Computing (2021) 103:589–612
https://doi.org/10.1007/s00607-020-00841-6

SPECIAL ISSUE ARTICLE

A framework for Model-Driven Engineering of resilient software-controlled systems

Jacopo Parri[1] · **Fulvio Patara**[1] · **Samuele Sampietro**[1] · **Enrico Vicario**[1]

Received: 12 February 2020 / Accepted: 17 August 2020 / Published online: 2 September 2020

Abstract
Emergent paradigms of Industry 4.0 and Industrial Internet of Things expect cyber-physical systems to reliably provide services overcoming disruptions in operative conditions and adapting to changes in architectural and functional requirements. In this paper, we describe a hardware/software framework supporting operation and maintenance of software-controlled systems enhancing resilience by promoting a Model-Driven Engineering (MDE) process to automatically derive structural configurations and failure models from reliability artifacts. Specifically, a *reflective* architecture developed around *digital twins* enables representation and control of system *Configuration Items* properly derived from SysML Block Definition Diagrams, providing support for variation. Besides, a plurality of distributed analytic agents for qualitative evaluation over executable failure models empowers the system with run-time self-assessment and dynamic adaptation capabilities. We describe the framework architecture outlining roles and responsibilities in a System of Systems perspective, providing salient design traits about digital twins and data analytic agents for failure propagation modeling and analysis. We discuss a prototype implementation following the MDE approach, highlighting self-recovery and self-adaptation properties on a real cyber-physical system for vehicle access control to Limited Traffic Zones.

Keywords Resilience · Software-controlled system of systems · Model-Driven Engineering · Reflection architectural pattern · Digital twins · Fault trees

✉ Fulvio Patara
fulvio.patara@unifi.it
https://stlab.dinfo.unifi.it/patara/

Jacopo Parri
jacopo.parri@unifi.it

Samuele Sampietro
samuele.sampietro@unifi.it

Enrico Vicario
enrico.vicario@unifi.it
https://stlab.dinfo.unifi.it/vicario/

[1] Department of Information Engineering, University of Florence, Florence, Italy

Mathematics Subject Classification 68M15 · 68T05 · 68T42

1 Introduction

1.1 Motivation

In the agenda of Industry 4.0 (I4.0), resilience of cyber-physical systems is expected to be supported by monitoring and control capabilities provided by software components exposing an agile interface for integration and processing of data carrying information at different levels of granularity, according to various pillars, notably Industrial Internet of Things (IIoT), big data and analytics, simulation, horizontal and vertical integration, and cloud computing [42,46]. This gives raise to a class of software-controlled systems that can afford functional, structural, and behavioural complexity while still maintaining commitment for high levels of reliability [1,34,45]. Effective exploitation of this potential largely depends on architectural choices that shape integration between physical, hardware, software, and human operators, and by development practices that preserve reliability while increasing agility and complexity.

1.2 Contribution

In this paper, we describe a hardware/software framework developed in an industrial research project [33], named JARVIS (Just-in-time ARtificial intelligence for the eValuation of Industrial Signals), which supports system reliability and recoverability through runtime monitoring, remote actuation capabilities, and predictive maintenance processes by leveraging a software architecture that facilitates development and verification of the integration of physical IoT devices, enterprise scale software agents, data analytics, and human operators.

At the core of its architecture, JARVIS supports creation of a Knowledge Base providing an executable representation of the system Configuration that mirrors each Item through a *digital twin* [48] implemented as a software object exposing methods for Item monitoring and control operations. Digital twins are organized according to the Reflection architectural pattern [6], separating a layer of concrete objects that represent state and identity of individual Items from an abstract layer that represents Item types and composition topology. This organization facilitates construction of digital twins and promotes maintainability [44] by accommodating changes over time or within a product line and by making the representation verifiable through concrete configurations.

Besides, a variety of analytic agents deployed incrementally and with loose coupling in the shape of microservices, empowers the framework with intelligence [43,47] for complex decision-making by interacting with digital twins based on extraction and processing of IoT data streams. In particular, analytic agents performing Fault Tree Analysis (FTA) over executable failure models provide runtime self-assessment and dynamic adaptation capabilities.

Both system configurations and failure models can be conveniently derived following a Model-Driven Engineering (MDE) process based on semi-formal artifacts from the practice of Reliability Engineering (i.e., SysML diagrams and Fault Trees) widespread in industrial documental standards and formal methods. This reduces the gap between reliability engineers and software architects, allowing a more direct cast of human expertise into the cyber-side logic of the system. In so doing, JARVIS supports system resilience by promoting semi-formal specification of structural aspects, functional requirements, and behavioural characteristics of subsystems in a System of Systems perspective.

According to this, the main contributions of this paper are: (i) a *framework* supporting development of resilient cyber-physical systems through connection of digital twins representing Configuration Items with a suite of diagnostic, predictive, and prescriptive analytics based on reliability engineering artifacts; (ii) an implementation of *digital twins* based on the architectural pattern of reflection that facilitates creation of digital twins, adaptation to configuration changes, and verification of coherence with the actual state of the system; (iii) the description of a *concrete cyber-physical system* that demonstrates the results of the JARVIS project in a Smart City context and illustrates applicability and positive consequences of the framework in an MDE approach.

The rest of the paper is organized as follows: Sect. 2 summarizes related works; Sect. 3 reports a general description of the framework, outlining its major requirements and describing its architecture, in the perspective of an IoT architectural stack; Sect. 4 illustrates salient traits of software design that permit digital twins to comprise a Knowledge Base and provide a high-level interface for remote commands actuation; Sect. 5 describes a data analytic agent that performs failure propagation analysis over executable representations of Fault Trees connected to digital twins of Configuration Items; Sect. 6 outlines an MDE approach that exploits a SysML BDD and a suite of Fault Trees to drive creation of objects representing the configuration of a system and the dynamic of its failure logic; Sect. 7 discusses a prototype instance of the proposed framework achieving self- recovery and adaptation in the application scenario of a real cyber-physical system for access control to Limited Traffic Zones (LTZ). Conclusions are drawn in Sect. 8.

The paper significantly extends preliminary results reported in [32,33] in three orthogonal directions: (i) from an *architectural perspective*, the JARVIS architecture has been enhanced with reference to the four layers of an IoT architectural stack, providing a complete description of structure of system/subsystems design, functional requirements, behavioral and dataflow characteristics, and contribution to system resilience by each subsystem; (ii) from a *functional perspective*, the Knowledge Base, designed over the digital twins abstraction, has been enriched with IoT remote commands representations, conferring to the framework dynamic actuation capabilities and enabling agile reconfiguration of physical devices; (iii) from a *methodological perspective*, in this paper we promote an MDE process to translate specifications of the system structure and failure modes combinations into executable representations of digital twins and Fault Trees, and we highlight the applicability of the methodology in a real operative scenario.

2 Related work

Technological and cultural changes promoted by IIoT and I4.0 pillars are encouraging the adoption of advanced software-controlled systems with a continuously increasing complexity, in terms of hardware/software components, implemented functionalities, and offered services. In contexts where malfunctions and services disruptions have societal, economic, or legal impact, these complex IT/OT [23] systems, based on many interconnected, autonomous and smart devices, must inevitably guarantee an acceptable level of reliability. In these scenarios, the cyber side may support system resilience, exploiting monitoring and control capabilities to provide fault-tolerance, rejuvenation, and self-adaptation mechanisms.

Software digital twins representations provide a key abstraction, providing an agile interface on hardware components, capturing operational behaviours of physical assets and processes, also providing refined and interpreted data, to enable diagnoses, predictive maintenance tasks, process plannings, process optimizations, virtual prototyping, and simulation. In [29], the definition, the role and the architectural aspects of digital twins in IIoT systems are elaborated. In [41], a hierarchical digital twin model framework is proposed as a way to handle advancement of product life cycle and to support 3D simulation of components for complex systems. In [20], an architectural framework based on an elementary meta model for simulation analyses and online planning is presented, primarily focusing on the event-driven information flow, extracted by a cognitive system monitoring physical components, rather than on the design and configuration of adopted digital twins.

Digital twins must be reliable, ensuring a consistent alignment with the characteristics of physical counterparts over time, facilitating the system verifiability during operation and maintenance stages. This is further exacerbated in cyber-physical systems where underlying software architectures must guarantee that the cyber side properly controls and orchestrates the physical side.

Reflective architectures can be considered as an effective solution by enabling dynamic strategies for changing structural aspect and runtime behaviour in response to unpredictable circumstances such as system faults or evolving requirements. Indeed, through the adoption of the *Reflection* Pattern [39], the architecture achieves self-awareness, adaptability and reusability capabilities by representing information about selected system properties; in particular, its meta level can be interpreted as a knowledge specification layer for the base level which includes the application logic. In [11], a reference architecture exploiting digital twins and *Reflection* principles to enhance flexibility and customization in trust-based digital ecosystems is proposed. In [4], a reference model for reflection and key reflection properties with their variation points are promoted as prerequisites for self-adaptive systems to support runtime reconfiguration of component-based architectures. In [16], a set of short-term and long-term research challenges about the adoption of model-driven approaches for assurance of self-adaptive software systems at runtime is identified, while in [8], authors propose an architectural approach for resilience evaluation of self-adaptive systems before deployment to increase the predictability of adaptation mechanisms with respect to dependability requirements. In [7], a self-adaptation paradigm including quantitative verification and model checking techniques at runtime is addressed to realize dependable self-adaptive software for detecting and predicting requirement violations and leading to adaptive maintenance. In [27], an automation agent-based architecture performing software reasoning processes over an ontological representation of low-level functionalities is presented, so as to provide online reconfiguration abilities to industrial manufacturing systems.

