Predictive formal analysis of resilience in cyber-physical systems

Sebti Mouelhi, Mohamed-Emine Laarouchi, Daniela Cancila, Hakima Chaouchi

Received January 15, 2019, accepted February 12, 2019. Date of publication March xx, 2019, date of current version March xx, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2903153

Predictive Formal Analysis of Resilience in Cyber-Physical Systems

SEBTI MOUELHI[1], MOHAMED-EMINE LAAROUCHI[2], DANIELA CANCILA[2] and HAKIMA CHAOUCHI [3] (MEMBER, IEEE)

[1]ECE Paris.Lyon – École d'ingénieurs, INSEEC U., 75015 Paris, France (e-mail: sebti.mouelhi@ece.fr)
[2]CEA LIST, CEA Saclay, 91191 Gif-sur-Yvette, France (e-mail: {mohamed-emine.laarouchi,daniela.cancila}@cea.fr)
[3]Télécom SudParis, Institut Mines-Télécom, 91000 Evry, France (e-mail: hakima.chaouchi@telecom-sudparis.eu)

Corresponding authors: Sebti Mouelhi (sebti.mouelhi@ece.fr) and Daniela Cancila (daniela.cancila@cea.fr).

ABSTRACT The behavioral analysis of cyber-physical systems in safety-critical scenarios is a challenging task. In this context, the endogenous and exogenous aspects of resilience are of a cornerstone importance in system design and verification. Endogenous resilience is the inherent ability of the system to detect and process internal faults and malicious attacks. Exogenous resilience is the permanent capability of the system to maintain a safe operation within its ambient environment. In this article, we present a predictive dual-sided contract-based formal methodology to address both aspects of resilience on top of a distributed object-oriented component-based software model. It is illustrated by a case study of urban drone rescue systems. We exploit the formalism of timed automata and the toolbox UPPAAL to predict by abstraction and analyze (simulate and verify) endogenous resilience. Instead of presenting the final models of the case study, we reflect our experience with UPPAAL in generic patterns of system design and contract specification, reusable in other contexts with adaptations. The analysis of exogenous resilience is specific to the considered drone rescue system. It consists of synthesizing by iterative model-checking safe flight paths for the drones within a 3D virtual model of urban surroundings true to modern cities.

INDEX TERMS Resilience, Safety, Distributed control, Object-oriented software, Component-based architectures, Design by contracts, Timed automata, 3D models, Model-checking, Temporal logic, Fairness.

I. INTRODUCTION

Industry 4.0 refers to the current trend of automation and data exchange technologies used for the manufacture of systems in many industrial sectors such as transportation, energy, and medicine. It fosters the so-called "smart technologies" which include Cyber-Physical Systems (CPSs), cognitive and cloud computing, and the Internet of Things (IoT). In CPSs, computing, communication and control technologies are tightly integrated to achieve efficiency, reliability, robustness, real-time, and stability when dealing with the physical devices. Actually, CPSs cover autonomous and adaptive operations and aim to include artificial intelligence.

Despite this trend, *safety* remains a thorny challenge as amply discussed in [1]. One major problem is to combine the system's operation within the environment and its characterization including the choices of embedded hardware platforms and software design approaches. In addition, a CPS needs to be *resilient*: the ability to withstand a major disruption with respect to permissible degradation parameters and to recover within acceptable times and composite risks and costs [2].

The resilience aspects of CPSs are well summarized in [3]. The authors highlight the *robust control* issues that require consideration of the following measures: 1) the adoption of *model-based approaches* in software design and verification; 2) the diagnostic of the *endogenous* resilience relying on the system ability to handle internal faults and malicious attacks; 3) the analysis of the *exogenous* resilience relying on the system ability to maintain a safe operation within its ambient environment by ensuring reliable control over sensors and actuators. Our work orbits around these topics.

Industrialists are still using common reactive component-based approaches to design software for CPSs [4]–[6]. Under these approaches, software is rigid and hard to maintain. The operation of a subsystem in the CPS is a row of cyclic reactions to environment inputs. This mode of operation does not allow a safe interactive behavior between the system parts. Indeed, data are crudely sent and received "asynchronously" between subsystems through wired and/or wireless networks. This exchange has three major drawbacks: it is error-prone by naming and processing the crude data, does not elucidate the

interaction scenarios between subsystems during design, and hardens their verification. The standards and norms [7]–[9] governing the design of safety-critical (aeronautic, railway, automotive, etc) CPSs recommend component-based development and formal methods for design and verification, while providing reliable solutions to the above issues at the risk of blaming the whole system integrity and certification [10].

In this article, we adopt a distributed and object-oriented component-based approach (extended from [10]) to design the software layer of a typical CPS: an urban drone rescue system with ground remote control. We define an object-oriented component as a software entity with the following functions: 1) it provides and requires environment-dependent synchronous service interfaces; 2) it exchanges directly crude data with its environment through asynchronous ports; 3) it might exhibit progressive behavior at the run-time [11]. By allowing distribution, the component instances can interact (synchronously or asynchronously) while being deployed in distant subsystem nodes. By allowing object-orientation, the safety-critical scenarios are easy to analyze since component interactions could be predicted early in design, and to trace and debug later in the implementation.

In order to analyze endogenous resilience in the considered CPS based on our software model, we establish a contract-based formal method to predict by abstraction and analyze the distributed interactive behavior of the system, the timing constraints of its networking activities, and its survivability in case of functional and timing faults, or detected malicious attacks during operation. We use the toolbox UPPAAL [12] to specify, simulate and verify endogenous resilience in our CPS case study. The formal language of UPPAAL, based on Timed Automata (TAs) networks [12], [13], is well-suited to meet our goals since it comes with various useful features for both design and verification: time-aware specifications, channel-based broadcasts and synchronization mechanisms, a C-like background language used to specify the behavioral effects of state transitions in TAs networks, graphical interfaces for behavior simulation and analysis of model-checking counterexamples, etc.

The main drawback we encountered by practicing model-checking using UPPAAL is that *fairness* conditions, required to define starvation-free models and to ensure liveness, cannot be expressed neither in the Computation Tree Logic (CTL) [14]–[16], used by UPPAAL to specify properties, nor in the model like in the TLA$^+$ language [17]. To overcome this problem, we consider a subset of CTL* [18] (combining CTL and the Linear Temporal Logic (LTL) [19]) as an extension of CTL to express fairness conditions. Then, we show how they could be exploited to guarantee liveness properties, and how TAs can be constrained in UPPAAL to meet them.

Despite its wide academic recognition and successful use in some industrial applications (as summarized in [20]), the common criticism against UPPAAL is about the current implementation of its model-checker based on hash tables [21]. It does not scale to large and complex applications due to the "wall problem" of combinatorial state explosion during verification. Therefore, to truly scale up, UPPAAL should be powered by efficient model-checkers and complemented by incremental design on top approaches [22]. We sustain our contributions by UPPAAL mainly for the reasons mentioned above, and hope that more scalable inductive model-checkers based on efficient SAT/SMT solvers [23], [24] with top-down component-based abstraction languages and code generation tools will be made available around it in the future.

The analysis of exogenous resilience is specific to our case study of urban drone rescue systems. Concretely, it is relative to the "safe presence" of drones in their ambient environment and should be checked on sensor inputs (positions, battery level, etc) and actuator outputs (flight signals). Being in early predictive design, we synthesize optimized flight paths for the drones in a scaled 3D virtualization (built using Blender) true to the modern big cities. The synthesis of flight paths is done by iterative model-checking with respect to input safety properties (mainly obstacle avoidance).

The related works are provided and discussed in Section II. In Section III, we present our urban drone rescue system and its behavioral requirements. They are recalled in the next sections to illustrate gradually our contributions. In Section IV, we present our work methodology going from the operational design to resilience analysis. The software model of the considered system, and its underlying aspects of interoperability and data exchange are provided in Section V. A feasibility study of the implementation is provided in the same section. The complete formal analysis of the endogenous resilience in the system using UPPAAL is described in Section VI. A theoretical body about the expression of fairness conditions in timed automata under UPPAAL is also provided in the same section. In Section VII, we present the 3D-driven formal analysis of exogenous resilience using Blender. Section VIII is dedicated to the conclusion discussions and perspectives.

II. RELATED WORKS

In [25], [26], the authors change the regular process of safety analysis, focusing mainly on negative causes and impacts of unwanted events, and take into account the success stories that deem insignificant. With this aim, the authors promote a new safety analysis classification (Safety-I, Safety-II and Safety-III), and open up new perspectives on the consideration of resilience aspects.

Safety-I is the traditional definition used in safety norms: the system quality should ensure that the number of failure events harmful to workers, the public, or the environment is acceptably low. It requires avoidance of failures and mitigates their eventual consequences. Safety-II focuses on the system ability to succeed in its daily operation instead of studying incidents and failure events and mitigates their consequences as in Safety-I. It assumes that the daily performance variability provides the adaptations needed to respond to varying conditions, and hence is the reason that things go right [25]. Safety-III is likely the combination of the two previous ones. It acknowledges that acceptable and adverse outcomes have common basis, namely everyday performance adjustments.

