2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) | 979-8-3503-2498-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/MODELS-C59198.2023.00089

Supporting Early-Safety Analysis of IoT Systems by Exploiting Testing Techniques

Diego Clerissi
Universiy of Milano-Bicocca
Milano, Italy
diego.clerissi@unimib.it

Juri Di Rocco
University of l'Aquila
L'Aquila, Italy
juri.dirocco@univaq.it

Davide Di Ruscio
University of l'Aquila
L'Aquila, Italy
davide.diruscio@univaq.it

Claudio Di Sipio
University of l'Aquila
L'Aquila, Italy
claudio.disipio@univaq.it

Felicien Ihirwe
University of l'Aquila
L'Aquila, Italy
jeanfelicien.ihirwe@graduate.univaq.it

Leonardo Mariani
Universiy of Milano-Bicocca
Milano, Italy
leonardo.mariani@unimib.it

Daniela Micucci
Universiy of Milano-Bicocca
Milano, Italy
daniela.micucci@unimib.it

Maria Teresa Rossi
Universiy of Milano-Bicocca
Milano, Italy
maria.rossi@unimib.it

Riccardo Rubei
University of l'Aquila
L'Aquila, Italy
riccardo.rubei@univaq.it

***Abstract*—IoT systems' complexity and susceptibility to failures pose significant challenges in ensuring their reliable operation. Failures can be internally generated or caused by external factors, impacting both the system's correctness and its surrounding environment. To investigate these complexities, various modeling approaches have been proposed to raise the level of abstraction, facilitating automation and analysis. Failure-Logic Analysis (FLA) is a technique that helps predict potential failure scenarios by defining how a component's failure logic behaves and spreads throughout the system. However, manually specifying FLA rules can be arduous and error-prone, leading to incomplete or inaccurate specifications. In this paper, we propose adopting testing methodologies to improve the *completeness* and *correctness* of these rules. How failures may propagate within an IoT system can be observed by systematically injecting failures, while running test cases to collect evidence useful to add, complete and refine FLA rules.**

***Index Terms*—Internet of Things, Software Analysis, Model-Driven Engineering, Model-Based Testing**

I. Introduction

IoT systems may experience different failures, either internally generated or caused by the surrounding environment [1]. Such failures may affect not only the correctness of the system, but also the environment in which it operates. Consider, for instance, Smart Irrigation Systems: they monitor parameters related to weather and soil to irrigate crop fields based on the data collected automatically. A failure affecting the behaviour of those IoT systems may cause a waste of water or loss to the farm's production. Since IoT systems are composed of components of different natures (e.g., temperature/humidity sensors, cloud servers, and irrigation units), studying how failures (e.g., caused by a malfunctioning component) may propagate within a system and impact its behaviour can be highly challenging, further than being of high importance [2], [3].

Developing IoT systems is complex due to several reasons. The integration of diverse components, the need to handle real-time data, and the distributed nature of IoT systems are just a few factors contributing to this complexity. Several modeling approaches have been proposed over the last few years to raise the level of abstraction (e.g., [1], [4]–[6]), promoting the adoption of models for increasing automation and easing analysis. These models help understanding systems behaviour, performance, and potential failure scenarios [7].

Failure-Logic Analysis (FLA) [8] is one of the analyses that can be applied to IoT systems. By using FLA, it is possible to define how a component's failure logic shall behave, which can help analyze how failures could potentially spread throughout a system and predict any potential issue. For FLA to work correctly, it is important to have accurate information about how failures may occur within each component and propagate between components. FLA relies on the manual specification of rules, that rigorously indicate the different kinds of failures that might occur and how they can propagate throughout the components. Specifying such rules is a strenuous and error-prone process, as identifying all possible fault scenarios and formulating accurate rules is challenging, possibly leading to incomplete or incorrect specifications.

In this paper, we propose to adopt testing methodologies to mitigate the issues related to the *completeness* and *correctness* of the manually specified FLA rules. By systematically introducing failures into an IoT system and running test cases, it is possible to observe how failures propagate. The collected evidence is then used to add, refine, and eliminate FLA rules, better capturing the behavior of the system in failure scenarios.

979-8-3503-2498-3/23/$31.00 ©2023 IEEE
DOI 10.1109/MODELS-C59198.2023.00089

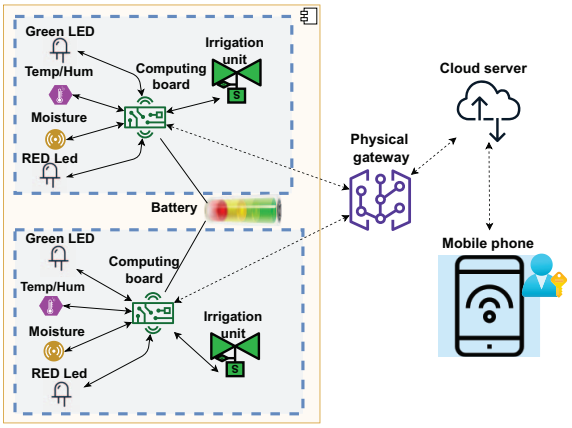

Fig. 1. The Smart Irrigation System use case.

The paper is organized as follows: In Section II we provide motivation for this work and present an explanatory example. We describe our approach in Section III. Section IV reports a preliminary evaluation of the proposed approach. In Section V, we discuss related work. Finally, Section VI concludes the paper and outlines future work.

II. Motivation and Background

Figure 1 represents a Smart Irrigation System (SIS) that includes all the building blocks of a typical IoT system, i.e., actuators, monitors, and sensors. The system analyzes the environmental conditions to automatically irrigate the soil using the classical MAPE-K control loop [9], [10]. In particular, each node (represented by the dashed line in Figure 1) is composed of different types of sensors, i.e., *Moisture*, *Temperature*, and *Humidity*.

Such sensors collect data at a given node and continuously feed it to the *Computing board*. Based on sensor data, the board decides whether to send a signal straight to the *Irrigation unit* actuator to start or stop the watering process. When the irrigation phase is ended, the *LED* indicators switch from green to red. The *Physical gateway* connects each irrigation node to the *Cloud server*, allowing users to remotely control, via *Mobile phone*, the irrigation nodes and analyze sensor data. Even though the presented system is simple, it represents a real-world application composed of miscellaneous IoT components that can be prone to critical malfunctioning. For instance, the Moisture sensor can send the wrong value, thus causing a waste of water or loss to the farm's production. Similarly, the user can erroneously decide to irrigate the field if the LED is malfunctioning. Therefore, failure propagation analysis plays an important role in understanding the system's behavior when it suffers from those faults.