Model-Driven Engineering (MDE) approaches may support architectural solutions for resilience, providing a way to connect different engineering processes through abstractions and artifacts that meet the needs for industrial acceptance and open the way to analytic methods, as reported in the empirical study described in [30]. The adoption of domain-specific modeling languages, transformation engines and generators, reverse engineering, and model checking techniques offers a means for achieving *correct-by-construction* executable software configurations, limiting complex and error-prone tasks [38]. In [40], a methodology to create digital twins representations using *AutomationML* meta language is described, where information about physical systems can be captured by the definition of *Computer Aided Engineering eXchange* meta models, for exchanging data between heterogeneous systems through an IoT middleware. In [3], an approach to integrate safety analysis within an MDE process is proposed, based on the automatic derivation of component-level and system-level Fault Trees (FT) from SysML models annotated with relevant safety information, with the aim of maintaining the traceability between system and analysis models. The problem of co-evolution of architectural and quality evaluation models is developed further in [22], where a set of transformation rules is designed to support the independent evolution and synchronization of inter-related models, reducing the engagement of developers expertise. In [5], a process exploiting UML specifications of software architectures for the generation of executable models for risk analysis is illustrated.

To the best of our knowledge, this is the first paper proposing a framework for MDE based on a *distributed and executable reflective architecture* leveraging on *digital twins*, so as to support system resilience through runtime monitoring, remote actuation capabilities, self-adaptability and reusability of the cyber-configurations: only a few works in literature describe an architecture including some of these concepts. In [26], a 5-level architecture is proposed, in a purely theoretical perspective, as a guideline for implementation of resilient, intelligent, and self-adaptable I4.0 manufacturing systems. In [49], a distributed communication platform for digital twins named *uDiT*, acting as a simulation environment and 3D visualizer for dependable cyber-physical systems, is proposed. In [2], a reference model for digital twin architectures is described, exploiting Bayesian belief networks and fuzzy logic to design smart interaction controllers for cloud-based systems, focusing on operational communication modes, while neglecting reliability and resilience properties and reusable configurations.

3 HW/SW architecture for resilience

JARVIS is a hardware/software framework developed along a V-model process [18] to evolve as a modular system within an incremental and iterative approach, so as to promote high levels of data ingestion, fault-tolerance, portability, and adaptability. The functional requirements of the framework, the system/subsystem design with allocated requirements, and the behavioral and dataflow characteristics are explained with the aim of highlighting their specific contributions to the system resilience, following an organization of contents in the style of the *MIL-STD-498 Software Development and Documentation* standard [35], with specific reference to the *System Requirements Specification (SRS)* and the *System/Subsystem Design Description (SSDD)* documents.

3.1 System Requirements Specification

Main system requirements and their consequent structural choices to improve the system resilience include: (RQ1) managing a plethora of *IoT devices* equipped with sensors and actuators to monitor the system status and to perform self-adaptation actions to handle disruptions through the execution of remote control commands; (RQ2) ingesting *big data* (characterized by Volume, Variety, Velocity, Value, and Veracity [13]) generated from IoT devices into a high-capacity data lake for more in-depth analysis to detect anomalies, predict failures, and schedule recovery interventions; (RQ3) providing *executable digital replicas* of physical assets, that update and change as the physical counterparts change, supporting virtual troubleshooting by promoting: the exploitation of simulation modeling and analysis to predict system failures and deviations from the expected behaviour; the implementation of different reconfiguration strategies in response to detected/forecast disruptions by working directly on its software representation; (RQ4) providing a swarm of *enterprise scale software agents* and *data analytics* with different business focuses (e.g., operational monitoring, anomaly detection, reactive/predictive/proactive maintenance), developed as loosely-coupled independently deployable services by different parties following the principles of a Microservice-Oriented Architecture [15]; (RQ5) *integrating data* residing in different sources to provide a unified view, and *integrating services* offered by different systems to enable cross-functional and cross-organizational collaborations; (RQ6) offering a presentation layer in the shape of *intelligent operational dashboards* to provide real-time access to the volumes of data collected from the infrastructure to expert end-users (e.g., help desk operators, domain experts, maintenance and system technicians) for inspections and interventions; (RQ7) providing *digital assistants* [14] to support human experts during configuration, monitoring, and control operations, by promoting inversion-of-responsibility mechanisms that improve the capacity for the system to make decisions autonomously and to perform some decision-making tasks (e.g., system reconfiguration for dynamic self-adaptation to preserve system goals during execution) without human intervention.

3.2 System/Subsystem Design Description

JARVIS is developed around a System of Systems (SoS) architecture [28] (see Fig. 1), designed with reference to the four layers of an IoT architectural stack [25], i.e. *perception layer*, *network/transport layer*, *processing layer*, and *application layer*.

The *Field System (FS)* plays the role of *IoT perception layer*, acquiring and generating data streams from a variety of heterogeneous IoT devices, logging information

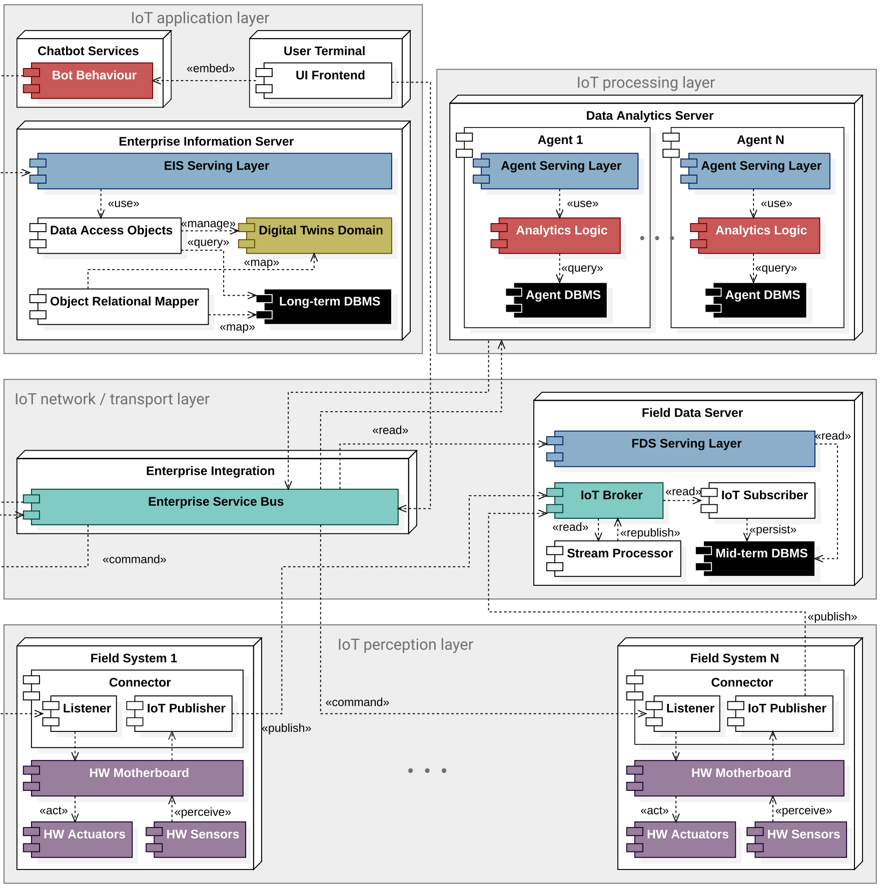

Fig. 1 UML diagram of the JARVIS architecture, where each subsystem belongs to a specific layer of an IoT architectural stack: purple components identify hardware items of monitored *Field Systems*; green components describe data carriers and orchestrators; red components represent behavioural algorithms implemented inside data analytics and chatbot agents; black components identify dedicated *DBMSs*; ocher components depict the digital twins core; finally, blue components illustrate web services exposed by server subsystems. (color figure online)

about operational status, and sampling environmental properties (i.e., *RQ1*). Each *FS* represents a physical device composed of hardware components (e.g., motherboard, sensors, actuators) and software controllers (e.g., embedded firmware): as an example in the industrial context, any interconnected machinery with IoT capabilities can be seen as a *FS*.

The *Field Data Server (FDS)* performs its functions in the *IoT network/transport layer*, storing raw and semi-structured data coming from the *Field System* in a high-capacity mid-term database, also applying analytic processes to filter, fix, and synthesise data (i.e., *RQ2*). In particular, the *IoT Broker* component acts as an asynchronous Message-Oriented Middleware (MOM) [12] based on the Publish-Subscribe pattern [24], and operates ingestion of IoT data streams.

The *Enterprise Integration (EI)* is responsible, in the *IoT network/transport layer*, for subsystems interoperability, orchestration of services, dynamic management of dependencies, authorizations and security of access to field devices (i.e., *RQ5*). The core component is represented by the *Enterprise Service Bus (ESB)* [10], which acts as a high-level broker for each message exchange (except for data ingestion) among subsystems, and intermediates command executions over physical *Field System* instances, implementing dedicated client processes. In so doing, the *ESB* guarantees high decoupling and push communications, also exploiting some major microservices patterns [31] to enhance availability and reliability.

