and (3) a configuration file to facilitate the integration of the test oracle. (1) and (2) together compose the organic test oracles executable code, and in order to be executed in any given target platform, they must be written in a general-purpose language. Organic oracles are intended to be run on an embedded system, and such systems are typically resource-constrained systems. Therefore, due to its portability, memory management, and performance, the C language is the one that fits the best. (3) is used by the microservice generation module. This file enables the integration and execution of the test oracles within the DevOps for CPSs infrastructure [11]. It follows the JSON standard, which is a text-based format for representing structured data. The selection was determined by the framework used for generating the microservices, Ansible, as it has native support for JSON.

As shown in Fig. 14 blue, the compilation process is as follows: First, the test oracles need to be modeled using the DSL. For this purpose, a plug-in for the Eclipse IDE is provided. The syntax is validated during the editing of the files, and the user is notified of the warnings and errors by the validation module. Additionally, the generation module compiles the code every time the files are saved. Notice that if the validation module reports an error, the compilation process is not performed. Therefore, after modeling the test oracles

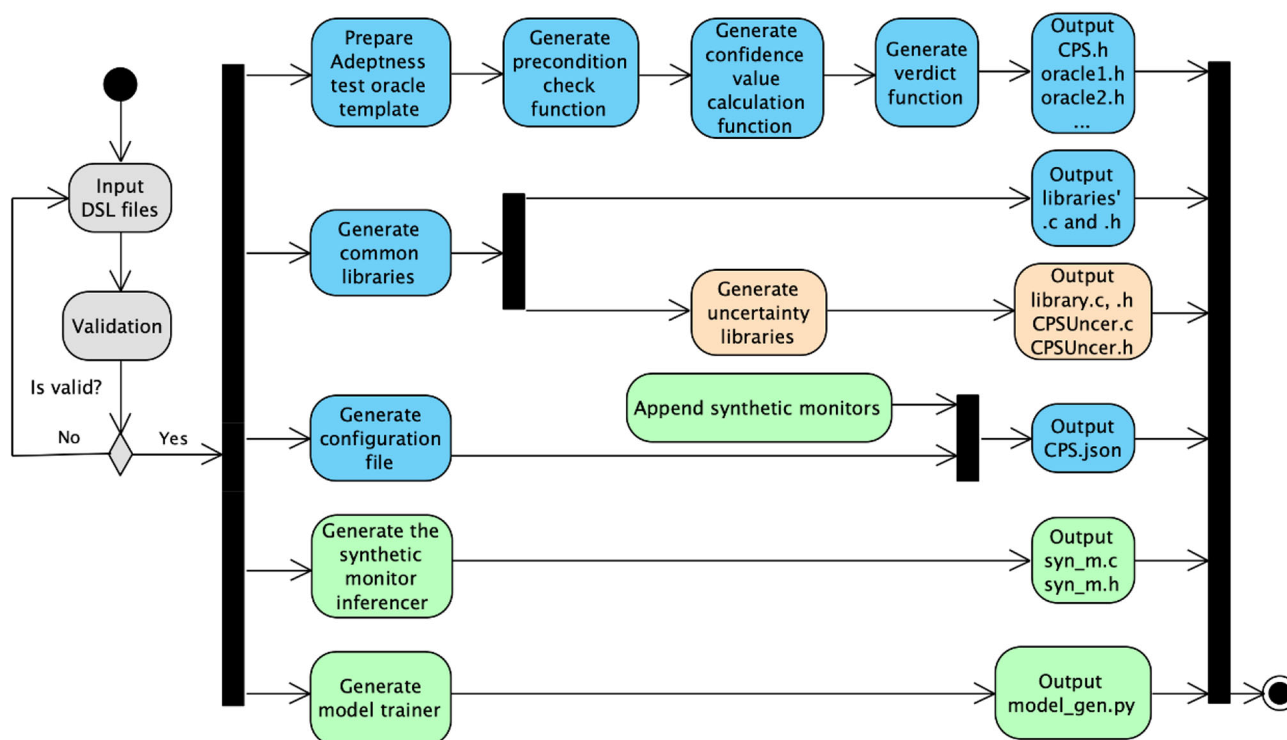


Fig. 14 In blue, the compilation process of an organic oracle; in blue and green, compilation process of an AI-based oracle; in blue and orange, uncertainty libraries compilation process

using the DSL, and the validation module does not report any error, the generation module takes the resulting files as input. These files are the monitoring plan file and the oracle files. The latter imports the former in order to have the monitors required by the test oracles available. Figure 14 blue shows the process from modeling in the corresponding files the Oracles using the DSL to the output files generated by the compiler, focusing on the stages that occur within the compilation phase.

Table 1 shows the six different stages that occur during compilation: (1) prepare test oracle template, (2) generate precondition check function, (3) generate assurance value calculation function, (4) generate verdict function, (5) prepare common libraries, and finally (6) generate configuration file.

As an example of the compilation process, consider the oracle snippet in Fig. 15. The process will be as follows: (1) the test oracle template is retrieved. The name for the oracle execution files will be the same as the oracle's: R1_72.c and the corresponding header file, R1_72.h. Additionally, the Autopilot.h file will be created, containing some constants and defining the structs to facilitate communication with the test oracle. (2) As no precondition is specified in this case, the precondition function is not inserted. (3) The template for the in-range assertion is used, along with the parameters -15 and 1, to complete the assurance level calculation function. The assurance value from the failure reasons is taken for the assurance level calculation. (4) The verdict function is completed according to the failure reason specified. In this case, there is only one type of failure: failure due to a high peak. (5) Common libraries for the test oracles are prepared: Array.h and Array.c. (6) The oracles' names, and the monitors' names are checked in order to complete the JSON file, Autopilot.json. In summary, a total of five different files are created: the R1_72.c and R1_72.h files, containing the test oracle's executable code, Autopilot.h file containing the constants and the verdict's and monitors' structure, the Array.h and Array.c, the library needed by the test oracles and the Autopilot.json file.