An early-safety analysis approach has been proposed by Ihirwe [2], by relying on Failure-Logic Analysis (FLA) [11] mechanisms. FLA allows modelers to specify a component's failure logic behavior to help analyze how failures propagate within a system to anticipate possible misbehaviors. To be effective, FLA requires accurate knowledge about how failure

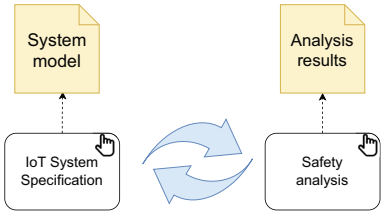

Fig. 2. Traditional failure analysis workflow.

may behave within the individual components. This can either be by means of propagation or transformation across components.

Figure 2 depicts the failure analysis workflow underpinning the approach proposed by Ihirwe [2]. First, the system is modeled by identifying all the needed components and communication channels. Afterward, the user has to check the system's safety by performing a proper failure propagation analysis. It is worth noting that the two phases are typically conducted manually with no or limited degree of automation [12], [13].

However, detecting those faults is a daunting task since thoroughly exercising an IoT system requires considering both the system and its environment. Therefore, a task of paramount importance is to detect how failures may propagate using early-safety analysis strategies. Even though several frameworks and techniques are in place [14]–[16], there is a need to verify the correctness of such rules at design time. Thus, the main challenges that need to be addressed when modeling IoT systems while supporting early-safety analysis are as follows:

- **CH1: Detecting fault propagation in IoT systems** While fault analysis has been studied in generic software systems [11], detecting failures in IoT systems has to consider real-time data that may introduce variability in the conducted analysis. Furthermore, failures that occur at the circuits-level should be considered in the analysis as they cause bugs that impact the source code [17], [18];
- **CH2: Verifying the completeness and correctness of fault propagation rules:** Even though fault propagation rules can be specified at the design time, their completeness and correctness cannot be granted *a priori*.

III. Proposed Approach

To detect fault propagation (CH1) and verify FLA rules (CH2), we propose a *Model-Based Test-Driven Safety Analysis* approach that allows engineers to identify potential failures and their propagation across components. The proposed approach implements and extends the one shown in Figure 2 and consists of the three main phases shown in Figure 3. The *IoT System Modeling* phase proposes tool-supported modeling of the IoT system-level architecture and the modeling of the system failure logic behavior. The *Fault-Tree Generation and Analysis* supports the analysis of failure propagation, with reference to the available rules. Finally, the *IoT System Testing* phase exploits the information in the model to execute

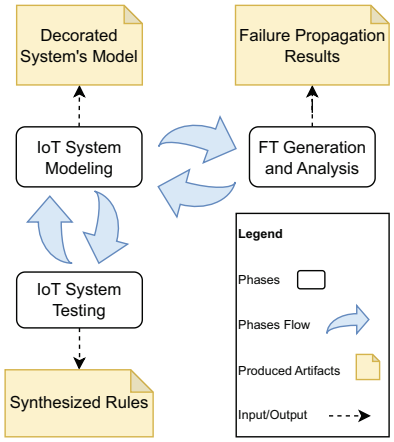

Fig. 3. Model-Based Test-Driven Safety Analysis.

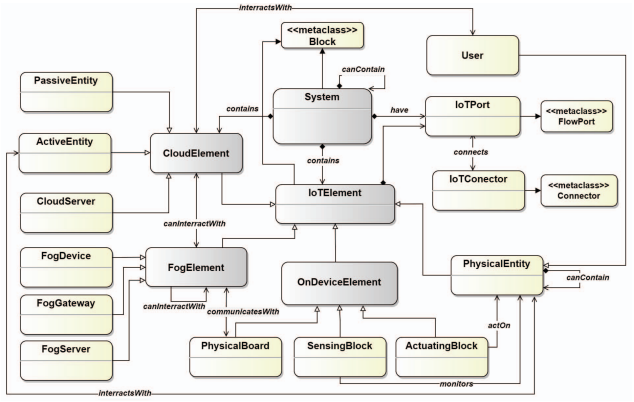

Fig. 4. CHESSIoT System-level meta-model [2].

the individual components while injecting failures on input ports and checking how they propagate to output ports. This phase can confirm or disprove the defined failure logic rules (correctness check) and discover new ones (completeness check). The outcome of both analysis and testing can be used to refine the system, and its model, to finally achieve a more reliable IoT system.

In the following, the three phases of the process shown in Figure 3 are described in details.

A. IoT System Modeling

In this phase, modelers specify the architecture of the IoT system and the failure-logic behaviour, as detailed below.

1) IoT system-level architecture: The proposed modeling approach runs on top of CHESSIoT [2], a model-driven environment to support the design and analysis of IoT systems. CHESSIoT provides a UML/SysML profile extension to reflect the constructs and semantics present in IoT system-level architectures. The CHESSIoT system-level modeling language was designed to satisfy the high-level specifications of a typical IoT system, supporting a multi-layered specification from the low-level edge layer to the fog layer and the cloud.

The language extends the SysML modeling language in terms of new IoT-specific stereotypes and their interrelations. Ports enable interactions among components and are fundamental for determining error propagation paths. Figure 4 presents the CHESSIoT system-level meta-model. It permits to specify IoT systems as a collection of physical devices and entities connected to collect, process, send, receive, and store data. The *IoTElement* represents physical entities, ranging from microcontrollers at the thing layer to cloud servers. The modeling layers can be grouped into *edge*, *fog*, and *cloud*.

OnDeviceElements are low-level IoT devices that contribute to the system's functional behavior, while *PhysicalBoard* represents hardware controllers and *PhysicalEntity* is any physical object or environment. *Fog devices* perform preliminary computations and convey results to on-device elements, with storage and processing capacities varying depending on the use case and hardware and software features.

On the cloud layer, devices operate at the cloud level and contribute to the overall functionality of the system. *Consumer entities* can be *active* or *passive*, with active consumer entities being computer-running software to monitor and control sensors remotely, and passive consumer entities being traffic light actuators.