The *Enterprise Information Server (EIS)* plays the role of *IoT application layer*—except for presentation—maintaining information on the status of monitored *Field System*s, adopting the abstraction of *digital twins* to keep a long-term consistent and context-unaware Knowledge Base about system configurations and available remote control commands, and interpreting and refining the mid-term *Field Data Server* raw data into a high-level semantic (i.e., *RQ3*). The *EIS* operates as a full-fledged passive module which defines a complex persistence logic with useful querying interfaces.

The *Data Analytics Server (DAS)* acts as *IoT processing layer* and is composed by a plurality of software agents executing dynamic context interpretation, data extrapolation, and data synthesis, and enabling descriptive (e.g., diagnoses on the actual status of monitored Configuration Items), predictive (e.g., forecasting of runtime failure events), and prescriptive analysis (e.g., decision-making processes, reacting or recovering in response to occurred system failures), through machine learning mechanisms and stochastic modeling techniques. Consequently, analytic modules can be deployed as a collection of loosely-coupled microservices to accomplish specific tasks, strictly dependent on the application context and also related to the quality of raw data stored into the *Field Data Server* (i.e., *RQ4*).

The *User Terminal (UT)* is responsible for exposing a decoupled presentation tier to the end-users in the *IoT application layer*, and can be concretely realized as a web application client which consumes the exposed services of other subsystems (i.e., *RQ6*).

The *Chatbot Services (CS)* expose alternative User Interfaces (UIs) in the *IoT application layer* and implement the internal logic of real-time messaging assistant, enabling both push and pull duplex communications among human operators and physical devices, and promoting machine-to-human and machine-to-machine interactions that can support in monitoring stage, exploiting instant messaging paradigms and remote command triggering (i.e., *RQ7*). In order to facilitate timely human reactions, the *CS* could be accompanied by smart devices able to receive push notifications.

The proposed framework implements and interprets all the responsibilities of a complete IoT architectural stack, supporting—in a holistic vision—system resilience at each tier: the *perception layer* enables monitoring and self-adaptation capabilities over *Field System* instances using smart motherboards equipped with sensors and actuators; the *network/transport layer* promotes a scalable message-oriented communication platform supporting the adaptation to unexpected conditions through the integration of *Field Data Server* and *Enterprise Integration* subsystems; the *processing layer* provides intelligence by a variety of *Data Analytics Server* agents capable

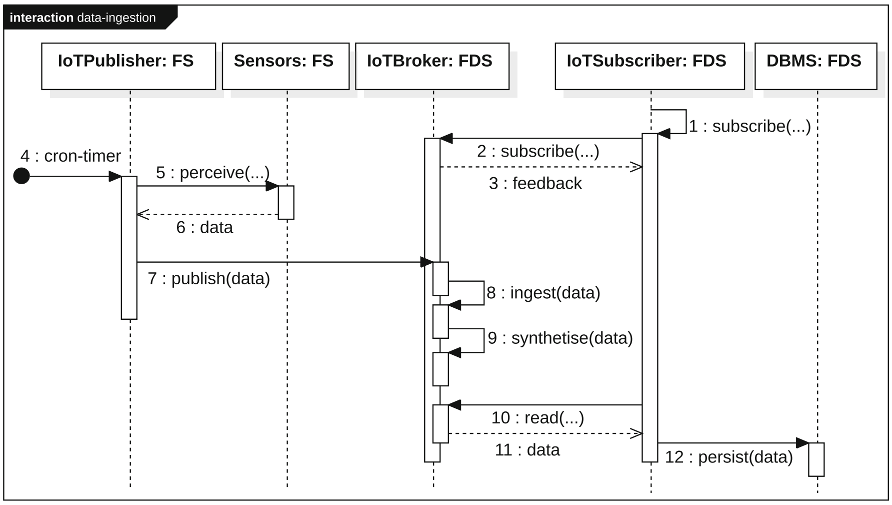

Fig. 2 The *data stream ingestion* process from *Sensors* of a *Field System (FS)* to the DBMS of the *Field Data Server (FDS)*

of detecting anomalies, predicting failure events, and applying adaptive just-in-time strategies starting from the analysis of raw and refined data; finally, the *application layer* contributes to system resilience in two ways: while the *Enterprise Information Server* exploits the digital twin representation to support real-time monitoring of physical components, *User Terminal* and *Chatbot Services* keep end-users in the loop through push notifications with the dual aim of providing up-to-date information about the system status and promoting timely manual and automated reconfigurations.

Note that, in the perspective of batch and stream processing of massive quantities of data, the system/subsystem design gives rise to a so-called Lambda architecture, where the *Enterprise Information Server* implements the *batch layer*, the *Field Data Server* acts as the *speed layer*, and together with the *Enterprise Information Server* and the *Data Analytics Server*, represents the *serving layer* [33].

Behavioral and dataflow characteristics of the framework and collaborations at interfaces are illustrated here by UML sequence diagrams in Figs. 2, 3, 4. The *data stream ingestion* process, started by monitored *Field System* sensors and terminated by DBMS storage in the *Field Data Server*, is shown in Fig. 2: on the one side, the *IoTPublisher* module of the *Field System* periodically perceives data and publishes them on the *IoTBroker* of the *Field Data Server* for big data ingestion and filtering operations; on the other side, the *IoTSubscriber* module of the *Field Data Server* periodically reads synthesised data from the *IoTBroker* and persists them into a mid-term *DBMS*.

Besides, the process of *failure detection*, performed by a dedicated *Data Analytics Server* agent which periodically consumes the REST API exposed by the *Field Data Server* through its *ServingLayer* module to read raw data stored in its *DBMS*, is depicted in Fig. 3.

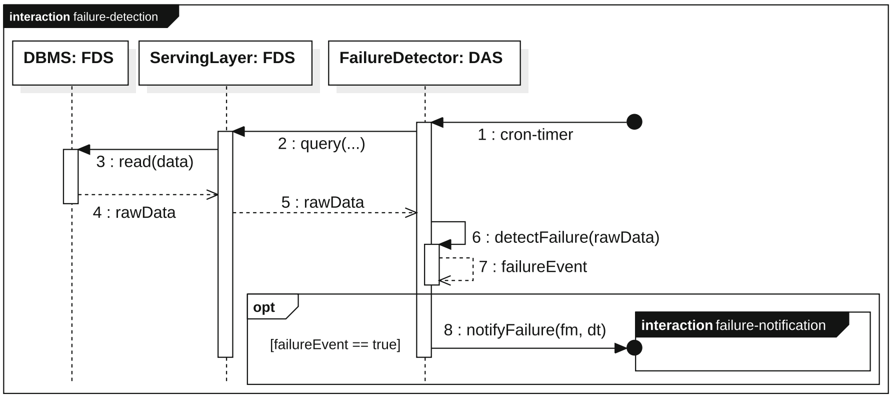

Fig. 3 The process of *failure detection* performed by a *FailureDetector* agent deployed inside the *Data Analytics Server (DAS)*. Detected failures activate the process of *failure notification* shown in Fig. 4

Finally, the process of *failure notification*, performed by the *Enterprise Integration* and activated by the *FailureDetector* agent in reaction to failures occurred in some monitored physical devices, is displayed in Fig. 4: (i) the *Enterprise Information Server*, which acts as a mediator between the agent and the other involved subsystems, updates its long-term Knowledge Base, if needed; (ii) the *Chatbot Services* support end-users in implementing self-recovery policies by suggesting available remote control commands (more details are reported in Sect. 4.2); and, finally, (iii), a dedicated *Data Analytics Server* for hazard analysis (i.e., *FaultTreeAnalyzer*) evaluates the impact of failure propagation on the overall system, restarting the process of failure notification in the case of new inferred failures (as further developed in Sect. 5).

4 A reflective Knowledge Base for online monitoring and reaction

In the JARVIS framework, subsystems cooperate in monitoring and controlling interconnected physical devices with IoT capabilities. To this end, the *Enterprise Information Server (EIS)* acts as a Knowledge Base by: (i) providing a digital representation about internal status and structural relationships of monitored components in different system configurations (Sect. 4.1); (ii) exposing remote control commands for each component, that can be invoked to enact self-adaptation mechanisms on reaction to human inputs or autonomous triggers raised by data analytic agents (Sect. 4.2).