It offers a new ontological understanding of safety as a new field in system design and development, called "Resilience Engineering". Horizon openings could be cast on the current safety issues under this new concept: given that the world has become more complex, the explanations of unwanted outcomes or success stories of system performance can no longer be limited to cause-effect relations described by linear models [25], [26].

Of the extensive literature around CPSs, we discuss some topicality and works related to the ours. The European project CPSwarm aims to define approaches and tool chains to develop and test collaborative and reconfigurable autonomous CPSs (see `www.cpswarm.eu`). The project partners provide use cases about the usage of autonomous UAV systems in ambient environments, similar to our drone rescue system. Although, the case study was extracted from the CPSwarm workbench [27] and tailored with our interest to resilience.

Few academic works already exist around the study of CPS resilience because it is a recent research topic. In [28], the proposed approach ensures resilience in CPSs through self-healing structural adaptation. This involves adding and removing components, or even changing their interaction at run-time. The authors in [29] propose a hybrid theoretical framework for robust and resilient control design. The proposed framework is applied to the design problem of voltage regulators in synchronous machines and motors.

In [30], the authors distinguish between the notions of information and infrastructure dependability, and clearly illustrate the need to formally model and reason about the dependability aspects of CPS applications. In [31], the authors focus on the CPSs communication aspects and study the effects of intermittent data integrity guarantees on system performance under stealthy attacks. The authors in [32] also tackle the security issues in CPSs by identifying the problem of secure control, investigating the defenses that information security and control theory provide, and proposing a set of challenges that need to be addressed to improve the survivability of a CPS in case of cyberattacks. In [33], the authors propose a software architecture of an agent-based production CPS and consider interoperability and data consistency aspects in their model. They also provide an implementation of such system to evaluate multi-agent approach with regards to conventional production processes.

Perhaps the closest work to ours is the one in [34] since the authors make use of formal methods. They define a generic component-based CPS meta-model in UML, and show how it could be instantiated in concrete systems using a pattern-based methodology based on the so called Formal Concept Analysis (FCA) approach [35]. They also apply a knowledge-driven process to determine all kind of relations between the "cyber-" and "physical-" components of different subsystems and their underlying functionalities to deal with their related resiliency and redundancy properties. Our predictive analysis of resilience in CPSs is specific compared to their approach. By cons, our software design model is scalable in concrete implementations as emphasized in Subsection V-D.

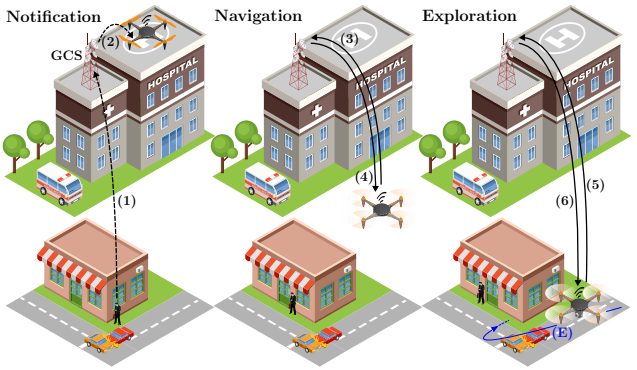

FIGURE 1. Drone/GCS crash exploration scenario. Notification phase: (1) a person on-site sends an alert to the GCS by smartphone; (2) GCS activates the drone; Navigation phase: (3) the drone continuously sends navigation data to GCS; (4) GCS controls the drone when necessary; Exploration phase: (E) while hovering, (5) the drone broadcasts the accident scene to GCS, which in turn (6) may order it to provide medical supply for victims if needed. Steps (3)/(4) and (5)/(6) are repeated with different frequencies (depicted by ⟶).

III. DRONE RESCUE SYSTEM

Unmanned Aerial Vehicles (UAVs) or *drones* are quite useful for healthcare organizations especially in emergency situations to avoid traffic jams, or when traditional transports are severely restricted, following a natural disaster for example. Common applications include live broadcasts of accidents to early grasp them, and deliver the required medical supplies for wounded people. UAVs are safe enough to transport disease test samples and kits in areas with high contagion. In emergency medicine, the studies have shown that drones are fast to deliver automated external defibrillator to rescue out-of-hospital heart attack victims using geographic information systems. Flying at speeds of up to 97 km/h, the drone can reach patients within a radius of 13 km^2 per mn versus 10 mn average for traditional services which increases the chance of survival to 80% [36]. Being connected to live-stream camera fixed on the drone, a Ground Control Station (GCS) instructs the first aid gestures for the patients, and provides the needed medical supply (transported as it happens by the drone).

Nevertheless, the usage of UAVs remains very challenging at the design and verification levels despite the fast progress of their related technologies. Our work tackles these issues for the software layer of a Drone/GCS-based urban rescue system. We provide the *system requirements* to deal with the crash scenario of Fig. 1. By receiving a notification from on-site persons who notice the crash severity (step 1), the GCS activates and sends immediately a drone to the place of the accident (step 2). Once flying, the drone sends continuously its navigation data (including its position composed of the latitude, longitude and altitude), the distance to the nearest obstacle, and the battery level (step 3). The GCS sends back control commands to the drone (when necessary) which may be either i) a next valid position to be reached from a set of possible paths, or ii) an emergency retrograde action if the drone is unable to continue the mission, because of a weak battery charge for example (step 4).

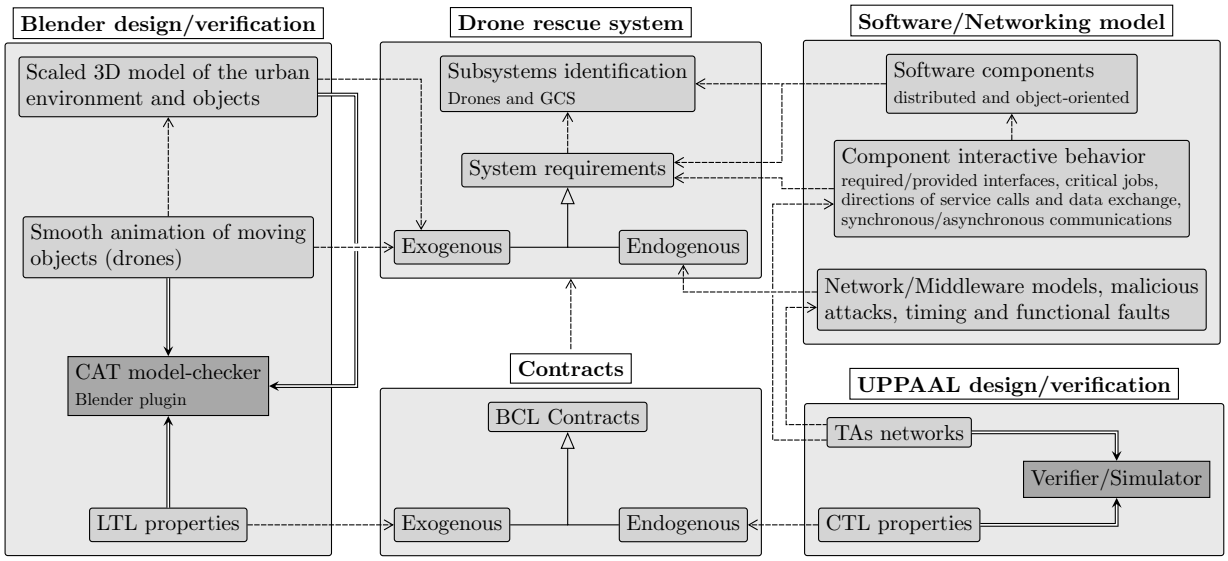

FIGURE 2. The diagram of our predictive methodology applied to analyze the endogenous and exogenous aspects of resilience in the considered drone rescue system; arrows significations: ⇢ uses or depends on, ⟹ tool input; ⊸▷ inherits from.

When the destination is reached, the drone hovers around the accident location and broadcasts a live video stream of the situation (step 5), and continues to receive commands from the GCS to provide the appropriate medical supply, and assist the first aid gestures for the victims if required (step 6).

IV. PREDICTIVE METHODOLOGY

The graphic of Fig. 2 represents our predictive methodology used to address the endogenous and exogenous aspects of resilience in our drone rescue system. The first step of any system design is the identification of the subsystems involved, and the definition of their behavioral requirements. A large panel of languages (and tools) is available around the subject of requirement engineering (like System Modeling Language SysML [37] and Lifecycle Modeling Language LML [38]). SysML has had a great success in the industry of safety-critical systems. The well-known use case and requirement diagrams of SysML are definitely useful to define high-level textual early specifications of systems. Given that the case study does not amount to significant complexity level, and only operational design is considered in this work, we do not exploit any of these languages to define system requirements; we settle for the informal description given in Section III.

Requirements are on the foundation of the design process, but are also used to define system *contracts* [39], required for the verification phase. A contract is semantically a Hoare triplet [40] that annotates a system *operation* by *assumptions* on the inputs and *guarantees* on the outputs. We use for that the Block Contract Language (BCL) [41] to extract contracts from requirements. BCL is pattern-based and semi-formal, easy to read, and convert to formal statements. For better readability, the endogenous and exogenous BCL contracts are given in Subsection VI-E and Section VII along with the low-level properties used to verify our system design.