2) Failure-Logic behavior modeling and analysis: Once the IoT system model is defined, the safety engineer derives and annotates the failure behavior rules for each modeled component by following the Failure Propagation Transformation Calculus (FPTC) [19] notation. Based on its nature, a component can propagate a failure (carrying a failure from input to output), transform a failure (changing the nature of a failure from input to output), act as a source of failure (creating a failure despite no failure in input), or act as a sink (avoiding the failure to be either propagated or transformed).

The following three abstract categories of failure types are assessed: *service provision failures*, such as the omission or commission of the output; *timing failures*, such as the early or late delivery of the output; and *value domain failures*, such as the output value being out of a valid range, stuck, or exhibiting erratic behavior. In addition, a *noFailure* annotation is used to indicate a no-failure type at the input port. Table I shows different failure types and their descriptions.

TABLE I
FAILURE TYPES.

Failure type	Description
Early	Output provided too early
Late	Output provided too late
ValueCoarse	Output out of range
ValueSubtle	Output in-range but erroneous
Omission	Output expected but not provided
Commission	Output provided but not expected

As previously mentioned, component failures can be propagated or transformed:

- ***Failure propagation:*** It occurs in a component when a single input port failure condition is directly transferred

to its output ports without changing its nature. For instance, Equation 1 shows a simple example of a failure propagation of $failure_1$ from port $p_{(in)}$ to port $p_{(out)}$ of a simple component. Propagation also occurs between two connected components when a failure condition at the output port of the preceding component is transferred to the input port of the following component.

$$p_{(in)}.failure_1 \rightarrow p_{(out)}.failure_1; \quad (1)$$

As an example from our specific scenario, this can happen when a board gets erroneous data from the sensors (i.e., `ValueCoarse`) and sends it directly to its output ports. This scenario would be expressed as in Equation 2, where $Bd_{(in)}$ and $Bd_{(out)}$ are the input and the output ports of the board, respectively.

$$Bd_{(in)}.valueCoarse \rightarrow Bd_{(out)}.valueCoarse; \quad (2)$$

- ***Failure transformation:*** It occurs within a component when a failure condition at the input port is converted into another type before reaching the output port. An example is shown in Equation 3. A failure transformation can also occur when more than one failure expression of any type, except a `NoFailure` or *wildcard* at multiple input ports, is transmitted on a single output port (see Equation 4). Even if the failure has the same type, the fact that the component converts two failures at its input ports to a single failure at the output port is regarded as a failure transformation.

$$p_{(in)}.failure_{(in)} \rightarrow p_{(out)}.failure_{(out)}; \quad (3)$$

$$p_{(in_1)}.failure_1, .., p_{(in_N)}.failure_N \rightarrow p_{(out)}.failure_{(out)}; \quad (4)$$

To make an example of failure propagations and transformations, let us consider the explanatory irrigation system with two motors controlled by a relay driver. The relay driver enables them to turn on and off depending on the location to be irrigated. To control the relay driver, the computer board sends the analog signal through two relay driver-controlling ports. To define the failure behavior of the irrigation unit, we must first understand the variety of failure scenarios that can occur with an irrigation unit. For example, two input ports may not get a signal from the board, causing the relay driver to be unable to switch on and off the motors. Another example is when the signal arrives at the input port later or earlier than anticipated. As a result, the relay will unexpectedly turn on and off the motors.

Table II shows a sample of the failure rules that specify the number of failure situations for the irrigation unit, including the ones described above. Note, $Irr_{(in_1)}$ and $Irr_{(in_2)}$ are defined as input ports, while $Irr_{(out_1)}$ and $Irr_{(out_2)}$ are defined as output ports.

When modeling the IoT system's Failure-Logic behavior is finished, the FLA analysis can be executed. This analysis

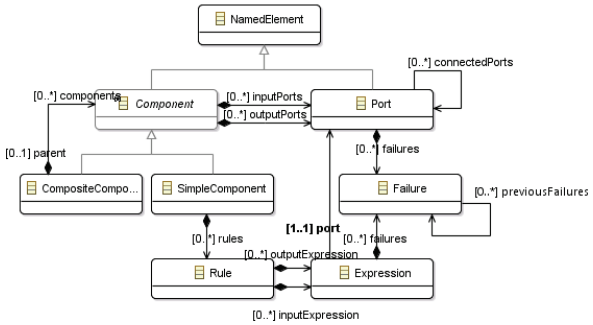

Fig. 5. FLA meta-model [2].

considers the annotated CHESSIoT model and transforms it into an FLA model [8]. The transformation calculates a complete system's failure behavior starting from the failure behavior rules of the system's composite components and their interconnections. This, in turn, means that the failure behaviors of composite elements are also determined by the failure behaviors of their individual simple components.

As shown in the FLA meta-model in Figure 5, FLA models consist of *composite components*, representing sub-systems containing one or more sub-components. These components do not possess failure behavior by themselves; instead, they rely on their sub-components to determine their failure conditions. On the other hand, a simple component represents a functional component whose failure may contribute to a system failure. Each component contains input and output ports with their corresponding failure rules.

B. Fault-Tree Generation and Analysis

The Fault-Tree Analysis (FTA) [11] aims to graphically analyze the system's final failure behavior based on the FLA input. Fault trees depict the system failure logic outcomes in a tree structure, making it simple to navigate and trace influences from a system-level danger to specific failures from system components and sub-components. In addition to that, it is also possible to perform analyses on it to determine minimal failure events that are required to trigger such hazards

The Fault-Tree generation is performed through a series of model-to-model transformations from the FLA model to a series of Fault-Tree (FT) models. An FT is generated for each of the failures that propagate to the targeted output port of the system, and contains logical networks of events and corresponding gates that together form a failure representation tree, reflecting the system's failure behavior set by the user and the system's functional architecture. Each FT event has its own unique identity in the tree and can be of type basic, intermediate, external, or undeveloped, depending on the stage at which it manifests.

In the FT generation process, each FT is built recursively. A top event is initially generated due to the failure's propagation to the system output port. In terms of logical gates used in the FT, only **AND** and **OR** gates are adopted. The events gate

TABLE II
SAMPLE FLA RULES OF THE IRRIGATION UNIT.