4.1 Digital twins for adaptable virtualisation of Configuration Items

The *EIS* supports resilience and adaptability by maintaining a living digital representation of the system configurations and by monitoring macroscopic events and failure modes through a *reflective* Knowledge Base populated by *digital twins* of integrated physical components, that update and change as their physical counter-

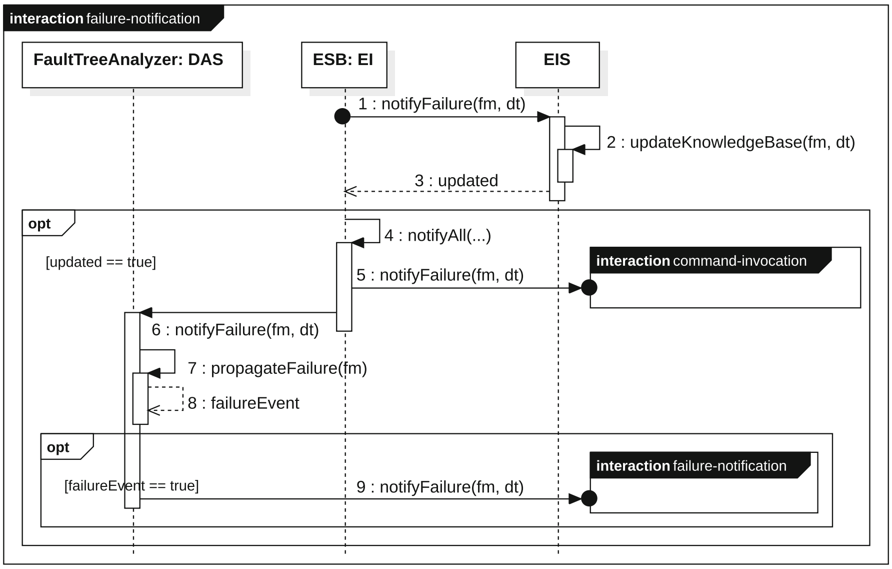
interaction failure-notification
FaultTreeAnalyzer: DAS
ESB: EI
EIS
1 : notifyFailure(fm, dt)
2 : updateKnowledgeBase(fm, dt)
3 : updated
opt
[updated == true]
4 : notifyAll(...)
interaction command-invocation
5 : notifyFailure(fm, dt)
6 : notifyFailure(fm, dt)
7 : propagateFailure(fm)
8 : failureEvent
opt
[failureEvent == true]
interaction failure-notification
9 : notifyFailure(fm, dt)

Fig. 4 The process of *failure notification* implemented by the *Enterprise Integration (EI)* in reaction to some failure detection, triggering the process of *command invocation* shown in Fig. 7 and restarting the *failure notification* procedure in the case of new discovered failures

parts change. Specifically, this develops around two modeling patterns. The *Reflection* architectural pattern [39] provides a mechanism to dynamically adapt, at runtime, the structure and behaviour of modeled digital twins, by splitting the domain logic in two parts: the *meta level* captures the types of physical components and their interconnections; the *base level* identifies concrete instances of physical components and their interfaces in the actual configuration of the system. From an architectural point-of-view, this pattern provides abstraction on how systems with a wide range of *variation points* can be efficiently configured and maintained at runtime, providing self-representation capabilities and moving configuration complexity from classes to objects. Besides, the *Composite* design pattern [21] represents the hierarchical compositions of *Field System* instances, in both the meta and base levels of the Reflection pattern. The combination of the two patterns enables the framework to evolve so as to cope with different *primary*/*alternative* hardware configurations and reliably adapt to changes in operation conditions by replacing the actual *running* compositional structure of some digital replica on the basis of switching policies and quantitative cost models implemented by specific decisional *Data Analytics Server* agents.

Figure 5 shows the *reflective* and *composite* nature of digital twins composing the Knowledge Base. At the base level, a *DigitalTwin* instance represents a *concrete* Configuration Item (e.g., a specific power supply unit with a *serial number* and a *version*), whose hierarchical structure enables it to become either an elementary *BasicComponent* or a *DigitalSystem* made by multiple digital twins. Each *DigitalTwin* is decorated with a register of *MacroscopicEvent* records classified over different

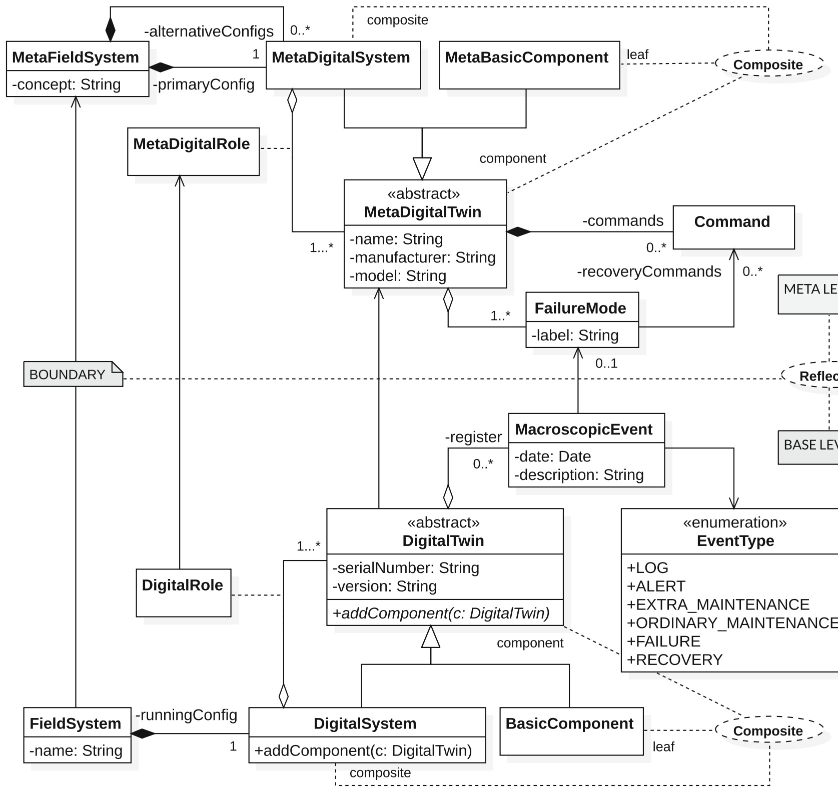

Fig. 5 UML class diagram of the Knowledge Base of the *Enterprise Information Server*, showing the *reflective* and *composite* nature of digital twins

EventTypes (e.g., *FAILURE*) and queried by analytic agents to get the status of digital twins for supporting system maintainability and reliability. On the other hand, at the meta level, an instance of *MetaDigitalTwin* represents a *type* of Configuration Items (e.g., a type of supply units with a *name*, a *manufacturer*, and a *model code*), with the same composite structure of digital twins so as to connect concrete items to types through the *MetaDigitalRole* association class, which supports reusable types among different instances of *MetaDigitalSystem* (e.g., a type of supply unit can accomplish both *main* and *spare* roles). Each *MetaDigitalTwin* is empowered with a set of remote control *Commands* which act as high-level interfaces for physical actuators promoting *recovery* interventions by chatbots and human operators on specific *FailureModes*.

4.2 IoT remote commands for agile reconfiguration of physical devices

In order to enable the framework with self-adaptation and self-recovery capabilities, the Knowledge Base of the *EIS* is enriched with a detailed representation of IoT *control commands*, distinguishing among: (i) high-level commands, intended as conceptual-

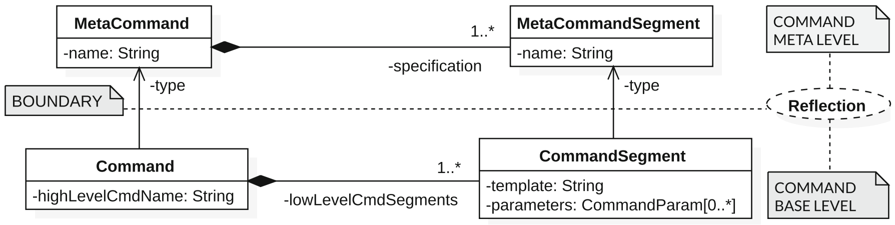

Fig. 6 UML class diagram focusing on the IoT control commands meta-model as part of the Knowledge Base of the *Enterprise Information Server*: the *Command* class reported here at the *command* base level corresponds to the same class depicted in Fig. 5 at the *digital twins* meta level, enabling each *MetaDigitalTwin* instance (and so any *DigitalTwin* with a counterpart in the physical world) to be controlled by one or more commands

izations about actuation and monitoring services offered by *Field System* instances; and, (ii) low-level commands, intended as executable operations written according to the native syntax of physical devices.

Control commands are further abstracted through an additional level of *Reflection* (see Fig. 6) to provide a common general representation applicable to a variety of physical devices: a second meta-tier provides a specification of supported remote *high-level* command types (e.g., a HTTP REST service, a bash script, a File Transfer Protocol connection) as expressed by instances of the *MetaCommand* class, while an extra base-tier enables the configuration of concrete *high-level* commands in terms of instances of the *Command* class. This reflective model empowers the *EIS* for transcoding *high-level* command invocations, generated by end-users or autonomous analytic agents, into *low-level* commands defined starting from a set of segments (i.e., *CommandSegment*), each containing a validable *template* definition and a collection of input *parameters* resolved at runtime.

As an example, in the scenario of a RESET command sent to a physical system in form of a JSON HTTP POST request to a REST endpoint URI, the HTTP represents a generic *high-level MetaCommand* instance, fully described by three *MetaCommandSegment* objects termed HTTP-verb, HTTP-body, and HTTP-URI, respectively. At the base level, a RESET *Command* of type HTTP is then instantiated, providing a real concretization for the HTTP-verb (i.e., POST), the HTTP-body (i.e, the JSON content), and the HTTP-URI (i.e., the REST endpoint URI, eventually in the form of a parameterized template) in terms of *CommandSegment* objects.