The software layer of our system is architected according to a Distributed Object-Oriented Component-Based Design (DOOCBD) approach extended from that of [10]. This new architecture embraces interface-driven composition, service-oriented interoperability, and direct data exchange between the behavioral jobs (tasks) of components. At run-time, these components are instantiated in distributed live objects that communicate (locally or remotely) while being deployed in connected software nodes embodied within the GCS and the drones (see Section V).

In order to bring in design the real conditions of operation, we need to analyze the system behavior within abstracted models of wireless network and middleware: the scenarios of exchanging (Tx/Rx) data through the network physical layer should be emulated with respect to a bounded latency. In addition, we should predict the system survivability when functional and timing faults or malicious attacks are detected.

The endogenous resilience involves the specification of all the critical scenarios and behavioral entities described above in TAs networks using UPPAAL. Endogenous BCL contracts are then translated to safety and liveness CTL properties and analyzed by the simulator and model-checker of the toolbox. Further details are provided in Section VI.

Concerning exogenous resilience, the design step involves 3D modeling of the urban environment and smooth animations of moving objects. The benefit of 3D models is stating the obvious: they schematize concepts more distinctly than any classic design language. Moreover, they allow an early feasibility of the system before realization. We use Blender with the built-in Contract Analysis Tool (CAT) to check if the exogenous BCL contracts, related to flight plannings and translated to LTL properties, are respected. If not, the flight path planning is corrected iteratively until the properties are met (see Section VII).

V. SOFTWARE MODEL

CPSs (and particularly the drone rescue system presented in Section III) are distributed and thoroughly compliant with the software component-based design presented in this section. It is an extended version of the DOOCBD approach already introduced in [10]. Before presenting the component architecture of the system, we start by introducing the principles of our approach, and its underlying properties of component interoperability and data exchange.

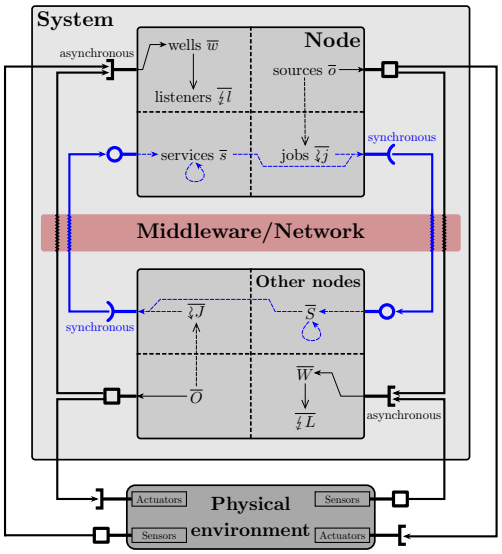

FIGURE 3. Distributed object-oriented model of a CPS; arrows signification: ⇢ executed periodically (resp.⟹ continuously) by, ⇢ call methods from, ⟶ data flow; $\overline{x}$ represent a sequence of elements $x_1, \ldots, x_n$.

A. THE DOOCBD APPROACH

Distributed systems are networks of computing *nodes* interacting with (and controlling) physical processes. According to the model of Fig. 3, a node is typically an open *big object* interacting with the physical environment and other nodes composing the system [10]. It *accepts* input data from (resp. *provides services* for) its environment, *generates* output data for (resp. *requires* services from) it, and may express progressive behaviors by running *periodic jobs*.

Crude (byte-coded) data can be exchanged directly with the physical environment and between nodes through *source* and *well asynchronous ports* (implemented resp. by *methods* $\overline{o}$ and $\overline{w}$). These ports are especially suitable for low-level exchange with the hardware sensors and actuators used resp. to sense and control the physical processes. On the other hand, the services $\overline{s}$ are *synchronous* and used for high-level inter-node interoperability. A service is a method with one-shot computation. It is callable locally by a node (private or public) or by its environment (only public), and may change the internal state (private attributes) of the node, and might require *remote* services from other distant nodes.

The jobs $\overline{\downarrow j}$ handle the actions and reactions of the node while interacting with its environment. Like services, a job may have impact the internal state of the node. It may also 1) require (private or public) methods, or 2) act as a periodic crude data *talker* task (by running *periodically* data sources). Data *listeners* $\overline{\lfloor l}$ are loops running *continuously* data wells in order to input aperiodic data flows. More details about talkers and listeners are presented in Subsection V-B.

The structure of components is almost the same as that of nodes but at finer levels of encapsulation and granularity. It is similar to the CORBA Component Model (CCM) [42] of OGM. Concretely, each node incorporates one or more *partitions*, each of which is a collection of component live *instances*. These instances may interact locally within their partitions, or with component instances of other partitions running on the same or distant nodes [10].

B. SYN VS. ASYNCHRONOUS COMMUNICATIONS

The classic scheme of synchronous inter-node interoperability is depicted by the scenario of Fig. 4. A node n requesting a service from a remote node m is blocked while waiting for it to return. The services $m.\text{srvc}(r_n^k)$ are handled by *aperiodic* tasks with *priorities* higher or lower than that of the nominal processes of m and should return before fixed time *deadlines*.

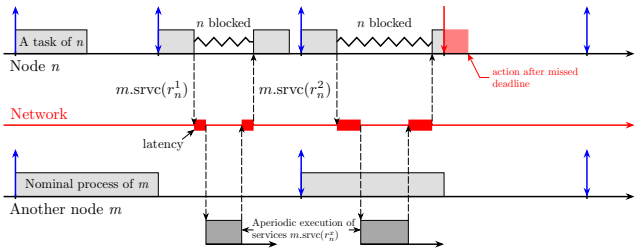

FIGURE 4. Classic synchronous inter-node interoperability.

These aperiodic executions of services may or not preempt the nominal process depending on their priorities and the processor architecture of the execution platform. Aperiodic service handlers in Fig. 4 might run on separate processors different from those used for nominal executions.

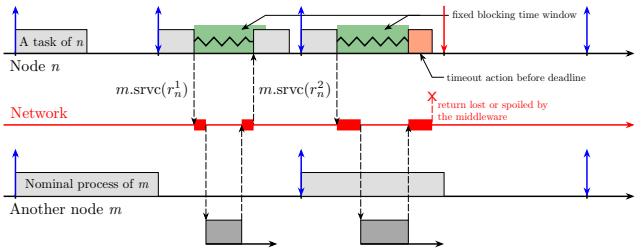

FIGURE 5. Timeout-constrained synchronous inter-node interoperability.

We adopted for our approach a more constrained model of synchronous interoperability based on *timeouts* (see Fig.5) which guarantee a better *timing predictability* than the classic one: if it does not receive the return value of the service call within a fixed time window, the caller process rejects the call and takes the appropriate decision based on the criticality of the service, and its impact on the system integrity. Timeouts shall be adjusted to meet the task deadline.

Real-time analysis of the caller task should consider the *worst-case blocking time* of method calls (including network *latency*). Low latency can be ensured by using adequate networking models and hardware platforms to improve real-time determinism. High priority aperiodic service calls should not disturb the nominal processes of the invoked node. They should not be excessively delayed (so that the caller task meets the best possible deadlines) especially when they are critical for the caller process.

By adopting suitably method-based synchronous communication, the interoperability between objects is explicit in design, and the implementation is easy to debug and maintain. Nonetheless, the synchronous communication is not the best suited in other contexts, especially when the synchronization is non critical for the system's integrity, or when the data exchange is with low-level hardware components (sensors and actuators). Fig. 6 depicts a simple scenario of asynchronous data exchange between two nodes.

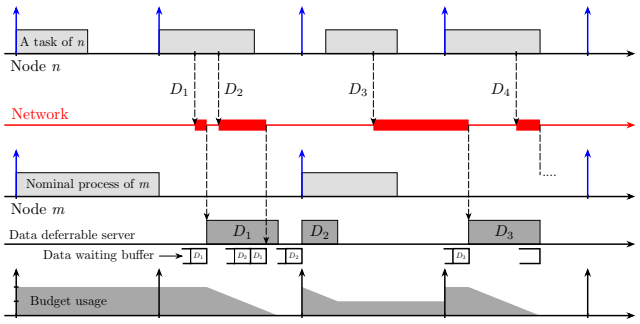

FIGURE 6. Asynchronous data acquisition. Sending data D_i is aperiodic in this generic scenario. In the software architecture discussed in Subsection V-C, data are assumed to be periodically communicated by the talker tasks.

When a byte-coded data D_i is aperiodically received by the node m, it is handled by a server in order to convert it to a software consumable information. A server is a periodic-like task defined by two static parameters: a *period* and a *time budget*. Once active, it reads the *pending* (threaded in a *buffer*) received data as soon as possible within the limit of its budget. Data are handled in general according to their *arrival instants*. If the budget is exhausted, the server must wait until the next cycle of the server to resume reading data. In the scenario of Fig. 6, we consider the example of a *deferrable* server [43] for asynchronous data acquisition.