Compilation of pre-specification oracles

As explained before, machine learning components could eventually infer some monitoring data. More in-depth analysis related to this can be found in previous works [17, 18]. Oracles containing ML components are named pre-specification oracles. These oracles take advantage of the organic oracles' framework explained in the previous subsection and add the necessary syntax to the DSL to enable the inference of synthetic monitors, which would emulate the execution of a regression test oracle. Afterward, these monitors could be used within the test oracles as if they were independent monitors. Therefore, a pre-specification oracle is an extended version of an organic test oracle that can obtain synthetic monitors. For this purpose, two additional files must be defined, one for the synthetic monitor's definition, and

Table 1 Steps for test oracle preparation

Step Number	Step Name	Explanation
1	Prepare the test oracle template	All the test oracles share the same structure and, consequently, a template that integrates all these pieces of code was prepared for the compiler to use. The first step is to take those pieces of code and generate the base of the test oracle. Each test oracle will have its own.c and.h files. Additionally, some constants and structs for the verdict and the monitors are created to facilitate communication with the microservice. With this aim, a header named the CPS is created and shared among the oracles
2	Generate precondition check function	At this stage, the precondition expression introduced by the user is taken from the input files and is located in the corresponding place within the precondition check function. The same procedure is carried out with the after clause, the duration specified is established in the wait function
3	Generate assurance value calculation function	We define six types of assertions: (1) same, (2) not same, (3) below, (4) above, (5) in range, and (6) gap. Each assertion generates an assurance value indicating how the CPS's current state deviates from expected performance. Templates for these assertions were prepared for the compiler, with specific implementations chosen based on user input. Signal and reference signal expressions are extracted from input files and placed accordingly within the function. Timing condition parameters, if provided, are also appropriately located
4	Generate verdict function	In Sect. 3.1.2, we differentiated four types of failure reasons: (1) failure due to a high peak, (2) failure due to a high time out of bounds, (3) failure due to more than X peaks in less than t seconds, and (4) failure due to constant degradation. As well as with the assertions, the piece of code that implements each one of the failure reasons was prepared for the compiler to be used. Then, the template is completed, taking the variables introduced by the user. Notice that the user can specify a single failure reason, or combine them all, and the verdict is emitted accordingly
5	Prepare common libraries	All the organic test oracles use common libraries. For example, we implemented a library to store values of several cycles of a monitor because some conditions require assessing different cycles' values simultaneously. At this stage, those libraries' files are generated
6	Generate configuration file	The JSON file is used afterward at the microservice generation stage covered in Sect. 3.3.2. It is named according to the user-specified name for the CPS. The main purpose of this file is to give enough information for the test oracle microservices to generate and execute properly. This file has two main sections: (1) the monitors' names and (2) the test execution functions' names. On the one hand, to complete section (1), and because there could be monitors declared in the monitoring file that are not used in the oracles, the monitors are extracted by analyzing the oracles. On the other hand, to complete section (2), test execution functions are extracted from oracles' names. There is a test execution function for each oracle, whose names are unique throughout the file

two for the specification of the models to infer those monitors. The generation module performs likewise, it extends the organic test oracles' generation module, appending the necessary pieces of code to complete the output files and generating some new files. Figure 14 blue and green shows the additional stages the generation module carries out when compiling an AI-based oracle (highlighted in green).

First, within the stage Append synthetic monitors, the names of the synthetic monitors and their specifications are appended to the JSON file created by the normal workflow of the generation module. By doing so, the microservice generation module will know which monitors must be inferred before executing any test. The required information is gathered by analyzing the synthetic monitor's file.

Second, within the stage Generate synthetic monitor inferencer, a.c and a.h files are created for each synthetic monitor. These are the files that the AI-based test oracle is going to execute to perform the inference. The technology used to make this possible is TensorFlow. A comparison of different frameworks was carried out before opting for this technology, which is included at the end of this section. A template is completed with information gathered from the synthetic monitor's file: the model to use and the required monitors for the inference to perform.

Additionally, if the pre-specification-based test oracle uses a synthetic monitor inferred by a trainable model, the Generate model trainer stage will be carried out by the compiler. The model trainer is written in Python, as it provides very useful APIs like the Keras API (Python deep learning API). Similarly to other compiler stages, the starting point is a template. Then, it is completed by the parameters specified in the model file: the monitors that are necessary to infer this synthetic monitor, the URL where the data is stored (monitor's names must be preserved), and finally, the layers and configurations that specify the model. This stage generates a.py file, which is used just before the microservice generation

```
5  CPS Autopilot: implements Autopilot_Monitors
6
7      ORACLE R1_72:
8          checks: AilCmd is in range between -15 and 1
9          fails if: assurance is below 0
10         description: "Autopilot R1_72 requirement"
11     ENDORACLE
12 ENDCPS
```

```
1 MONITORINGPLAN Autopilot_Monitors:
2     MONITOR AilCmd:
3         type: double
4         max: 180
5         min: -180
6     ENDMONITOR
7
8 ENDMONITORINGPLAN
```

Fig. 15 Simple example of an oracle

process starts. It uses the data provided at the specified URL, and as a result, a TensorFlow Lite model is generated.