For a better understanding of the role of control commands, Fig. 7 reports the UML sequence diagram showing the interactions in a scenario of command invocation in response to a failure detection: first, the event is notified by a *Chatbot Services* agent to some engaged end-users (in so realizing a kind of machine-to-human inversion of responsibility process), which can timely react querying the Knowledge Base of the *Enterprise Information Server* to obtain the current status of the failed device and a list of available remote commands; then, in accordance with human choice, the command execution process performed by the *Enterprise Integration* starts with a *high-level* command transcoding operation and ends with the effective remote invocation of a *low-level* command over the *Field System*. To this end, command templates are transposed at invocation time by the ESB within the *Enterprise Integration* subsystem in

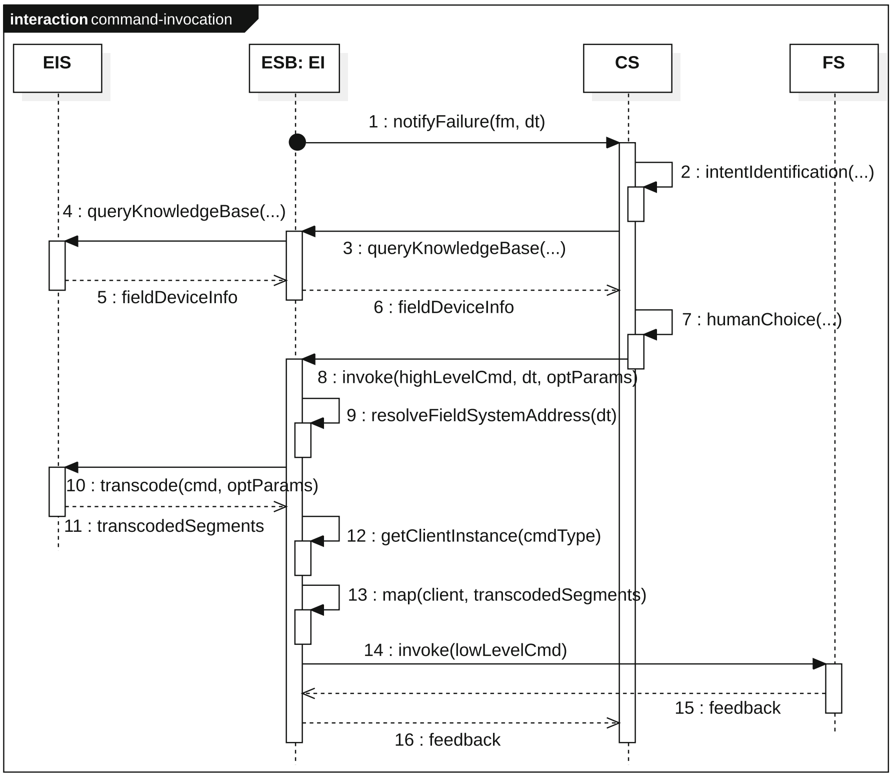
interaction command-invocation
EIS
ESB: EI
CS
FS
1 : notifyFailure(fm, dt)
2 : intentIdentification(...)
4 : queryKnowledgeBase(...)
3 : queryKnowledgeBase(...)
5 : fieldDeviceInfo
6 : fieldDeviceInfo
7 : humanChoice(...)
8 : invoke(highLevelCmd, dt, optParams)
9 : resolveFieldSystemAddress(dt)
10 : transcode(cmd, optParams)
11 : transcodedSegments
12 : getClientInstance(cmdType)
13 : map(client, transcodedSegments)
14 : invoke(lowLevelCmd)
15 : feedback
16 : feedback

Fig. 7 The process of *command invocation* performed by the *Enterprise Integration (EI)*, which is responsible for: (i) the address resolution; (ii) the mapping of transcoded segments in the low-level syntax; (iii) the client identification and instantiation; and, finally, (iv) the concrete remote invocation of low-level commands on the monitored *Field System (FS)*

a client-specific implementation constrained to the *MetaCommand*, mapping, orchestrating, and combining transcoded segments, as prescribed by the *low-level* command specification.

5 Fault Tree Analysis for vulnerability and resilience assessment

The *Data Analytics Server* is responsible for supporting operation and maintenance processes exploiting the Knowledge Base maintained by the *Enterprise Information Server* as well as raw data generated by the *Field Data Server*. In so doing, the *Data Analytics Server* contributes to system resilience by facilitating development and integration of intelligent data agents able to detect failure events occurred in accordance with a set of failure modes. This enables reactive, predictive, and proactive fault management approaches over system failures [37] and risk analysis and assessment techniques.

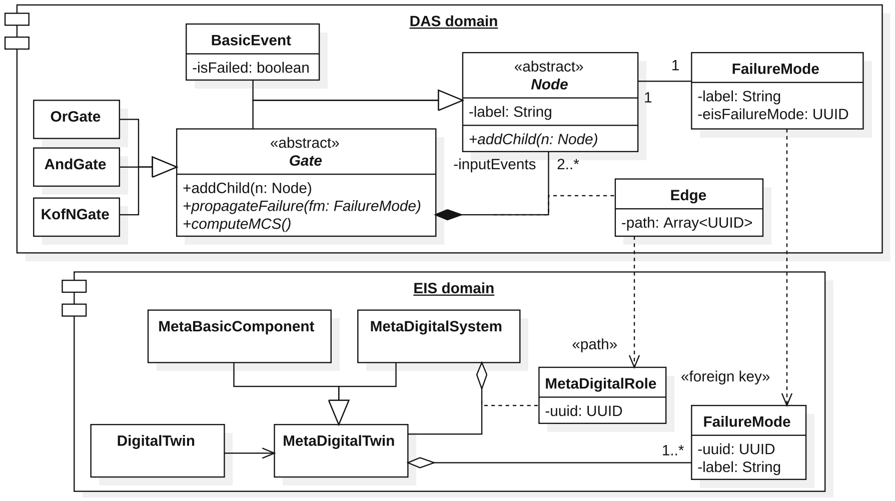

Fig. 8 UML class diagram of the domain logic of the *FaultTreeAnalyzer* agent, showing the modeling of Fault Tree digital representations in the *Data Analytics Server (DAS)* domain respect to its implicit relationships in the *Enterprise Information Server (EIS)* domain

By design, the *Data Analytics Server* provides an agent named *FaultTreeAnalyzer* which reacts to detected failure events performing Fault Tree Analysis (FTA) [17] on available failure models typically expressed using the *Fault Tree (FT)* formalism and built by reliability experts on top of some system's configurations. The domain logic of the *FaultTreeAnalyzer* agent is reported in Fig. 8 and mainly focuses on the design of the compositional structure of FTs, expressed in terms of *BasicEvent*s and logical *Gate*s (e.g., OR, AND, K-of-N), and its mapping with related Configuration Items. In this perspective, each *Node* has a structural dependency on a *FailureMode*, whose definition is directly derived from the Knowledge Base of the *Enterprise Information Server*, while each *Edge* decorates the association between *Gate* and *Node* so as to identify at runtime the corresponding concrete *DigitalTwin* starting from a unique path of *MetaDigitalRoles* in the compositional hierarchy of the *Enterprise Information Server* meta-level. In so doing, the FT is built on top of a *meta* system rather than on its *base* concretisations: this choice properly reflects the causal relationships among failure modes (possibly comprising a family of devices belonging to the same hardware/software specification), enabling qualitative and quantitative analysis starting from a collection of *MacroscopicEvent* of type *FAILURE* generated by some instances of *DigitalTwin*.

In order to infer latent high-level failures (not directly observable through monitoring techniques or data-driven diagnosis agents), to identify which combinations of component failures lead to system failures, and to plan for system self-recovery, the *FaultTreeAnalyzer* agent supports, natively, a mechanism of failure propagation and Minimal Cut Sets (MCS) determination provided by the *propagateFailure* and *computeMCS* methods, implementing recursive bottom-up and top-down resolutions of the structure according to the boolean semantics of supported logical gates, respectively.

Finally, note that the domain logic of the *FaultTreeAnalyzer* agent is intentionally designed to be general-purpose, opening the way to further extensions for quantitative analysis techniques, including a variety of stochastic methods to estimate system reliability and availability in terms of failure probabilities, in so driving the implementation of other analytics agents in support of self-adaptability and self-recoverability processes (as concretely shown in the case study discussed in Sect. 7.2).

6 A Model-Driven Engineering process for resilient systems

In designing and shaping complex systems, reliability of physical installations must be assured and, *a fortiori*, correctness of software implementations and configurations of digital representations must be guaranteed, so as to obtain functional compliance while satisfying Service Level Agreements.

The JARVIS framework promotes a *correct-by-construction* strategy, exploiting artifacts of the industrial practice to drive the necessary configuration of executable software instances under a Model-Driven Engineering (MDE) approach, rather than a *construct-by-correction* strategy where defects and errors are discovered following a *code-and-fix* style [38]. In particular, structural artifacts may capture hierarchical system compositions and dependencies, while reliability artifacts may capture reliability requirements and system failure logic, thus implying the definition of a set of fully-disciplined configuration guidelines and the specification of *ad hoc* data contracts for any serving layer. In so doing, guidelines drive domain experts and system engineers in a *step-by-step* manual configuration of the cyber side, while data contracts drive automated services integration and adaptation to different data inputs generated by third-parties tools and applications.

