Journal of Computer Languages 78 (2024) 101243

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

Supporting model-based safety analysis for safety-critical IoT systems

Felicien Ihirwe [a,b,*], Davide Di Ruscio [a], Katia Di Blasio [c], Simone Gianfranceschi [b], Alfonso Pierantonio [a]

[a] Department of Information Engineering Computer Science and Mathematics, University of L'Aquila, L'Aquila, Italy
[b] Innovation Technology Services Lab, Intecs Solutions S.p.A, Pisa, Italy
[c] Automotive Division, Intecs Solutions S.p.A, Pisa, Italy

ARTICLE INFO

Keywords:
Model-driven engineering
Model-based safety analysis
Failure-Logic Analysis
Fault-Tree Analysis
Internet of Things

ABSTRACT

Dependability is regarded as the ability of the system to provide services that can be trusted within a specific period. As the complexity and heterogeneity of Internet of Things (IoT) systems rise, so does the possibility of errors and failure. Early safety analysis not only reduces the cost of late failure but also makes it easier to trace and determine the source of the failure beforehand in case something goes wrong. In this paper, we present an early safety analysis approach based on Failure-Logic Analysis (FLA) and Fault-Tree Analysis (FTA) for safety-critical IoT systems. The safety analysis infrastructure, supported by the CHESSIoT tool, takes into account the system-level physical architecture model annotated with the component's failure logic properties to perform different kinds of automated failure analyses. In addition to its ability to generate the system Fault-Trees (FTs), the new FTA analysis approach automatically performs qualitative and quantitative analyses which include the elimination of redundant events, unnecessary failure paths, as well as automatic probabilistic calculation of the undesired events. To assess the effectiveness of the approach, a comparative study between our propose approach with 19 existing approaches in both academia and industry was conducted showcasing its contribution to the state of the art. Finally, a Patient Monitoring System (PMS) use case has been developed to demonstrate the capabilities of the supporting CHESSIoT tool, and the results are thoroughly presented.

1. Introduction

In our daily lives, we encounter several intelligent Internet of Things (IoT) systems in various domains, such as smart cities, health care, home automation, and industrial production. IoT systems integrate different intelligent features into our daily human activities through the automation of services. Aside from the inherent difficulties in developing multi-device IoT applications for diverse platforms, software developers often make false assumptions. One of these assumptions is that devices will never fail [1]. Indeed, IoT systems might fail because of a wide range of reasons: devices' age, data sources, communication protocols, deployment environment, as well as human errors. In IoT ecosystems, different types of errors can occur. They can be local (e.g., failing sensors) or involve multiple devices at the same time (e.g., network failures or missing communication patterns), causing the entire system to fail [2].

A significant challenge to be recognized in IoT ecosystems is providing a reliable infrastructure for the billions of expected devices and delivering their intended services without failing in unexpected and catastrophic ways [3]. A system is considered fail-safe if it has none or harmless failures. In contrast, a safety-critical system can have catastrophic failures that sometimes result in human life loss. For instance, in the healthcare domain, the monitoring of hospitalized patients must be done with extreme caution. For instance, in a safety critical system such as a *"Patient Monitoring System"*, a simple failure, such as a false sensor data reading, can have catastrophic consequences, including the patient's death. Because these systems are at the intersection of information technology and biomedical sciences, it is necessary to understand how the connected components work and the ability to make perfect decisions either manually or through automated software. Furthermore, these systems are among the riskiest in engineering because they interact directly with sick patients.

Engineering such systems is challenging and complex primarily due to the ever-increasing heterogeneity in every aspect that needs to be combined to fully produce a well-sensed and functional system [4]. Through a high-level abstraction, Model-Driven Engineering (MDE) can provide a unique means for representing many aspects of heterogeneous systems all in one place thanks to modeling languages,

* Corresponding author at: Department of Information Engineering Computer Science and Mathematics, University of L'Aquila, L'Aquila, Italy.
E-mail addresses: jeanfelicien.ihirwe@graduate.univaq.it (F. Ihirwe), davide.diruscio@univaq.it (D. Di Ruscio), katia.diblasio@intecs.it (K. Di Blasio), simone.gianfranceschi@intecs.it (S. Gianfranceschi), alfonso.pierantonio@univaq.it (A. Pierantonio).

https://doi.org/10.1016/j.cola.2023.101243
Received 31 May 2022; Received in revised form 19 September 2023; Accepted 10 November 2023
Available online 22 November 2023
2590-1184/

specifically domain-specific ones (DSMLs). Tackling such heterogeneity, it is essential to look at every system sub-component as a black box, where both the physical characteristics and the software that manages them are highly linked [5]. These sub-systems can be designed, developed, tested, and analyzed independently, and later, they can be integrated to form a fully functioning system. However, it is unavoidable that such systems might fail. Therefore, assessing their safety and trustworthiness beforehand is crucial to guarantee their reliability in case something goes wrong.

Dependability is regarded as the ability of the system to provide services that can be trusted within a specific period [6]. It is mainly characterized by five essential attributes: *availability*, *reliability*, *maintainability*, *integrity*, and *safety*. In this paper, we focus on *safety*, which is defined as the absence of catastrophic effects for the user(s) and the environment [7]. Safety is one of the major features that must be investigated and considered during the development phase of such systems to prevent further disasters. Again, from the healthcare system's point of view, when something goes wrong, it is critically important to react quickly and with high precision to isolate the danger before it happens. However, these systems are very complex and involve high-tech interconnected devices in which the process of discovering faults can be very long and tedious.

In the past, safety engineers relied on different informal design artifacts and documents, such as requirements documents, to measure the safety compliance of the system with less or no involvement of system engineers. Later, several approaches, such as [8–12] (to mention a few), have emerged in the field. These approaches add a degree of automation during the analysis process, bridging the gap between the system and safety engineers. However, these approaches were designed and developed to fit domains such as aerospace, automotive, and cyber–physical systems. In some cases, they might not fully suit the IoT domain. This is mainly due to the large degree of heterogeneity in IoT ecosystems, not to mention its current rapid evolution.

This paper presents a new approach for modeling an early safety analysis for safety-critical IoT systems based on the Fault-Tree Analysis (FTA) approach. The approach runs on top of CHESSIoT, a model-driven environment for engineering industrial IoT systems. The CHESSIoT environment is built on top of the CHESS tool [13], an open-source model-driven tool that offers cross-domain modeling, development, and analysis of high-integrity systems. CHESS supports different kinds of analysis, including but not limited to real-time schedulable analysis [13], Contract-Based Analysis [14], and Quantitative Reliability Analysis based on Mobius [15]. In addition to this, CHESS also offers means to perform early safety analysis based on Failure Logic Analysis (CHESS-FLA) [16]; however, that existing infrastructure is not suitable enough to support the IoT domain as well as not mature enough to support the Fault-Tree Analysis as one of the common and necessary artifacts used in the process of safety analysis [17,18].

The presented approach relies on and extends the existing CHESS-FLA infrastructure [16]. Normally, CHESS-FLA offers means to: *(i)* model the system's failure behavior through the decoration of the system's simple components following the Failure Propagation Transformation Calculus (FPTC) annotation [19], *(ii)* run the Failure Logic Analysis (FLA), *(iii)* and propagate the analysis results back onto the original model [20]. Throughout the CHESS-FLA analysis process, the entire system's behavior is automatically determined solely from the composition of its elements. Thus the potential of automatic model-based safety analysis is significant. It is achieved by calculating the failure behavior from the composite parts up to the system level. This can help predict the impact of a component change or architectural change on a system very cheaply [19]. Furthermore, suppose an essential failure behavior occurs at the model system level. In that case, it will be easy to discover the source of the fault immediately and identify where the fault tolerance measures should be directed in the architecture to mitigate them.

The new proposed extension extends the CHESS-FLA by supporting the definition of the failure behavior of a simple component with no input ports. In addition, the extension supports the generation of the system's complete Fault-Trees and performs qualitative and quantitative FTA Analysis. In the proposed approach, the qualitative analyses help eliminate unnecessary paths and redundancies in the FTs' events. On the other hand, quantitative analysis allows the user to set the basic event probabilities and calculate the failure probabilities of an entire system from its constituent parts' failure event probabilities. This calculation is automatically performed following the well-known logic probabilities calculation mechanism techniques [21–23]. Aside from the FTA, the existing CHESS infrastructure also supports the generation of the Failure Mode Effect Analysis (FMEA) table [24], one of the common and necessary artifacts used in the safety analysis process.

Throughout the paper, we present the evaluation mechanism and the experimental evaluation outcomes to better assess how the proposed approach performs compared to the 19 existing approaches drafted from the academic literature and the industry. In doing so, six main features have been considered: *(i)* IoT-specific support, *(ii)* support for system design modeling, *(iii)* support the failure behavior modeling, *(iv)* automated FT generation, *(v)* support for automated qualitative FT analysis, *(vi)* and support for automated quantitative FT analysis. In addition, we used a Patient Monitoring System (PMS) case study to demonstrate the effectiveness as well as the capability of the supporting tool. As a result, we summarize this paper's key contribution as follows:

- We present CHESSIoT's system-level modeling environment for designing IoT systems, considering the system's physical aspects.
- We presented the CHESS-FLA extension to support the safety analysis suitable for the IoT domain.
- We introduce an automated Fault Tree generation approach that can handle large and complex models while still supporting critical features like event tracking and component sub-tree generation.
- We present both qualitative and quantitative FTA analysis approaches on the generated Fault-Tress.
- We present the experimental results from a relative evaluation mechanism conducted in comparison with 19 existing approaches in the literature.
- We present a Patient Monitoring System (PMS) case study to demonstrate the effectiveness of the proposed approach as well as the capability of the supporting tool

The rest of the paper is arranged into seven sections as follows: Section 2 provides a general background of the paper; Section 3 presents the proposed CHESSIoT system-level modeling approach; Section 4 presents the proposed safety analysis approach; Section 5 presents the evaluation mechanism highlighting comparative study results as well as the developed PMS running example. In Section 6 we discuss the related work, whereas Section 7 concludes the paper and draws future perspective work.

2. Background

2.1. Safety critical systems

The term "safety-critical system" was created in response to growing concern and awareness about the use of computers in situations where human lives could be jeopardized if an error occurs [25]. Safety-critical systems are those whose failure could result in loss of life, significant property damage, or damage to the environment [26]. A safety-critical system should be ideally designed to lose no more than one life per billion hours of operation [27]. We can see many examples of such systems in the following domains: aviation, railways, medicine, nuclear engineering, and military. Engineering such systems is difficult and must be done with extreme caution, as even the smallest error in the

process can have disastrous consequences. More and more modern safety-critical systems are incorporating new technologies, such as machine learning techniques, to reduce the possibility of failure through intelligent responses provided by artificially trained robots [28]. According to [29], a system's *fault* occurs when the functional behavior of a component is disrupted due to either internal or external reasons. This may eventually result in an error, which is when the system deviates from correct behavior to erroneous behavior. When such a deviation results in a total interruption of the system or the interruption of a component's functionality, it is referred to as a failure.

A significant challenge recognized in the IoT ecosystem is how to provide a reliable infrastructure for the billions of expected devices and how to deliver their intended services without failing in unexpected and catastrophic ways [3]. In the IoT context, safety is often considered to be the ability to detect and prevent any unintended failure behavior in IoT systems [30]. In the past, IoT systems were considered fail-safe because of their size, as their failures mostly had no or harmless consequences. However, due to the system's rise in size and complexity and the increased demand for IoT systems in the industry, errors and failures for such systems are unavoidable. For instance, IoT systems, such as Intelligent traffic lights, smart homes, smart manufacturing systems, as well as patient monitoring systems, can suffer from potential failures generated internally in the system due to several issues such as age or poorly connected or failures caused by external influences such as weather or human error.

As research in this area continues, their developers deem existing proposed concepts and architectures safe. Still, they are frequently found to be impractical for real-life applications because safety-critical systems involve unpredictable behavior of lives, properties, or the environment [31]. In addition, as the technologies evolve in some domains, such as IoT, new failure modes, such as denial-of-service attacks against networked information systems, are emerging. Failures occur through physical effects and service disruption or data loss. The lack of a systematic, disciplined, and quantifiable software engineering methodology, as well as a comprehensive abstraction mechanism for dealing with the increasing complexity of safety-critical systems, results in a wide variety of similar, but not congruent, isolated solutions that cannot be easily reused and combined [30].

The number of computer systems that we consider safety-critical is expected to grow significantly in the future. In addition, the declining cost of hardware, improvements in hardware quality, and other technological advancements ensure that new applications will be sought in a wide range of domains [26]. However, for the analysis of the safety-critical systems, there is no universally accepted rigorous dependability analysis process, which helps in choosing the metrics, techniques, and methodologies for the dependability evaluation of such critical systems [6]. In any case, analysis of software development approaches, as well as safety-critical software, is required to determine the most appropriate techniques for use in the production of future software for high-integrity systems [25].

2.2. Model-driven engineering

Model-driven engineering (MDE) seeks to support the automation of the software development process by employing models as the primary artifact in the development of complex systems [32]. MDE has been applied to almost every research domain, and its success has demonstrated a significant push toward better and faster software development [33]. In addition, MDE can generate fully functional code through a series of model transformations [34], potentially reducing development time and costs.

MDE can provide a unique means for representing many aspects of heterogeneous systems all in one place thanks to modeling languages, specifically domain-specific ones (DSMLs). Models defined by these languages are intended to be far more human-oriented than common code artifacts, which are inherently machine-oriented [35]. Engineering platforms such as MDE4IoT [35], ThingML [36], IoTML [37] and Montithings [2] (to name a few), have demonstrated the potentiality of MDE to be a realistic alternative for developing scalable IoT systems. However, developing IoT code generators that are perfectly capable of handling large models and accommodating a huge set of requirements provided by the client is still an open issue. This is due to the high degree of heterogeneity in their hardware devices, data sources, protocols, deployment levels, and so on [38].

In terms of analysis, the IoT domain still presents a significant gap in the validation, verification, and analysis of such systems under development [32,39]. The main challenge is the scope of the analysis; this is mainly due to the fact that the number of IoT devices and application's complexity is already huge and is only likely to grow in the future. Furthermore, the physical replication of such systems is challenging and much more complex due to their scale [40]. This potentially contributes to the long-standing lack of standardized, realistic reference models that can perfectly capture the interactions between sensors, apps, and actuators.

2.3. Model-based safety analysis

Failures that could risk human life and environmental or property injuries are considered safety hazards. Safety analysis should run concurrently with system design, including interactions between the two, and it should be kept up-to-date throughout the system life cycle. Risks of this sort are usually managed with the methods and tools of safety engineering. The safety analysis is conducted initially by safety engineers, which is one of the dependability analysis techniques that aim to study system response in case of an unwanted failure behavior that can hinder system safety compliance. In safety-critical systems, it is often required to maintain a high level of safety to prevent potentially catastrophic consequences [16].

Failure logic approaches map the reliability concepts (produced by reliability engineers) to reflect the underlying fault-to-failure and failure-to-fault propagations within the systems [41]. In practice, a component can act as a source of failure (for example, by causing a failure in output due to the activation of internal faults) or as a sink (a component can avoid failure propagation by detecting and correcting the failure in input). Furthermore, failures in a component can be propagated (i.e., a failure can be passed from input to output) or transformed (by changing the nature of the failure from one type to another from input to output) [42]. The Failure Logic Analysis (FLA) [16] allows the possibility of defining such failure behavior of the system following the Failure Propagation Transformation Calculus (FPTC) notations [19]. In the end, a combination of different rules can also be expressed in terms of logic combination and later analyzed to estimate the failure behavior of the entire system

The Fault-Tree Analysis (FTA) [43] technique is currently one of the most widely used methodologies when performing safety analysis. The purpose of an FTA is to graphically represent and trace down influences from a system-level hazard to individual failures of distinct system components and sub-components. The graphical representation of the scenarios can aid in explaining these causal chains that can lead to a hazard, followed by an analysis to determine the combination of events that trigger such hazards or compute the chance that such a hazard could occur. During the analysis, the safety engineer starts with the actual hazard, referred to as a "top event", and traces down different event combinations that might contribute to such a hazard until the actual cause is reached. This is referred to as a "basic event" in this case. Fig. 1, shows an illustrative FT example.

Starting from the FLA results, FT combines logic representation and relies on logic gates to determine the output of a component's failure behavior. This representation is performed in an automated fashion. For instance, when two or more failure events are needed to represent a certain component failure behavior, an "AND" gate can be used; while

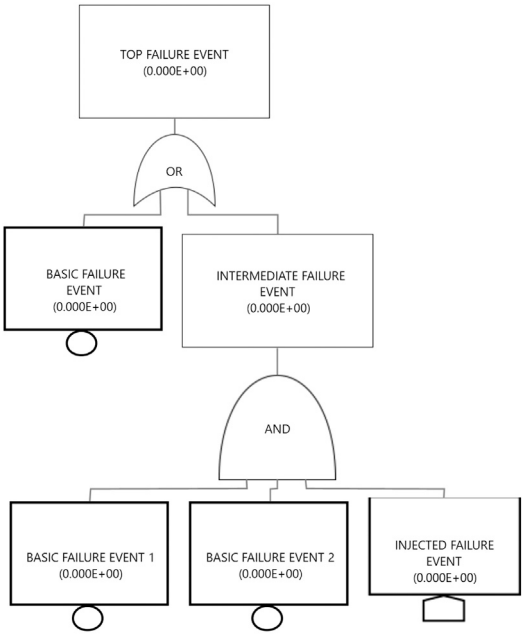


Fig. 1. An FT example.

if one event is enough to determine the component's failure, an "OR" gate is used. Other known logic gates can also be used based on the desired system failure behavior. Depending on the scope of the system, the root failure can be caused by internal components or by an external impact, and such failure events are referred to as injected in the system.