The technology we selected for the inference was TensorFlow Lite due to its popularity and its lightweight module that can be adequate for edge devices. As TensorFlow Lite is written in C++, it can be easily linked to the microservice architecture [11] compatible with this DSL. Moreover, the TensorFlow framework enables the implementation of the proposed architecture, separating the training and inference phases. First, we can use the full functionality of TensorFlow to train the model, that in the case of trainable models would be in the cloud infrastructure and, then, transform this model into a lightweight representation (.tflite file) to be used with TensorFlow lite at inference time in the edge devices.

Compilation of Uncertainty libraries

Analogously to the pre-specification oracles, the uncertainty-wise oracles extend the generation module of the Organic Oracles. If any of the uncertainty library functions are specified through the DSL at the oracle definition phase, the generation module executes one more stage in order to get the corresponding uncertainty library functions' source code. Figure 14 orange and blue shows the stage added to the compilation process (in orange) and the files obtained as output.

To facilitate the integration and execution of the uncertainty library with the rest of the oracle's source code, it is also compiled into the C language. The methodology used to compile these library functions is the same as used in the organic oracles and the pre-specification oracles. The Uncertainty library functions template is pre-written in C; therefore, if the generation module detects that a library is used within the DSL syntax, the corresponding template is extracted, in addition to the corresponding parameters defined through the DSL syntax. The resulting source code is inserted within the CPS_Uncer.c and CPS_Uncer.h files. Notice that these files uniquely contain the library files used within the syntax.

3.3.2 Microservice generation for multi-level oracle execution

After the DSL compilation phase finishes, the generation of a microservice that makes use of the generated oracles takes place. This phase consists of integrating the generated executable code on a microservice template for embedded C programs that allows for adopting a DevOps workflow within the realms of CPSs. A detailed description of these microservices is available in a related publication [11]. After the executable code is integrated into the template, this newly created microservice is compiled and packed in a Docker container. As a final step of generation, all the microservices are made available in different Docker registries which can be reused at different test levels (SiL, HiL), as well as in operation.

Figure 16 summarizes the different steps undertaken during the automated microservice generation process. Within the first step, the generator takes the DSL files as inputs and provides the compiled artifacts as output, which are the ones presented in Sect. 3.3.1. Additionally, if we are working with pre-specification oracles, and specifically if we have defined at least one trainable model, the.py file to train the model will be included in the output. Moreover, if we work with uncertainty-wise oracles, we could also find the corresponding files.

After generating the source code and configuration artifacts by the DSL, there are integrated into the DevOps microservice template. On top of the DevOps microservice, additional functionality is also required to configure the oracle through REST calls and manage the execution status. To add the required functionality in mind, a wrapper built on top of the DevOps microservice template has been developed. This wrapper provides all the functionality that the oracles generated by the DSL need to communicate with the outside world and gives the oracles the microservice nature that allows them to be integrated into the DevOps ecosystem. The wrapper performs the following functions (explanations are detailed in Table 4 in Appendix):

1. Implementation of the REST API input configuration call.
2. Implementation of the REST API URN binding configuration call.
3. Transformation of monitor sensor values to oracle structures.
4. Transformation of verdicts to SenML messages.
5. Integration of DSL Artifacts in the oracle microservice:
   (a) Interpretation of the JSON configuration file.

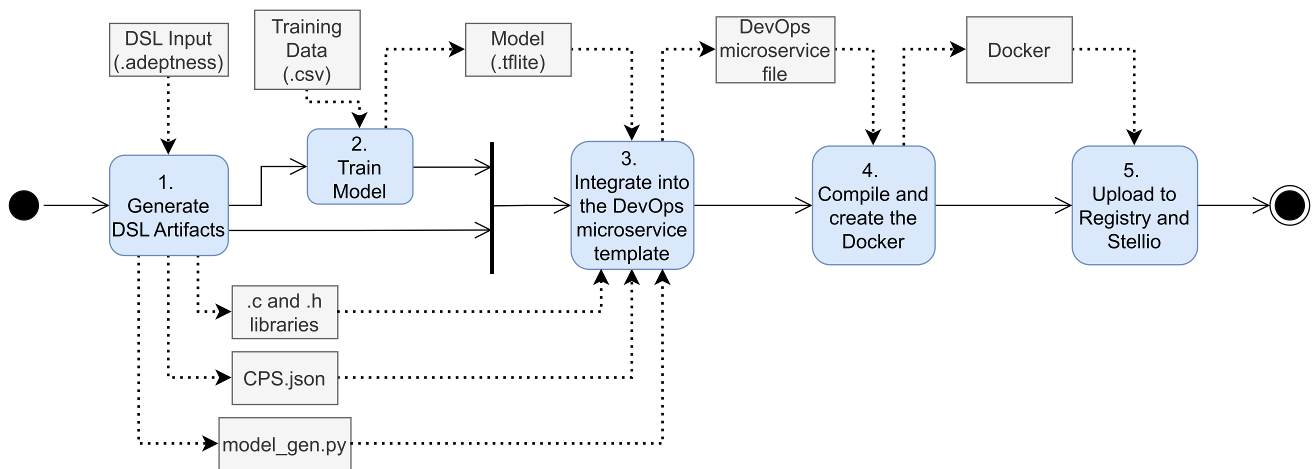


Fig. 16 Overview of the oracle microservice generation workflow, which is part of the *microservice generation* in Fig. 2

(b) Preparation and copying of the required files to target directories.
(c) Filling of template files.

After the required source code has been consolidated and modified to fit the oracles, the compilation and Docker image generation phase occurs. The main motivation for generating 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 our DevOps infrastructure, the required portability goes further, having to provide the ability to execute the microservices across multiple architectures. To achieve this portability for the microservices, the newly added Docker multi-arch support and the buildx project have been used. The Docker image generation stage runs as part of an Ansible task, generating the Docker image name based on the image name set when running the playbook and the CPS name. For each CPS, the Docker builder used the Dockerfile available in each microservice output directory. The used Dockerfile is a multistage definition, with an initial stage for compiling the project and a second stage for the runtime image. The first stage installs all the required dependencies for development and builds the project with CMake and GCC. The second stage installs the runtime dependencies, copies the build artifacts, and sets the microservice executable as the entry point for the image.