The proposed MDE process has been addressed from two orthogonal perspectives. On the one hand, a specification of the hierarchical composition of Configuration Items through the adoption of SysML Block Definition Diagram (BDD) artifacts [19] can be directly translated into a Knowledge Base of software digital twins, following the meta-model depicted in Fig. 5. In principle this translation can be fully automated with relatively minor effort: each block of the BDD results into a *MetaDigitalTwin* instance at runtime, so that basic and composite blocks are implemented as *meta-objects* of type *MetaBasicComponent* and *MetaDigitalSystem*, respectively, while each association results in a *MetaDigitalRole* instance to distinguish redundant configurations. From this meta-level specification, leveraging the reflective structure of digital twins, concrete installations of *Field Systems* can be configured as *base-objects* of type *DigitalTwin* (and its specializations) deriving valued attributes from the *Enterprise Information Server* serving layer (e.g., the *serial number* contains the unique identifier assigned to each deployed component).

On the other hand, reliability requirements can be mapped to executable models of failure modes, which can be exploited in descriptive, predictive, and prescriptive analysis driving operation and maintenance. Specifically, identification of failure modes can be conveniently guided by artifacts of the practice of reliability engineering such as Failure Mode and Effects Analysis (FMEA) [5], Failure Mode, Effects, and Criticality Analysis (FMECA) [9], or Fault Tree Analysis (FTA) [36]. Also on this side, the specification of failure modes in terms of Fault Tree (FT) artifacts can be translated into objects of the *FaultTreeAnalyzer* domain logic within the *Data Analytics Server*, following the meta-model depicted in Fig. 8: this can be automated traversing the FT structure and instantiating a *Node* instance (in the shape of *BasicEvent* or *Gate* objects) for each corresponding element of the FT. Moreover, FMEA and FMECA artifacts may drive stochastic characterisations of *FailureMode* occurrences to enable the exploitation of quantitative analysis over the FT.

In conclusion, following the proposed MDE approach, structural aspects of system configuration are preserved *by construction* guaranteeing that the meta-model complies to, and is isomorphic with, the structure of the specification artifact. This circumvents the need of testing the implementation of each analytic component, which becomes particularly hard in self-adaptive systems. Synoptically, the MDE approach results in three relevant positive side-effects: (i) bridging the gap between the insight of SoS experts and the actual implementation of running software; (ii) reducing the probability of introducing defective system configurations; and (iii) limiting time and effort required by human operators in tuning the cyber side.

7 Case study

In this section, we provide concrete arguments about the practical implications, in terms of advantages and enhancements for the overall system resilience, of applying the proposed framework on a real case study related to a cyber-physical system prototype for Limited Traffic Zone (LTZ) access control, developed within the Research & Development JARVIS project [32] following the MDE approach illustrated in Sect. 6.

7.1 A LTZ cyber-physical system prototype

The increasing number of vehicles on urban roads and the consequent management problems in terms of traffic congestion control, environmental impact, and safety driving, push the adoption of sensitive and adaptive systems to support sustainable forms of transportation based on real-time information as demanded by future Smart Cities. To this end, LTZs have been introduced in most of the historical centers of cities, where only a limited number of authorized vehicles (whose license plates have been registered with the Municipal Police) are allowed to drive. The LTZ access control system typically consists of one or more entrances equipped with two video surveillance cameras: the first one, named *violation* camera, automatically takes photos of all entering vehicles; the second one, named *OCR* camera, is responsible for recognising the registration numbers that are then checked with respect to authorized licenses to detect illegal accesses.

Figure 9 illustrates the concept with reference to the *primary configuration* of the LTZ gate system used for the prototype implementation of the framework, developed by Sismic Sistemi srl[1] and derived by their commercial product line (which is con-

[1] Sismic Sistemi srl is an Italian company specialized in LTZ gate systems, law enforcement operations centers, and radio systems.

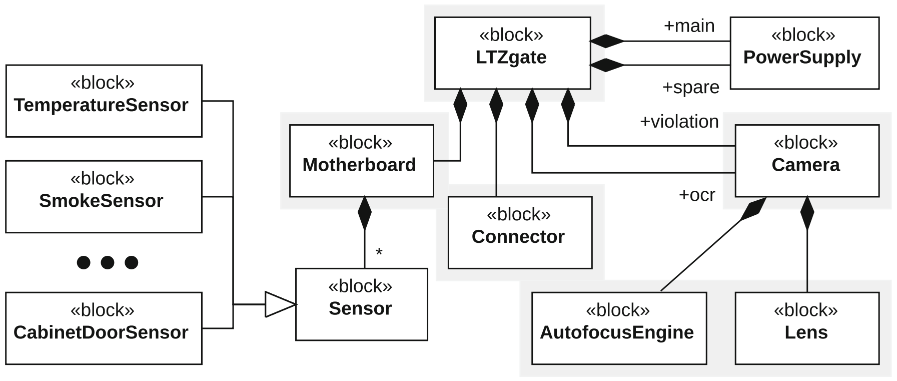

Fig. 9 SysML BDD of the LTZ gate implemented in JARVIS. Blocks internal properties, including *name*, *model*, and *manufacturer*, have been omitted here for clarity of presentation

cretely installed in several Italian municipalities), here modeled as a SysML BDD composed by: a redundant *PowerSupply* block containing two power supply units; two *Camera* blocks, each composed by a set of *Lens* and an *Autofocus engine* which adjusts the camera lens to focus on a target object; a *Connector* block that acts like a communication system, sending raw data generated by the LTZ gate (e.g., acquired images, telemetries) to external systems, and receiving external inputs (e.g., actuation commands); finally, a *Motherboard* component hosting a series of *Sensors* for self-monitoring purposes. An *alternative configuration* of the LTZ gate system consists of just one camera for both violation and registration number detection.

Following the MDE approach described in Sect. 6, the BDD depicted in Fig. 9 is translated into meta-objects (Fig. 10) so as to comprise the Knowledge Base of the *Enterprise Information Server* on the primary hardware configuration of the LTZ gate system. Besides, Fig. 11 shows one of the Fault Trees implemented in the JARVIS framework associated to the LTZ system and a portion of the corresponding objects in the *FaultTreeAnalyzer* domain logic. Note that in so doing, configurations (in terms of Configuration Items and related failure modes) and artifacts comprise a coherent and executable representation, that can support co-evolution at changes in the system configuration or health status.

7.2 A real scenario of self-adaptive system with self-recovery capabilities

We illustrate here how the JARVIS framework can concretely support LTZ gate system resilience through the exploitation of inversion-of-responsibility mechanisms and self-adaptation capabilities.

In normal conditions the system operates in its *primary configuration* with both cameras properly functioning, i.e., the *violation* camera looking down on the gate's entrance to monitor all accesses to a restricted vehicle area, while the *OCR* camera pointing forward and zooming in on the license plate to automatically extract the registration number. On detection of a *Camera failure* for the *violation* camera module, caused by an uncaught *Dirty Lens* basic event, the *FailureDetector* agent notifies