Since the system's simple FTs might be difficult to understand and challenging to derive information from, especially when the system is huge, both quantitative and qualitative analyses are frequently performed to extract useful information from the resulting trees. The primary goal of qualitative FTA is to discover the logical relationships between events that contribute to a system failure [44]. It is focused on determining the minimum number of failure pathways required to cause a system to fail. Quantitative FTA, on the other hand, seeks to compute the probability of a top-level event (system failure) based on the probabilities of individual component failures and their logical relationships. This approach involves assigning numerical probabilities or failure rates to each basic event (component failure) using component data, expert judgment, or experimental results [44].

Failure Mode and Effects Analysis (FMEA) [24] is among the earliest known failure analysis techniques, and it is frequently used as the first stage in a system reliability analysis. Reliability engineers originally developed it to investigate problems that could come from military system failures. It is used to examine as many components, assemblies, and subsystems as feasible to determine failure modes and their causes and effects. The failure modes of each component, as well as their consequences for the rest of the system, are recorded in a separate FMEA worksheet [45]. Unlike the FTA, which follows a top-down deductive approach from the top event to specify its possible causes, the FMEA follows an inductive reasoning approach. Using a forward logic approach, FMEA separates a system into small components, analyzes failures that each component may cause, and assesses the effects of those failures on the system. As a result, the FMEA performer must properly understand the system safety context and software requirements or design specifics to ensure the comprehensiveness of the system decomposition and the validity of each component's usability.

FTA, as well as FMEA, are already mandatory analysis approaches for performing safety analysis in domains like automotive and aerospace [17,18], and more domains are subject to follow that suit [11]. From a technical point of view, FTA and FMEA analysis seek to support the improvement in safety analysis, and they both rely on the same system definitions and failure criteria defined by the user. In our case, CHESS-FLA analysis results are used in both the FTA and FMEA analysis process (we will discuss this in the following sections)

3. CHESSIoT system-level modeling

Due to the heterogeneity present in the IoT ecosystem, deciding the levels of abstraction can be tricky. Therefore, the CHESS tool provides a multi-view modeling environment by providing six main views: system view, component view, deployment view, analysis view, requirement view, and platform-specific view. Each supported view has its own underlined constraints that enforce its specific privileges on model entities and properties that can be manipulated. Furthermore, depending on the current stage of the design process, CHESS sub-views are adopted to enhance specific design properties or steps of the current process. More specifically, the system view aims to provide a system-level modeling environment focusing on the physical aspects of the system to support early-stage analysis.

As the CHESS tool is a cross-domain modeling and development environment, depending on the domain, concepts can vary accordingly. To tackle that, it allows extending the current view with additional domain-specific sub-views, which can be activated based on the domain or the design development stage. To support IoT system-level modeling, CHESSIoT introduces *"IoTSub-view"*. Once applied at any design stage, the user will benefit from a dedicated IoT-specific modeling infrastructure.

In addition to that, we have defined a high-level UML/SysML profile extension to reflect the construct and semantics present in IoT system-level architectures. This was achieved by defining IoT-specific stereotypes and nomenclatures that better fit an IoT perspective. Fig. 2 presents the abstract syntax of the CHESSIoT system-level profile. Such a level considers only the main physical architectures involved in achieving a full multi-level IoT system without covering any aspects related to system functional behavior.

3.1. CHESSIoT system-level DSL

By referring to Fig. 2, the *System* element defines a conceptual representation of an IoT system under development as a whole, and all the other elements are defined under it. Other sub-systems can also be contained, making it possible to design IoT system-of-systems architectures. To better support all the layers present in the IoT ecosystem, the CHESSIoT System profile spans all the layers, typically from the edge to the fog and the cloud layer.

At the edge layer, a *PhysicalElement* can be of any type, ranging from a tiny microcontroller to an element as big as a car, a plane, or a house. Any physical component that can play a part at the edge layer is represented as a physical element. The system can have one or more physical elements; each can have one or more communicating ports. The following four main element types extend the physical element. A *PhysicalEntity* on the other hand, can be almost any object or environment on which a running physical element can act. A self-driving car software, for instance, runs on various boards attached to the car but not on the car itself. So a car is a physical entity in this case. Furthermore, a *PhysicalBoard*, as expected, represents a physical controller where the software runs. For example, a Raspberry Pi board processes data collected by different sensors deployed in different building parts. Finally, the *SensorBlock* represents a sensor component at the thing layer that is in charge of collecting environmental data. In contrast, the *ActuatingBlock* in this case represents any actuating component.

At the fog layer, we only have the *Gateway*, which is just any kind of device that serves as a link between the physical world of things and the virtual world, in this case, a cloud infrastructure. This element communicates with both the remote server and the physical board. The

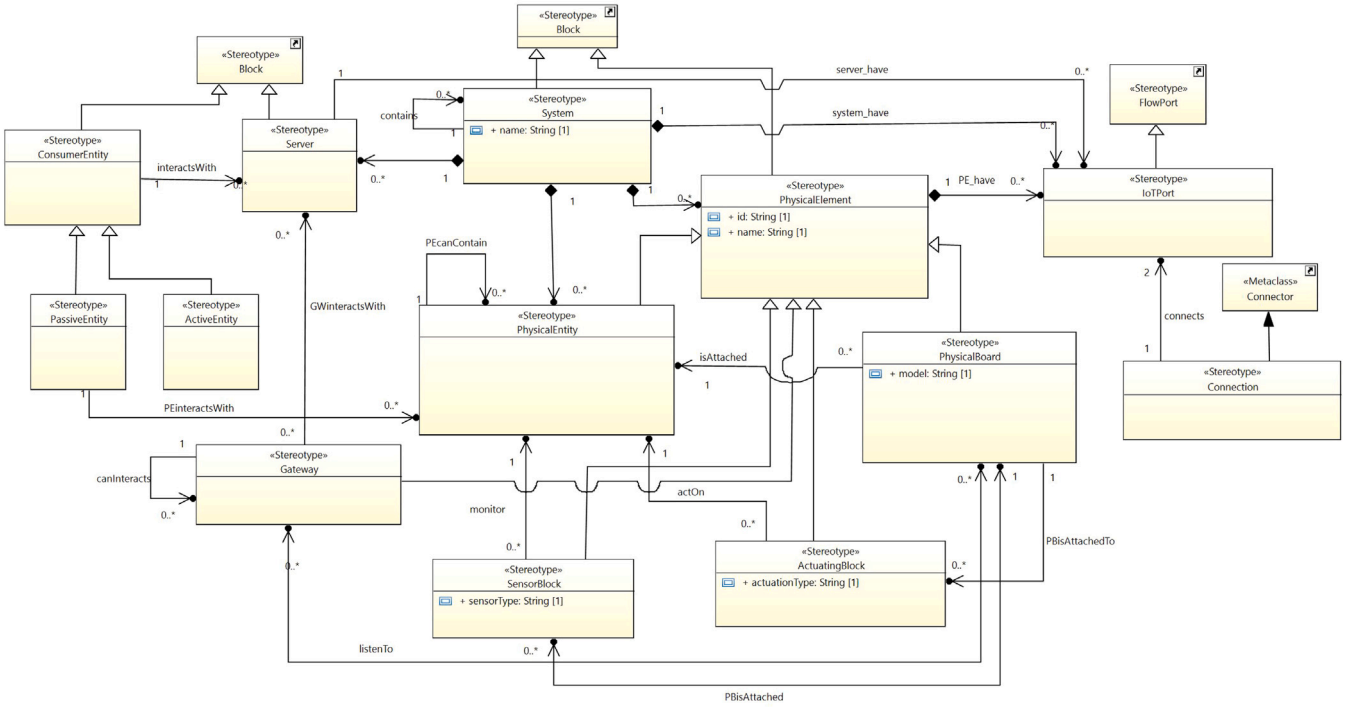


Fig. 2. IoT System-level meta-model.

CHESSIoT system-level model does not include any information related to any functional behavior of such a component but the main physical construct of the system. The cloud layer, on the other hand, is made up of a *Server* and a *ConsumerEntity*. A server is an element of the cloud infrastructure that stores data and cloud services. A consumer entity is any third-party element that can communicate with the server to consume its data. Depending on its role in the system, this component can be *active* or *passive*. A computer, for instance, that runs software to visualize and control remotely deployed sensors qualifies as an active consumer entity. In contrast, a traffic light actuator that receives commands from the server to function will be qualified as a passive consumer entity.

4. Safety analysis approach

This section presents the technical aspects of the proposed automated model-based safety analysis approach and is divided as follows: In Section 4.1, we present a general overview of the proposed safety analysis process supported by the proposed tool; in Section 4.2, we describe the extended FPTC syntax; and in Section 4.3 describe the CHESS-FLA transformation process. In Section 4.4, we introduce the new FT generation process, whereas Section 4.5 introduces the novel FTA analysis, covering the technical aspects behind both the qualitative and quantitative analysis mechanisms.

4.1. Safety analysis process overview

IoT systems can experience failures due to various factors, including device age, data source problems, network issues, deployment environment, and external constraints. For instance, human error can also cause problems. The CHESSIoT safety analysis approach proposes an early safety analysis method using Fault-Tree Analysis, which involves annotating a system model with failure behavior rules using the Failure Propagation Transformation Calculus (FPTC) notation [12].

As illustrated in Fig. 3, the safety analysis process typically commences with the *IoT system engineer* creating a model based on the gathered *system functional requirements* (1) in Fig. 3. These requirements are mainly acquired through close collaboration with the *client*. The system-level model encompasses the system's major functional components, sub-components, and interconnections. These system components can be represented as blocks in SysML Block Definition Diagrams (BDD), which align with the abstract syntax meta-model illustrated in Fig. 2. Internal Block Diagrams (IBD) illustrate the interdependencies between these components, facilitating the identification of error propagation paths. Each part or block can be assigned to a specific architectural subsystem or component. The physical architecture should be extensible to add new components or blocks as necessary. The entire safety analysis process is fully detailed in the following sections.

Once the system model is complete (see the *CHESSIoT model* (2) in Fig. 3), it can be handed to the *safety engineer* for further safety analysis. Similarly to the system engineer, the safety expert can derive *safety requirements* (3) from the client's needs, domain standards, and expertise to ensure optimal safety. Starting from identifying the typical system-level failures, the safety engineer identifies the failure behavior for each component following the Failure Propagation Transformation Calculus (FPTC) notation. The FPTC technique enables the analysis of component-based systems with cyclic data, control-flow structures, and closed feedback loops. Such failure behavior, referred to as *FLA rules* (4) are annotated to the system's simple comments to illustrate how failures might occur in a system component and how they are propagated from one component to another. At this stage, the safety engineer can additionally set *the component's failure rates* (5) to be used for quantitative analysis.

4.2. FPTC calculus

In contrast to other prevailing approaches that heavily depend on monitoring the state events of individual system components, the Fault Propagation and Transformation Calculus (FPTC) technique excels in scenarios involving intricate interplays of potential cyclic paths, intricate control flow dynamics, and intricate closed feedback loops, where traditional methodologies may fall short. [46]. The extended annotations explain how failures might occur in a system component and how they are propagated from one component to another. Based

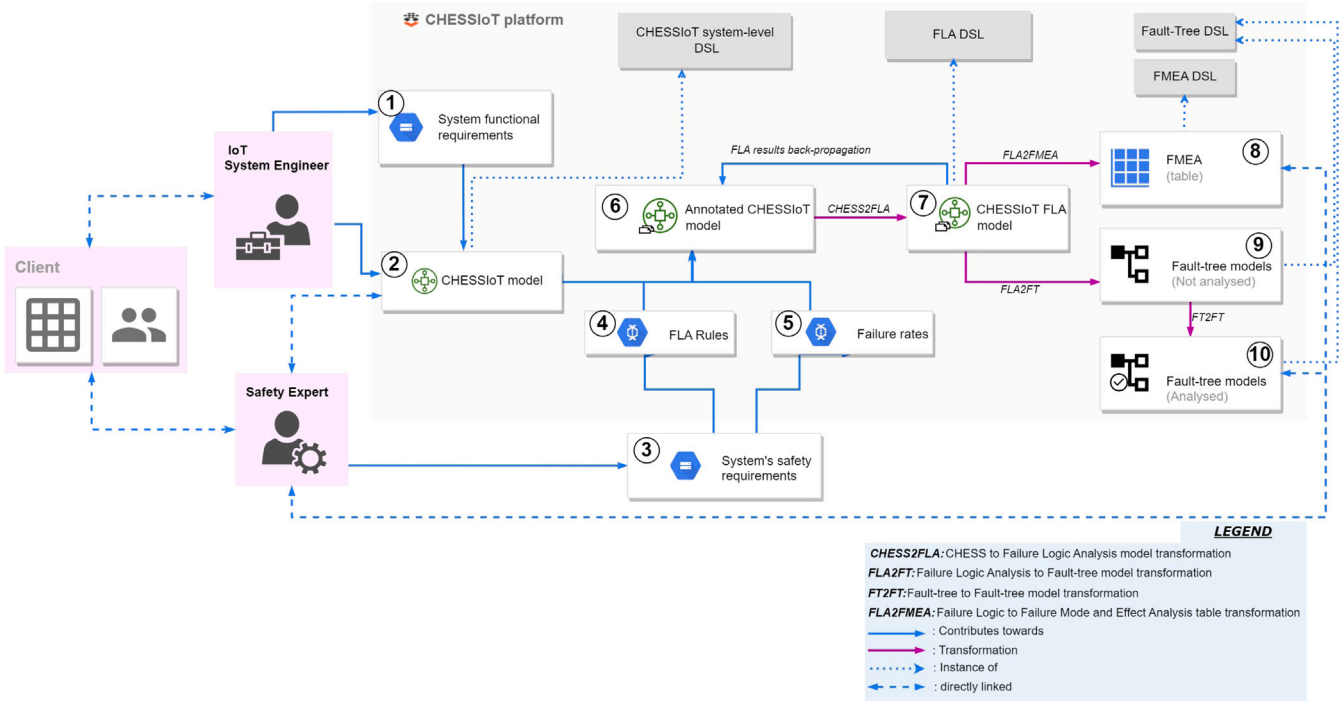


Fig. 3. Safety analysis process.

Table 1
Failure types.

Failure type	Description
Early	output is provided too early
Late	output is provided too late
ValueCoarse	output out of range in a detectable way
ValueSubtle	output not in range in an undetectable way
Omission	no output is provided
Commission	an output is provided when not expected

on its nature, a function/component can propagate a failure (carrying a failure from its input onto its output without changing its nature), transform a failure (change the nature of a failure from input onto the output), act as a source of failure (creating a failure despite no failure in input), or act as a sink (avoiding the failure being either propagated or transformed).

When defining the failure behavior of a component, the following failure abstraction categories are supported: *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, biased, exhibiting a linear or non-linear drift, or erratic behavior. In addition to that, a *noFailure* annotation is used to indicate a no-failure mode at the input port. Table 1 describes different failure modes and their descriptions.

The first FLA integration in CHESS was done in [16] with the support for FI^{4}FA (Formalism for Incompletion, Inconsistency, Interference, and Impermanence Failures Analysis) [47], which is an extension of FPTC that takes into consideration Incompletion, Inconsistency, Interference, and Impermanence Failures, and their corresponding countermeasures. However, in the previous version, it was impossible to express a component's failure behavior with no input ports. To model the failure expression on a certain port, for instance, the port's name and the failure type are always necessary; in the case of no failure at a given component port, a "noFailure" mode is annotated to the port to identify the internal failure. Nevertheless, a component may not need to have an input port to fail. For instance, a temperature sensor's role is to sense the surrounding environment and relay the information to connected parties. In such a case, it only needs an output port. Such internal failure behavior can now be expressed using a "(*)" notation. The new extended syntax, with respect to the one introduced in [16], is shown in Listing 1.

```
1 FLA: "LHS"=>"RHS"; #left/right side of an
      expression
2 LHS=portName."bL"|portName."bL"(,portName."bL")
      +|"(*)"
3 RHS=portName."bR"|portName."bR"(,portName."bR")+
4 bL=wildcard|"bR"
5 bR=noFailure|"FAILURE"
6 FAILURE=early|late|commission|omission|
      valueSubtle
7                                    |valueCoarse
```

Listing 1: Extended FLA syntax

4.3. CHESS2FLA transformation

The *Annotated CHESSIoT model* (6), produced by the system expert as previously explained, gets automatically transformed by means of the *CHESS2FLA* model-to-model transformation to generate the *CHESSIoT FLA model* (7). During the transformation, each component is essentially a black box in the CHESS environment that can only exchange data through its ports. The FLA technique automates the calculation of a complete system's failure behavior starting from the failure behavior rules of its separate composite components and interconnections. This, in turn, means that the failure behavior of composite elements is also determined by the failure behaviors of their instantiated simple components and their internal decomposition. Simple components have no other parts and rely on the failure behavior defined in the previous stages. When the failure modeling is finished, the model undergoes a CHESS2FLA model-to-model transformation, which transforms it into an FLA model following the meta-model presented in Fig. 4. This transformation is domain agnostic, and it does not consider any domain-specific construct other than linking each component and its final state of failure into a single model instance.