Rule	Description
$Irr_{(in_1)}.omission, Irr_{(in_2)}.omission \rightarrow Irr_{(out_1)}.omission, Irr_{(out_2)}.omission$	The input ports receive no signal, causing the relay driver to be unable to turn on/off the motors.
$Irr_{(in_1)}.early, Irr_{(in_2)}.early \rightarrow Irr_{(out_1)}.commission, Irr_{(out_2)}.commission$	The input ports receive the signal earlier than expected, causing the relay driver to unexpectedly turn on/off the motors.
$Irr_{(in_1)}.late, Irr_{(in_2)}.late \rightarrow Irr_{(out_1)}.commission, Irr_{(out_2)}.commission$	The input ports receive the signal later than expected, causing the relay driver to unexpectedly turn on/off the motors.
$Irr_{(in_1)}.valueSubtle, Irr_{(in_2)}.valueSubtle \rightarrow Irr_{(out_1)}.early, Irr_{(out_2)}.early$	The input ports receive an erroneous but in-range signal, causing the relay driver to turn on/off the motors earlier than expected.
$Irr_{(in_1)}.valueSubtle, Irr_{(in_2)}.valueSubtle \rightarrow Irr_{(out_1)}.late, Irr_{(out_2)}.late$	The input ports receive an erroneous but in-range signal, causing the relay driver to turn on/off the motors later than expected.

is created systematically. An **AND** gate is used to indicate a failure transformation from an input to an output port of a component. On the other hand, an **OR** gate is used to depict a failure propagation case. The **OR** gate can also depict a scenario in which one or more failure outputs from distinct components are passed to the input of the following component.

The intermediate events are created and populated into the FT based on the failure expressions and the components they are assigned to. The FT population involves a recursive transformation process in which components, ports, and their corresponding rules are recursively parsed. So, at this stage, the only crucial stopping case is reached when the transformation hits a condition matching a basic failure, an underdeveloped failure (i.e., an insufficient source failure), or an externally injected failure. For instance, Figure 6 depicts a simple transformation example with indications showing a transformation mapping of Equation 4. From the example, each of the output expressions is mapped to an output event of a logical combination of the input expressions. Each input expression is mapped to an event, and the type of such event is determined by the expression condition. In addition to that, the logical gates are defined based on the nature of the input expressions to satisfy the failure propagation and transformation concepts.

As the system gets bigger and more complex, which in turn requires a large number of rules to better cover all possible failure scenarios, the generated FTs inevitably become even bigger and harder to grasp. To tackle such a challenge, different analysis mechanisms are used to systematically extract meaningful insight from the generated tree. CHESSIoT supports *"Qualitative"* fault tree analysis mechanisms in which only the essential FT representations are kept. This process involves the removal of internal component failure propagations, external component-to-component failure propagations, and basic event redundancies. In addition, CHESSIoT also supports *"Quantitative"* analysis that automatically calculates the failure probabilities of an entire system from its constituent parts' failure probabilities.

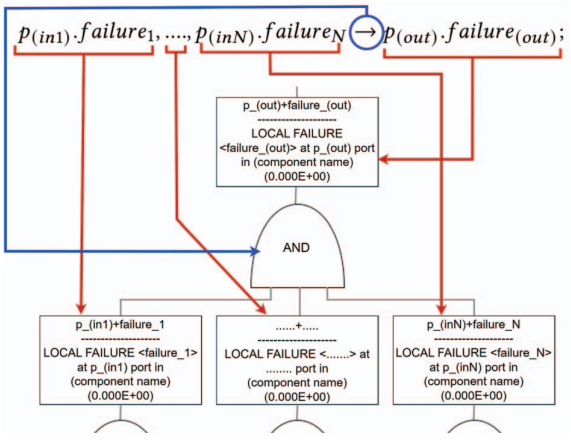

Fig. 6. FT corresponding to Equation 4.

C. IoT System Testing

The third phase of the approach shown in Figure 3 concerns testing the modeled IoT system to confirm or disprove the defined failure logic rules (correctness check) and discover new ones (completeness check). This phase is guided by the information collected from the system's model and consists of three main activities: the *Isolation* of the components to be exercised, the *Testing* of the isolated components to collect observations about how failures are propagated from inputs to outputs, and the *Rules Generation* from the observations, as shown in Figure 7.

The *Isolation* activity isolates the component under test from the rest of the system using stubs and probes. The stubs are connected to the input ports of the component while removing the original connections. The probes are connected to the output ports of the components, again removing the original connections. In this configuration, the stubs generate the input values in a controlled and coordinated way, whereas monitoring probes capture and record the output values produced by the component under test. Figure 8 shows how the explanatory component *Irrigation unit* is isolated to support the discovery of failure propagation patterns. Isolation is a simple activity performed visually on the model.

The *Testing* activity consists of exercising the isolated component through the stubs to log evidence about how failures are propagated through the probes. Since the inputs of an isolated component are fully controllable and its outputs can be fully observed, it is possible to systematically generate tests that include failures in the inputs and observe how and if they propagate to the outputs.

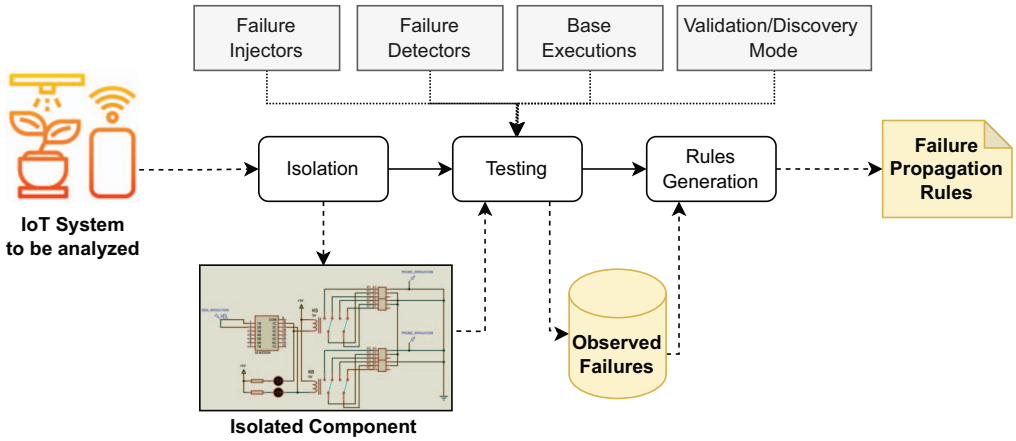

Fig. 7. IoT System's testing process.