After the newly created microservices have been made available on the Docker Registry, they are ready to be used as validation components in the DevOps for CPSs workflow. In order to make the other components in the DevOps ecosystem aware of the existence of the microservice images and Oracles, entities have to be uploaded to Stellio. To accomplish the publication of entities to Stellio, a script reads the.json file generated by the DSL, and based on the CPS, evaluation function, and inputs section, it defines the entities to be created. It then uses the PyNGSI-LD library provided by a supplier to connect to Stellio and publish the entities.

3.3.3 Continuous integration pipeline

As an effort to facilitate the generation process of the oracle microservice images to the end-user, a streamlined process of image generation on GitLab has been implemented. This process uses the CI/CD infrastructure provided by GitLab to automate the microservice generation process. The generation process requires several components to be executed. To ease this execution and to provide a self-contained approach to the microservice generation, a Docker image shipping all the required components has been created. This image contains all the system requirements and components required for generating microservices, along with several scripts that provide an easy-to-use interface for the users. Input files are shared to a pre-defined path inside the container through Docker bind mounts.

A repository containing the Dockerfile required to generate the image, along with the microservice generator and Stellio entity uploader as GIT submodules, has been created. The image is automatically generated using the GitLab CI/CD infrastructure and published to the GitLab Docker registry, where it can be used in other pipelines. With the generator image, an example pipeline is shipped. This pipeline definition includes a set of steps or stages required to generate microservices from DSL input files. The developed pipeline is being defined as a GitLab pipeline. GitLab pipelines are defined in a gitlab-ci.yml file, following the YAML file syntax and the keywords GitLab runners use to execute the pipeline. The structure of GitLab pipelines consists of several execution stages, each containing steps to be executed. The defined pipeline has been divided into four different stages,

each of which contains a single step of generation: (1) DSL generation stage, (2) model training stage, (3) microservice generation stage, and (4) Stellio publication stage. All the steps are executed inside the previously created generator image.

4 Evaluation of the approach

In our evaluation, we tried to answer the following two Research Questions:

- RQ1—How good is our DSL-based tool's expressiveness for designing test oracles for CPSs?
- RQ2—How efficient is the tool at generating multi-level test oracles?

With RQ1, we aimed to assess whether our DSL is expressible enough to define test oracles to check whether a CPS is compliant with its requirements. With RQ2, we aimed to assess whether our approach is fast enough to generate multi-level test oracles and therefore, scale for being integrated into a DevOps infrastructure for CPSs.

4.1 Experimental setup for RQ1

4.1.1 Industrial datasets

We employed two different case study systems in our dataset to assess the expressiveness of our DSL. On the one hand, we employed Orona's case study system that encompassed a total of 37 requirements extracted from their typical test process. These 37 requirements were extracted together with domain experts by analyzing the documentation of the validation plans, which were conducted for testing various versions of the dispatching algorithm. Moreover, the authors of this paper interviewed engineers from Orona to derive interesting properties (by using natural language) that they would like to consistently check through the different versions of the dispatching algorithm.

On the other hand, we employed a second case study system related to a satellite provided by LuxSpace Sàrl. While we did not have full access to the case study system, its requirements were openly available [27] and involved a total of 41 requirements. Therefore, in total, we employed 78 requirements from two industrial case study systems of two CPSs.

4.1.2 Open-source dataset

To complement our two industrial case study systems, we employed 9 additional open-source case study systems. 7 out of 9 case study systems (all except Autopilot and TwoTanks) were related to the "Ten Lockheed Martin Challenges" [25] The largest case study system had a total of 33 requirements, whereas the smallest one had only 2. In total, the 9 case study systems provided 87 additional requirements to our experiments. Therefore, considering both, industrial and open-source case studies, our benchmark involved a total of 165 requirements.

4.1.3 Evaluation metrics

To assess the expressiveness of our DSL, two metrics were proposed.

On the one hand, we measure the *expressible requirements rate (ERR)*. This metric measures the percentage of requirements of the system that are possible to be modeled by our DSL. We divide this metric into (1) ERR and (2) $ERR_{new_monitors}$. The former refers to the percentage of requirements that are modelable without the need for modifying any monitoring variable, whereas the latter refers to the percentage of requirements that can be modeled through our DSL but by modifying at least one monitoring variable (e.g., performing a small modification of it).

On the other hand, we measure the *number of test oracles to model each requirement (# of Oracles)*. For each modelable requirement, we measure how many test oracles are required to model the requirement. This is because some requirements are too complex and therefore require several test oracles to model a single requirement. The lower this metric, the better the expressiveness of our approach. For each case study system, we provide the average, maximum, and minimum numbers for this metric.

4.2 Experimental setup for RQ2

With RQ2, we aimed to assess how efficient the test oracle generator is at generating multi-level test oracles. To this end, we employed the industrial case study system provided by Orona along with its 37 requirements and followed the test oracle generation pipeline in Fig. 16. That is, we commit the sources to a GitLab repository and it triggers an automated CI/CD pipeline that encompasses three different steps: (1) generation of the source code; (2) generation of the microservice; and (3) publication of the entities in Stellio. To answer RQ2, we measured the time taken by each of these steps.