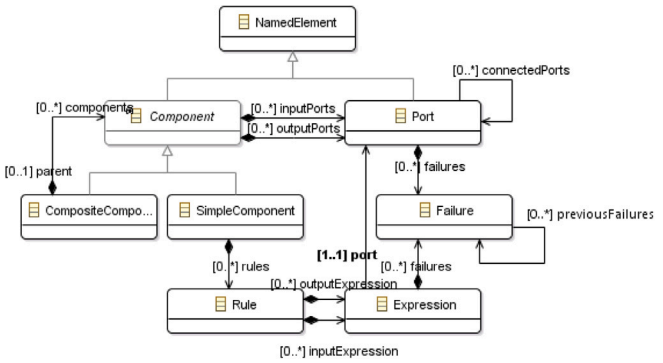


Fig. 4. CHESS-FLA meta-model [42].

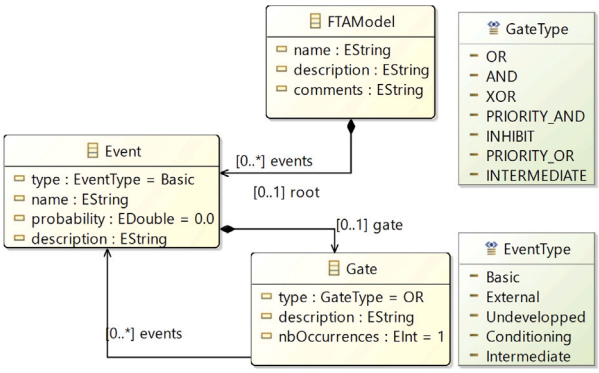


Fig. 5. FT meta-model [8].

As seen from the FLA meta-model, a composite component represents a subsystem that contains one or more sub-components. As mentioned in the previous stages, this component does not possess the failure behavior by itself, but it relies on its sub-components to determine its failure behavior. On the other hand, a simple component represents a functional component that can contribute to system failure. Each component being simple or composite has one or more input and output ports which are referred to when deriving failure rules. A rule is composed of a set of input expressions and output expressions and a prefix always starts it "*FLA:*". An expression being either from the input or the output side is made by a combination of a port and a failure type associated with it. An illustrative rule is shown in Eq. (1)

$$FLA : input\_expression\_1, \ldots., input\_expression\_N \rightarrow output\_expression\_1, \ldots .., output\_expression\_M; \quad (1)$$

In composite components, the propagated failures from a simple component output ports are automatically transferred to its connected output ports. Furthermore, each simple component is assigned rules where each rule contains input and output expressions reflecting failures and their respective ports. During the transformation, the extended notation of the internal failure of a component with no input ports creates a unique virtual port assigned with a "noFailure" failure type at the component input port to reflect the idea of the component's internal failure source. Although it might appear to be a minor improvement with respect to the existing FPTC infrastructure, it eliminates a significant amount of confusion during the system modeling process because otherwise, input ports with no reasonable connections will be left hanging which may mislead the user.

At each level of the FLA analysis, the results are back-propagated onto the original model to assign each component's failure state to be reflected in the model. The final failure state at simple and composite components as well as at the system level is reflected when the analysis is done. The system *FT* (9) and FMEA (8) table can be automatically generated and analyzed before being sent back to the safety expert for consultation. If something is wrong, changes can be made before the final inspection. In the following sections, we briefly review each step of the supported analysis process. Although the FMEA analysis is part of the safety analysis process as it relies on CHESS-FLA analysis results, this paper focuses primarily on the FT-based analysis approach.

4.4. Fault-tree generation

The FT generation process is performed following a FLA2FT model for the transformation of a CHESS-FLA model into a conforming FT model. Fig. 5 shows an FT meta-model adopted from [8]. The FTModel element is the top element of the tree, and it is instantiated for each failure that propagates to the system output ports during the FLA analysis. Each FTModel element contains a logical network of events and gates that together form an FT. The entire FT generation process will be covered in the following sections.

4.4.1. Fault-tree events

In our proposed approach, each event can be graphically identified in the FT from its unique identifier (ID). An event ID is defined as a pair of "failure_name" and "port_name" in a given component. This ID never changes through the FT generation as well as in the FT analysis process. This can potentially help in event tracking when comparing generated and analyzed FTs. In addition to that, each event has its own name which by default combines the information regarding its corresponding failure and its effect in a given component. The effects can be of type *top failure* type at the system level, *local failure* caused by the system's intermediate failures across the tree, *injected failure* resulted from external faults, and *internal failure* resulted from the component's internal faults. In the next listing, we go over various event types that we use to construct the FT and we describe how those events are generated throughout the transformation process.

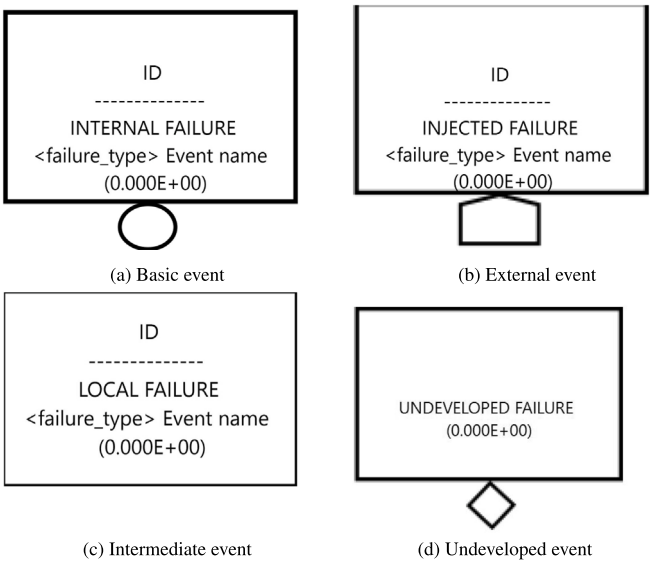
- **Basic events:** A simple component may suffer different kinds of malfunctions, generating either one or more kinds of internal failures. One or more notations may be required to define such events for a given component. A basic event is used to represent a failure that is initiated inside a simple component. This can be basically referred to as a simple component acting as a source of failure. In this case, a failure condition is present on any of the output ports despite no failure at its input port. In case a simple component does not possess any input port, the newly developed approach allows the definition of such condition following Expression (2). On the other hand, in case a simple component possesses one or more input ports, its failure behavior can be defined by explicitly initializing all the input failures of the component with a *"noFailure"* condition (Expression (3)). Fig. 6(a) shows the basic event representation in an FT resulting from an internal failure of a component.

$$FLA : (*) \rightarrow p.failure_{(out)}; \quad (2)$$

$$FLA : p_1.noFailure, \ldots., p_n.noFailure \rightarrow p.failure_{(out)}; \quad (3)$$

NOTE: *Considering p_1, p_2 to p_n to be the input ports of a simple component while p is an output port. If any of their corresponding failure condition is different to "noFailure", then the above condition is not met, so, all "noFailure" conditions on other ports are ignored as they do not contribute toward the logical failure behavior of the component.*

- **External events:** External events are used to represent failures that can be introduced from the environment outside the system boundaries. CHESS provides the possibility to inject failures in the system through the system-level input ports. These faults are modeled with a comment annotation with *<<FPTCSpecification>>* stereotype attached to the relevant input port where the



(a) Basic event (b) External event

(c) Intermediate event (d) Undeveloped event

Fig. 6. Event types.

fault is being injected in SysML block diagrams (the case study in Section 5.5.2 will demonstrate this concept in more details). The injected fault specifies the type of failure attribute being injected into the system and the port to which it is introduced on. This fault injection can also be done on a composite sub-system under analysis. Fig. 6(b) shows a graphical representation of an external event resulting from an injected fault in the system.

- **Intermediate events:** An intermediate event is used to describe the local failure effects resulting from a logical output of one or many events. In the presented approach, these events are generated to represent the failure condition at the input or output port(s) of the simple component resulting from an internal failure or other failure condition from the outside of that simple component. It is also used to represent the top event of an FT. Fig. 6(c) is used to represent an intermediate event.
- **Underdeveloped events:** The underdeveloped event is used to specify an event resulting from a failure in which we do not have sufficient information about it. This can basically happen when a failure is introduced on the input port of a simple component without a preceding definition of how it was been propagated or injected at that input port. During the FT generation process the symbol in Fig. 6(d) is used.

4.4.2. Failure propagation

Failure propagation occurs in a component when a single input port failure condition of a component is directly transferred on its output ports of the same component without changing its nature. This failure propagation can be modeled in CHESS-FLA using the notation in Expression (4). A propagation also occurs between two connected components, when a failure condition at the output port of the preceding component is transferred on the input port of the following component.

$$FLA : p_{(in)}.failure_1 \rightarrow p_{(out)}.failure_1; \tag{4}$$

4.4.3. Failure transformation

A failure transformation occurs within a component when a failure condition present at the input port of a simple component is converted into another type before reaching the output port (Expression (5)). A failure transformation can also occur when more than one failure expression of any type the exception of a "noFailure" or "wildcard" at multiple input ports is transmitted on a single output port (Definition in Eq. (6)). 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.

$$FLA : p_{(in)}.failure_1 \rightarrow p_{(out)}.failure_{(out)}; \tag{5}$$

$$FLA : p_{(in1)}.failure_1, \ldots, p_{(inN)}.failure_N \rightarrow p_{(out)}.failure_{(out)}; \tag{6}$$

4.4.4. Fault-tree generation process

The system FTs are generated through a series of model-to-model transformation mechanisms written using the Epsilon Transformation Language (ETL) [48]. The process starts by instantiating a number of FT objects equal to the number of failures that propagates to the output port(s) of the system. At this stage, each error that propagates to the output port(s) of the system is represented into its own FT. Note that when a "noFailure" condition is propagated to the output, it is ignored. This technically means that the system acts like a failure sink and it is able to mitigate its propagation to the output of the system, which is also true for all other sub-systems. To achieve that, the Algorithm 1 is followed.

```
for p in allPorts do
    if p is output of the system then
        for f in failures assigned to p do
            if f is not "noFailure" then
                Create an FT relative to the failure f and p;
                Add FT to FTs sequence;
            end
        end
    end
end
```

Algorithm 1: Instantiate fault-tree

In the next steps, each FT is built separately and recursively. The initial action involves the creation of a top event among all. A top event is generated as a result of the failure propagation to the system output port. In terms of logical gates used in the FT, only "AND" and "OR" gates are adopted. An **AND** gate is used to indicate a failure transformation from an input to an output port of a component (see Section 4.4.3). An **OR** gate, on the other hand, is used to depict a failure propagation situation (see Section 4.4.2). 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 whole FT generation population algorithm is described in Algorithm 2.

```
Data: FLA composite component (system-level)
Result: FT model
for port in allPorts do
    if port is an output of the system then
        for f in failures assigned to port do
            if f.name is correspond to an FT then
                Create an intermediate event ← TOP FAILURE;
                Assign an OR gate to it;
                Add it to its corresponding FT;
                for con_port in port.connectedPorts do
                    recurseFT(f,con_port,FT,topEvent);
                    // Call Algorithm 3
                end
            end
        end
    end
end
```

Algorithm 2: FLAComposite2FT rule algorithm

When the top-event creation is done, the intermediate events are created and populated into the FT, based on the failure expressions and their components they are assigned to. The FT population involves a recursive transformation process in which, as indicated from FLA meta-model (Fig. 4), from a component, we can have information on ports and from ports we can get to rules, rules to expressions and back to the

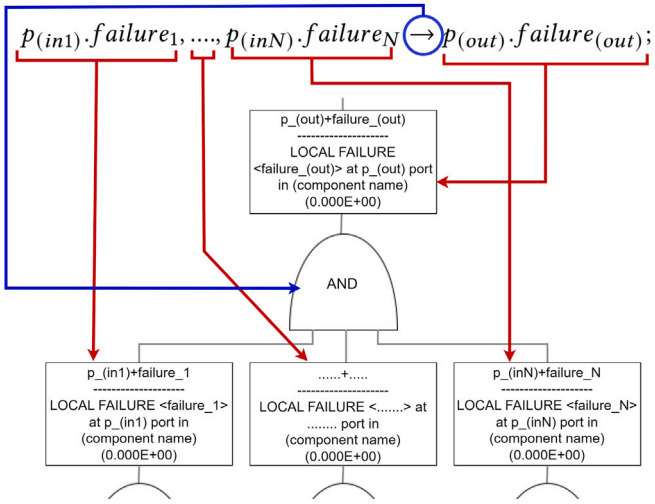


Fig. 7. Expression (6) corresponding tree.

components. So, at this stage the only crucial stopping case is when the transformation hits a condition matching an internal, underdeveloped or an injected failure. For instance, in Fig. 7, a simple transformation example with indications showing a simple transformation mapping of the Expression (6) is shown. From the example, each of the output expression 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 gate is defined based on the nature of the input expressions to satisfy the failure propagation and transformation concepts. A brief description of the followed algorithm is described in Algorithm 3.

In the newly developed environment, it is possible for a safety expert to assign failure rates of any of the events leading to basic failure conditions. These events include the internal failure condition and the injected failures. In addition to the failure rate assignment, the user is also able to add a failure description in a textual format to reflect the proper cause of the failure. This is potentially important when, for instance, a simple component might have two different internal failure conditions leading to two different outputs. For instance, an aging device can still work by providing the wrong output leading to a *valueCoarse* or *valueSubtle* failure at the output port, whereas a blown device fuse will automatically halt its functionality therefore leading to an "omission" failure at the output port. Assigning such information to the FT model will eventually improve its readability. This assignment is done long before the FLA2FT transformation process and when the transformation finishes, a probability file is generated separately from the model in an Excel file format. This file is later loaded into the model to support the quantitative probabilistic analysis process.

4.5. Fault-tree analysis

The proposed approach supports both the qualitative and the quantitative FT analysis through the use of rigorous model transformation techniques. In this section, we will go through the supported analysis.

4.5.1. FT qualitative analysis

The FT qualitative study is conducted using an FT2FT model-to-model transformation (ie: transformation from ⑨ to ⑩ in Fig. 3) in which the FT meta-model in Fig. 5 serves as both the source and the target meta-model. This effectively creates new FT representations in the workspace, permitting users to reuse both the generated and the analyzed FTs at the same time. The goal of this qualitative analysis approach is to provide a new representation of the existing FT that only includes the essential representations. Although the current qualitative analysis does not fully reflect the calculation of the minimal event sets needed for a system to fail (minimal cut-sets [49]), it does provide a much shorter and more readable FT that still reflects the goal for the analysis.

During the qualitative analysis process, the following actions are performed:

- ***1. Removal of internal component failure propagation:*** FT helps users discover and trace down the source event of a system failure in a more easy and intuitive fashion. As described in Section 4.4.2, internal component failure propagation occurs when a single input port failure condition of a component is directly transferred to the output ports of the same component without changing its nature. Although keeping such information in the FT is important, as the model becomes big and complex, this information can be difficult to look over. Therefore, each path meeting such a condition is omitted and removed from the FT. This process drastically reduces the vertical magnitude of a FT but it does not change its nature.
- ***2. Removal of external component-to-component failure propagation:*** This refers to a condition in which a component-to-component propagation is solicited from a single channel in the FT. For instance, if a single failure condition at the input port of a component is propagated from a single source, then this information is omitted in the analyzed FT. Mainly on the basic events, events involved in a failure transformation, and the top event are kept in the FT.
- ***3. Removal of basic event redundancy:*** Typically, a failure is initiated from a single source and propagates over many propagation pathways until it reaches the output port(s). If, in all the propagation channels, no transformation occurred, then, from the output, only one path is considered, and, from there, all intermediate propagation representations are removed accordingly.

During the transformation process, only the paths that satisfy the internal or external failure transformation are kept in the tree. This is to help users only care about the important information when tracing the origin of the failure. One special case of propagation that is kept in the tree is when the FT that has to be analyzed contains a single path in which a single basic event propagates up the tree all the way to the top event. Then, in such cases, only the top event and the basic event are kept in the analyzed FT. Finally, each of the omitted intermediate paths as well as each gate resulting from a simple component internal failure transformation is replaced by a feed-forward intermediate gate to enhance the FT readability.

For example, taking the generated FT branch shown in Fig. 8, the event ⓪ is obtained from a logical "AND" output from 3 subsequent paths, which makes this event a result of a component-to-component external transformation. Starting from event ② ("omission") failure at the input port to the event ① ("commission") event at the output port, indicates internal failure transformation. So in such a case, event ① and its next gate are kept permanently, while event ② is kept temporally for future analysis. Further down to event ③ ("omission") up to event ② ("omission") is component-to-component failure propagation, so event ③ will be permanently removed while event ② will be kept again temporarily for further analysis.