The server may have higher or lower priority compared to that of nominal processes depending on the data importance. Like for synchronous interactions, the implementation should ensure that important data processing is not excessively delayed, and does not disturb the real-time determinism of the nominal process.

Asynchronous communication has drawbacks: 1) it is rigid and difficult to maintain; 2) it is error-prone by wrongly reading data; 3) it may suffer from non-deterministic processing delays because data are buffered and handled according to reception order; 4) the underlying communication semantics between nodes is not explicit in the implementation.

C. COMPONENT-BASED ARCHITECTURE

The component architecture, depicted in Fig. 7, presents the software structure of the drone rescue system, wherein nodes and partitions are not depicted. Given that it involves the use of several drones remotely guided by a GCS, its main components are highlighted: *Drone* and *GCS*. The middleware behavior is discussed in Subsection VI-B.

As briefly stated in Section III, *Drone* represents the main software unit of a connected drone. It is to perform the minimum of internal computations, and strictly follows the *GCS* commands, making it regularly aware of its state. All the heavy control operations, likely to consume an important amount of the drone energy (particularly the battery charge), are performed by *GCS* whose role is to fully control the drone mission remotely: paths computation, assistance decisions, retrograde actions, etc. It also controls the drone action on the crash site when filming the accident, and supplying the needed medical material.

Before starting the flight, the drone is activated by *GCS* by invoking the public (+) method *Activate*, provided remotely (@) from an instance of *Drone* running on the drone's embedded platform. Since it may simultaneously communicate with several drones, *GCS* maintains a map attribute *paths* (private –), associating each connected drone to a mission path, which is a cursor-moving array of records *Coordinates*, consisting of three fields: latitude, longitude and altitude. The flight path of each drone is initialized by *GCS*, and can be updated following unforeseen positions the drone may reach during its flight (*Control*). As soon as a position in the path is reached, the *Drone* instance updates (by the periodic job *Flight*) the next position by calling *Fly_To* from *GCS*.

In order to decide about the next position, *GCS* needs to be regularly informed of the drone's state (by invoking the method *Get_Data*). *Drone* periodically collects the following sensing data for *GCS*: 1) the current position (*Coordinates*) from the component *Position_Updater* (that acquires the latitude and the longitude from a GPS unit, and the altitude from a barometric sensor); 2) the distance to the nearest obstacle from the component *Sonar* (using an ultrasonic sensor); 3) the battery level from the component *Battery_Checker*; 4) a stability boolean computed by *Stability_Controller* based on three-dimensional linear and angular acceleration parameters (resp. provided by an accelerometer and a gyroscope).

To reach the next position, *Drone* launches the speed control process by calling the method *Actuate_Rotors* (procedure) of the component *Navigation_Controller*. It computes the *Speed_Request* for the whole drone based on the individual *Current_Speed* of each rotor (provided by an instance of the component *Rotor* showed in Fig. 8) and its current *Position*. The actual speed is computed for each rotor using the actual number of revolutions from the last speed computation (determined from the *pulses* generated by the motor *Encoder*). This sensor has a predefined resolution in PPR (Pulses Per motor Revolution), and is asynchronously read by the instance of *Rotor* using the data listener *Update_Pulses* (server) through the data well *Read_Pulses* (see Fig. 6).

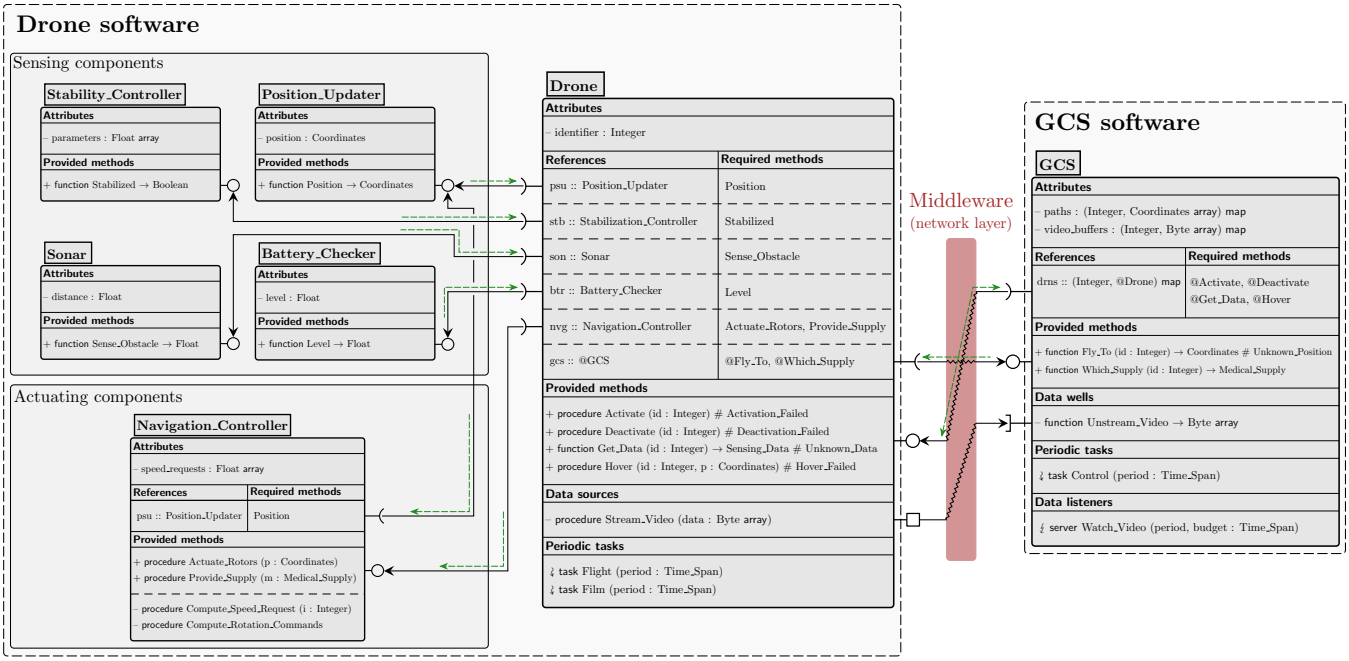

FIGURE 7. The software component architecture of the drone rescue system. Dashed arrows ⇢ represent the direction of data transitions when synchronous services are invoked: i) they are in the opposite direction of method call (continuous arrows →) because transiting data are values returned or exceptions thrown to the caller components; ii) they are in the same direction of a method call arrow when data are directly passed as argument(s) to the called component.

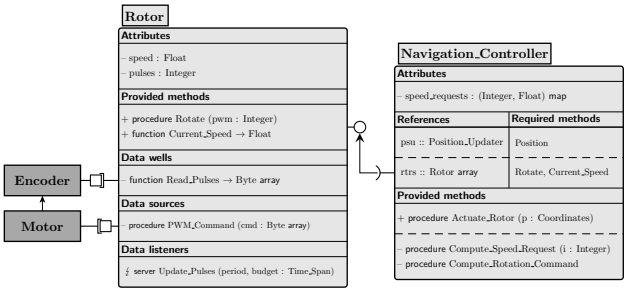

FIGURE 8. Concrete architecture and dependencies of *Navigation_Controller*.

Once the speed request is computed, *Navigation_Controller* actuates each rotor using a Pulse-Width Modulation (PWM) command (*Compute_Rotation_Command*) passed to the method *Rotate* for each instance of *Rotor* interfaced with each motor (see Fig. 8). The actual PWM value is proportionally calibrated in relation to the *error* between the speed request and the current speed. Each *Rotor* then applies the PWM command through the data source *PWM_Command*.

When the destination is reached, *GCS* signals the drone to hover around the accident (by invoking the method *Hover* provided by *Drone*), to broadcast a live stream video of the accident using the data source *Stream_Video*, and to provide the appropriate medical supply (*Which_Supply*), if needed. When the mission ends, the drone flies back to the starting position based on the same principle of the outward flight, and is deactivated (*Deactivate*) by *GCS* upon arrival.

D. IMPLEMENTATION FEASIBILITY

Object-oriented development has often been a hard sell in safety-critical systems industry [44]. The applied standards require extensive verification processes and real-time difficult to carry on by the dynamic aspects and flexibility of object-oriented paradigms (polymorphism, dynamic dispatch, late binding, overriding, etc). The distribution is also penalizing because of its semantics (message passing, remote dispatch and procedure call, etc). The Ada programming language is sufficiently expressive to implement our software model and can decidedly deal with these disadvantages. It is strongly typed and object-oriented with a powerful and explicit support for tasking, concurrency, mutliprocessor architectures, compiler directives (*pragmas*), design by contracts [39] (the SPARK language [45]), etc. It allows developers to exploit the object-oriented assets while avoiding vulnerabilities and ensuring real-time [10], [44]. Besides, subsets of Ada are the target of many design and code generation toolboxes widely used in the industry (like SCADE Suite and Atelier B).

A speed control application for connected wheeled robot platoons[1], based on the DOOCBD approach, was discussed in [10]. The implementation is mainly in Ada, and based on annexes D and E of the Ada Reference Manual resp. of real-time and distributed systems. Annex E (abbreviated DSA) provides support for efficient distribution by making the middleware layer completely transparent and the development easier. The distribution in the application is managed by the middleware PolyORB [46] (maintained by AdaCore).