4.3 Analysis of the results and discussion

4.3.1 RQ1—expressiveness

Table 2 shows the results of the first metric, which assesses the percentages of requirements that can be modeled through our DSL for defining test oracles. In total, 94% of the requirements, i.e., 156 out of 165, could be modeled through our

DSL without requiring any modification in the monitoring file. This percentage could be increased to 98.18% in the case by adding slight modifications in the monitoring file. When considering individual systems, in 8 out of the 11 case study systems, the oracles could be modeled for 100% of the requirements. In the regulator case study system, the oracles for 4 of the requirements could not be modeled. However, with a slight modification in the monitoring file, all the requirements could have an oracle associated with this case study system. In the neural network case study, there was a requirement whose behavior could not be modeled through our DSL to generate an oracle, and subsequently, in this case, a custom oracle would need to be developed, which is also supported by our DSL.

When considering both industrial case study systems, in the case of Orona, related to the vertical transport domain, all the requirements were modelable through our DSL. This was expected, as we involved engineers from this company when designing the tool. As for the industrial case study from LuxSpace Sàrl [27], related to the aerospace domain, 90.12% of the requirements could be modeled directly and 95.12% of the requirements could be modeled by adding slight modifications to the monitoring variables. The requirements that could not be modeled with our DSL had low-level signal properties that the current version of the DSL does not include (e.g., how a signal decreases monotonically). Those cases, however, can be implemented once the test oracles are generated in C by using the custom oracle functionality.

Table 3 reports the results for the second metric, which measures the number of oracles that were necessary to specify a requirement. On average, from 1 to 4.32 test oracles were required for modeling the requirements of the system through our DSL. Two case study systems had a requirement that required up to 9 oracles to model. Conversely, in 5 out of the 11 case study systems, with a single test oracle it was possible to model each requirement. This included Orona's case study, one of the two industrial case studies employed to assess our approach. For the specific case of LuxSpace, the second industrial case study system, the maximum amount of test oracles to define a requirement was 3, although most of the requirements could be modeled with a single test oracle. The reason why more than 1 oracle is needed to model more than 1 requirement was that (1) to keep the oracles simple, we only allowed to assert one specific condition, and (2) our DSL allows expressing one single precondition. That is, in cases where multiple checks are needed to validate a requirement or a requirement has different preconditions, our DSL is not expressible enough to model them within one oracle. However, instead, our approach tries to keep individual oracles simple.

Having discussed this, the first RQ can be answered as follows:

RQ1: Our DSL is capable of modeling a large portion of requirements by employing only a few test oracles for each requirement.

4.3.2 RQ2—oracle generation efficiency

Within RQ2, we wanted to assess how efficient our approach is at generating multi-level test oracles. To that end, within the realm of our industrial case study system provided by Orona, we measured the time taken by our CI/CD pipeline to generate (1) the source code of the oracles; (2) generation of the microservice; and (3) publication of the entities in Stellio. In total, we measured the time taken in each of these steps. The shortest times were produced by the first and last steps, which took each of them less than 45 s. Conversely, the longest one was the second one, the generation of the microservice, which took around 15 min. Nonetheless, our tool is sufficiently fast to be adopted by industrial practitioners for several reasons. Firstly, the common industry practice involves the manual generation of multi-level test oracles based on requirements, which is extremely time-consuming. In contrast, our automated tool significantly reduces this effort. Secondly, test oracle generation typically occurs after test cases are designed and before they are executed. This step can be scheduled to run during periods where immediate results are not critical, such as overnight. Thirdly, test oracle generation is not a continuous activity; it is usually performed once per testing cycle or when significant changes are made to the software.

Therefore, RQ2 can be answered as follows:

RQ2: Our test generator takes around 17 minutes from the commit of the test oracle files to the generation of the multi-level test oracle through a microservice, which is fast enough to be adopted in practice.

4.4 Threats to validity

The main threats to the validity of our evaluation are related to *external validity*. On the one hand, we only employed two different industrial case study systems, which poses a threat in generalizability. We note, however, that both case study systems were from two different CPS domains. Furthermore, we complemented these two case study systems with 9 open-source case study systems, encompassing a total of 165 requirements to model. On the other hand, the requirements needed to be modeled by humans, which may incur bugs when defining test oracles. We mitigated this by involving domain experts in the process of defining test oracles.

An internal validity threat in our evaluation could relate to the time it takes to generate the microservices, which

Table 2 Results for the % of modelable requirements in the selected open-source case studies

Case study	#of requirements	ERR	$ERR_{new_monitors}$
Orona	37	100%	-
LuxSpace Sàrl	41	90.2%	95.12%
Autopilot	11	100.0%	-
EffectorBlender	3	100.0%	-
Euler	8	100.0%	-
Finite State Machine	13	100.0%	-
Neural Network	3	66.6%	66.66%
Nonlinear Guidance	2	100.0%	-
Regulator	10	60.0%	100.00%
Tustin	5	100.0%	-
TwoTanks	32	100.0%	-
TOTAL	165	94.54%	98.18%

Table 3 Summary of the results for the number of test oracles required to model a requirement with our DSL

Case study	Avg. # of oracles	Maximum # of oracles	Minimum # of oracles
Orona	1.00	1	1
LuxSpace Sàrl	1.07	3	1
Autopilot	1.15	5	1
EffectorBlender	4.32	9	1
Euler	2.61	9	1
FiniteStateMachine	1.00	1	1
NeuralNetwork	1.00	1	1
Nonlinear Guidance	1.00	1	1
Regulator	1.00	1	1
Tustin	1.74	4	1
TwoTanks	1.35	8	1

may vary from run to run. Unfortunately, since the generation takes significant time (i.e., 17 min), we were unable to conduct thorough experiments with this. However, based on internal validation of the tool, this time is always similar.