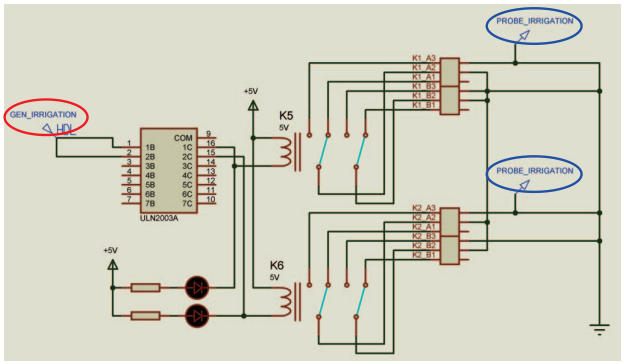

Fig. 8. Irrigation unit component isolated.

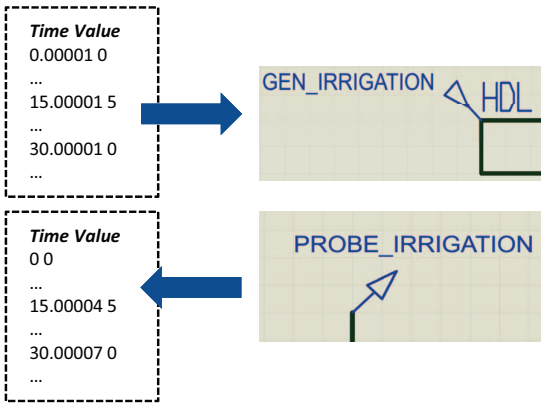

Fig. 9. Input & output time series of Irrigation unit component.

The strategy to observe how failures propagate is based on a variant of *differential testing* [20], [21]. In differential testing, the same inputs are executed on two comparable implementations (e.g., two compilers for the same language), and the outputs are directly compared to discover possible faults. In this case, we start from a *base execution* `t=(I,O)`, where `I` is a set of time series values, each one representing a sequence of input values for an input port, and `O` is a set of time series values, each one representing a sequence of output values observed for an output port. Figure 9 shows an example of a base execution of the Irrigation Unit component. The time series provided by the stub in input activates the component from second 15.00001 to second 30.00001, by sending 5 Volts to the circuits to turn on the water fans and the LED associated with that input port; instead, when the value is set to 0, the component is not active as not stimulated by any Volt. The time series in output reflects the behavior instrumented in input, as the component results active, for each output port, from second 15.00004 to second 30.00007 (differences with respect to input are minimal and depend on the precision of Proteus tool[1], which is used for this work as design and simulation environment).

To discover how failures propagate, our approach automatically modifies the inputs I used in a base execution t by systematically injecting failures of the given types on the inputs. The approach then executes the modified inputs I' and collects the outputs, namely O'. The comparison of the output produced by the base execution O and the output generated by the mutated execution O' reveals if and how the input failure(s) propagated to the output.

We need two main elements for each supported failure type to execute this process: a failure injector and a failure detector. The *failure injector* is a function that, given an input time series, modifies its values to obtain a minimally modified time series that includes the failure of the given type. The *failure detector* is a function that, given two time series, one obtained from the base execution and another obtained from a mutated execution, can tell if the failure of the given type is present in the output.

Table III summarizes the failure injectors and the failure detectors defined for the fault types currently supported by our implementation for IoT systems. Note that both injectors

[1]https://www.labcenter.com/iotbuilder/

and detectors can be parameterized with respect to the actual use case, to reflect the characteristics and volatility of the signals. TS and $TS^{'}$ represent base and mutated time series, respectively, ε_t and ε_v determine the tolerance on timing and values variations, and $[v_{min}, v_{max}]$ are the range of accepted values for the considered use case. As an example, an `Early` failure injector may mutate the base execution of Figure 9, by changing the component activation time from second 15.00001 to 12.00001, whereas an `Early` detector may compare base and mutated executions to determine how this propagates in output.

These elements are combined into a fully automated testing process that repeatedly instantiates the isolated component, generates the mutated execution (using the failure injectors), runs the component with the mutated execution, collects observations checking for the presence of failures (using the failure detectors), and destroys the instance of the isolated component. This process is repeated multiple times for all the combinations of failures to be investigated.

Additional data can be collected by considering multiple base executions. In particular, we envision the possibility to start from a set of base executions that cover the various possible states of the components under test, so that the propagation of failures can be studied in multiple contests (i.e., states). For the moment, we assume the user of the approach shall define the states that must be used in this process, and thus provide a set of base executions that cover the relevant states. In the future, we would like to explore the automatic generation of the base executions.

At the end of this process, for each combination of input failures $p_{(in1)}.failure^{i_1}_{(in1)}, .., p_{(inN)}.failure^{i_N}_{(inN)}$, a set of observations Obs_i where $Obs_i \in \{failure^1 \dots failure^k\} \cup NoFailure$, are available.

This testing phase can be configured to work in two different modes: (i) *validation* of existing failure propagation rules (validation mode) or (ii) *discovery* of new failure propagation rules (discovery mode).

The *validation mode* provides a cost-effective targeted exploration of the combinations of failures, and their propagation. In particular, for each failure propagation rule $p_{(in1)}.failure_{(in1)}, .., p_{(inN)}.failure_{(inN)} \rightarrow p_{(out)}.failure_{(out)}$, the validation mode investigates how failures propagate to the output when the failures in the input ports are consistent with the left-hand side of the rule, i.e., $failure_{(in1)}, .., failure_{(inN)}$. The final set of derived rules will either confirm or disprove the existing rules.

The *discovery mode* is a more expensive but systematic exploration of the possibile combinations of failures to derive rules about their propagation. Given a set of k failure types $failure^1 \dots failure^k$ and N input ports, this mode investigates how, and if, failures propagate for every combination of input failures $p_{(in1)}.failure^{i_1}_{(in1)}, .., p_{(inN)}.failure^{i_N}_{(inN)}$, where each tuple indicates the failures present in the input ports, and $failure^i$ is any of the considered failure types or `NoFailure`. For instance, if the `Late (L)`, and `Early (E)` failure types are investigated for two input ports, six combinations of inputs failures ($\langle L, - \rangle, \langle -, L \rangle, \langle R, - \rangle, \langle -, R \rangle, \langle L, R \rangle, \langle R, L \rangle$, where $-$ represents `NoFailure`), obtained by every possible permutation of the considered failures, are considered. The final set of rules will provide a comprehensive view about how failures are propagated by the considered component.