[1] `https://github.com/mouelhis/adawrplatoon`

The covered control scenarios and interoperability aspects in [10] are very close to those presented and discussed in this article. The Ada application is a proof of concept of our formal design, and we expect to contemplate prototyping real drones in the future. The development process will be much easier to approach. The predictive analysis presented in the next two sections has pre-chewed a lot of the work.

Concerning wireless connectivity in UAVs, an in-depth analysis of the implementation opportunities and challenges is provided in [47]. According to the authors, low-altitude short-range line-of-sight communication scheme seems to be the best suited to our case study, and may potentially lead to significant performance gains. This modality of wireless connectivity allows the dynamic adjustment of the UAV states to well suit the networking environment. For example, when a UAV experiences good channels with ground terminals, it can conserve energy to sustain good wireless connectivity in order to transmit more data to terminals. This is entirely what is needed in our medical rescue context. The standards IEEE 802.11p [48] and ITS-G5 [49] can be adopted to implement Drone-to-Drone or Drone-to-GCS communications since they suit such highly dynamic networking topology.

VI. ANALYSIS OF ENDOGENOUS RESILIENCE

We distinguish four features to handle endogenous resilience in our context: 1) the interactive behavior between drones and the GCS during their flights, 2) their behavioral actions (and reactions) while interacting with the GCS, 3) wireless network latency impact on timing predictability, and 4) survivability in the presence of malicious attacks and functional or timeout faults. Design and verification approaches dealing with the second feature, which cover particularly the individual internal behavior of component units, are well established both in academia and industry.

In this section, we provide a formal methodology to reason about (by abstract prediction) and verify together the first, the third, and the fourth features since they are challenging and lack of a deep investigation in the literature. We reflect in our formal design under UPPAAL the relevant parts of the components behavior, which are related to their synchronous interoperability in the presence of middleware and network abstract models. Asynchronous data exchange is restricted to non-critical communications and hardware management (see Subsection V-B) and not considered in our design.

First, we start by recalling the definition of timed automata and their transitional semantics. Second, we present UPPAAL design patterns of the interactive behavior of the software and middleware components under an abstract network model. These patterns are to avoid hindering the article with the final models, and to make them easy to approach. Then, we provide an ample formal body on the expression of fairness conditions in CTL, and how timed automata in UPPAAL could be constrained to meet fairness, and ensure starvation-free behaviors and liveness. We end the section with the CTL properties, that we extract from BCL generic contracts of the UPPAAL design patterns, and use for verification.

A. NETWORKS OF TIMED AUTOMATA

No better than the UPPAAL tutorial [12] to recall the common definition of timed automata: finite-state machines extended by synchronous-evolving *clocks* used to abstract and reason about the real-time behaviors of systems. UPPAAL extends them by bounded discrete integer variables. In this pargaraph, we slightly revisit the succinct and intuitive definitions of the formalism and its semantics given in [12] for more precision.

We denote by $\mathbf{X}$ the universal set of discrete variables. Given a set $X \subset \mathbf{X}$, we define by $\mathbb{T}[x]$ the type (possible values) of $x \in X$, written $x : \mathbb{T}[x]$ for short. $\mathbb{T}[\![X]\!]$ is the type of X (the union set of cartesian products $\prod_{x_i \in X} \mathbb{T}[\sigma(x_i)]$ for all permutations $\sigma : X \to X$). We write $\mathbb{T}[\![X_1, ..., X_n]\!]$ for $\mathbb{T}[\![\bigcup_{1 \leq i \leq n}(X_i)]\!]$ with $X_i \subset \mathbf{X}$. We denote by $\mathbf{C}$ the universal set of *clocks*. We have $\mathbb{T}[c] = \mathbb{R}^+$ for any $c \in \mathbf{C}$, and $\mathbb{T}[C] = (\mathbb{R}^+)^n$ for any $C \subset \mathbf{C}$ with a cardinality $|C| = n$. The left shifted (or the *next*) version of $x \in X$ with a one logic step is denoted by $x' : \mathbb{T}[x]$ *e.g.,* let us consider two logic successive states s_1 and s_2, if $x = x_1$ at s_1 and $x = x_2$ at s_2, then $x' = x_2$ at s_1 and so on. We generalize the notation for sets, X' is the set of next versions x' of all $x \in X$.

Conditions are predicates of the First-Order Logic (FOL). Given $X = \{x_1, ..., x_n\} \subset \mathbf{X}$, a *predicate* p on $x \in X$ represents a subset of the possible values of x *i.e.,* p is a sub-type of $\mathbb{T}[x]$. A predicate Q on X is then a sub-type of $\mathbb{T}[\![X]\!]$. The *projection* of Q on $Z = \{z_1, ..., z_k\} \subseteq X$ is written $Q[\![Z]\!]$. The syntax of predicates, functions, operators and constants are defined according to variable types under specific theories (equality, linear arithmetic, etc). $Q\langle x/x' \rangle$ is Q by substituting $x \in X$ by its primed version x'. We generalize the notation for sets: $Q\langle\!\langle X/X' \rangle\!\rangle = Q\langle x_1/x'_1 \rangle \langle x_2/x'_2 \rangle ... \langle x_n/x'_n \rangle$.

Definition 1. *A* timed automaton (TA) *A on $C, X \subset \mathbf{C}, \mathbf{X}$ is a tuple $(\Upsilon, \imath, \Sigma, \Psi, \mathcal{G}, \mathcal{E}, \mathcal{J}, \mathcal{I})$ consisting of:*

- *a set of* locations Υ *with* $\imath \in \Upsilon$ *is the* initial *state;*
- *a set of* actions Σ*;*
- *a* transition function $\Psi \subseteq \Upsilon \times \Sigma \to \Upsilon$*;*
- *a* guard *function* $\mathcal{G} \subseteq \Psi \to \mathbb{T}[\![C, X]\!]$*;*
- *an* update *function* $\mathcal{E} \subseteq \Psi \times \mathbb{T}[\![C, X]\!] \to \mathbb{T}[\![C', X']\!]$*;*
- *an* initialization *function* $\mathcal{J} \subseteq \{\imath\} \to \mathbb{T}[\![C, X]\!]$*;*
- *an* invariant *function* $\mathcal{I} \subseteq \Upsilon \to \mathbb{T}[\![C, X]\!]$.

For all $c \in (\mathbb{R}^+)^n = \mathbb{T}[\![C]\!]$ where $n = |C|$, we denote by $c \oplus \tau$ the tuple $(c_1 + \tau, ..., c_n + \tau)$. To save space, we write $e_1...e_k$ instead of $(e_1, ..., e_k)$ with $k \in \mathbb{N}^+$. We write $A.K$ for any component $K \in \{\Upsilon, \imath, \Sigma, \Psi, \mathcal{G}, \mathcal{E}, \mathcal{J}, \mathcal{I}\}$ of A.

Definition 2. *The* semantics $\mathfrak{S}(A)$ *of A is a labeled transition system LTS $(S, s_0, \mathcal{R})$ where $S \subset \Upsilon \times \mathbb{R}^{|C|} \times \mathbb{T}[\![X]\!]$ is the set of* states *with $s_0 = (\imath, \mathcal{J}(\imath)[\![C]\!], \mathcal{J}(\imath)[\![X]\!]) \in S$ is* initial*, and $\mathcal{R} \subseteq S \times (\mathbb{R}^+ \cup \Sigma) \to S$ is a* transition function *such that:*

- $\mathcal{R}(lcx, \tau) = l(c \oplus \tau)x$ *if* $(\forall t \in [0, \tau])\, c \oplus t \in \mathcal{I}(l)[\![C]\!]$*;*
- $\mathcal{R}(lcx, a) = l'c'x'$ *such that*
  - $l' = \Psi(la)$,
  - $c \in \mathcal{I}(l) \wedge \mathcal{G}(la \mapsto l')[\![C]\!]$, $c' \in \mathcal{E}(la \mapsto l', c)[\![C']\!]$,
  - $x \in \mathcal{I}(l) \wedge \mathcal{G}(la \mapsto l')[\![X]\!]$, $x' \in \mathcal{E}(la \mapsto l', x)[\![X']\!]$,
  - $c' \in \mathcal{I}(l')\langle\!\langle C/C' \rangle\!\rangle$, *and* $x' \in \mathcal{I}(l')\langle\!\langle X/X' \rangle\!\rangle$.

In UPPAAL's graphical language, a system is modeled by a Network of Timed Automata (NTA) with concurrent behavioral semantics over sets of common clocks and variables $C, X \subset \mathbf{C}, \mathbf{X}$, and actions Σ split into two disjoint subsets Σ^{asy} and Σ^{sy} of resp. *asynchronous* and *synchronous* actions. This NTA is written $A_1 \| ... \| A_n$ (or $A_{1..n}$), and consists of n TAs A_i for $i \in \{1, ..., n\}$. We write K_i, $\bar{\Upsilon}$ and $\bar{\imath}$ resp. for $A_i.K$, the product $\prod_i \Upsilon_i$, and the vector $\imath_1 ... \imath_n$. We define $\mathcal{I} \subseteq \bar{\Upsilon} \to \mathbb{T}[\![C, X]\!]$ and $\mathcal{J} \subseteq \{\bar{\imath}\} \to \mathbb{T}[\![C, X]\!]$ resp. such that $\mathcal{I}(\bar{l}) = \bigwedge_i \mathcal{I}_i(l_i)$ with $\bar{l} = l_1 ... l_n$ and $\mathcal{J}(\bar{\imath}) = \bigwedge_i \mathcal{J}_i(\imath_i)$. We write $\bar{l}[l_i/l_i']$ to denote $\bar{l}$ with l_i replaced by l_i'.