Next, we remain with event ④ ("omission") up to ② ("omission") which is a propagation as well (omission-to-omission). From here, normally event ④ is supposed to be removed; however, as event ④ is a basic event, event ② will be removed instead. Finally, the whole omitted part of the tree will be substituted by a feed-forward intermediate gate to enhance the readability of the FT. The final version of the FT is provided in the right-hand figure, 8(b). To sum up, we would say that "the internal failure ④ leading to an *"omission"* at the output port of a basic component had transformed into event a *"commission"* failure event ① at some point in the system, in which then combined with the other two failure sources had caused an *"omission"* at the top level of the system.

```
Data: current_failure as f, current_port as p, current_FT as FT, event_to_construct as eG
Result: final populated FT
if p is of not system's port then
    if p is of a "Simple Component" then
        for outExp in p.owner.rules.outputExpressions do
            for f1 in outExp.failures do
                if f1==f then
                    Create an intermediate event e0 ← LOCAL FAILURE;
                    Assign gate to e0 based failures at the p port;
                    Append e0 to eG;
                    Add e0 to FT;
                    for inpExp in p.owner.rules.inputExpressions do
                        for f2 in inpExp.failures do
                            if f2 is not a "wildcard" then
                                if f2 is a "noFailure" then
                                    Create a basic event eT ← INTERNAL FAILURE;
                                    Append eT to eG;
                                    Add eT to FT;
                                else
                                    Create a intermediate event e1 ← LOCAL FAILURE;
                                    Assign an OR gate to e1;
                                    Append e1 to e0;
                                    Add e1 to FT;
                                    for p1 in inpExp.port.connectedPorts do
                                        recurseFT(f2,p1,FT,e1); // Recurse from the start of the Algorithm 3
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    else
        for p1 in p.connectedPorts do
            recurseFT(f,p1,FT,ev);// Recurse from the start of the Algorithm 3
        end
    end
else
    Create a external event eX ← INJECTED FAILURE;
    Append eX to eG;
    Add eX to FT;
end
```

Algorithm 3: Recurse FT operation

4.5.2. FT quantitative analysis

The quantitative probabilistic analysis is meant to automatically calculate the system-level (top event) failure rate. In the proposed approach, the user is able to assign the failure probability rates of the basic failure events, such as internal failure and injected failure. This information can be obtained from the device manufacturer's data sheet as well as from safety experts. The probability calculation follows a widely used formula for conducting a logical output of an "AND" or an "OR" gate in the FT [21,22]. The output of an "AND" gate means that the output event will only happen when a combination of independent events occurs at the same time. On the other hand, the output of an "OR" gate implies that the output event will occur if any one of the input events occurs.

The system failure rate (the top event probability) is calculated following a recursive calculation of the intermediate probabilities for the intermediate events. Based on the probabilities of the basic events, the probability values of their parent events can be calculated from the input event probabilities. The probability calculation follows the formula in Fig. 9. Let N be the number of input events and P_{in} the probability of the input event; the output probability P_{out}, for both "AND" and "OR" gate types, is calculated as follows:

Calculating the probability of a single failure propagation or transformation for a component involves transmitting the probability from an input port to the event's output port. This same principle holds when conducting component-to-component failure propagation, where the probability of the failure event from the previous component's output port is transferred to the event linked with the immediate component's input port. Furthermore, we follow a specific protocol when dealing with branch probabilities stemming from an underdeveloped failure event. Specifically, when these probabilities are directed into an "OR" gate, they are initially set to 0. Conversely, when they are channeled into an "AND" gate, they are set to 1. This approach is deliberately adopted to maintain neutrality during the probability calculation process, as 0 and 1 are considered neutral values in the context of addition and multiplication.

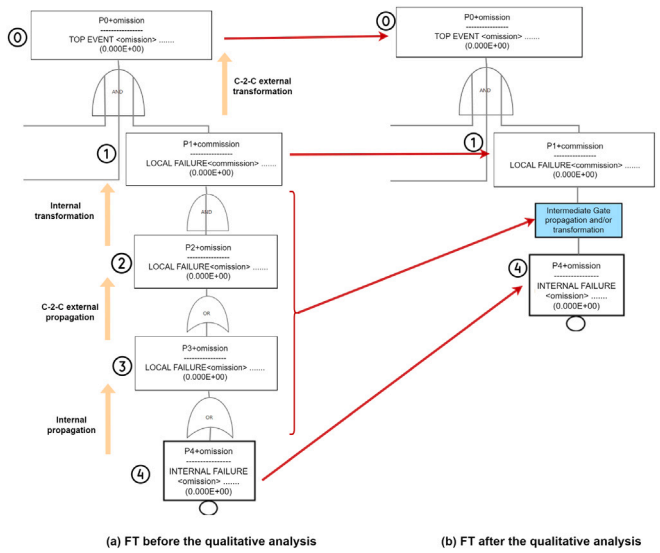


Fig. 8. Qualitative transformation example (a) before, (b) after.

$$P_{out} = \begin{cases} \prod_{in=1}^{N} P_{in} & \text{for an } AND \text{ gate} \\ 1 - \prod_{in=1}^{N} (1 - P_{in}) & \text{for an } OR \text{ gate} \end{cases}$$

Fig. 9. Probability calculation formula.

5. Evaluation

In this section, we describe the approach used to evaluate the proposed approach and the tool in supporting the safety analysis of safety-critical IoT systems with respect to other existing approaches. In Section 5.1, we present briefly the evaluation process we used; in Section 5.2, we introduce the considered case study scenario while Section 5.3 presents the research questions. In Section 5.4, we presents the results of the conducted comparative study. In Section 5.5, the physical architecture of the PMS system is presented whereas Section 5.6 presents the PMS system failure-logic modeling approach. Finally, Section 5.7 presents the FTA analysis results showcasing both qualitative and quantitative aspects.

5.1. Evaluation process

The evaluation procedure we adopted followed six different steps. To demonstrate the potential of our proposed approach, we first identified a use case that precisely fits in the context of the safety analysis of safety-critical IoT systems. Next, we defined research questions that primarily focus on evaluating and demonstrating the effectiveness and potential of the proposed approach. Third, we briefly assessed the key features offered by our system and the supporting tool. Following that, we presented the experimental results, emphasizing answering the posed research questions. Because our proposed approach can be employed in areas other than IoT, the evaluation approach will consider the IoT domain and the widely used FTA approach both in academia and in system engineering industry. To this end we included the most industrial state-of-the-art platforms, including but not limited to ISOGRAPH Reliability Workbench [50], xSAP [51], AltaRica 3.0 [52], EMv2 for AADL [53], HiP-HOPS [54], and MediniAnalyse [55].

5.2. Motivating example: Patient monitoring system (PMS)

Due to the general rapid evolution of electronics and information technology, more powerful bedside patient monitors capable of complex bio-signal processing and interpretation are becoming available, and they are usually equipped with some highly specialized communication interfaces [56]. This goes hand in hand with the huge advances in IoT technologies that allow the integration of such devices with the capability to connect to the internet, which makes it possible to monitor the health state of multiple patients remotely. To support our evaluation process, we adopted the "Efficient Patient Monitoring for Multiple Patients Using WSN" case study [57]. The case study is an advanced system capable of reliably monitoring the multiple parameters of up to six hospitalized patients simultaneously in real time.

The system investigates the potential of employing Wireless Sensor Networks (WSN) to reliably and wirelessly collect multiple parameters such as blood pressure, temperature, electrocardiography (ECG), electroencephalography (EEG), and pulse oximeter (SPO2). These parameters are collected through a set of sensors placed on different parts of the patient's body. For instance, the ECG Sensor is placed on the chest and on the limbs to extract the patient's heart rate data, and the EEG sensor is placed on the patient's head to read electrical activity generated by the brain. Furthermore, the Blood Pressure sensor is placed on the arm to detect the level of pressure in the blood; the SPO2 sensor is placed on the patient's finger to measure the oxygen saturation of a patient's blood; and, finally, the Temperature sensor is placed on any part of the body to measure the temperature. Fig. 10 describes the high-level architecture of the system.

As a result, the recorded parameters are wirelessly transmitted to a computer running PMS software, which feeds them onto the monitor screen in the doctor's office. The software can also wirelessly send alarming messages to the doctor's phone if they are not present to respond to the patient's requests. We will describe the architecture in great detail in Section 5.5.

5.3. Research questions

We study the performance of our proposed approach by considering the following research questions:

- ***RQ1: How does the presented approach distinguish itself from state-of-the-art techniques?*** We simply did a short review of the existing methodologies in relation to what the proposed approach offers to derive our approach's contribution.
- ***RQ2: Does the suggested modeling infrastructure address all the aspects of designing an IoT system, including the need for safety analysis?*** Apart from the modeling language, we looked at the capability of the approach to capture all the required information to facilitate the safety analysis as well as the degree to which such data are taken into consideration when performing the analysis
- ***RQ3: How well do the proposed FLA rules efficiently reflect the system failure behavior leading to the top failure events?*** We look at the efficiency of the derived rules from the possible system failure events in clearly supporting the actual logical analysis leading to the expected results.
- ***RQ4: Does the proposed FT qualitative and quantitative analysis improve the existing FT analysis techniques?*** We will concentrate on the reliability of the result from the supported automated calculus about the existing approaches. We will also have a look at the final FTs and how they better describe the important system failure paths.

5.4. Short literature review (RQ1)

RQ1: How does the presented approach distinguish itself from state-of-the-art techniques?

To sufficiently answer the research question, we needed to first understand the available tools that employ the FTA technique to conduct the safety analysis in the literature. We did not only focus on tools that solely support the IoT domain since we wanted to understand the

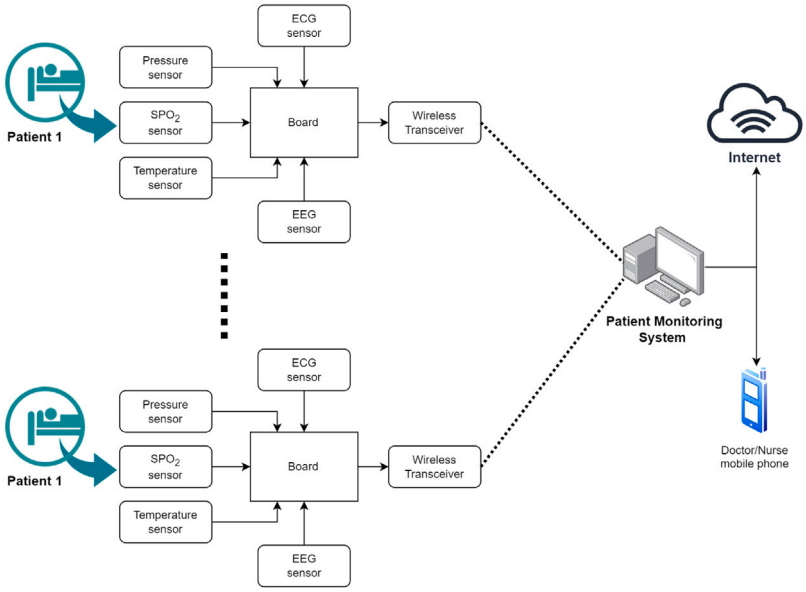


Fig. 10. Basic architecture of Patient Monitoring System [57].

methodologies used by the existing tools in their FT formalism and analysis process in comparison with our proposed approach. In the end, 19 different platforms were found in the literature with much closer ties to our approach.

5.4.1. Search and selection process

The search and selection procedure was divided into four major stages. Since the purpose of the paper is not to undertake an empirical study, we did not conduct this review using well-known databases; we only relied on Google Scholar results. We performed an automatic search using the keywords "Model-based Safety analysis in IoT systems", "Fault Tree analysis in IoT systems", and "Fault Tree analysis from SysML models". The goal was to offer us a sample of current publications on topics related to the search strings. Because each search returned a large number of results, we only analyzed the first two pages of the results. The second phase was conducted for each search, in which we filtered out such results by reading through the titles and abstracts. To pass this step, a paper title and abstract must include at least one of the terms "Fault-Tree Analysis" or "Model-Based Safety Analysis". Following that, we skimmed through the selected articles to exclude those that merely presented an analysis approach but with no supporting tool, resulting in a total of 13 approaches. Finally, we included 6 most industrial tools and workbenches that are extensively considered namely ISOGRAPH [50], xSAP [51], AltaRica 3.0 [52], EMv2 for AADL [53], HiP-HOPS [54], and MediniAnalyse [55]. This led us to a total of 19 primary studies.

5.4.2. Results

To better answer RQ1, we have defined a set of fundamental features we think a safety analysis tool should possess. These features were evaluated on the selected approach in comparison with the proposed infrastructure. The features include supporting system design modeling, failure behavior modeling, automated FT generation, performing qualitative FT analysis, and performing quantitative FT analysis. Table 3 summarizes the findings of the study, in which for each approach, a "Yes" or "No" label was used to indicate if that approach supports that particular feature. As indicated, all 19 approaches are represented against the 6 main evaluation factors. In this section, we will go over the results of the study and have some discussion in line with the factors defined Table 2.

- *IoT-specific*: Given our focus on the IoT domain, our initial concern naturally revolved around evaluating IoT support within existing tools and approaches. After a thorough examination, we found that only 4 out of 19 tools explicitly target the IoT domain. While we acknowledge that certain underlying modeling languages, such as AADL [53], possess design elements that can capture IoT-related aspects to some extent, we firmly believe that being IoT-specific remains a crucial factor for usability and community adoption. Furthermore, it is important to recognize that the IoT domain is characterized by rapid and continuous evolution, with new technologies and concepts emerging daily. Tools lacking IoT-specificity may eventually struggle to effectively accommodate these changes, and extending their capabilities can become challenging. In this dynamic landscape, we view CHESSIoT's contribution as pivotal in advancing the state of the art in IoT system safety analysis. By explicitly addressing the unique requirements of the IoT domain, we aim to provide a solution that not only meets current needs but also adapts seamlessly to the evolving IoT ecosystem, ultimately benefiting both researchers and practitioners in the field.
- *Support for system modeling*: This feature assesses whether the tool supports system-level design before proceeding with its safety analysis. Conducting a safety analysis needs to go hand in hand with the design of the system under analysis. We believe that integrating the design and analysis infrastructure can improve transparency and consistency among system and safety experts. Table 3, on the other hand, indicates that approaches such as [62,63,66] which we considered "academic" do not provide such a feature and instead rely on manually created FT models, which are then transformed into FT graphs. On the other hand, except xSAP [51] which does not clearly provide infrastructure for modeling the system rather it relies on the behavioral-level models of the system. In contrast to the previous approach, our proposed approach completely supports this feature by providing an environment in which system main blocks and sub-systems can be decomposed and analyzed separately.
  Even though approaches such as [7,8,59,69,70] extend the SysML language in the same way that we do, our environment is more user-friendly due to the advanced component-based and multi-view modeling infrastructure, where each view has its own underlined constraints that enforce its specific privileges on model entities and properties that can be manipulated [32]. We also

Table 2
Selected approaches for the experimental analysis.

Domain	Approach	Title	Year	Type
Academic approaches	*JARVIS* [58]	A framework for Model-Driven Engineering of resilient software- controlled systems	2021	Journal
	Z. Haider et al. [16]	FLA2FT: Automatic Generation of Fault Tree from ConcertoFLA Results	2018	Conference
	F. Mhenni et al. [59]	Automatic fault tree generation from SysML system models	2014	Open tool
	I. Silva et al. [60]	A dependability evaluation tool for the Internet of Things	2013	Journal
	H. Fazlollahtabar et al. [61]	Fault Tree Analysis for Reliability Evaluation of an Advanced Complex Manufacturing System	2018	Journal
	MetaFPA [62]	Transformation of Failure Propagation Models into Fault Trees for Safety Evaluation Purposes	2016	Conference
	K. Clegg et al. [63]	Integrating Existing Safety Analyses into SysML	2019	Conference
	smartIflow [64]	Model Based Safety Analysis with smartIflow	2017	Journal
	J. Xiang et al. [65]	Automatic Static Fault Tree Analysis from System Models	2010	Conference
	Y. Chen et al. [66]	Application of Fault Tree Analysis and Fuzzy Neural Networks to Fault Diagnosis in the Internet of Things (IoT) for Aquaculture	2017	Journal
	L. Xing et al. [67]	Reliability Modeling of Mesh Storage Area Networks for Internet of Things	2017	Journal
	B. Alshboul et al. [8]	Obtaining Fault Trees Through SysML Diagrams: A MBSE Approach for Reliability Analysis	2020	Conference
	N. Yakymets et al. [68]	Model-based System Engineering for Fault Tree Generation and Analysis	2013	Conference
Industrial tools	*ISOGRAPH* [50]	ISOGRAPH: Fault Tree Analysis in Reliability Workbench	1986	Industrial tool
	xSAP [51]	xSAP: The XSAP Safety Analysis Platform	2016	Book Chapter
	AltaRica 3.0[52]	AltaRica 3.0: a Model-Based approach for Safety Analyses	2014	Book
	EMv2 for AADL [53]	Automated Fault Tree Analysis from AADL Models	2017	Journal
	HiP-HOPS [54]	A Conceptual Framework to Incorporate Complex Basic Events in HiP-HOPS	2017	Book Chapter
	MediniAnalyse [55]	Toward the Adoption of Model Based System Safety Engineering in the Automotive Industry	2022	Conference

have to recognize that, most of the industrial approaches considered in this short study can still support the modeling of IoT systems, however, as mentioned above, their coherence with IoT-specific complexities could be a potential blocker as the system gets more complex.

- *Support for failure logic behavior modeling*: This feature determines whether the proposed approach provides mechanisms for explicitly defining the failure modes and failure logic behavior of the system's components, which contribute explicitly to the safety analysis process. In our approach, we relied upon FPTC [12] notation to express the system failure behavior logic due to its simplicity. Some of the considered academic approaches use *"IF-THEN"* notation or logical math association expressions to formalize failure behavior including but not limited to that [59,60,62,66]. Nonetheless, all these approaches lack the concepts of external failure injection as well as internal failure transformation and propagation. Furthermore, [68] uses a formal method approach in modeling system failure logic while [8] depends on annotating failure information in the model state machines. However, the complexity behind formal method formalism or state machine definition can be difficult and time-consuming to handle tasks.