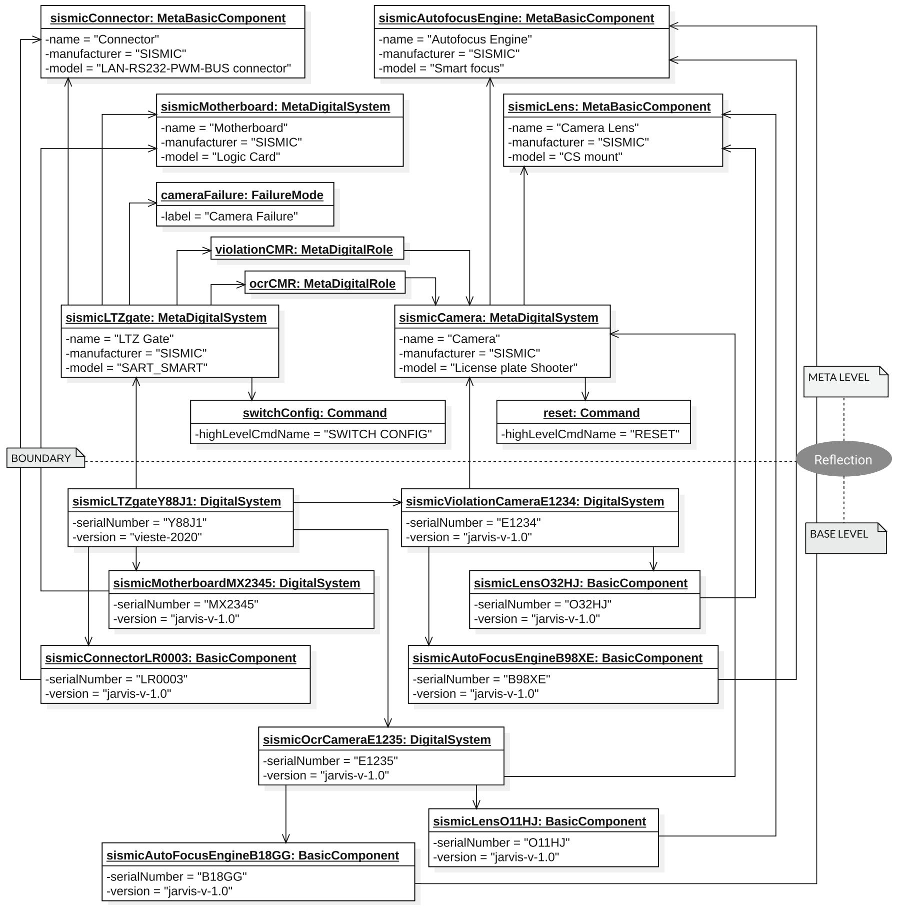
sismicConnector: MetaBasicComponent
-name = "Connector"
-manufacturer = "SISMIC"
-model = "LAN-RS232-PWM-BUS connector"
sismicAutofocusEngine: MetaBasicComponent
-name = "Autofocus Engine"
-manufacturer = "SISMIC"
-model = "Smart focus"
sismicMotherboard: MetaDigitalSystem
-name = "Motherboard"
-manufacturer = "SISMIC"
-model = "Logic Card"
sismicLens: MetaBasicComponent
-name = "Camera Lens"
-manufacturer = "SISMIC"
-model = "CS mount"
cameraFailure: FailureMode
-label = "Camera Failure"
violationCMR: MetaDigitalRole
ocrCMR: MetaDigitalRole
sismicLTZgate: MetaDigitalSystem
-name = "LTZ Gate"
-manufacturer = "SISMIC"
-model = "SART_SMART"
sismicCamera: MetaDigitalSystem
-name = "Camera"
-manufacturer = "SISMIC"
-model = "License plate Shooter"
META LEVEL
switchConfig: Command
-highLevelCmdName = "SWITCH CONFIG"
reset: Command
-highLevelCmdName = "RESET"
BOUNDARY
Reflection
sismicLTZgateY88J1: DigitalSystem
-serialNumber = "Y88J1"
-version = "vieste-2020"
sismicViolationCameraE1234: DigitalSystem
-serialNumber = "E1234"
-version = "jarvis-v-1.0"
BASE LEVEL
sismicMotherboardMX2345: DigitalSystem
-serialNumber = "MX2345"
-version = "jarvis-v-1.0"
sismicLensO32HJ: BasicComponent
-serialNumber = "O32HJ"
-version = "jarvis-v-1.0"
sismicConnectorLR0003: BasicComponent
-serialNumber = "LR0003"
-version = "jarvis-v-1.0"
sismicAutoFocusEngineB98XE: BasicComponent
-serialNumber = "B98XE"
-version = "jarvis-v-1.0"
sismicOcrCameraE1235: DigitalSystem
-serialNumber = "E1235"
-version = "jarvis-v-1.0"
sismicLensO11HJ: BasicComponent
-serialNumber = "O11HJ"
-version = "jarvis-v-1.0"
sismicAutoFocusEngineB18GG: BasicComponent
-serialNumber = "B18GG"
-version = "jarvis-v-1.0"

Fig. 10 UML object diagram related to highlighted BDD elements of Fig. 9. For the sake of conciseness, the diagram hides the *MetaDigitalRole* objects implied in one-to-one relations between *MetaDigitalTwins*

the failure to the *Enterprise Information Server (EIS)* which updates the status of the *DigitalSystem* representing the *violation* camera with a new *MacroscopicEvent* instance of type *FAILURE*.

Two sub-scenarios are then automatically activated. On the one hand, the *FaultTreeAnalyzer* agent is awakened in order to perform a *top-down* analysis for MCS determination over the FT associated with the running system configuration: for each identified basic event, available recovery commands associated with involved digital twins are retrieved from the *EIS* and suggested to end-users through the *Chatbot Services* subsystem.[2] On the other hand, the *FaultTreeAnalyzer* agent starts a *bottom-up* failure propagation analysis to evaluate the negative impact of the observed *violation* camera failure event, highlighting the failure of the overall LTZ gate system.

[2] Note that, in our scenario, the RESET command provided by the *Camera* component can fix only an *Autofocus Engine issue*, while *Lens failures* can only be recovered through human on-site interventions.

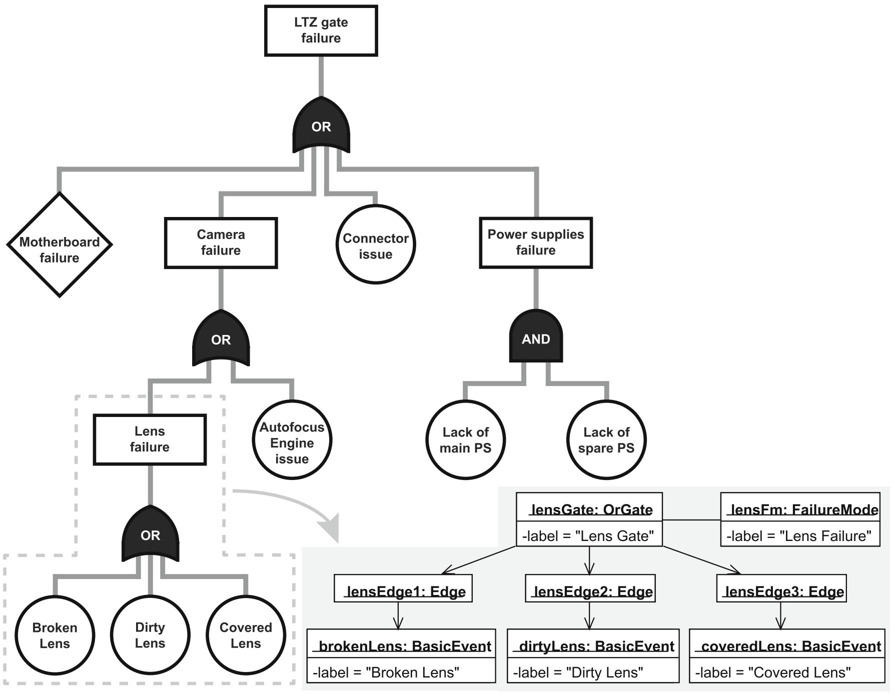

Fig. 11 Fault Tree (FT) associated to the *primary configuration* of the LTZ system depicted in Fig. 9. Note that, switching to the *alternative configuration* does not affect the structure of the FT, comprising only a change about the camera to be considered, i.e., from the *violation* to the *OCR* camera. The internal UML object diagram depicts the executable instances of *FaultTreeAnalyzer* domain logic, related to the dashed grey area of the FT

This implies the re-evaluation of the Knowledge Base by the *EIS* (i.e., a new *MacroscopicEvent* object of type *FAILURE* is instantiated and associated to the LTZ gate *DigitalSystem*), and the recommendation by the *Chatbot Services* to reconfigure the system (i.e., moving from the current *primary* configuration to the *alternative* configuration through the *SWITCH CONFIG* command). In so doing, the LTZ gate system can continue to properly operate, exploiting the *OCR* camera for both violation and registration number detection tasks. Specifically, after command resolution and transcoding processes, a set of *low-level* commands are sent by the *EIS* to: (i) switch off the *violation* camera; (ii) adjust the *OCR* camera position for looking at the gate's entrance; and, (iii) zoom out the *OCR* camera for getting a wider view of the scene.

In summary, the experimented case study lends itself very well to highlighting and discussing the role played by the proposed framework to enhance the overall system resilience. Specifically, data analytics agents provide advanced capacity in real-time monitoring the status of physical components by synthesising and interpreting perceived raw data in high-level events of the digital twin domain model. This empowers the system with self-assessment capabilities by which major disruptions can be promptly detected and handled through the scheduling of reactive, predictive, and proactive maintenance operations with the aim of recovering the proper functioning, even exploiting self-adaptation and remote actuation mechanisms to move the system to alternative configurations. Finally, the concrete implementation of the proposed MDE approach on a real system demonstrates how semi-formal specification of structural aspects and failure logics can support agile construction and verification of executable models.

8 Conclusions

This paper presented a hardware/software framework, designed around a SoS architecture, supporting resilience at runtime of cyber-physical systems, exploiting digital twins and failure models to improve operation, integration, maintenance, and recoverability for many application scenarios, notably including Smart City and Industrial Internet of Things contexts. In so doing, the framework promotes an MDE process adopting SysML BDDs to automatically derive validated executable software representations reflecting physical Configuration Items, and FTs to decorate structural aspects of the system with reliability requirements, respectively. Moreover, the Knowledge Base developed around digital twins has been enriched with IoT control commands design with the aim of empowering the framework with remote actuation capabilities, enabling both recoverability and adaptability mechanisms in a proactive way: in fact, data analytic agents can perform failure detection and propagation analysis to infer disruptions and to react autonomously with resolutive actions or by activating chatbot agents for human interventions.

The framework opens the way to various further developments. On the one hand, the MDE process in combination with an extensive adoption of artifacts produced by different areas of expertise inevitably leads to the problem of co-evolution of architectural and quality evaluation models. This might be, to some extent, solved exploiting artifacts able of subsuming information contained in BDDs and FTs, such as Reliability Block Diagrams. The consequent synchronisation of running digital twins instances and related reliability objects expressed in terms of failure modes and FTs, located in separated subsystems, should also be considered, demanding for advanced orchestration solutions to maintain aligned the digital information distributed across subsystems. Besides, the actual design of the Knowledge Base in the *Enterprise Information Server* lends to investigate fault/failure propagation mechanisms, i.e., how a failure of one component causes a fault of another component in the digital twin hierarchy, in so enabling various quantitative analysis and simulation techniques to drive the reconfiguration process.