Definition 3. *The semantics $\mathfrak{S}(A_{1..n})$ of $A_{1..n}$ is a LTS $(S, s_0, \mathcal{R})$ consisting of a set of states $S \subset \bar{\Upsilon} \times \mathbb{R}^{|C|} \times \mathbb{T}[\![X]\!]$ with $s_0 = (\bar{\imath}, \mathcal{J}(\bar{\imath})[\![C]\!], \mathcal{J}(\bar{\imath})[\![X]\!]) \in S$ initial, and a transition relation $\mathcal{R} \subseteq S \times (\mathbb{R}^+ \cup \Sigma) \to S$ defined such that:*

- $\mathcal{R}(\bar{l}cx, \tau) = \bar{l}(c \oplus \tau)x$ *if* $(\forall t \in [0, \tau])\ c \oplus t \in \mathcal{I}(\bar{l})[\![C]\!]$;
- $\mathcal{R}(\bar{l}cx, a) = \bar{l}'c'x'$ *such that*
  - $a \in \Sigma^{\text{asy}}$, $\bar{l}' = \bar{l}[l_i/\Psi_i(l_i a)]$,
  - $c \in \mathcal{I}(l) \wedge \mathcal{G}_i(l_i a \mapsto l_i')[\![C]\!]$, $c' \in \mathcal{E}_i(l_i a \mapsto l_i', c)[\![C']\!]$,
  - $x \in \mathcal{I}(l) \wedge \mathcal{G}_i(l_i a \mapsto l_i')[\![X]\!]$, $x' \in \mathcal{E}_i(l_i a \mapsto l_i', x)[\![X']\!]$,
  - $c' \in \mathcal{I}(\bar{l}')\langle\!\langle C/C' \rangle\!\rangle$, *and* $x' \in \mathcal{I}(\bar{l}')\langle\!\langle X/X' \rangle\!\rangle$;
- $\mathcal{R}(\bar{l}cx, b) = \bar{l}'c'x'$ *such that*
  - $b \in \Sigma^{\text{sy}}$, $\bar{l}' = \bar{l}[l_i/\Psi_i(l_i b)][l_j/\Psi_j(l_j b)]$,
  - $c \in \mathcal{I}(\bar{l}) \wedge \mathcal{G}_i(l_i b \mapsto l_i') \wedge \mathcal{G}_j(l_j b \mapsto l_j')[\![C]\!]$,
    $v \in \mathcal{I}(\bar{l}) \wedge \mathcal{G}_i(l_i b \mapsto l_i') \wedge \mathcal{G}_j(l_j b \mapsto l_j')[\![X]\!]$,
  - $c' \in \mathcal{E}_i(l_i b \mapsto l_i', c) \wedge \mathcal{E}_j(l_j b \mapsto l_j', c)[\![C']\!]$,
    $x' \in \mathcal{E}_i(l_i b \mapsto l_i', x) \wedge \mathcal{E}_j(l_j b \mapsto l_j', x)[\![X']\!]$,
  - $c' \in \mathcal{I}(\bar{l}')\langle\!\langle C/C' \rangle\!\rangle$, *and* $x' \in \mathcal{I}(\bar{l}')\langle\!\langle X/X' \rangle\!\rangle$.

From a given state of the resulted LTS, every A_i may 1) fire a transition separately by elapsing time, or by enabling an asynchronous action and applying its individual update on variables and clocks, or 2) synchronize with another TA A_j through transition(s) labeled by a synchronous action a enabled as output (depicted $a!$ in UPPAAL) in one of them and as input (depicted by $a?$ in UPPAAL) in the other such that $a!$ transition update applies before that of $a?$ transition.

The toolbox UPPAAL offers additional design features like time urgency, synchronous and broadcasting (asynchronous) channels, data types (arrays and structures), etc. It has a query language to specify CTL properties for model-checking; their taxonomy is introduced in Subsection VI-C.

B. UPPAAL DESIGN PATTERNS

To preserve space and hide complexity, instead of presenting the whole UPPAAL models of our CPS case study, we present our experience as generic design patterns, reusable in other contexts with adaptations. The models of the jobs *Flight* and *Control*, resp. of *GCS* and *Drone*, are discussed in reference to those patterns (see Subsection VI-D). As stated in Section I, we mainly use UPPAAL to sustain our predictive analysis approach while being aware that the verification of an exhaustive design will not scale due to the state explosion problem. Nevertheless, the provided models are verifiable in an acceptable time (see Subsection VI-D).

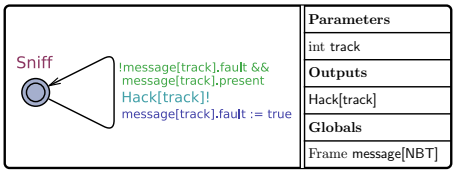

FIGURE 9. Pattern *Hacker*.

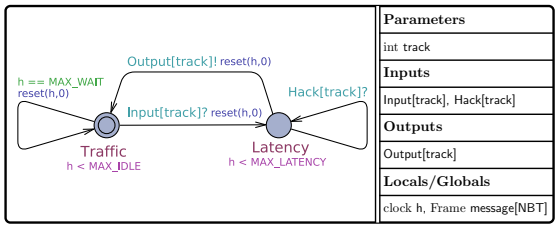

FIGURE 10. Pattern *Network track*.

We consider a wireless channel with a frequency band divided into a fixed number NBT of half-duplex tracks (sub-frequencies). The exchanged messages are defined as an array message[NBT] of data frames. It contains among others the coded message, and the identities of the transmitter and the receiver. It is updated when a message is posted through the channel Input[track], or when a message is received from the channel Output[track] (see patterns 3 and 4).

Pattern 1 (Network track). A network frequency track (see Fig. 10) can be designed as a timed automaton where a trace starts by an input action on the channel Input[track] if data are posted through the corresponding track, and finishes by an output action from the *broadcast* channel Output[track] to provide the frame message[track] to the destination(s). The network latency is measured by a clock h local to each track and bounded by MAX_LATENCY. It is reset to 0 by firing each input/output synchronization edge or if no traffic is present on the track for a delay equal to MAX_IDLE.

Broadcast channels allow *one-to-many* synchronizations: an edge with an output action a! on a broadcast channel a can always fire (if the guard is satisfied), no matter if any receiving edges (labeled by a?) are enabled immediately or not. Those receiving edges once enabled *will* synchronize [12].

Pattern 2 (Hacker). A hacker can be designed as a single state self-loop automaton injecting faults in message[track], transiting on track (see Fig. 9). The hack is assumed to intervene during the transmission phase (in the location Latency) by enabling the input event Hack[track]?.

The above patterns may seem very simplistic since wireless networking and hacking malicious attacks cannot be resumed by these light abstractions. Instead of investigating concrete hacking scenarios, our goal here is to predict not only the system survivability if such attack is detected, but also its reaction when latency exceeds the prefixed timing bound for synchronous communications. References to some research studies on networking analysis and attack resilience in CPSs are discussed in the related works (see Section II).

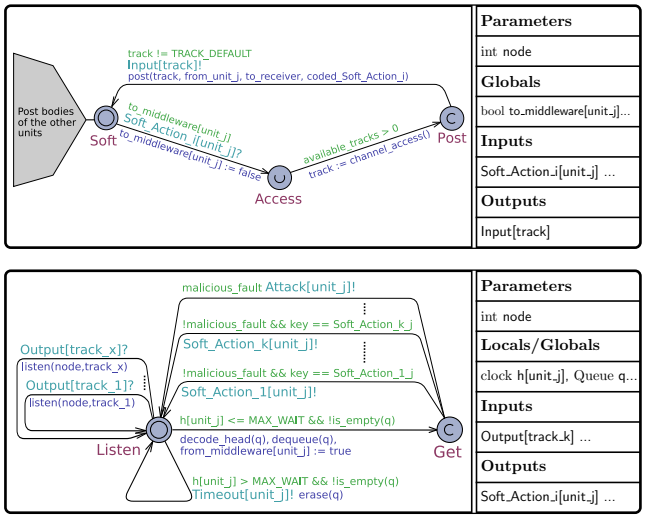

FIGURE 11. Pattern *Middleware Tx* (top); Pattern *Middleware Rx* (bottom).

Pattern 3 (**Middleware Tx).** The middleware transmitter Tx (see Fig. 11, top) is defined separately for each subsystem node. It is designed as a *wing*-automaton looping on the initial state for each software action Soft_Action_i[unit_j] (method calls, return values or exceptions) made by a component unit_j (method and jobs bodies). The middle *urgent* location Access is used for network channel_access(). If a track is available, access will be given to post() the coded message atomically via the *commited* (see the explanation below) location Post, while synchronizing with the network on Input[track].