  Regarding the industrial tools, the approaches such as Hip-HOPS [54], xSAP [51], and EMV2 for AADL [53] rely on annotating the existing model with non-functional properties which describes failure behavior associated with any given element and the logic with which they are propagated from the element's inputs to its output and adjacent components. Although this is the case none of the approaches relies on FPTC notation. For instance, tools like xSAP [51] rely on the model extension routine which takes input from the *nominal model* (describing behavior in the absence of faults), *the fault library* (containing templates for faults and their dynamics), and the fault extension instructions (specifying directives to instantiate the fault templates) to analyze finite- and infinite-state synchronous transition system. Although this can be handy in performing safety analysis, such a combination can add more complexity to its usability. In contrast, tools such as ISOGRAPH [50] and AltaRica 3.0 [52] do not support in any way any kind of failure behavior modeling of the system.
- *Perform automated FT generation*: This feature determines whether the proposed approach automatically generates the FT from the model rather than manually constructing it. This is one of the main motivations for our proposed approach since we believe that automating the FT generation process is critical to reducing the time safety engineers spend performing the safety analysis as well as increasing transparency in the process. Although most of the industrial tools considered to support the automatic Fault-Tree generation, ISOGRAPH [50], does not support this; this could be due to the scale at which FTs are used in their ecosystem. Not only in safety analysis but also in other domains like reliability analysis, and risk analysis which are supported by the ISOGRAPH workbench. As technology advances, we strongly believe that this should change in order to remain relevant in the market. Our approach provides an FT generation infrastructure that can support large and complex models with advanced features, including but not limited to event tracking, component sub-tree generation and analysis, undeveloped branch identification, and so on.
- *Perform automated qualitative FT analysis*: This feature determines whether or not the proposed approach supports any means for performing qualitative analysis on the generated FT, including detecting minimal cut-sets, FT path reduction, FT event redundancies, and so on. Since FTs can be large, depending on the

Table 3
Analyzed approaches.

Domain	Approach	IoT-specific	System-level modeling	Failure behavior modeling	Automated FT generation	Perform qualitative FT analysis	Perform quantitative FT analysis
Academic	Z. Haider et al. [42]	No	Yes	Yes	Yes	No	No
	JARVIS [58]	Yes	Yes	Yes	Yes	Yes	No
	F. Mhenni et al. [59]	No	Yes	Yes	Yes	No	No
	B. Alshboul et al. [8]	No	Yes	Yes	Yes	No	No
	N. Yakymets et al. [68]	No	Yes	Yes	Yes	Yes	Yes
	H. Fazlollahtabar et al. [61]	No	Yes	No	No	Yes	Yes
	I. Silva et al. [60]	Yes	Yes	Yes	Yes	Yes	Yes
	Y. Chen et al. [66]	Yes	No	Yes	No	Yes	No
	L. Xing et al. [67]	Yes	Yes	No	No	Yes	Yes
	MetaFPA [62]	No	No	Yes	No	No	No
	K. Clegg et al. [63]	No	No	Yes	Yes	No	No
	smartIflow [64]	No	Yes	Yes	Yes	Yes	No
	J. Xiang et al. [65]	No	Yes	Yes	Yes	No	No
Industrial	ISOGRAPH [50]	No	No	No	No	Yes	Yes
	xSAP [51]	No	No	Yes	Yes	Yes	Yes
	AltaRica 3.0 [52]	No	Yes	No	Yes	Yes	Yes
	EMv2 for AADL [53]	No	Yes	Yes	Yes	Yes	Yes
	HiP-HOPS [54]	No	Yes	Yes	Yes	Yes	Yes
	MediniAnalyse [55]	No	Yes	Yes	No	Yes	Yes
	CHESSIoT	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*

system size and complexity as well as the individual component failure behavior, it is vital for an FTA platform to make it easier for the user to navigate through the system's main failure paths in order to better help in defining how they might be easily mitigated. This can be accomplished either graphically, through actions such as path reduction, or textually, through deriving the minimal sets of events required for a system to fail (minimal cut-sets). Aside from that, alternative approaches may be completely platform-specific and dependent on the failure behavior modeling approach and FT generation mechanisms. It can be seen that all of the industrial tools support this feature while 6 out of 13 academic approaches considered have a means to support this. This mainly due to maturity of such tools and the fact that they have been implemented and being used for many years. It should be noted again that although CHESSIoT does support the FT qualitative analysis through unessential FT path removal, FT event redundancies removal, and so on, it does not fully support the deduction of minimal cut-sets. This is currently foreseen as our future work.

- *Perform automated quantitative FT analysis*: This feature determines whether the proposed approach allows quantitative analysis, mainly the top failure event probability estimation. Offering such support could potentially aid in quantifying the risk and determining how to manage it. However, this is regarded as optional in the FTA mechanism due to the lack of a standard means of determining individual component failure rates, since basic event failure rates encompass not just hardware failures but also software, human, and environmental factors. Only 4 out of 13 of the academic approaches evaluated support such a feature, namely [60,61,67,68]. Our same as all of the industrial tools considered that already support this, our proposed approach not only computes the intermediate and top event probabilities from the basic events, but it also recognizes underdeveloped branches and injected failures failure rate computations. It is also worth noting that, during the FT generation process, a file containing the probability information is generated, which allows you to still update the component failure rates and re-run the analysis, in which the new values are picked up by the tool without having to re-generate the FT again.

Looking at the discussions above, it can be seen that the CHESSIoT approach has a point to contribute in regard to the factors considered for evaluation. Although the industrial safety analysis tools seem to be very mature and very advanced in the field, the approach they use still differs from our proposed approach. In addition to that, although tools like AADL [53], xSAP [51], and AltaRica 3.0 [52] are open and free to use, other tools such as ISOGRAPH [50] and MediniAnalyse [55] are not free. Furthermore, we can see from the Table 3, that among all the considered approaches none of them fully support all of the considered six features. Although this might look subjective taking the fact that some of the industrial tools could indirectly support the IoT modeling, however, as highlighted above their scalability in terms of IoT-specific context complexity can be an issue in the long run. In conclusion, the proposed approach is novel and unique in terms of advancing the state-of-the-art in the IoT domain through different novel mechanisms supported as presented in this paper and we believe its contribution in the field is handy.

5.5. PMS system design (RQ2)

RQ2: Does the suggested modeling infrastructure address all the aspects of designing an IoT system, including the need for safety analysis?

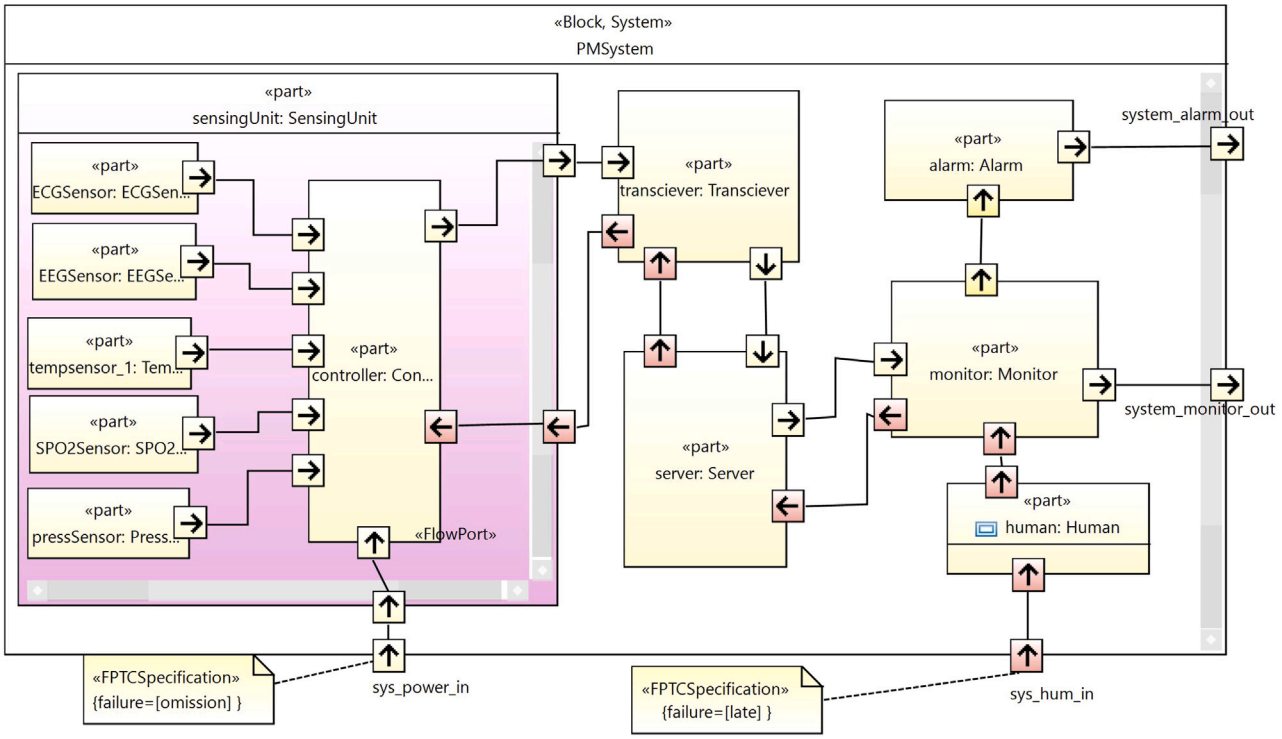


Fig. 11. Patient monitoring system.

5.5.1. PMS system-level model

In order to better tackle this question, we will showcase the capability of our proposed modeling environment by employing the case study presented in Section 5.2. As can be seen in Fig. 10, the Patient Monitoring System (PMS), uses a set of sensors to collect sick patient data and send it to a remote server. The system can display the data on the monitor as well as send an alarm signal when something goes wrong. Fig. 11 represents the internal physical architecture of the proposed system. For the sake of simplicity and to facilitate the analysis process to produce presentable results in a paper, we have considered the following changes to the architecture presented in 10. Firstly, we designed a PMS that only monitors a single patient. Secondly, we introduced a remote server component that acts as a bridge by hosting the service that saves the received data and exposes it to other third-party services that might need it. Thirdly, we replaced the doctor's phone sub-system with an alarming system component that receives data from the PMS software on the monitor side. Finally, we added a "Human" component to reflect the role of a doctor in the overall system functionality.

As shown in Fig. 11, a *"SensingUnit"* composite component consisting of five sensors, namely ECG, EEG, SPO2, pressure, and temperature sensors. All the sensors are placed on a patient's body to collect the patient's health parameters. They are directly sent to the controller, which aggregates all of that information and sends it to a gateway (in this case, a transceiver). The gateway processes the data and forwards it wirelessly to a remote server. The server hosts the software services that save the data and expose it to other authenticated parties in need. On the other hand, the monitoring software deployed on the computer accesses such information and sends it to a display screen. When something goes wrong, for instance, in terms of reading sensor values that exceed or fall below a certain threshold, the monitoring software can decide to raise an alarm in order to alert the doctor about the unusual condition of the patient. In this case, a doctor checks the displayed data and decides to act accordingly by either shutting off the alarm, changing the configuration of the systems, or fixing some issues that might be related to the sensors.

5.5.2. PMS model including failure behavior informations

To facilitate the modeling of the system failure behavior data needed for the safety analysis data, the infrastructure must allow one to annotate the failure logic behavior rules as well as the failure rates on each of the low-level simple components, and this information is fully part of the model itself. As we all know, external influences can cause a system to fail. Through two system-level input ports, the presented architecture allows for simulation effects in which an external failure introduced in the system would affect the overall system functionality. For instance, the *sys_power_in* port, used to model the power source outlet, was injected with an "omission" failure, which basically models the event in case there is a power outage. On the other hand, the *sys_hum_in* port is used for modeling the external influence of the doctor. In our case, a "late" failure was used to simulate an event in which a doctor reacted late due to some external factors. Finally, the system contains two output ports, namely system_monitor_out for modeling the output port from the monitor, and the system_alarm_out to model the output of the alarm system. According to the direction of the ports at the system level as well as the types of failures that will be able to propagate to them, different FTs will be generated accordingly.

5.6. System failure behavior (RQ3)

RQ3: How well do the proposed FLA rules efficiently reflect the system failure behavior leading to the top failure events?

As was previously anticipated, the above system is subjected to different kinds of failures, either internally generated by the system or coming from the surrounding environment. As we described in the previous section, it is possible to model an individual component's failure behavior that later gets assessed in determining the failure behavior of a sub-system or an entire system. Note that we will not be focusing on software-level functional behavior but on physical failure behavior, which can even be understood by nonprofessional users. In order to understand the need for the conducted analysis, let us first discuss the different top failure scenarios that we have taken into account in defining individual component rules.

Table 4
PMS failure behavior table.

Component	Rules	Description
ECG, EEG,	(1) FLA:(*) → ecgsens_out.**omission**;	Sensor fails internally which makes it unable to read and push any at output port
Temp, SPO2, Pressure	(2) FLA:(*) → ecgsens_out.**valueCoarse**;	Sensors begin to fail as they age and provide incorrect data to the output; this can also be caused by sensor components that are not properly mounted to the patient body.
sensor	Rules (1) and (2) apply to other sensors	Same as other sensors
Controller	(3) FLA:ecg_cont_in.**noFailure**, eeg_cont_in.**noFailure** → cont_trans_out.**omission**;	Controller fails completely omitting to send the data
	(4) FLA:mon_power_in.**omission**, trans_cont_in.**omission** → cont_trans_out.**omission**;	The controller fails to function due to a power outage at its input power port, no backup solution is available *(trans_cont_in port)*
	(5) FLA:ecg_cont_in.**omission**, eeg_cont_in.**omission**, press_cont_in.**omission**, spo_cont_in.**omission**, temp_cont_in.**omission** → cont_trans_out.**omission**;	All of the sensors simultaneously stop sending data, preventing the controller from sending any data to the server
	(6) FLA:ecg_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	The controller receives inaccurate data from the ECG sensor and sends it to its output port
	(7) FLA:eeg_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	controller receives inaccurate data from the EEG sensor and sends it to its output port
	(8) FLA:press_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	controller receives inaccurate data from the blood pressure sensor and sends it to its output port
	(9) FLA:spo_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	controller receives inaccurate data from the SPO2 sensor and sends it to its output port
	(10) FLA:temp_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	controller receives inaccurate data from the temperature sensor and sends it to its output port
	(11) FLA:trans_cont_in.**valueSubtle** → cont_trans_out.**valueCoarse**;	The controller receives an undetected error at its from-system port, which impedes sensor data transmission.
	(12) FLA:trans_cont_in.**valueSubtle** → cont_trans_out.**omission**;	The controller receives an undetected error at its from-system port halting the data transmission process.
Transceiver	(13) FLA:trans_in_fr_unit.**valueCoarse** → trans_out.**valueCoarse**;	The transceiver received the wrong data and transmitted to its output port
	(14) FLA:trans_in_fr_unit.**noFailure**, trans_in_f_serv.**noFailure** → trans_out.**omission**;	The transceiver fails internally causing the halt of data transmission process
	(15) FLA:trans_in_fr_unit.**omission** → trans_out.**omission**;	The transceiver receives no data and fails to transmit data to its output
	(16) FLA:trans_in_f_serv.**valueSubtle** → trans_o_2_unit.**valueSubtle**;	The transceiver receives an undetected error at its server port and forwards it back to the sensing unit
Server	(17) FLA:server_in.**noFailure** → server_out.**omission**;	The server fails, bringing the transmission process to a halt
	(18) FLA:server_in.**valueCoarse** → server_out.**valueCoarse**;	The server routes the incorrect data received at the input to the output port
	(19) FLA:server_in.**omission** → server_out.**omission**;	The server receives no data from its input port, and this error is forwarded to its output
	(20) FLA:serv_in_f_mon.**valueSubtle** → serv_2_trans_out.**valueSubtle**;	The server sends an undetected error from the monitor back to the transceiver's port
Monitor	(21) FLA:monitor_in.**noFailure** → monitor_out.**omission**, mon_alarm_out.**omission**;	The monitor fails internally omitting to display the data on the screen as well as not communicating to the alarm component
	(22) FLA:monitor_in.**omission** → monitor_out.**omission**, mon_alarm_out.**omission**;	The monitor receiving no data from the server omitting to display the data as well as not sending any communicating signal to the alarm component
	(23) FLA:monitor_in.**valueCoarse** → monitor_out.**valueCoarse**, mon_alarm_out.**commission**;	The monitor receives inaccurate data and displays it on the screen, potentially sending an unexpected notification to the alarm component *(Commission)*
	(24) FLA:hum_mon_in.**valueSubtle** → mon_2_serv_o.**valueSubtle**, mon_alarm_out.**commission**, monitor_out.**noFailure**;	The monitor receives an unpredicted error from the human nurse component, the failure propagates in the system in various ways with no direct effect on the data displayed on the screen before (Refer to Rules 30 and 31 for possible causes)
	(25) FLA:hum_mon_in.**omission** → mon_2_serv_o.**valueSubtle**;	The monitor receives no engagement from the nurse intended to resolve the issue in the system, the cause of which we do not know. As a result, such a failure will go unnoticed by the system. (Refer to Rules 11 and 12 for possible effects)
Alarm	(26) FLA:mon_alrm_in.**commission** → alarm_out.**commission**;	The alarm component received an inaccurate notification and immediately rings because it lacks any type of logical reasoning on the signal receiving other than ringing.
	(27) FLA:mon_alrm_in.**noFailure** → alarm_out.**commission**;	The alarm starts failing due to internal failure which can make it malfunction by giving false alerts

(*continued on next page*)

Table 4 (*continued*).