5 Related work

CPSs are typically modeled through generic modeling languages, including MATLAB/Simulink, Modelica, or SysML. Besides these generic modeling notations and tools, there are others based on DSLs that cover specific particularities of CPSs. Pradhan et al. [28] proposed a DSL to support CPSs' heterogeneous communications and resilience systems. Van den Berg et al. [29] proposed a DSL to specify (1) the interactive model description, (2) the computation of possible operational modes of the CPS, and (3) design alternatives of the CPS. Nandi et al. [30] proposed to deploy runtime monitors of CPSs. Klikovits et al. [31] proposed a DSL for modeling reactive CPSs. All these approaches are, however, intended for the design and modeling of CPSs, whereas our DSL focuses on modeling test oracles for CPSs.

As for the testing dimension, several DSLs have been proposed. Vernotte et al. [32] proposed an attack-based testing approach modeled through a DSL oriented to air traffic control systems. A similar approach was proposed by Barreto et al. [33] for the same application domain. These approaches are focused on testing the security aspects of the CPSs. Maciel et al. proposed the test specification language (TSL) to produce test cases following the Gherkin format [34]. In the automotive domain, there have also been artifacts targeting DSLs to specify test cases. For instance, Menzel et al. proposed a DSL to create test cases from scenarios with different levels of abstraction [35]. Queiroz et al. proposed a DSL for representing autonomous driving scenarios for simulation-based testing [36]. Afzal et al. [37] proposed a DSL to generate scenes in Gazebo. Arrieta et al. [38] proposed a DSL to generate test cases as well as test oracles for simulation-based testing of CPSs. These approaches, however, focus on generating test cases (including test inputs), while our approach focuses on the test oracle part, i.e., the

mechanism that asserts whether the system is behaving as expected.

Regarding the test oracle problem, there are tools supported by a DSL that aim at modeling and generating test oracles for CPSs. As the development of CPSs' software is deemed to shift to a DevOps paradigm, our DSL is intended to support such development methodology. Specifically, as explained in the introduction, our tool supports six key characteristics: ***C1****: Streamlined test oracles* (i.e., oracles that can be reused across the different simulation-based test levels as well as in operation); ***C2***: *Support for time-continuous behaviors*; ***C3***: *Support for uncertainty modeling*; ***C4***: *Quantitative verdicts*; ***C5***: *Support for synthetic data*; and ***C6***: *Use of feedback from operation to reduce false positives*. Similar to our work, only four papers cover the modeling of test oracles for CPSs through a DSL [12, 14, 15, 39]. Given that time is a key characteristic of CPSs, all these works cover the second characteristic, i.e., support for the time-continuous behaviors of CPSs. Due to the properties of functional requirements of CPSs, this characteristic is necessary to provide enough expressiveness to model test oracles in the CPS context; therefore, all the works cover this characteristic. Similar to our work, Bernaerts et al. [39] and Menghi et al. [14] considered the use of quantitative verdicts (i.e., C4) to assess the severity level when a CPS violates a requirement, or how far the system is from violating it. This can be a useful characteristic for other testing activities, such as test case generation, where test oracles can be integrated for falsification-based testing. Last, only one study incorporates uncertainty in their DSL [14]. Nevertheless, the covered uncertainty part only considered (1) uncertainty due to unknown hardware choices and (2) uncertainty due to the noise in the inputs received from the environment (e.g., sensor readings). In contrast, our DSL covers a wider range of uncertainties of CPSs, identified by a previous work [6]. For this reason, we consider that the work by Menghi et al. [14] covers uncertainty only partly. The remaining three characteristics (i.e., C1, C5, and C6) are only covered by our tool. We note that these three characteristics make our approach unique in terms of modeling and generating test oracles for CPSs to validate their software in a DevOps development context.

6 Conclusion and future work

In this paper, we propose a tool supported by a DSL to define test oracles for CPSs and generate them automatically in order to be reusable across different testing levels as well as in operation. The DSL supports important CPS characteristics, including the time-continuous behavior of the systems, their inherent uncertainties, and different failure types. On the other hand, we support multi-level execution by generating microservices that are compatible with a DevOps infrastructure for CPSs [11]. We assessed the approach by employing several case study systems, including two different industrial case studies. The results suggest that our tool can define relevant test oracles for CPSs and can generate the microservices at an affordable time to apply to industrial contexts. However, for other CPSs there may be oracles that can be used at the SiL level that cannot be used at HiL level and operation, or vice versa. In the future, we plan to further evaluate our DSL in these CPSs and further investigate the generation of oracles for specific testing levels by considering this option in the DSL.

One key disadvantage of our approach is that the manual definition of DSL models is required. Given the recent advances in the field of large language models (LLM), as a next step, we would like to explore the option of fine-tuning an LLM to allow for the automated generation of test oracles within our proposed syntax directly from natural language. This would allow for saving a significant amount of time to test engineers.

Acknowledgements This project is partially supported by the Adeptness project funded by the European Union's Horizon 2020 program under Grant Agreement no. 871319. Shaukat Ali is also supported by the Co-evolver project (Project #286898), funded by the Research Council of Norway. Pablo Valle and Aitor Arrieta are part of the Software and Systems Engineering research group of Mondragon Unibertsitatea (IT1519-22), supported by the Department of Education, Universities and Research of the Basque Country. Aitor Arrieta is supported by the Spanish Ministry of Science, Innovation and Universities (project PID2023-152979OA-I00), funded by MCIU /AEI /10.13039/501100011033 / FEDER, UE. Pablo Valle is supported by the Pre-doctoral Program for the Formation of Non-Doctoral Research Staff of the Education Department of the Basque Government (Grant n. PRE_2024_1_0014).

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Appendix

See Table 4.