An urgent location is equivalent to a location such that its incoming edges reset a clock c, and it is labeled with an invariant $c \leq 0$. Time does not progress in urgent locations. However, interleaving with other clock-free edges is allowed. A committed location is more restrictive: its outgoing edges can be only interleaved with those of the other committed locations [12]. In the Tx pattern (see Fig. 11, top), Access is urgent since interleaving is allowed only when no tracks are available. The location Post is however commited because when access is given, Tx is intended to post immediately.

Pattern 4 (**Middleware Rx).** The middleware receiver Rx (see Fig. 11, bottom) is also defined separately for each node for all software unit_j of component instances. At the location Listen, the receiver Rx listens the incoming message[track_k] from each track_k by synchronizing with the Output[track_k]! broadcasts. If the message is destined for a specific unit_j, it is enqueued in q, and waits to be dequeued and decoded in order to synchronize with the unit_j process on the adequate software action Soft_Action_i[unit_j].

Pattern 5 (**Attack detection).** In case where a malicious_fault is detected, Rx synchronizes on the channel Attack[unit_j] with the body of unit_j. Attacks could be only notified by the non-critical bodies. However, they should be handled, by the control critical jobs, as security faults.

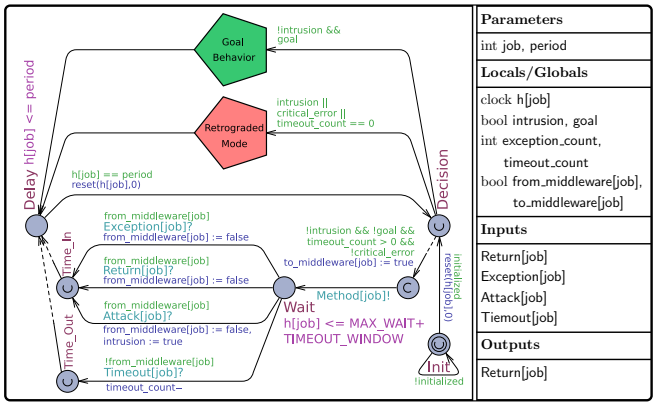

FIGURE 12. Pattern *Periodic job*.

Pattern 6 (**Timeout).** In case the message is not delivered within MAX_WAIT, Rx erases the content of q, and synchronizes on the channel Timeout[unit_j] with the body of unit_j in order to handle timing faults as explained in Subsection V-B, and illustrated by the scenario of Fig. 5. The delay MAX_WAIT is assumed to be equal to MAX_LATENCY with an extra time window for the message delivery on Soft_Action_i[unit_j].

Pattern 7 (**Periodic job).** A periodic job can be designed in UPPAAL as an automaton complying to the pattern showed in Fig. 12. The parameter job is used to instantiate several times the body for interactions if needed. After initialization, job takes a Decision at the beginning of each periodic life cycle until reaching – if possible – a desired goal behavior. Three main causes can prevent it to reach that goal: the detection of an intrusion, reaching the maximum number of authorized timeouts (timeout_count is decremented until reaching 0), or being aware of a critical_error. In this case, the retrograded operation mode is activated if the situation is deemed critical. Otherwise, job executes its nominal body by calling local or remote methods, and performing local computations resulted from their return and exception events. In Fig. 12, we show the simple interactive scenario of calling a remote Method, if a Return value is received, job exploits it to check whether the goal behavior is reached or not. By catching Exception, job – according to its impact on the system operation – decides if critical_error should be set or not. If an Attack is detected, intrusion is set to true. If MAX_WAIT is elapsed and nothing is received, the job synchronizes on the channel Timeout and decretments timeout_count.

In order to reach timeout scenarios in the job traces during model-checking, a TIMEOUT_WINDOW extra delay is added to the invariant of location Wait. During this delay Rx and job may synchronize on Timeout (see Subsection VI-E). The global variables from_middleware[call] and to_middleware[call] are used to force the output (resp. input) actions of job to synchronize with the transmitter Tx (resp. the receiver Rx) corresponding actions, instead of synchronizing directly with the method bodies as depicted in Fig. 11.

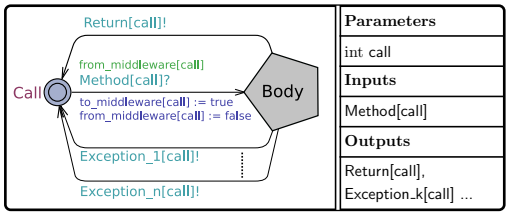

FIGURE 13. Pattern *Remote method*.

Pattern 8 (Remote method). A remote method (see Fig. 13) can be designed as an automaton where every trace starts by an input action on the channel Method[call], and finishes by an output action on the channel Return[call] or one of the channels Exception_k[call]. The input parameter call is used to instantiate several time the method body for open use.

Methods are considered to be stateless one-shot programs, and are invoked if needed by their environments. No infinite loop behaviors are expected by running a method, and consequently possible traces of its behavior should be constrained by the presence of a *final* (implicit) state equal to the initial one. This choice was made to guarantee that an instance of the method body could be instantiated (called) for a limited number of times: we cannot expect unlimited method calls in formal design, which may be the case in real executions.

C. FAIRNESS CONDITIONS

The query language of UPPAAL utilizes a subset of CTL to define properties. It consists of *state* and *trace* formulae. State formulae describe individual states, whereas trace formulae quantify over traces of the model semantics (see definitions 2 and 3). Trace formulae are classified into *safety*, *reachability* and *liveness*. Safety properties state that "something bad will never happen". These properties are usually formulated positively, something good is invariantly true, *i.e.,* given ϕ a state formula ϕ, $\mathbf{A}\Box\phi$ expresses that ϕ should be true in all states. Reachability states that "something expected to happen will occur under some circumstances", *i.e.,* there is a trace starting from the initial state, such that ϕ eventually holds (denoted $\mathbf{E}\Diamond\phi$). Liveness states that "something good will eventually happen". Liveness could be simply expressed by the formula $\mathbf{A}\Diamond\phi$. A more useful form is the *leads to* property $\phi \rightsquigarrow \psi$: whenever ϕ is satisfied, then eventually the state formula ψ will be satisfied [12].

Since we are only interested in the correctness of execution starvation-free fair traces when verifying distributed systems. A system model in UPPAAL is usually reduced to a collection of deadlock-free processes (specified in a NTA), and we need to consider only those traces in which each process resumes *infinitely often*. *Unfair* traces, in which one process always executes while preventing the others to resume (starvation), are ignored. Since only finite state semantics are considered, *fairness* requires that some of their states are visited infinitely often in every fair computation (especially states that should be reached to satisfy liveness properties) [15].

By default, the fairness conditions cannot be expressed in CTL *i.e.,* CTL cannot express that some proposition should eventually hold on all fair traces [14]. A common solution to this problem is to modify the CTL semantics. Traditionally, the semantics of CTL formulae is defined with respect to a labeled state-transition graph $M=(S,R,P)$ consisting of: a finite set of states S, a transition relation $R\subseteq S\times S$, and an assignment $P:S\rightarrow 2^{\mathbf{AP}}$ of atomic propositions in $\mathbf{AP}$ to states. To express fairness, the semantics M is extended by $\mathcal{F}\subseteq 2^S$. A trace of M is fair iff for each state set $F\in\mathcal{F}$, there are infinitely many positions on it at which some state of F appears. The new semantics is the same as that of CTL except that trace quantifiers range over only fair traces [15].

Exercising model-checking in UPPAAL led us to reason about fairness to guarantee liveness properties. Given that the query language of UPPAAL does not support fair semantics, we need to intervene at the level of the model by acting on TAs transitions activation. We first present an extension of CTL syntax to express fairness conditions. It is actually a subset of CTL* [18]. Then, practical examples are given to show how to constrain UPPAAL models to meet fairness.

CTL extension

The semantics $\mathfrak{S}(A_{1..n})$ of NTA (Definition 3) and $\mathfrak{S}(A)$ of single timed automata (Definition 2) are equivalent except the presence of composite states $\bar{l}\in\bar{\Upsilon}$ in the first. Otherwise we have always *time* or *channel* transitions in both of them. Thence, we found the following notations on Definition 2 by assuming A to be an NTA. A *trace* σ derived from $\mathcal{R}$ is an infinite sequence $s_0[a_0]s_1..s_k[a_k]s_{k+1}...$ of transitions $s_ka_k\mapsto s_{k+1}\in\mathcal{R}$ where $k\rightarrow\infty$. A transition of $sa\mapsto t\in\mathcal{R}$ covered by σ is written $s[a]t:\sigma$. We write $s:\sigma$ when the state s is traversed by σ. We consider infinite traces since $\mathfrak{S}(A)$ is assumed to be deadlock-free. The traces of $\mathfrak{S}(A)$ combine together in an infinite labeled transition tree $\mathfrak{T}(\mathfrak{S}(A))$ obtained by unfolding $\mathfrak{S}(A)$ from the initial state s_0. We denote by σ_s any trace of $\mathfrak{T}(\mathfrak{S}(A))$ starting from a state s in $\mathfrak{T}(\mathfrak{S}(A))$ reachable from s_0. We denote by $\dot{s}\in S$ the *image* of s in $\mathfrak{S}(A)$. We define by $\mathfrak{vi}_\sigma^\infty$ an infinite sequence of monotonic natural indexes $k_1..k_l...$ of states traversed by σ such that the difference between any two successive indexes of the sequence is variable *i.e.,* for all k_ik_{i+1}, k_jk_{j+1} where $i\neq j$, $k_{i+1}-k_i$ might be equal or different from $k_{j+1}-k_j$. We write $k:\mathfrak{vi}_\sigma^\infty$ if the index k is in the sequence $\mathfrak{vi}_\sigma^\infty$.