Component	Rules	Description
	(28) FLA:mon_alrm_in.**noFailure** → alarm_out.**omission**;	The alarm component fails which makes it unable to make any alert
	(29) FLA:mon_alrm_in.**omission** → alarm_out.**noFailure**;	The alarm receives no data but that will not affect the internal functionality of the alarm
Human nurse	(30) FLA:human_in.**late** → human_out.**valueSubtle**;	The human nurse reacts very slowly in the event of a system failure, which may or may not affect the system in some way, which is why a "valueSubtle" is considered.
	(31) FLA:human_in.**noFailure** → human_out.**omission**;	The absence of the doctor results in an omission at the output port

- **The alarm sub-system malfunctions by sending out a false signal:** In normal settings, this can occur when the alarm component receives the wrong alarm notification. This is usually caused by the monitor software making a decision based on incorrect data from one of its input ports. The alarming system, on the other hand, can send false signals due to its internal failure for a variety of reasons, such as poor internal configuration or simply aging.
- **The alarm subsystem has completely stopped working:** It is possible that the alarm system no longer works completely. This can be caused by several reasons; for example, the alarm system being physically disconnected or an internal failure that causes a total blackout.
- **The monitor is displaying incorrect data:** As is obvious, the main cause of this could be incorrect data being sent to the monitor. However, other factors, such as a faulty monitor, losing connection to the internet, and so on, make it display the last received data. It should be noted that these are only generalized assumptions; the extensive individual study, as well as their corresponding failure rules, is shown in Table 4.
- **The monitor completely fails to display data on the screen:** This can occur due to internal and/or external monitor issues such as the monitor not being physically connected to the system power source, being unable to connect to the server, internal monitor malfunction due to aging, and so on. On the other hand, this could be caused by the monitor being properly connected but the server not receiving any data from the sensing unit.

The next step is to derive internal failure rules as well as propagation rules for the basic components. For instance, for each sensor, two rules were defined to model two different scenarios in which a sensor can fail. A sensor can fail internally, leading to a complete omission in providing the data to the output port; thus, an "omission" failure will be propagated to the output port of the sensor. On the other hand, a sensor can start to fail, but not completely, due to age. This may result in providing incorrect data to the output; this can also be caused by sensor components that are not properly placed in the patient's body. In this case, we consider that the value sent to the output port is of the "valueCoarse" type. Hence, the two different types of failure can be propagated to the same output port in different scenarios, and they will be represented as indicated in Expressions (7) and (8) respectively. We considered only the two failure conditions to apply to all of the sensors. As it can be seen from Table 4, a detailed set of failure behavior rules and their descriptions are represented.

$$FLA : (*) \rightarrow ecgsens\_out.\textbf{omission} \tag{7}$$

$$FLA : (*) \rightarrow ecgsens\_out.\textbf{valueCoarse} \tag{8}$$

In CHESSIoT, to facilitate the quantitative analysis, the failure rates of the component internal failure events as well as the injected failure events have to be set separately. As we did not have the exact failure rates of the basic components, we considered the arbitrary failure rates of any component to be practically small in a range of 10^{-8} to the 10^{-7}. Fig. 12 depicts the interfaces in which the internal failure and its description are set. The event description is useful for improving the readability of the FT, but it is not required in conducting qualitative analysis. When no data is provided for any of the rows, the default values are used. For instance, an unset basic event probability is assigned a value of zero in the FT, while the unset basic event description will still follow the naming convention of ''<failure type> at <port name> in <component name>''.

As previously discussed, our proposed approach is capable of satisfying all potential failure behaviors prescribed by the safety expert. As shown in Fig. 11, our approach is capable of modeling the backward failure propagation paths. For instance, a server failure will affect the monitor's behavior, preventing data from being displayed on the screen. On the other hand, an erroneous command sent in the doctor's absence (for example, to fix an unmounted sensor) may eventually propagate back to the sensing unit, causing a wrong value error to be transmitted at the controller output port (valueCoarse failure) or possibly suspending the data transmission process (omission failure). It is also worth noting that the ability to integrate all of their component failure rules as well as their failure rates into the same model has the potential to boost model consistency as well as transparency in the modeling process.

5.7. PMS system fault tree analysis (RQ4)

RQ4: Does the proposed FT qualitative and quantitative analysis improve the existing FT analysis techniques?

The FT analysis begins after the CHESS-FLA transformation, as described in Section 4.1. The FT generation process is performed prior to running the FT analysis, in which each of the top events described in Section 4 results in its own FT. For instance, FT leading to a "omission" failure at the system monitor out port is generated to show the entire failure contribution leading to that top event. Other FTs representing the remaining 3 top events are generated as well. At this stage, the generated FTs are very large as they contain every detail related to failure propagation and transformation from component to component, making them tricky to read. Therefore, FT analysis can then be launched to automatically perform both qualitative and quantitative analysis on the model. Fig. 13 shows the analyzed FT of the event "the monitor fails to display data completely on the screen". The presented FT showcases only the important events and logical gate combinations. It can be clearly anticipated that the analyzed FT makes it easier to identify and trace any failure source events in their contribution to the top failure event. For instance, we can easily grasp that the monitor would display no data on the screen completely when any of the following events occur:

- Internal failure in the monitor (10^{-8} probability)
- Server is down (2×10^{-8} probability)
- The transceiver (gateway) fails completely to transmit the data (3×10^{-8} probability)
- A combination of events (low-left AND gate) in which there is a problem with the sensing unit power source and there is no one to fix that at the moment. (5×10^{-15} probability)
- The controller of the sensing unit fails completely, which halts the transmission process (1.2×10^{-8} probability)

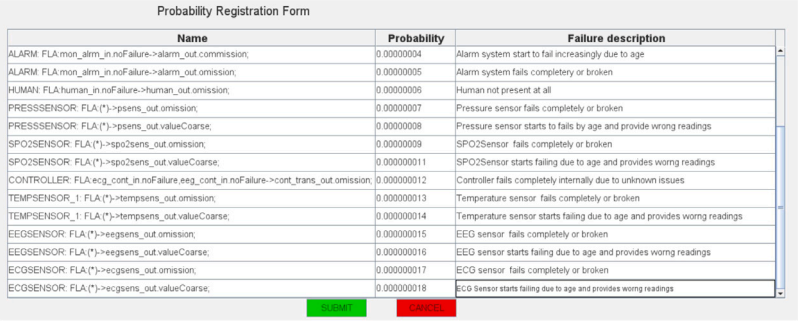


Fig. 12. PMS components failures rates set.

- An event in which all the sensors do not send the data at all (This is more unlikely but possible; that is why we have a lower-middle and gate combination with 2.088×10^{-38} probability).
- An unknown human error occurred from the monitor side (2×10^{-7} probability)

When the monitor fails to provide any data, medical workers can rely on just a handful of events to pinpoint the source of the problem. Furthermore, medical workers can use the probability associated with each basic event in the list to swiftly identify the source of failure, progressing from the most likely basic event (highest probability) to the least probable event (lowest probability). The overall probability of this system-level failure event occurring is calculated to be 2.72 × 10−7. This probability is practically small; however, it can be looked at as due to the fact that it is calculated automatically and solely dependent on the arbitrary basic event failure rates.

Other analyzed FT on the event in which "monitor displaying incorrect data" and "PMS alarm sub-system alert false signal" are shown in Figs. 14 and 15 respectively. As it can be seen from Fig. 14, the event in which the monitor will display incorrect data can be caused by any of the sensors (OR gate), as well as an unforeseen human error that transforms throughout the system and hinders the data that are being transmitted. The overall probability in which such an event can occur is calculated to be about 3.39×10^{-7} which is higher than the event in which the monitor can stop working at all.

On the other hand, as shown in Fig. 15, the same events that cause the monitor to display incorrect information can also cause the monitor to send a false signal via a failure transformation, resulting in a false alarm event in the alarm system. It is also worth noting that an event like the alarm sub-system failing with a probability of 4×10^{-8} would also contribute to that cause. The overall probability of such an event occurring is projected to be around 5.79×10^{-7} which is much higher than the previous two top events. Furthermore, while the "Human error" basic event appears twice in the tree, such failure passes through different channels and eventually transforms into other types throughout the system. This is practically important to understand which component of a system's error would change its nature, potentially causing a lot more damage than expected. Finally, It is worth noting that this FT does not include the top event in which the alarm sub-system stops working completely. An FT reflecting such an event was generated and analyzed separately.

Typically, safety engineers will collaborate with system engineers to keep the safety model up-to-date during the development process. Maintaining coherence between system architecture and the safety model can be difficult as the model gets larger and more complicated. Having a framework that can integrate modeling and analysis processes from a single place would potentially improve consistency, increase transparency, and minimize analysis time. Overall, the proposed analysis approach is capable of achieving that by means of automated qualitative and quantitative calculus.

6. Related work

Fault tree analysis is one of the hugely used and suggested methods when performing different dependability analysis studies, including safety analysis [30]. We have also mentioned that FTs are among the mandatory artifacts that should be provided for performing Safety Analysis in different domains and IoT is yet to follow [11]. However, most of the approaches presented in the literature still rely on the manual construction of the FTs, which still makes the process time-consuming. In this section, we will present go over two categories of related tools namely general-purpose industrial tools and UML-based approaches.

6.1. General purpose industrial tools

ISOGRAPH Reliability workbench [50] is a powerful integral visual modeling and analysis environment in which all the aspects of the reliability analysis are managed. This software provides comprehensive tools for assessing and improving system reliability and safety. It allows engineers to model complex systems and analyze potential failure scenarios by making creating fault trees, event trees, and reliability block diagrams easier. ISOGRAPH supports failure rate and maintainability prediction, Failure Mode Effects, Criticality Analysis (FMECA), and Reliability Allocations (Reliability Block Diagram as well as Fault Tree, Event Tree, and Markov analysis combined). Although this tool is seemingly powerful in terms of what can be covered, differently from our approach in which the system FTs are automatically generated from the analysis, the FTs are still manually constructed from the system failure requirements provided by the safety experts.

AltaRica 3.0 [52] is a formal modeling language that is widely used in safety-critical system analysis and design. It was originally developed by the French company France Telecom R&D (now part of Orange Labs) and is commonly used in the aerospace, automotive, and nuclear industries. Since version 3.0, it has been developed by the non-profit AltaRica Association,[1] along with the associated modeling environment AltaRica Wizard.[2] It excels at safety analysis by allowing users to create precise models of complex systems, including potential faults and failures. Safety properties and critical failure modes can be formally specified for rigorous mathematical verification. Altarica's support for fault tree and event tree generation and representations simplifies the visualization of fault scenarios and their consequences. It also supports redundancy and fault tolerance modeling, ensuring that safety-critical systems remain resilient in the face of failures.

[1] https://www.altarica-association.org/

[2] https://www.altarica-association.org/Products/Software/AltaRicaWizard/AltaRicaWizard.html

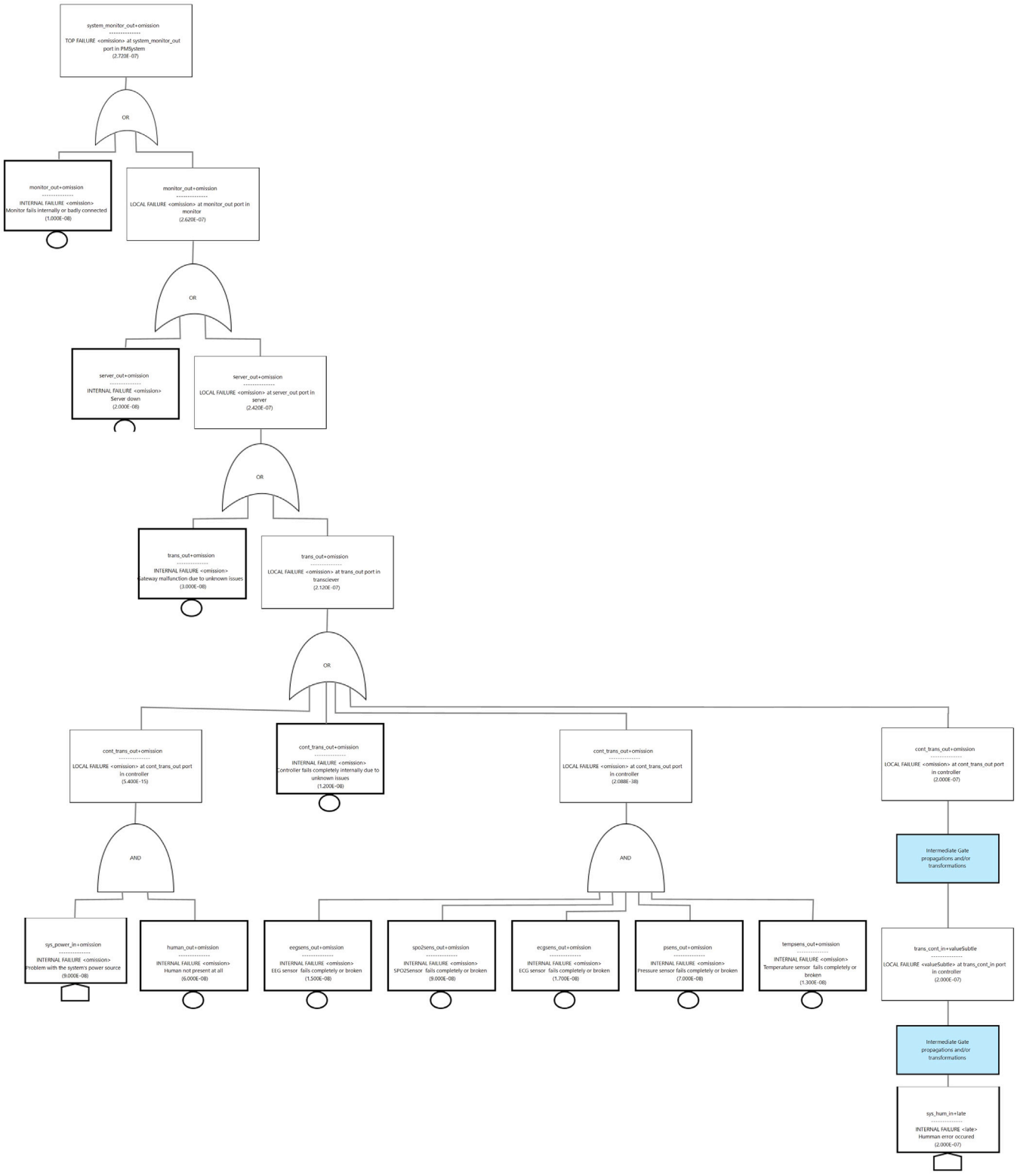


Fig. 13. PMS monitor screen shown no data.

xSAP [51] is a tool for the safety assessment of synchronous finite-state and infinite-state systems. Based on symbolic model-checking techniques, XSAP includes several model-based safety analysis features for finite- and infinite-state synchronous transition systems. It supports explicitly library-based fault mode definition, an automatic model extension facility, and the generation of safety analysis artifacts such as Dynamic Fault Trees (DFTs) and Failure Mode and Effects Analysis (FMEA) tables. Furthermore, it supports the probabilistic evaluation of

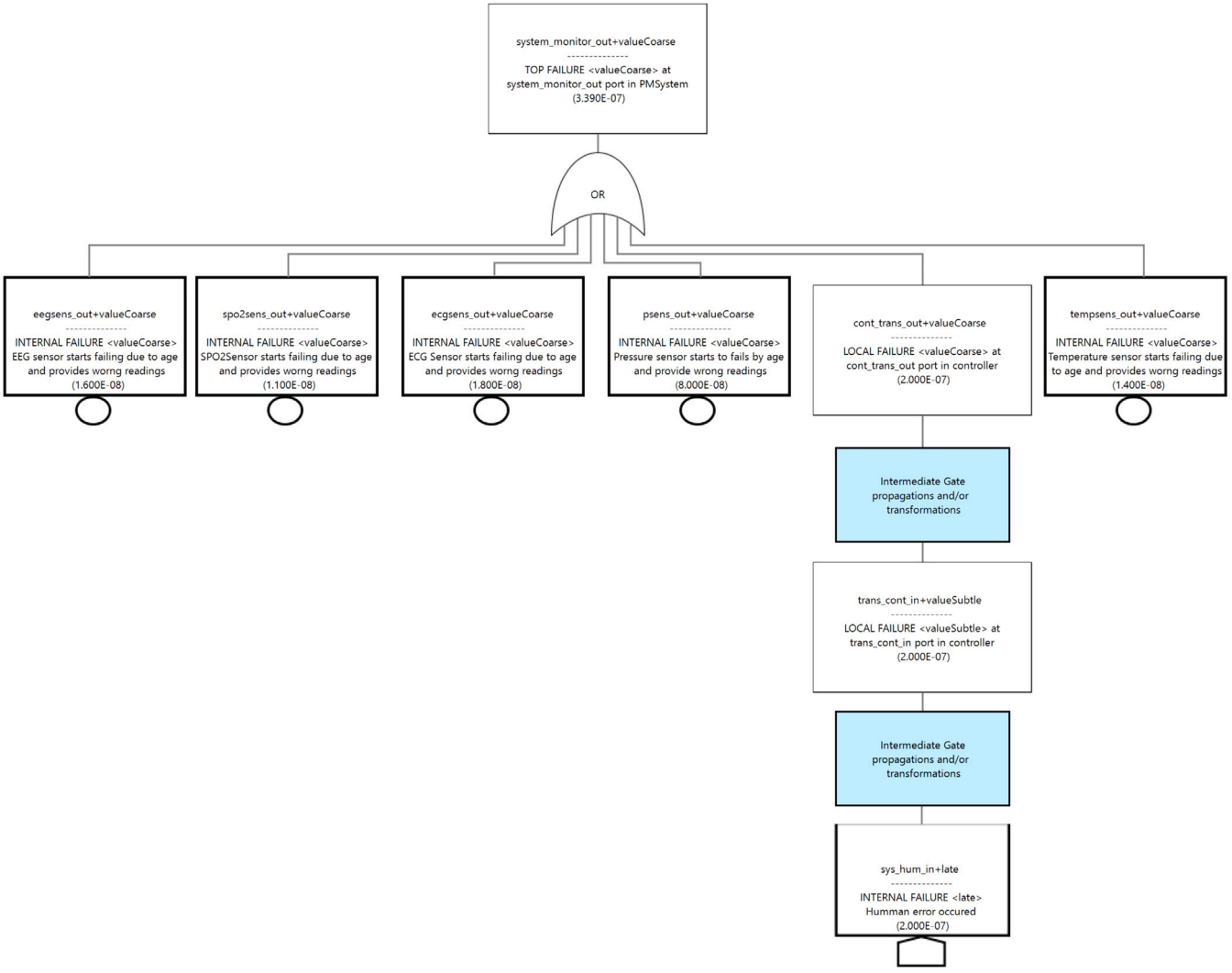


Fig. 14. The monitor fails to display data completely on the screen.