Acknowledgements This work is supported in part by the JARVIS Project, funded by the Tuscany Regional Government (Italy) in the POR FESR 2014-2020 program, under contract D55F17000480009. We thank Sismic Sistemi srl for collaboration and support in the engineering and concrete implementation of the JARVIS framework in a real cyber-physical system for Limited Traffic Zone (LTZ) access control.

Funding Open access funding provided by Universitá degli Studi di Firenze within the CRUI-CARE Agreement.

References

1. Abreu DP, Velasquez K, Curado M, Monteiro E (2017) A resilient internet of things architecture for smart cities. Ann Telecommun 72(1–2):19–30
2. Alam KM, El Saddik A (2017) C2ps: a digital twin architecture reference model for the cloud-based cyber-physical systems. IEEE Access 5:2050–2062
3. Alshboul B, Petriu DC (2018) Automatic derivation of fault tree models from SysML models for safety analysis. J Softw Eng Appl 11(5):204–222
4. Andersson J, De Lemos R, Malek S, Weyns D (2009) Reflecting on self-adaptive software systems. In: 2009 ICSE workshop on software engineering for adaptive and self-managing systems. IEEE, pp 38–47
5. Bonfiglio V, Montecchi L, Rossi F, Lollini P, Pataricza A, Bondavalli A (2015) Executable models to support automated software FMEA. In: 2015 IEEE 16th international symposium on high assurance systems engineering. IEEE, pp 189–196
6. Buschmann F, Meunier R, Rohnert H (1996) Pattern-oriented software architecture: a system of patterns, vol 1. Wiley, Hoboken
7. Calinescu R, Ghezzi C, Kwiatkowska M, Mirandola R (2012) Self-adaptive software needs quantitative verification at runtime. Commun ACM 55(9):69–77
8. Cámara J, de Lemos R, Vieira M, Almeida R, Ventura R (2013) Architecture-based resilience evaluation for self-adaptive systems. Computing 95(8):689–722
9. Catelani M, Ciani L, Cristaldi L, Faifer M, Lazzaroni M, Khalil M (2015) Toward a new definition of FMECA approach. In: Proceedings 2015 IEEE international instrumentation and measurement technology conference (I2MTC). IEEE, pp 981–986
10. Chappell DA (2004) Enterprise service bus. O'Reilly Media, Inc, Newton
11. Cioroaica E, Chren S, Buhnova B, Kuhn T., Dimitrov D (2019) Towards creation of a reference architecture for trust-based digital ecosystems. In: Proceedings of the 13th European conference on software architecture-vol 2, pp 273–276
12. Curry E (2004) Message-oriented middleware. Middleware for communications pp 1–28
13. Demchenko Y, Grosso P, De Laat C, Membrey P (2013) Addressing big data issues in scientific data infrastructure. In: 2013 international conference on collaboration technologies and systems (CTS). IEEE, pp 48–55
14. Di Prospero A, Norouzi N, Fokaefs M, Litoiu M (2017) Chatbots as assistants: an architectural framework. In: Proceedings of the 27th annual international conference on computer science and software engineering. IBM Corp, pp 76–86
15. Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, Safina L (2017) Microservices: yesterday, today, and tomorrow. In: Present and ulterior software engineering. Springer, pp 195–216
16. Eder KI, Villegas NM, Trollmann F, Pelliccione P, Müller HA, Schneider D, Grunske L, Rumpe B, Litoiu M, Perini10 A, et al.: Assurance using models at runtime for self-adaptive software systems
17. Ericson CA (1999) Fault tree analysis. System safety conference, Orlando, Florida vol 1, pp 1–9
18. Forsberg K, Mooz H (1991) The relationship of system engineering to the project cycle. In INCOSE international symposium, vol. 1. Wiley Online Library, pp 57–65
19. Friedenthal S, Moore A, Steiner R (2014) A practical guide to SysML: the systems modeling language. Morgan Kaufmann, Burlington

20. Gabor T, Belzner L, Kiermeier M, Beck MT, Neitz A (2016) A simulation-based architecture for smart cyber-physical systems. In: 2016 IEEE international conference on autonomic computing (ICAC). IEEE, pp 374–379
21. Gamma E (1995) Design patterns: elements of reusable object-oriented software. Pearson Education India, New Delhi
22. Getir S, Grunske L, van Hoorn A, Kehrer T, Noller Y, Tichy M (2018) Supporting semi-automatic co-evolution of architecture and fault tree models. J Syst Softw 142:115–135
23. Hahn A (2016) Operational technology and information technology in industrial control systems. In: Cyber-security of SCADA and other industrial control systems. Springer, pp 51–68
24. Hohpe G, Woolf B (2004) Enterprise integration patterns: designing, building, and deploying messaging solutions. Addison-Wesley Professional, Boston
25. Khan R, Khan SU, Zaheer R, Khan S (2012) Future internet: the internet of things architecture, possible applications and key challenges. In: 2012 10th international conference on frontiers of information technology. IEEE, pp 257–260
26. Lee J, Bagheri B, Kao HA (2015) A cyber-physical systems architecture for industry 4.0-based manufacturing systems. Manufact Lett 3:18–23
27. Lepuschitz W, Zoitl A, Vallée M, Merdan M (2010) Toward self-reconfiguration of manufacturing systems using automation agents. IEEE Trans Syst Man Cybern C 41(1):52–69
28. Maier MW (1998) Architecting principles for systems-of-systems. Syst Eng J Int Council Syst Eng 1(4):267–284
29. Malakuti S, Grüner S (2018) Architectural aspects of digital twins in iiot systems. In: Proceedings of the 12th European conference on software architecture: companion proceedings, pp 1–2
30. Mohagheghi P, Gilani W, Stefanescu A, Fernandez MA (2013) An empirical study of the state of the practice and acceptance of Model-Driven Engineering in four industrial cases. Empir Softw Eng 18(1):89–116
31. Montesi F, Weber J (2016) Circuit breakers, discovery, and api gateways in microservices. arXiv preprint arXiv:1609.05830
32. Parri J, Patara F, Sampietro S, Vicario E (2019) JARVIS, A Hardware/Software Framework for Resilient Industry 4.0 Systems. In: International workshop on software engineering for resilient systems. Springer, pp 85–93
33. Parri J, Sampietro S, Vicario E (2018) Deploying digital twins in a lambda architecture for industry 4.0. ERCIM NEWS (115), pp 30–31
34. Pradhan S, Dubey A, Gokhale A (2016) Designing a resilient deployment and reconfiguration infrastructure for remotely managed cyber-physical systems. In: International workshop on software engineering for resilient systems. Springer, pp 88–104
35. Radatz J, Olson M, Campbell S (1995) Mil-std-498, Crosstalk. J Defen Softw Eng 8(2):2–5
36. Ruijters E, Stoelinga M (2015) Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools. Comput Sci Rev 15:29–62
37. Salfner F, Lenk M, Malek M (2010) A survey of online failure prediction methods. ACM Comput Surv 42(3):10
38. Schmidt DC (2006) Model-driven engineering. Comput IEEE Comput Soc 39(2):25
39. Schmidt DC, Stal M, Rohnert H, Buschmann F (2013) Pattern-oriented software architecture, patterns for concurrent and networked objects, vol 2. Wiley, Hoboken
40. Schroeder GN, Steinmetz C, Pereira CE, Espindola DB (2016) Digital twin data modeling with automationml and a communication methodology for data exchange. IFAC-PapersOnLine 49(30):12–17
41. Shangguan D, Chen L, Ding J (2019) A hierarchical digital twin model framework for dynamic cyber-physical system design. In: Proceedings of the 5th international conference on mechatronics and robotics engineering, pp 123–129
42. Shrouf F, Ordieres J, Miragliotta G (2014) Smart factories in industry 4.0: A review of the concept and of energy management approached in production based on the internet of things paradigm. In: 2014 IEEE international conference on industrial engineering and engineering management. IEEE, pp 697–701
43. Simmons AB, Chappell SG (1988) Artificial intelligence-definition and practice. IEEE J Oceanic Eng 13(2):14–42
44. for Standardization/International Electrotechnical Commission IO, et al (2011) Iso/iec 25010: Systems and software engineering-systems and software quality requirements and evaluation (square)-system and software quality models. Authors, Switzerland

45. Suciu G, Vulpe A, Halunga S, Fratu O, Todoran G, Suciu V (2013) Smart cities built on resilient cloud computing and secure internet of things. In: 2013 19th international conference on control systems and computer science. IEEE, pp 513–518
46. Vaidya S, Ambad P, Bhosle S (2018) Industry 4.0–a glimpse. Proc Manufact 20:233–238
47. Wang Y (2009) On abstract intelligence: toward a unifying theory of natural, artificial, machinable, and computational intelligence. Int J Softw Sci Comput Intell 1(1):1–17
48. Weippl ER, Sanderse B (2018) Digital twins: introduction to the special theme. ERCIM News 2018 (115)
49. Yun S, Park JH, Kim WT (2017) Data-centric middleware based digital twin platform for dependable cyber-physical systems. In: 2017 ninth international conference on ubiquitous and future networks (ICUFN). IEEE, pp 922–926