In practice, the validation mode constraints the discovery to the subset of failures used in the rules to be validated, while the discovery mode considers every possible combination of failures.

Finally, the *Rules Generation* activity of Figure 7 consists of the generation of the actual set of failure propagation rules. This is done by extracting the set of failures types $f_1, \dots f_p$ present in a set of observations Obs_i associated with a same pattern of failures $p_{(in1)}.failure^{i_1}_{(in1)}, .., p_{(inN)}.failure^{i_N}_{(inN)}$ and generating the rule $p_{(in1)}.failure^{i_1}_{(in1)}, .., p_{(inN)}.failure^{i_N}_{(inN)} \rightarrow f_1 or \dots or f_p$, which is finally encoded in the failure analysis tool as a set of p non-deterministc rules $p_{(in1)}.failure^{i_1}_{(in1)}, .., p_{(inN)}.failure^{i_N}_{(inN)} \rightarrow f_i$, where $i = 1 \dots p$.

The discovered rules may confirm or disprove the existing failure propagation rules and discover new ones. This phase may thus trigger an evolution of the system's model by refining the system failure logic behaviour definition. From this evolution, the whole process may be re-triggered to achieve satisfactory reliability for the candidate IoT system.

IV. Evaluation

To evaluate our approach, we selected the Irrigation unit component, introduced in Figure 7 and shown in more details in Figure 8. The component, designed with Proteus, comprises two main circuit streams that decide the activation of two water fans and two LEDs (each circuit controls a water fan and a LED). Once isolated, the input signals of the circuit are generated by the *GEN_IRRIGATION* stub (red circle in Figure 8), while the output signals are linked to the *PROBE_IRRIGATION* probes that record the data produced by the circuits (blue circles in Figure 8).

We use the base execution described in Figure 9 as a basis to study failure propagation. In this scenario, *both the water fans and LEDs start as turned off, then they are turned on for a fixed timespan by an external request, and finally they are turned off again*. As mutated scenarios, we considered every possible combination of the currently supported failure types (i.e., `Early`, `Late`, `ValueCoarse`, and `ValueSubtle`, from Table I) for the two input ports available in the component. Every combination was repeated 3 times, in order to capture variations in the executions, for a total of 48 different mutated scenarios produced by our *injectors*. We injected each combination separately, ran the mutated scenario, and compared the results via *detectors* to discover how failures propagate.

The experiment's results are summarized in Table IV. The column labeled *IN Failures* shows the combinations of failures

TABLE III
FAILURE INJECTORS & DETECTORS.

Type	Injectors	Detectors
`Early`	$(t_i, v_i) \in TS \rightarrow (t_i - x, v_i) \in TS^{'}$, where $(x > \varepsilon_t)$	$(t_i, v_i) \in TS, (t_j, v_j) \in TS^{'} \rightarrow true$, if $(v_i = v_j) \wedge (t_j < t_i - \varepsilon_t)$
`Late`	$(t_i, v_i) \in TS \rightarrow (t_i + x, v_i) \in TS^{'}$, where $(x > \varepsilon_t)$	$(t_i, v_i) \in TS, (t_j, v_j) \in TS^{'} \rightarrow true$, if $(v_i = v_j) \wedge (t_j > t_i + \varepsilon_t)$
`ValueCoarse`	$(t_i, v_i) \in TS \rightarrow (t_i, v_i \pm x) \in TS^{'}$, where $(x > \varepsilon_v) \wedge ((v_i - x < v_{min}) \vee (v_i + x > v_{max}))$	$(t_i, v_i) \in TS, (t_j, v_j) \in TS^{'} \rightarrow true$, if $(t_i = t_j) \wedge ((v_j < v_{min}) \vee (v_j > v_{max}))$
`ValueSubtle`	$(t_i, v_i) \in TS \rightarrow (t_i, v_i \pm x) \in TS^{'}$, where $(x > \varepsilon_v) \wedge (v_{min} \leq \lvert v_i \pm x \rvert \leq v_{max})$	$(t_i, v_i) \in TS, (t_j, v_j) \in TS^{'} \rightarrow true$, if $(t_i = t_j) \wedge (\lvert v_i - v_j \rvert > \varepsilon_v) \wedge (v_{min} \leq v_j \leq v_{max})$

that were injected into the two input ports, while the column labeled *OUT Failures* displays the combination of failures that were observed on the two output ports. As each combination of input failures was executed multiple times, it was possible to observe various combinations of failures on the output. For example, when the first and second unit ports were given the failures `Early` and `ValueSubtle`, respectively, two possible combinations of output failures were identified: `Early-Late` and `Early-NoFailure`.

The `green color` highlights a failure type that propagates unchanged from the input to the output. The `red color` indicates a failure type that is transformed as a failure of a different type. Finally, the `blue color` determines failures that are masked by the implementation and thus do not propagate to the output.

Notably, `Early` and `Late` failures are always propagated to output ports, whereas `ValueSubtle` and `ValueCoarse` produce different outcomes depending either on the position within the time series of the value affected by the mutation or the magnitude of the mutation. In fact, when a wrong voltage value, injected as either a `ValueSubtle` or a `ValueCoarse`, occurs at the beginning of the time series, changing the original value to a value near or below 0, the component responsible for turning on the water fans and the LEDs is markedly delayed, resulting in the detection of a `Late` failure on output ports. Instead, when the mutated voltage value is set close to 5 Volts, i.e., the maximum accepted value provided by the injector in input according to the use case, or even higher, no notable changes are detected, resulting in a `NoFailure`. This is because the Irrigation unit component in Proteus is configured to flatten voltage values up to 5 Volts. Interestingly, depending on the context of the failure and the specific value, it may either mask the failure or transform the failure into a failure of a different kind.

These results contributed to improving the knowledge of the engineers about the fault tolerance of the system. In fact, the engineers' supposed failures would only propagate unchanged through the component, while failure propagation rules show more complicated, sometimes context-dependent, patterns. Further, the component could sometime mask the effect of the same failures. For instance, by referring to Table II, just one of the sample FLA rules (last row) was actually confirmed by our testing techniques. This is primarily because our testing approach has yet to cover the `Commission` and `Omission` failure types, which appear in the prior three rules in Table II. Please note that not all the recommended tests listed in Table IV will necessarily be used in the FLA analysis. However, having these results can provide the user with greater clarity regarding potential failure combinations, which can improve the accuracy and comprehensiveness of the FLA rules and generated fault trees.