Fault Trees, failure propagation analysis using Timed Failure Propagation Graphs (TFPGs), and Common Cause Analysis (CCA). XSAP has been used as a verification back-end in several industrial projects, and it is currently being evaluated in a joint R&D project involving FBK[3] and The Boeing Company.[4]

The Error Model Annex (EMV2) [53] is an Architecture Analysis and Design Language (AADL) extension that provides a formal framework for performing safety analysis and modeling in complex embedded systems. In EMV2, users can specify error sources, error types, and their relationships, resulting in a comprehensive view of potential errors within a system. To represent how errors propagate through the system architecture, error propagation, and transformation rules can be defined to analyze fault effects on system components. Furthermore, EMV2 facilitates quantitative analysis by associating probabilistic information with error models, allowing the calculation of metrics such as failure rates and system failure probabilities under various conditions. Although this approach seems to match our proposed approach in terms of failure behavior modeling, their approach relies on descriptive error definition while we follow FPTC notation [12].

HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) [71] enables integrated assessment of a complex system from the functional level through to the low level of component failure modes. In HiP-HOPS, the user begins by annotating a model of the system architecture with mostly local (per system element) failure behavior information. This information describes any basic events associated with the given element and the logic propagating them from the element's inputs to its outputs. Once the annotation of a system model is complete, the HiP-HOPS tool can be invoked, automatically synthesizing local fault trees for each system element. Once the resulting minimal fault tree is complete, it can be analyzed qualitatively and quantitatively [54].

Medini Analyze [55] is a powerful tool for analyzing safety in complex engineering systems, particularly in the automotive, aerospace, and rail transportation industries. Developed by Ansys, it includes several features that aid in identifying, assessing, and managing safety-related issues throughout the system analysis lifecycle. Medini Analyze provides Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) to examine how faults and failures in a system can lead to hazardous conditions. Users may generate thorough fault trees and FMEA worksheets within the application, allowing them to simulate and evaluate the consequences of various failure scenarios. This aids in understanding the criticality of failure modes and prioritizing safety improvements. Medini Analyze also interfaces with model-based systems engineering methodologies, making it easy to correlate safety analysis artifacts with system models, maintaining consistency and traceability throughout the development process.

[3] https://xsap.fbk.eu/
[4] https://www.boeing.com/

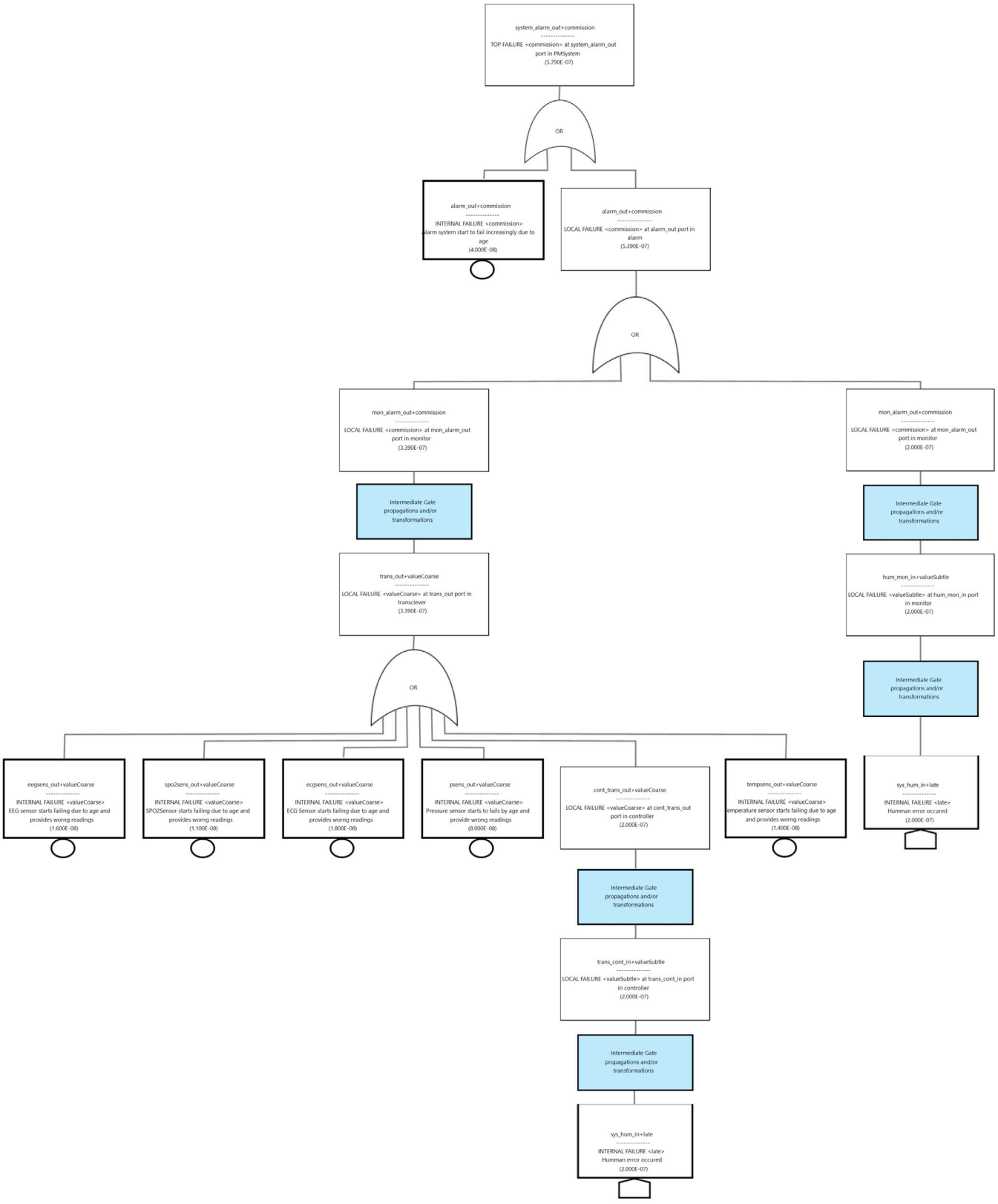


Fig. 15. Alarm sub-system alert false signal.

6.2. UML/SysML-based approaches

The approach presented in [42] employs the CHESS-FLA transformation results to build ECSS-compatible FTs. Although their approach is related to ours, it differs significantly in various points. To name a few, their approach only supports system-level component composition while creating FT, while our approach supports any level of composition. For instance, with our approach, FTs of a single composite

component can be generated and analyzed individually. Furthermore, in their approach, only basic events from the system-level input ports can be generated, whereas in our case, any component can initiate a basic failure event. Finally, their approach only supports FT generation, with support for neither qualitative nor quantitative FT analysis, whereas our proposed approach supports both qualitative and quantitative analysis by removing unnecessary paths and redundancy, as well as quantitative analysis through the probabilistic calculation approach.

The authors of [58] present JARVIS (Just-in-time ARtificial intelligence for the eValuation of Industrial Signals), a model-driven tool that facilitates the development and verification of the integration of physical IoT devices, enterprise-scale software agents, data analytics, and human operators. JARVIS promotes semi-formal specification of structural elements, functional requirements, and behavioral characteristics of subsystems from a System of Systems perspective. JARVIS employs agents to facilitate the development and integration of intelligent data agents capable of detecting failure events that occur in accordance with a set of failure modes. Eventually, a FaultTreeAnalyzer agent is used to perform Fault Tree Analysis on detected failure events. Although their approach performs a qualitative analysis, the quantitative one is not supported. Finally, their FT generation approach relies on the practical data model constructed by the deployed agent, while our approach relies on FLA for the FT generation.

Several approaches have been proposed for the automatic generation of FTs from SysML models. For example, the authors in [59] present an approach for generating FTs from SysML models, relying on a combination of information provided in activity and IBD diagrams as well as information in the FMEA table. Although the current tool generates a single FT picture representing the system failure paths, no FT models are generated. Another critical difference with respect to our proposed approach regards the use of *directed graph traversal* and *block design patterns* to generate FTs which is nothing but using the component-directed edge relationships to determine how the next component has to be represented in an FT. Even though the presented block design patterns are useful to derive the component failure propagation behaviors, they do not cover certain topics such as "internal failure of the components", since this information is probably picked from the FMEA table, as well as they do not provide any support for any kind of automated qualitative or quantitative FT analysis.

The approach in [8], as well as that in [69] presents an MDE environment for performing preliminary safety analysis from SysML models. The approaches use UML state machines to model the component functional behavior and annotate them with failure behaviors; later such information is used to generate the system FTs. Although the proposed approach generates the FTs, certain aspects of the safety analysis are not covered such as injected or external failures, "AND" gate logic, undeveloped events as well as the qualitative or quantitative analysis of the generated FTs. On another hand, in [68], the authors presented a framework that integrates the formal method approach for facilitating the automatic FT generation within an MDE workflow. The approach annotates to the SysML model elements the formal analytical expressions showing how deviations in the block outputs can be caused by internal failures of the block and/or possible deviations in the block inputs. Later this information is transformed into an AltaRica model [52] representation which is used to perform qualitative and quantitative analysis using the XFTA tool provided by the framework. Although this approach seems very interesting, the process of annotating the model with formal analytical expression can be very complex to grasp whereas, in our proposed approach, failure logic behavior rules following FPTC notation are used and we retain they are simpler and straightforward to be used.

In terms of Safety-critical systems, the authors of [45] present an approach for performing a combination of FMEA and FTA analysis on safety-critical systems starting from the Preliminary Hazard Analysis (PHA) method, initially conducted by the safety experts. However, no supporting tool is provided. The same occurs with [61], which presents an approach for manually deriving FT diagrams from the Reliability Block Diagram (RBD) and, later, the qualitative and quantitative analysis are manually performed, differently from our approach where the analysis is performed automatically. Furthermore, unlike our approach which models the system architecture, annotates the model with safety-related information, and later generates and analyzes FTs, several approaches, such as [63,65,70], propose SysML profiles which are used to create FT models and later translate them into FT graphs without any support for system modeling itself. Another approach, such as [62], proposes a Meta-modeling-based Failure Propagation Analysis (MetaFPA) framework to support the synthesizing of the system failure propagation models in order to help the creation of the system FTs. Although the presented framework presents an alternative to FPTC on how system failure propagation rules can be modeled, unlike our proposed approach, the framework does not generate the system FTs but relies on the ISOGRAPH tool [50] to perform the FTA. The same goes with the approaches proposed in [61,72] which rely on the ISOGRAPH software to manually construct and analyze the FTs.

6.3. IoT-specific safety analysis approaches

In the IoT domain, very few approaches specifically target the execution of safety analysis on IoT systems. As briefly mentioned in Section 2, this might be mainly caused by the lack of systematic, disciplined, and quantifiable software engineering standards, as well as comprehensive abstraction models for dealing with the increasing complexity and safety requirement heterogeneity present in the IoT domain. In [60], the authors present a dependability evaluation tool for IoT applications, when hardware and permanent link faults are considered. The tool supports the modeling of system network architecture and, later, the so-called network failure condition events (*nfc*) are defined to help in generating the FT. The *nfc* formalism somehow follows the logical association rules for addition and multiplication in order to reflect the "OR" and the "AND" gates respectively. Finally, the tool supports the qualitative analysis, by generating minimal cut-sets, as well as the quantitative analysis. Although this tool supports the automatic generation and the analysis of the FTs, it differs from our approach both in terms of system failure behavior formalism and because it does not support any mechanism related to failure transformation, propagation, and injection.

In [66], the author presents an intelligent method for fault diagnosis based on a combination of FTA and fuzzy neural networks in the aquaculture IoT systems. In their approach, the FT is manually constructed for each component of the system and later the "IF-THEN" rules are extracted from the FT to be fed into the fuzzy neural network to train the relationship model between fault symptoms (failures) and faults. Although this method uses the FTA for the safety analysis process, it differs from our approach since the generation of the FT is done manually, while in our case it is performed automatically. Furthermore, our approach conducts a quantitative analysis by calculating the system failure probability, while their approach does not. Finally, the approach in [67], presents an FT modeling infrastructure in which different reliability analyses for mesh topology IoT networks are performed taking into account the quantitative analyses. However, same as the ISOGRAPH tool, the aspect of the FT construction is still manually done from the system failure requirements provided by the safety expert.

From the above discussions, we can see that, none of the approaches has addressed the safety analysis for safety-critical IoT systems with both automated generation with qualitative and quantitative FTA analysis. As a result, we believe that our proposed approach is considered to be unique with respect as well as contributing to the literature.

7. Conclusion and future work

Automated safety analysis is critical for increasing transparency and reducing the time required for manual analysis. However, when the system becomes too large and complex, it is challenging to maintain the coherence between the safety analysis model and the corresponding system architecture. In addition, the architecture usually has to be reworked many times, which can hinder the consistency of the process. This paper presented CHESSIoT, a novel approach for developing and performing safety analysis on safety-critical IoT systems. The proposed method combines rigorous automated analysis procedures with annotated failure behavior on components and associated failure rates to generate fault trees. The supporting tool can perform both qualitative and quantitative analysis on generated FTs. We presented an evaluation mechanism compared to existing techniques, taking into account the academic research and industrial tools to showcase its contributions better, and the results were promising. The approach improves model composability and reuse while reducing the time required to perform the safety analysis of safety-critical IoT systems.

In the future, we plan to implement the infrastructure for deriving minimal tree representation based on minimum cut-set events. In addition, we plan to integrate time-based failure logic analysis and the severity aspects into our approach. This is mainly to reflect the effect that a component failure may cause on the entire system taking into account short or longer periods and how severe it could be. We intend to improve our system failure mode abstraction method by making it easily customizable from one domain to another, as well as providing testing support to potentially assist in the recommendation of any potentially missing safety rules [73]. In conclusion, our plans include expanding the proposed approach to incorporate dynamic fault trees, enabling real-time analysis capabilities.

CRediT authorship contribution statement

Felicien Ihirwe: Conceptualization, Methodology, Software, Writing – original draft, Writing – review & editing. **Davide Di Ruscio:** Conceptualization, Resources, Methodology, Writing – original draft, Writing – review & editing, Supervision, Project administration, Funding acquisition. **Katia Di Blasio:** Methodology, Writing – original draft. **Simone Gianfranceschi:** Resources, Supervision, Project administration, Funding acquisition. **Alfonso Pierantonio:** Supervision, Writing – review & editing, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Felicien Ihirwe reports financial support was provided by European Union. Katia Di Blasio reports financial support was provided by European Union. Davide Di Ruscio reports financial support was provided by PRIN 2020 program. NONE

Data availability

No data was used for the research described in the article.

Acknowledgments

This work has received funding from the Lowcomote project under European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement n°813884 (Italy); This work also received funding from the VALU3S H2020-C Funds (Italy) through the ECSEL Joint Undertaking (JU) under grant agreement n°876852; This work has also been partially supported by the EMELIOT national research project (Italy), which has been funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY).

References

[1] A. Taivalsaari, T. Mikkonen, A roadmap to the programmable world: Software challenges in the IoT era, IEEE Softw. 34 (1) (2017) 72–80, http://dx.doi.org/10.1109/MS.2017.26.

[2] J.C. Kirchhof, B. Rumpe, D. Schmalzing, A. Wortmann, MontiThings: Model-driven development and deployment of reliable IoT applications, J. Syst. Softw. 183 (2022) 111087, http://dx.doi.org/10.1016/j.jss.2021.111087.

[3] A. Power, 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, in: IoT 2019, Association for Computing Machinery, New York, NY, USA, 2019, http://dx.doi.org/10.1145/3365871.3365872.

[4] F. Ihirwe, D. Di Ruscio, S. Mazzini, P. Pierini, A. Pierantonio, Low-code engineering for internet of things: A state of research, in: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, Association for Computing Machinery, New York, NY, USA, 2020, http://dx.doi.org/10.1145/3417990.3420208.

[5] A. Gómez, M. Iglesias-Urkia, L. Belategi, X. Mendialdua, J. Cabot, Model-driven development of asynchronous message-driven architectures with AsyncAPI, Softw. Syst. Model. (2021) 1–29, http://dx.doi.org/10.1007/s10270-021-00945-3.

[6] R. kamal Kaur, B. Pandey, L.K. Singh, Dependability analysis of safety critical systems: Issues and challenges, Ann. Nucl. Energy 120 (2018) 127–154, http://dx.doi.org/10.1016/j.anucene.2018.05.027, URL https://www.sciencedirect.com/science/article/pii/S030645491730213X.

[7] S. Aircraft, S. Dev, S.A. Committee, Guidelines for development of civil aircraft and systems, 2010, http://dx.doi.org/10.4271/ARP4754A.

[8] B. Alshboul, D.C. Petriu, B. Alshboul, D.C. Petriu, Automatic derivation of fault tree models from SysML models for safety analysis, J. Softw. Eng. Appl. 11 (2018) 204–222, http://dx.doi.org/10.4236/jsea.2018.115013.