Table 4 Functions performed by the wrapper

Functions	Descriptions
Implement the REST API input configuration call	The oracles receive inputs that are defined by a name and data type within a configuration JSON file. In a microservices environment, these inputs are disseminated via MQTT by monitors and organized as subscriptions in SenML format. The REST API input configuration call facilitates the binding of a sensor, subscription, and monitoring representation to an oracle's input
Implement the REST API URN binding configuration call	While oracles are named within the DSL, they require unique identifiers during runtime to distinguish between multiple instances of the same oracle that may be deployed simultaneously. This is accomplished by binding each oracle's name to a unique URN, which is subsequently transformed into a distinct MQTT topic for publishing the oracle's verdicts. The URN binding API call ensures that each oracle instance has a unique topic for verdict publication
Transform monitor sensor values to oracle structures	Oracles operate on a C structure, referred to as MonitorInputs, which encapsulates all relevant inputs. Upon receiving sensor data from monitors, the wrapper function parses and converts these values into the MonitorInputs structure. This transformation ensures the consistency between the data received and the internal representation utilized by the oracles, enabling them to process the inputs effectively and accurately
Transform verdicts to SenML messages	The verdict, represented by a C structure known as Verdict, comprises the discrete verdict, assurance level, and verdict type. The wrapper is responsible for both the transformation and the publication of these verdicts, thereby ensuring that they are properly formatted and accessible to other services within the system
Integrate DSL Artifacts in the Oracle microservice	This process is managed using Ansible. Initially, Ansible interprets the JSON configuration file, which contains the definitions for each CPS, including the oracles, variation points, and synthetic variation points for pre-specification oracles. Ansible then proceeds to prepare and copy the required files into the designated target directories. For each CPS definition, a distinct microservice is generated, with the corresponding template and wrapper files, as well as the.c and.h files generated by the DSL, being copied into the appropriate output directory. Following the file preparation, Ansible fills the template files with the necessary data

References

1. Derler, P., Lee, E.A., Vincentelli, A.S.: Modeling cyber-physical systems. Proc. IEEE **100**(1), 13–28 (2011)
2. Baheti, R., Gill, H.: Cyber-physical systems. Impact Control Technol. **12**(1), 161–166 (2011)
3. Alur, R.: Principles of Cyber-physical Systems. MIT press, Cambridge, MA (2015)
4. Ayerdi, J., Garciandia, A., Arrieta, A., Afzal, W., Enoiu, E., Agirre, A., Sagardui, G., Arratibel, M., Sellin, O.: Towards a taxonomy for eliciting design-operation continuum requirements of cyber-physical systems. In: 2020 IEEE 28th International Requirements Engineering Conference (RE), pp. 280–290. IEEE (2020)
5. Zampetti, F., Tamburri, D., Panichella, S., Panichella, A., Canfora, G., Di Penta, M.: Continuous integration and delivery practices for cyber-physical systems: an interview-based study. ACM Trans. Softw. Eng. Methodol. **32**(3), 1–44 (2023)
6. Zhang, M., Ali, S., Yue, T., Norgren, R., Okariz, O.: Uncertainty-wise cyber-physical system test modeling. Softw. Syst. Model. **18**, 1379–1418 (2019). https://doi.org/10.1007/s10270-017-0609-6
7. Zhang, M., Selic, B., Ali, S., Yue, T., Okariz, O., Norgren, R.: Understanding uncertainty in cyber-physical systems: a conceptual model. In: Wąsowski, A., Lönn, H. (eds.) Modelling Foundations and Applications, pp. 247–264. Springer, Cham (2016)
8. Arrieta, A., Wang, S., Arruabarrena, A., Markiegi, U., Sagardui, G., Etxeberria, L.: Multi-objective black-box test case selection for cost-effectively testing simulation models. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1411–1418 (2018)
9. Menghi, C., Nejati, S., Briand, L., Parache, Y.I.: Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 372–384 (2020)
10. Wang, C., Pastore, F., Briand, L.: Oracles for testing software timeliness with uncertainty. ACM Trans. Softw. Eng. Methodol. (TOSEM) **28**(1), 1–30 (2018)
11. Aldalur, I., Arrieta, A., Agirre, A., Sagardui, G., Arratibel, M.: A microservice-based framework for multi-level testing of cyber-physical systems. Softw. Quality J. 1–31 (2023)
12. Boufaied, C., Menghi, C., Bianculli, D., Briand, L., Parache, Y.I.: Trace-checking signal-based temporal properties: A model-driven approach. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 1004–1015 (2020)
13. Boufaied, C., Menghi, C., Bianculli, D., Briand, L.C.: Trace diagnostics for signal-based temporal properties. IEEE Trans. Softw. Eng. **49**(5), 3131–3154 (2023)
14. Menghi, C., Nejati, S., Gaaloul, K., Briand, L.C.: 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, pp. 27–38 (2019)
15. Menghi, C., Viganò, E., Bianculli, D., Briand, L.C.: Trace-checking cps properties: bridging the cyber-physical gap. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 847–859. IEEE (2021)
16. Han, L., Ali, S., Yue, T., Arrieta, A., Arratibel, M.: Uncertainty-aware robustness assessment of industrial elevator systems. ACM Trans. Softw. Eng. Methodol. **32**(4), 1–51 (2023)
17. Gartziandia, A., Arrieta, A., Ayerdi, J., Illarramendi, M., Agirre, A., Sagardui, G., Arratibel, M.: Machine learning-based test oracles for performance testing of cyber-physical systems: an industrial case study on elevators dispatching algorithms. J. Softw.: Evol. Process **34**(11), 2465 (2022)
18. Arrieta, A., Ayerdi, J., Illarramendi, M., Agirre, A., Sagardui, G., Arratibel, M.: Using machine learning to build test oracles: an industrial case study on elevators dispatching algorithms. In: 2021