V. RELATED WORK

In this section, we review *i)* the most relevant Model-Driven Engineering (MDE) approaches applied in the context of IoT *ii)* applications of mutation testing in fault analysis, and, *iii)* specification mining in software verification.

A. MDE for IoT development

Ciccozzi *et al.* [22] exploits the MDE paradigm to enable the abstraction of IoT systems. They propose exploiting the MDE paradigm to enable the abstraction of IoT systems, the easy handling of the various degrees of automation in software development, and the performance analysis of the system from different perspectives.

Thramboulidis *et al.* [23] developed an MDE approach to face the complexity of IoT-based cyber-physical manufacturing systems. The conceived language allows domain experts to integrate IoT protocols during the system specification.

ThingML [24] is an IoT engineering platform that combines well-proven textual software-modeling constructs aligned with UML, such as statecharts and components, with an imperative platform-independent action language for developing IoT applications.

Fortas *et al.* [25] exploit MDE techniques to build an approach supporting the development and testing of IoT applications. In particular, they use ThingML [24] in the modeling process and Proteus for simulation.

Monitor-IoT *et al.* [6] is a graphical designer based on the Obeo Designer Community and Eclipse Sirius tools. The framework allows developers to model IoT multi-layer monitoring architectures. The tool enables the definition of computing nodes and their resources that support the monitoring processes, i.e., data collection, transport, processing, and storage. Monitor-IoT is flexible enough to support the modeling at the edge, fog, and cloud layers.

B. Mutation testing in fault analysis

Praphamontripong *et al.* [26] present an approach to testing Web applications by applying mutation analysis to the connections among Web application software components. The authors showed the effectiveness of mutation analyses in

TABLE IV
FAILURES PROPAGATION IN IRRIGATION UNIT EXPERIMENT.

IN Failures	OUT Failures
Early – Early	Early – Early
Early – Late	Early – Late
Early – ValueCoarse	Early – Late, Early – NoFailure
Early – ValueSubtle	Early – Late, Early – NoFailure
Late – Early	Late – Early
Late – Late	Late – Late
Late – ValueCoarse	Late – Late, Late – NoFailure
Late – ValueSubtle	Late – Late, Late – NoFailure
ValueCoarse – Early	Late – Early, NoFailure – Early
ValueCoarse – Late	Late – Late, NoFailure – Late
ValueCoarse – ValueCoarse	NoFailure – Late, Late – NoFailure, Late – Late, NoFailure – NoFailure
ValueCoarse – ValueSubtle	NoFailure – Late, Late – NoFailure, Late – Late, NoFailure – NoFailure
ValueSubtle – Early	Late – Early, NoFailure – Early
ValueSubtle – Late	Late – Late, NoFailure – Late
ValueSubtle – ValueCoarse	NoFailure – Late, NoFailure – NoFailure, Late – Late, Late – NoFailure
ValueSubtle – ValueSubtle	NoFailure – Late, NoFailure – NoFailure, Late – Late, Late – NoFailure

creating tests supporting fault detection. The proposed type of analysis is able to discover also new mutation operators.

Similarly, Moran *et al.* [27] applied mutation testing in the context of mobile Android applications. Besides the application of empirically derived operations, the proposed tool supports the automation of the process of detecting potential mutant locations, generating mutants, and discovering new operations.

Humbatova *et al.* [28] present an approach for testing Deep Learning (DL) solutions. The authors extracted mutation operators from existing fault taxonomies. Then, they assessed the mutation operators to understand whether they produce killable, but not trivial, mutations. Eventually, they evaluated the approach by comparing it with the existing DeepMutation++ DL mutation tool. The results showed that their operators can discriminate more effectively between a weaker from a more robust test set.

Belli *et al.* [29] propose a model-based mutation testing approach for industrial systems based on directed graphs. The approach generates mutants and injects faults at the model level. In such a way, the mutation testing strategy can be applied even when the source code is unavailable. They only use two mutation operators, omission and insertion, by means of directed graphs. Then, these graphs are semantically enriched and exemplified using a collection of graph-based models to generate other operators.

C. Specification Mining

Concerning the understanding of system behaviour, *specification mining*, intended as the extraction of high-level specifications from existing code, may play a key role. Approaches exploiting mined specifications can be used for program understanding but also for formal verification.

Dallmeier *et al.* [30] propose *TAUTOKO*, a typestate miner that combines systematic test case generation and typestate mining. Using those strategies. the approach systematically extends the execution space and enriches the final specification by increasing true positives.

The need for good specifications for effective system verification is also highlighted by Cao *et al.* [31]. They adopted a rule-based specification mining approach that explores the search space of all possible rules and uses interestingness measures to differentiate specifications from false positives. Then, the authors propose a learning-to-rank-based approach to consider 38 available interestingness measures together and investigate their combinations. Their experiment results show that the learning-to-rank-based approach can improve the best ranking performance using a single measure by up to 66%.

ARTINALI++ [32] tool dynamically mines specifications of Complex Cyber-Physical Systems (CPS) to manage security issues. The approach generates a multi-dimensional model that is capable of embodying time, data, and events into the specifications. *ARTINALI++* has been validated using three CPS platforms for intrusion detection. The results showed an average of 97.7% detection accuracy across platforms while incurring reasonable performance and memory overheads.

VI. CONCLUSION AND FUTURE WORK

This paper discussed the challenges and importance of supporting early safety analysis of IoT systems, which are susceptible to various failures that can impact their functionality and the environment they operate in. Failure propagation within these systems is complex due to their diverse components and distributed nature. To address this, the paper discusses using Failure-Logic Analysis (FLA) to understand how component failures may propagate and affect the system's behavior. However, FLA relies on manually specified rules, which can be error-prone and incomplete. The paper proposes adopting testing methodologies to mitigate the issues with manually specified FLA rules. Potential faults can be observed and identified by subjecting the IoT system to various test cases. By means of the proposed testing techniques, it is possible to support the validation of the correctness of the system's behavior and the effectiveness of the specified rules in capturing fault scenarios. Future plans include the support of all the fault types that can be specified at the level of IoT system modeling. Moreover, we intend to investigate the generalizability of the proposed technique by considering different execution environments than Proteus.