[9] A.A. Abdellatif, F. Holzapfel, Model based safety analysis (MBSA) tool for avionics systems evaluation, in: 2020 AIAA/IEEE 39th Digital Avionics Systems Conference, DASC, 2020, pp. 1–5, http://dx.doi.org/10.1109/DASC50938.2020.9256578.

[10] A. Joshi, S. Miller, M. Whalen, M. Heimdahl, A proposal for model-based safety analysis, in: 24th Digital Avionics Systems Conference, Vol. 2, 2005, p. 13, http://dx.doi.org/10.1109/DASC.2005.1563469.

[11] G. Girard, I. Baeriswyl, J.J. Hendriks, R. Scherwey, C. Müller, P. Hönig, R. Lunde, Model based safety analysis using sysml with automatic generation of FTA and FMEA artifacts, in: Proceedings of the 30th European Safety and Reliability Conference and the 15th Probabilistic Safety Assessment and Management Conference, Esrel 2020 PSAM 15, 1–5 November 2020, Venice, Italy, 2020.

[12] R.F. Paige, L.M. Rose, X. Ge, D.S. Kolovos, P.J. Brooke, FPTC: Automated safety analysis for domain-specific languages, in: M.R.V. Chaudron (Ed.), Models in Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 229–242.

[13] A. Debiasi, F. Ihirwe, P. Pierini, S. Mazzini, S. Tonetta, Model-based analysis support for dependable complex systems in CHESS, in: Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD, SciTePress, INSTICC, 2021, pp. 262–269, http://dx.doi.org/10.5220/0010269702620269.

[14] A. Cimatti, M. Dorigatti, S. Tonetta, OCRA: A tool for checking the refinement of temporal contracts, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2013, pp. 702–705, http://dx.doi.org/10.1109/ASE.2013.6693137.

[15] T. Courtney, S. Gaonkar, K. Keefe, E.W.D. Rozier, W.H. Sanders, Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models, in: 2009 IEEE/IFIP International Conference on Dependable Systems & Networks, 2009, pp. 353–358, http://dx.doi.org/10.1109/DSN.2009.5270318.

[16] B. Gallina, M.A. Javed, F.U. Muram, S. Punnekkat, A model-driven dependability analysis method for component-based architectures, in: 2012 38th Euromicro Conference on Software Engineering and Advanced Applications, 2012, pp. 233–240, http://dx.doi.org/10.1109/SEAA.2012.35.

[17] ESA Requirements, Standards Division, Space Product Assurance - Fault Tree Analysis - Adoption Notice ECSS/IEC 61025, ESA Publ. Division, 2008, URL https://ecss.nl/standards/active-standards/.

[18] ESA Requirements, Standards Division, Failure Modes, Effects (and Criticality) Analysis (FMEA/FMECA), ESA Publ. Division, 2009, URL https://ecss.nl/standards/active-standards/.

[19] M. Wallace, Modular architectural representation and analysis of fault propagation and transformation, Electron. Notes Theor. Comput. Sci. 141 (3) (2005) 53–71, Proceedings of the Second International Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA 2005).

[20] B. Gallina, E. Sefer, A. Refsdal, Towards safety risk assessment of socio-technical systems via failure logic analysis, in: ISSRE Workshops, 2014, pp. 287–292.

[21] A. Cheliyan, S. Bhattacharyya, Fuzzy fault tree analysis of oil and gas leakage in subsea production systems, J. Ocean Eng. Sci. 3 (1) (2018) 38–48, http://dx.doi.org/10.1016/j.joes.2017.11.005.

[22] S. Markulik, M. Šolc, J. Petrík, M. Balážiková, P. Blaško, J. Kliment, M. Bezák, Application of FTA analysis for calculation of the probability of the failure of the pressure leaching process, Appl. Sci. 11 (15) (2021) http://dx.doi.org/10.3390/app11156731, URL https://www.mdpi.com/2076-3417/11/15/6731.

[23] R. Ferdous, F. Khan, B. Veitch, P.R. Amyotte, Methodology for computer aided fuzzy fault tree analysis, Process Saf. Environ. Protect. 87 (4) (2009) 217–226, http://dx.doi.org/10.1016/j.psep.2009.04.004, URL https://www.sciencedirect.com/science/article/pii/S0957582009000421.

[24] D.H. Stamatis, Failure Mode and Effect Analysis: FMEA from Theory to Execution, Quality Press, 2003.

[25] J. Ostroff, S. Gerhart, D. Craigen, T. Ralston, N.G. Leveson, J. Bowen, V. Stavridou, Real-time and safety-critical systems, in: High-Integrity System Specification and Design, Springer London, London, 1999, pp. 359–528, http://dx.doi.org/10.1007/978-1-4471-3431-2_6.

[26] J.C. Knight, Safety critical systems: Challenges and directions, in: Proceedings of the 24th International Conference on Software Engineering, ICSE '02, Association for Computing Machinery, New York, NY, USA, 2002, pp. 547–550, http://dx.doi.org/10.1145/581339.581406.

[27] J. Bowen, The ethics of safety-critical systems, Commun. ACM 43 (4) (2000) 91–97, http://dx.doi.org/10.1145/332051.332078.

[28] J. An, A. Mikhaylov, K. Kim, Machine learning approach in heterogeneous group of algorithms for transport safety-critical system, Appl. Sci. 10 (8) (2020) http://dx.doi.org/10.3390/app10082670, URL https://www.mdpi.com/2076-3417/10/8/2670.

[29] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Secure Comput. 1 (1) (2004) 11–33, http://dx.doi.org/10.1109/TDSC.2004.2.

[30] F. Ciccozzi, I. Crnkovic, D. Di Ruscio, I. Malavolta, P. Pelliccione, R. Spalazzese, Model-driven engineering for mission-critical IoT systems, IEEE Softw. 34 (1) (2017) 46–53, http://dx.doi.org/10.1109/MS.2017.1.

[31] W. Dunn, Designing safety-critical computer systems, Computer 36 (11) (2003) 40–46, http://dx.doi.org/10.1109/MC.2003.1244533.

[32] F. Ihirwe, D. Di Ruscio, S. Mazzini, A. Pierantonio, Towards a modeling and analysis environment for industrial IoT systems, in: STAF Workshops, 2021, pp. 90–104, URL http://ceur-ws.org/Vol-2999/messpaper1.pdf.

[33] A. Bucchiarone, F. Ciccozzi, L. Lambers, A. Pierantonio, M. Tichy, M. Tisi, A. Wortmann, V. Zaytsev, What is the future of modeling? IEEE Softw. 38 (2) (2021) 119–127, http://dx.doi.org/10.1109/MS.2020.3041522.

[34] D. Di Ruscio, R. Eramo, A. Pierantonio, Model transformations, in: M. Bernardo, V. Cortellessa, A. Pierantonio (Eds.), Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 91–136, http://dx.doi.org/10.1007/978-3-642-30982-3_4.

[35] F. Ciccozzi, R. Spalazzese, MDE4IoT: Supporting the Internet of Things with model-driven engineering, in: C. Badica, A. El Fallah Seghrouchni, A. Beynier, D. Camacho, C. Herpson, K. Hindriks, P. Novais (Eds.), Intelligent Distributed Computing X, Springer International Publishing, Cham, 2017, pp. 67–76.

[36] N. Harrand, F. Fleurey, B. Morin, K.E. Husa, ThingML: A language and code generation framework for heterogeneous targets, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 125–135, http://dx.doi.org/10.1145/2976767.2976812.

[37] D. Conzon, M.R. Ahmmad, Rashid, X. Tao, A. Soriano, R. Nicholson, E. Ferrera, BRAIN-IoT: Model-based framework for dependable sensing and actuation in intelligent decentralized IoT systems, in: Oct 2019 4th International Conference on Computing, Com and Security, ICCCS, LINKS Foundation, 2019.

[38] F. Ihirwe, A. Indamutsa, D. Ruscio, S. Mazzini, A. Pierantonio, Cloud-based modeling in IoT domain: a survey, open challenges and opportunities, in: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C, IEEE Computer Society, Los Alamitos, CA, USA, 2021, pp. 73–82, http://dx.doi.org/10.1109/MODELS-C53483.2021.00018, URL https://doi.ieeecomputersociety.org/10.1109/MODELS-C53483.2021.00018.

[39] F. Ihirwe, D. Di Ruscio, S. Mazzini, A. Pierantonio, A domain specific modeling and analysis environment for complex IoT applications, 2021, arXiv preprint arXiv:2109.09244.

[40] D.T. Nguyen, C. Song, Z. Qian, S.V. Krishnamurthy, E.J.M. Colbert, P. McDaniel, IotSan: Fortifying the safety of IoT systems, in: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 191–203, http://dx.doi.org/10.1145/3281411.3281440.

[41] J. Parri, S. Sampietro, E. Vicario, FaultFlow: a tool supporting an MDE approach for timed failure logic analysis, in: 2021 17th European Dependable Computing Conference, EDCC, 2021, pp. 25–32, http://dx.doi.org/10.1109/EDCC53658.2021.00011.

[42] Z. Haider, B. Gallina, E.Z. Moreno, FLA2FT: Automatic generation of fault tree from ConcertoFLA results, in: 2018 3rd International Conference on System Reliability and Safety, ICSRS, 2018, pp. 176–181, http://dx.doi.org/10.1109/ICSRS.2018.8688825.

[43] L. Xing, S.V. Amari, Fault tree analysis, in: K.B. Misra (Ed.), Handbook of Performability Engineering, Springer London, London, 2008, pp. 595–620, http://dx.doi.org/10.1007/978-1-84800-131-2_38.

[44] E. Ruijters, M. Stoelinga, Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools, Comp. Sci. Rev. 15–16 (2015) 29–62, http://dx.doi.org/10.1016/j.cosrev.2015.03.001, URL https://www.sciencedirect.com/science/article/pii/S1574013715000027.

[45] X. Han, J. Zhang, A combined analysis method of FMEA and FTA for improving the safety analysis quality of safety-critical software, in: 2013 IEEE International Conference on Granular Computing, GrC, 2013, pp. 353–356, http://dx.doi.org/10.1109/GrC.2013.6740435.

[46] L. Grunske, J. Han, A comparative study into architecture-based safety evaluation methodologies using AADL's error annex and failure propagation models, in: 2008 11th IEEE High Assurance Systems Engineering Symposium, 2008, pp. 283–292, http://dx.doi.org/10.1109/HASE.2008.32.

[47] B. Gallina, S. Punnekkat, FI4FA: A formalism for incompletion, inconsistency, interference and impermanence failures' analysis, in: 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications, 2011, pp. 493–500, http://dx.doi.org/10.1109/SEAA.2011.80.

[48] D.S. Kolovos, R.F. Paige, F.A.C. Polack, The epsilon transformation language, in: A. Vallecillo, J. Gray, A. Pierantonio (Eds.), Theory and Practice of Model Transformations, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 46–60.

[49] H. Ren, X. Chen, Y. Chen, Chapter 6 - Fault tree analysis for composite structural damage, in: H. Ren, X. Chen, Y. Chen (Eds.), Reliability Based Aircraft Maintenance Optimization and Applications, in: Aerospace Engineering, Academic Press, 2017, pp. 115–131, http://dx.doi.org/10.1016/B978-0-12-812668-4.00006-X, URL https://www.sciencedirect.com/science/article/pii/B978012812668400006X.

[50] ISOGRAPH, Fault tree analysis in reliability workbench, 2022, URL https://www.isograph.com/. (Last accessed May 2022).

[51] B. Bittner, M. Bozzano, R. Cavada, A. Cimatti, M. Gario, A. Griggio, C. Mattarei, A. Micheli, G. Zampedri, The xSAP safety analysis platform, in: M. Chechik, J.-F. Raskin (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 533–539.

[52] T. Prosvirnova, AltaRica 3.0: a Model-Based approach for Safety Analyses (Theses), Ecole Polytechnique, 2014, URL https://pastel.archives-ouvertes.fr/tel-01119730.

[53] P. Feiler, J. Delange, Automated fault tree analysis from AADL models, Ada Lett. 36 (2) (2017) 39–46, http://dx.doi.org/10.1145/3092893.3092900.

[54] S. Kabir, K. Aslansefat, I. Sorokos, Y. Papadopoulos, Y. Gheraibia, A conceptual framework to incorporate complex basic events in HiP-HOPS, in: Y. Papadopoulos, K. Aslansefat, P. Katsaros, M. Bozzano (Eds.), Model-Based Safety and Assessment, Springer International Publishing, Cham, 2019, pp. 109–124.

[55] T. Chiang, R.G. Mendoza, J. Mahmood, R.F. Paige, Towards the adoption of model based system safety engineering in the automotive industry, in: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 579–587, http://dx.doi.org/10.1145/3550356.3563130.

[56] P. Varady, Z. Benyo, B. Benyo, An open architecture patient monitoring system using standard technologies, IEEE Trans. Inf. Technol. Biomed. 6 (1) (2002) 95–98, http://dx.doi.org/10.1109/4233.992168.

[57] R.K. Megalingam, D.M. Kaimal, M.V. Ramesh, Efficient patient monitoring for multiple patients using WSN, in: 2012 International Conference on Advances in Mobile Network, Communication and Its Applications, 2012, pp. 87–90, http://dx.doi.org/10.1109/MNCApps.2012.23.

[58] J. Parri, F. Patara, S. Sampietro, E. Vicario, A framework for model-driven engineering of resilient software-controlled systems, Computing 103 (2021) http://dx.doi.org/10.1007/s00607-020-00841-6.

[59] F. Mhenni, N. Nguyen, J.-Y. Choley, Automatic fault tree generation from SysML system models, in: 2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics, 2014, pp. 715–720, http://dx.doi.org/10.1109/AIM.2014.6878163.

[60] I. Silva, R. Leandro, D. Macedo, L.A. Guedes, A dependability evaluation tool for the Internet of Things, Comput. Electr. Eng. 39 (7) (2013) 2005–2018, http://dx.doi.org/10.1016/j.compeleceng.2013.04.021, URL https://www.sciencedirect.com/science/article/pii/S0045790613001171.

[61] H. Fazlollahtabar, S. Niaki, Fault tree analysis for reliability evaluation of an advanced complex manufacturing system, J. Adv. Manuf. Syst. 17 (2018) 107–118, http://dx.doi.org/10.1142/S0219686718500075.

[62] M. Chaari, W. Ecker, T. Kruse, C. Novello, B.-A. Tabacaru, Transformation of failure propagation models into fault trees for safety evaluation purposes, in: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop, DSN-W, 2016, pp. 226–229, http://dx.doi.org/10.1109/DSN-W.2016.18.

[63] K. Clegg, M. Li, D. Stamp, A. Grigg, J. McDermid, Integrating existing safety analyses into SysML, in: Y. Papadopoulos, K. Aslansefat, P. Katsaros, M. Bozzano (Eds.), Model-Based Safety and Assessment, Springer International Publishing, Cham, 2019, pp. 63–77.

[64] P. Hönig, R. Lunde, F. Holzapfel, Model based safety analysis with smartiflow, Information 8 (1) (2017) http://dx.doi.org/10.3390/info8010007, URL https://www.mdpi.com/2078-2489/8/1/7.

[65] J. Xiang, K. Yanoo, Automatic static fault tree analysis from system models, in: 2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing, 2010, pp. 241–242, http://dx.doi.org/10.1109/PRDC.2010.35.

[66] Y. Chen, Z. Zhen, H. Yu, J. Xu, Application of fault tree analysis and fuzzy neural networks to fault diagnosis in the internet of things (IoT) for aquaculture, Sensors 17 (1) (2017) http://dx.doi.org/10.3390/s17010153, URL https://www.mdpi.com/1424-8220/17/1/153.

[67] L. Xing, M. Tannous, V.M. Vokkarane, H. Wang, J. Guo, Reliability modeling of Mesh Storage Area networks for internet of things, IEEE Internet Things J. 4 (6) (2017) 2047–2057, http://dx.doi.org/10.1109/JIOT.2017.2749375.

[68] N. Yakymets, H. Jaber, A. Lanusse, Model-based system engineering for fault tree generation and analysis, in: Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD, SciTePress, INSTICC, 2013, pp. 210–214, http://dx.doi.org/10.5220/0004346902100214.

[69] A.H.d.A. Melani, G.F.M.d. Souza, Obtaining fault trees through sysml diagrams: A MBSE approach for reliability analysis, in: 2020 Annual Reliability and Maintainability Symposium, RAMS, 2020, pp. 1–5, http://dx.doi.org/10.1109/RAMS48030.2020.9153658.

[70] K. Clegg, M. Li, D. Stamp, A. Grigg, J. McDermid, A sysml profile for fault trees—Linking safety models to system design, in: A. Romanovsky, E. Troubitsyna, F. Bitsch (Eds.), Computer Safety, Reliability, and Security, Springer International Publishing, Cham, 2019, pp. 85–93.

[71] Y. Papadopoulos, J.A. McDermid, Hierarchically performed hazard origin and propagation studies, in: M. Felici, K. Kanoun (Eds.), Computer Safety, Reliability and Security, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 139–152.

[72] S. Katsavounis, N. Patsianis, E. Konstantinidis, P. Botsaris, Reliability analysis on crucial subsystems of a wind turbine through FTA approach, 2014, http://dx.doi.org/10.13140/2.1.2524.3849.

[73] D. Clerissi, J.D. Rocco, D.D. Ruscio, C.D. Sipio, F. Ihirwe, L. Mariani, D. Micucci, M.T. Rossi, R. Rubei, Supporting early-safety analysis of IoT systems by exploiting testing techniques, 2023, arXiv:2309.02985.