IEEE/ACM International Conference on Automation of Software Test (AST), pp. 30–39. IEEE (2021)
19. Ayerdi, J., Terragni, V., Arrieta, A., Tonella, P., Sagardui, G., Arratibel, M.: Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1264–1274 (2021)
20. Ayerdi, J., Segura, S., Arrieta, A., Sagardui, G., Arratibel, M.: Qos-aware metamorphic testing: an elevation case study. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), pp. 104–114. IEEE (2020)
21. Arrieta, A., Otaegi, M., Han, L., Sagardui, G., Ali, S., Arratibel, M.: Automating test oracle generation in devops for industrial elevators. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 284–288. IEEE (2022)
22. Valle, P., Arrieta, A., Arratibel, M.: Applying and extending the delta debugging algorithm for elevator dispatching algorithms (experience paper). In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1055–1067 (2023)
23. Xtext - Language Engineering Made Easy! https://eclipse.dev/Xtext/
24. Nejati, S., Gaaloul, K., Menghi, C., Briand, L.C., Foster, S., Wolfe, D.: Evaluating model testing and model checking for finding requirements violations in simulink models. In: Proceedings of the 2019 27th Acm Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1015–1025 (2019)
25. Mavridou, A., Bourbouh, H., Giannakopoulou, D., Pressburger, T., Hejase, M., Garoche, P.-L., Schumann, J.: The ten lockheed martin cyber-physical challenges: formalized, analyzed, and explained. In: 2020 IEEE 28th International Requirements Engineering Conference (RE), pp. 300–310. IEEE (2020)
26. Barney, G.: Transportation Systems in Buildings: CIBSE Guide D: 2010. Chartered Institution of Building Services Engineers, London (2010)
27. Boufaied, C., Jukss, M., Bianculli, D., Briand, L.C., Parache, Y.I.: Signal-based properties of cyber-physical systems: taxonomy and logic-based characterization. J. Syst. Softw. **174**, 110881 (2021)
28. Pradhan, S.M., Dubey, A., Gokhale, A., Lehofer, M.: Chariot: A domain specific language for extensible cyber-physical systems. In: Proceedings of the Workshop on Domain-specific Modeling, pp. 9–16 (2015)
29. Van Den Berg, F., Garousi, V., Tekinerdogan, B., Haverkort, B.R.: Designing cyber-physical systems with adsl: a domain-specific language and tool support. In: 2018 13th Annual Conference on System of Systems Engineering (SoSE), pp. 225–232. IEEE (2018)
30. Nandi, G.S., Pereira, D., Proença, J., Tovar, E.: Work-in-progress: a dsl for the safe deployment of runtime monitors in cyber-physical systems. In: 2020 IEEE Real-Time Systems Symposium (RTSS), pp. 395–398. IEEE (2020)
31. Klikovits, S., Linard, A., Buchs, D.: Crest-a dsl for reactive cyber-physical systems. In: System Analysis and Modeling. Languages, Methods, and Tools for Systems Engineering: 10th International Conference, SAM 2018, Copenhagen, Denmark, October 15–16, 2018, Proceedings 10, pp. 29–45. Springer (2018)
32. Vernotte, A., Cretin, A., Legeard, B., Peureux, F.: A domain-specific language to design false data injection tests for air traffic control systems. Int. J. Softw. Tools Technol. Transfer, 1–32 (2022)
33. Barreto, A.B., Hieb, M., Yano, E.: Developing a complex simulation environment for evaluating cyber attacks. In: Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC), vol. 12248, pp. 1–9 (2012)
34. Maciel, D., Paiva, A.C., Da Silva, A.R.: From requirements to automated acceptance tests of interactive apps: An integrated model-based testing approach. In: ENASE, pp. 265–272 (2019)
35. Menzel, T., Bagschik, G., Maurer, M.: Scenarios for development, test and validation of automated vehicles. In: 2018 IEEE Intelligent Vehicles Symposium (IV), pp. 1821–1827. IEEE (2018)
36. Queiroz, R., Berger, T., Czarnecki, K.: Geoscenario: An open dsl for autonomous driving scenario representation. In: 2019 IEEE Intelligent Vehicles Symposium (IV), pp. 287–294. IEEE (2019)
37. Afzal, A., Goues, C.L., Timperley, C.S.: Gzscenic: Automatic scene generation for gazebo simulator. arXiv preprint arXiv:2104.08625 (2021)
38. Arrieta, A., Agirre, J.A., Sagardui, G.: 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), pp. 1–5. IEEE (2020)
39. Bernaerts, M., Oakes, B., Vanherpen, K., Aelvoet, B., Vangheluwe, H., Denil, J.: Validating industrial requirements with a contract-based approach. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 18–27. IEEE (2019)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Pablo Valle is a PhD student in the Software Engineering and Systems research gorup at Mondragon University. His thesis focuses on simulation-based automated program repair for AI-enabled cyber-physical systems (CPSs). His research interests include software engineering, debugging, fault localization and isolation, repair, autonomous systems, and reliability and robustness of AI-driven CPSs.

Aitor Arrieta Aitor Arrieta is a Professor at Mondragon University. He got his PhD in software engineering at Mondragon University in 2017. His research interests lie in the field of software testing (e.g., search-based software testing, test oracle problem, regression testing, debugging, and repair), especially on testing untestable systems, such as cyber-physical systems, highly configurable systems, and machine learning-based systems.

Liping Han Her research interests include but are not limited to: uncertainty in software engineering, cyber-physical systems, software testing, and empirical software engineering.

Tao Yue Tao Yue's main research interests include model-driven engineering, uncertainty-aware software engineering, quantum software engineering, etc.

Shaukat Ali Shaukat Ali focuses on devising novel methods for verifying and validating classical and quantum software systems.