ACKNOWLEDGMENT

This work has been partially supported by the EMELIOT national research project, which has been funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY)

REFERENCES

[1] J. C. Kirchhof, B. Rumpe, D. Schmalzing, and A. Wortmann, "Montithings: Model-driven development and deployment of reliable IoT applications," *Journal of Systems and Software*, vol. 183, p. 111087, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001849

[2] F. Ihirwe, "Low-Code Engineering for the Internet of Things," Ph.D. dissertation, University of L'Aquila, 2023.

[3] A. Power and G. Kotonya, "Providing fault tolerance via complex event processing and machine learning for IoT systems," in *Proceedings of the 9th International Conference on the Internet of Things*, ser. IoT '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3365871.3365872

[4] F. Ciccozzi and R. Spalazzese, "MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering," in *Intelligent Distributed Computing X*, C. Badica, A. El Fallah Seghrouchni, A. Beynier, D. Camacho, C. Herpson, K. Hindriks, and P. Novais, Eds. Cham: Springer International Publishing, 2017, pp. 67–76.

[5] H. Muccini and M. Sharaf, "CAPS: Architecture Description of Situational Aware Cyber Physical Systems," in *2017 IEEE International Conference on Software Architecture (ICSA)*, 2017, pp. 211–220.

[6] L. Erazo-Garzón, P. Cedillo, G. Rossi, and J. Moyano, "A domain-specific language for modeling IoT system architectures that support monitoring," *IEEE Access*, vol. 10, pp. 61 639–61 665, 2022.

[7] F. Ihirwe, D. Di Ruscio, S. Mazzini, and A. Pierantonio, "Towards a modeling and analysis environment for industrial IoT systems," 2021.

[8] B. Gallina, M. A. Javed, F. U. Muram, and S. Punnekkat, "A Model-Driven Dependability Analysis Method for Component-Based Architectures," in *Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications*, 2012.

[9] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[10] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, *Engineering Self-Adaptive Systems through Feedback Loops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 48–70. [Online]. Available: https://doi.org/10.1007/978-3-642-02161-9_3

[11] L. Xing and S. V. Amari, "Fault Tree Analysis," in *Handbook of Performability Engineering*, K. B. Misra, Ed., 2008, pp. 595–620.

[12] J. P. Dias, F. Couto, A. C. Paiva, and H. S. Ferreira, "A Brief Overview of Existing Tools for Testing the Internet-of-Things," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018, pp. 104–109.

[13] P. Giménez, B. Molína, C. E. Palau, and M. Esteve, "SWE Simulation and Testing for the IoT," in *2013 IEEE International Conference on Systems, Man, and Cybernetics*, 2013, pp. 356–361.

[14] R. Cristea, M. Feraru, and C. Paduraru, "Building blocks for IoT testing - A benchmark of IoT apps and a functional testing framework," in *2022 IEEE/ACM 4th International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)*, May 2022, pp. 25–32.

[15] M. Bures, B. S. Ahmed, V. Rechtberger, M. Klima, M. Trnka, M. Jaros, X. Bellekens, D. Almog, and P. Herout, "PatrIoT: IoT Automated Interoperability and Integration Testing Framework," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2021, pp. 454–459, iSSN: 2159-4848.

[16] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated IoT safety and security analysis," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 147–158. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/celik

[17] T. Booth, S. Stumpf, J. Bird, and S. Jones, "Crossed Wires: Investigating the Problems of End-User Developers in a Physical Computing Task," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 3485–3497. [Online]. Available: https://doi.org/10.1145/2858036.2858533

[18] A. Makhshari and A. Mesbah, "IoT Bugs and Development Challenges," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 460–472.

[19] M. Wallace, "Modular Architectural Representation and Analysis of Fault Propagation and Transformation," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 3, pp. 53–71, 2005.

[20] G. Barany, "Finding Missed Compiler Optimizations by Differential Testing," in *Proceedings of the 27th International Conference on Compiler Construction*, ser. CC 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 82–92. [Online]. Available: https://doi.org/10.1145/3178372.3179521

[21] C. Chen, P. Ren, Z. Duan, C. Tian, X. Lu, and B. Yu, "SBDT: Search-Based Differential Testing of Certificate Parsers in SSL/TLS Implementations," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 967–979. [Online]. Available: https://doi.org/10.1145/3597926.3598110

[22] F. Ciccozzi, I. Crnkovic, D. Di Ruscio, I. Malavolta, P. Pelliccione, and R. Spalazzese, "Model-Driven Engineering for Mission-Critical IoT Systems," *IEEE Software*, vol. 34, no. 1, pp. 46–53, 2017.

[23] K. Thramboulidis, P. Bochalis, and J. Bouloumpasis, "A Framework for MDE of IoT-Based Manufacturing Cyber-Physical Systems," in *Proceedings of the Seventh International Conference on the Internet of Things*, ser. IoT '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3131542.3131554

[24] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, "ThingML: A Language and Code Generation Framework for Heterogeneous Targets," ser. MODELS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 125–135.

[25] A. Fortas, E. Kerkouche, and A. Chaoui, "Application of MDE in the Development of IoT Systems: A Simulation-Based Approach," in *2022 First International Conference on Computer Communications and Intelligent Systems (I3CIS)*, 2022, pp. 93–98.

[26] U. Praphamontripong and J. Offutt, "Applying Mutation Testing to Web Applications," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010, pp. 132–141.

[27] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, and D. Poshyvanyk, "MDroid+: A Mutation Testing Framework for Android," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 33–36. [Online]. Available: https://doi.org/10.1145/3183440.3183492

[28] N. Humbatova, G. Jahangirova, and P. Tonella, "DeepCrime: Mutation Testing of Deep Learning Systems Based on Real Faults," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 67–78. [Online]. Available: https://doi.org/10.1145/3460319.3464825

[29] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, and W. E. Wong, "Model-based mutation testing—Approach and case studies," *Science of Computer Programming*, vol. 120, pp. 25–48, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642316000137

[30] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating Test Cases for Specification Mining," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 85–96. [Online]. Available: https://doi.org/10.1145/1831708.1831719

[31] Z. Cao, Y. Tian, T.-D. B. Le, and D. Lo, "Rule-based specification mining leveraging learning to rank," *Automated Software Engineering*, vol. 25, pp. 501 – 530, 2018.

[32] M. R. Aliabadi, M. V. Asl, and R. Ghavamizadeh, "ARTINALI++: Multi-dimensional Specification Mining for Complex Cyber-Physical System Security," *Journal of Systems and Software*, vol. 180, p. 111016, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001138