We have already provided the needed notations to define our CTL extension: for any trace $\sigma=s_0[a_0]s_1..s_k[a_k]s_{k+1}...$ of $\mathcal{M}=\mathfrak{T}(\mathfrak{S}(A))$, any state s reachable in $\mathcal{M}$, and any well-formed CTL formulae ϕ and ψ, we consider the following inductive semantic assertions (the original syntax and semantics of CTL are given in [14], [15]); $\models$ means "satisfy".

Trace formulae

$$\mathcal{M},\sigma\models\Diamond\phi \;\Leftrightarrow\; (\exists s:\sigma)\,\mathcal{M},s\models\phi$$

$$\mathcal{M},\sigma\models\Box\phi \;\Leftrightarrow\; (\forall s:\sigma)\,\mathcal{M},s\models\phi$$

$$\mathcal{M},\sigma\models\Box\Diamond\phi \;\Leftrightarrow\; (\exists \mathfrak{vi}_\sigma^\infty)(\forall k:\mathfrak{vi}_\sigma^\infty)\,\mathcal{M},s_k\models\phi$$

$$\mathcal{M},\sigma\models\Diamond\Box\phi \;\Leftrightarrow\; (\exists k{>}0)(\forall j{\geq}k,i{<}k)\,\mathcal{M},s_j\models\phi\wedge\mathcal{M},s_i\not\models\phi$$

State formulae

$$\begin{aligned}
\mathcal{M}, s \models \mathbf{A}\Diamond\phi &\Leftrightarrow (\forall\sigma_s)\, \mathcal{M}, \sigma_s \models \Diamond\phi \\
\mathcal{M}, s \models \mathbf{A}\Box\phi &\Leftrightarrow (\forall\sigma_s)\, \mathcal{M}, \sigma_s \models \Box\phi \\
\mathcal{M}, s \models \mathbf{E}\Diamond\phi &\Leftrightarrow (\exists\sigma_s)\, \mathcal{M}, \sigma_s \models \Diamond\phi \\
\mathcal{M}, s \models \mathbf{E}\Box\phi &\Leftrightarrow (\exists\sigma_s)\, \mathcal{M}, \sigma_s \models \Box\phi \\
\mathcal{M}, s \models \phi \rightsquigarrow \psi &\Leftrightarrow \mathcal{M}, s \models \mathbf{A}\Box(\phi \Rightarrow \mathbf{A}\Diamond\psi) \\
\mathcal{M}, s \models \mathbf{A}\Box\Diamond\phi &\Leftrightarrow (\forall\sigma_s)\, \mathcal{M}, \sigma_s \models \Box\Diamond\phi \\
\mathcal{M}, s \models \mathbf{A}\Diamond\Box\phi &\Leftrightarrow (\forall\sigma_s)\, \mathcal{M}, \sigma_s \models \Diamond\Box\phi \\
\mathcal{M}, s \models \mathbf{E}\Box\Diamond\phi &\Leftrightarrow (\exists\sigma_s)\, \mathcal{M}, \sigma_s \models \Box\Diamond\phi \\
\mathcal{M}, s \models \mathbf{E}\Diamond\Box\phi &\Leftrightarrow (\exists\sigma_s)\, \mathcal{M}, \sigma_s \models \Diamond\Box\phi
\end{aligned}$$

Note that the first five state formulae are already defined in CTL [14], [15]. We introduce the last four formulae; they could be read by the following statements:

- $\mathcal{M}, s \models \mathbf{A}\Box\Diamond\phi$: for all traces σ_s of $\mathcal{M}$ starting from s, ϕ is *infinitely often* satisfied by an infinity of the states traversed by σ (may be not all of them) *i.e.,* ϕ is said here to be *unconditionally fair* [50];
- $\mathcal{M}, s \models \mathbf{A}\Diamond\Box\phi$: for all traces σ_s of $\mathcal{M}$ starting from s, ϕ is *eventually always* satisfied from a state r posterior to s in σ_s;
- $\mathcal{M}, s \models \mathbf{E}\Box\Diamond\phi$: $\Box\Diamond\phi$ holds only for some traces σ_s;
- $\mathcal{M}, s \models \mathbf{E}\Diamond\Box\phi$: $\Diamond\Box\phi$ holds only for some traces σ_s.

Strong, weak and eventual fairness conditions

In order to express fairness conditions using the previous state formulae, we need to introduce the facts of enabling and firing actions in $\mathcal{M}$ as state formulae, Let $a \in \mathbb{R}^+ \cup \Sigma$, the formula $\mathsf{Enabled}(a)$ is a state predicate asserting that a is enabled. $\mathcal{M}, s \models \mathsf{Enabled}(a)$ iff $(\exists t \in S)\, t = \mathcal{R}(\dot{s}a)$. $\mathsf{Fired}(a)$ is a state predicate asserting that a is fired. $\mathcal{M}, s \models \mathsf{Fired}(a)$ iff $(\exists r, \sigma_r)\, \mathcal{M}, r \models \mathsf{Enabled}(a) \wedge r[a]s : \sigma_r$. We write $\mathcal{M} \models \phi$ if $(\forall s)\, \mathcal{M}, s \models \phi$.

Definition 4. *$\mathcal{M}$ is* strongly fair *for an action a iff there is no trace σ in $\mathcal{M}$ in which a is infinitely often enabled by $\mathfrak{S}(A)$ but never infinitely often fired in σ. This statement is written formally $\mathcal{M} \models \mathrm{SF}(a)$ where $\mathrm{SF}(a)$ is defined as follows.*

$$\mathrm{SF}(a) \triangleq \mathbf{A}(\Box\Diamond\mathsf{Enabled}(a) \Rightarrow \Box\Diamond\mathsf{Fired}(a))$$

Definition 5. *$\mathcal{M}$ is* weakly fair *for an action a iff there is no trace σ in $\mathcal{M}$ in which a is always enabled by $\mathfrak{S}(A)$ but never infinitely often fired in σ. This statement is written formally $\mathcal{M} \models \mathrm{WF}(a)$ where $\mathrm{WF}(a)$ is defined as follows.*

$$\mathrm{WF}(a) \triangleq \mathbf{A}(\Diamond\Box\mathsf{Enabled}(a) \Rightarrow \Box\Diamond\mathsf{Fired}(a))$$

We also consider the case where the action a is eventually enabled but never fired, hence we introduce a new condition called *eventual fairness*.

Definition 6. *$\mathcal{M}$ is* eventually fair *for an action a iff there is no trace σ in $\mathcal{M}$ in which a is occasionally enabled by $\mathfrak{S}(A)$ but never fired in σ. This statement is written formally $\mathcal{M} \models \mathrm{EF}(a)$ where $\mathrm{EF}(a)$ is defined as follows.*

$$\mathrm{EF}(a) \triangleq \mathbf{A}(\Diamond\mathsf{Enabled}(a) \Rightarrow \Diamond\mathsf{Fired}(a))$$

$\mathcal{M}$ is assumed to be unconditionally fair for time actions $\tau \in \mathbb{R}^+$ *i.e.,* the axioms (ufe_τ) and (uff_τ) below are assumed to hold in $\mathcal{M}$.

$$\frac{}{\mathcal{M} \models \mathbf{A}\Box\Diamond\mathsf{Enabled}(\tau)}\ (\mathrm{ufe}_\tau)$$

$$\frac{}{\mathcal{M} \models \mathbf{A}\Box\Diamond\mathsf{Fired}(\tau)}\ (\mathrm{uff}_\tau)$$

Proving liveness properties under fairness conditions

A liveness formula $\phi \rightsquigarrow \psi$ holds under a strongly fair $\mathcal{M}$ for an action $a \in \Sigma$ if the rule $(\mathrm{sf}_\rightsquigarrow)$ is backward satisfied.

$$\frac{\begin{array}{ll} 1) & \mathcal{M} \models \mathsf{Fired}(a) \rightsquigarrow \psi \\ 2) & \mathcal{M} \models \phi \rightsquigarrow \mathsf{Enabled}(a) \\ 3) & \mathcal{M} \models \mathbf{A}\Box\Diamond\mathsf{Enabled}(a) \end{array}}{\mathcal{M} \models \mathrm{SF}(a) \Rightarrow \mathcal{M} \models \phi \rightsquigarrow \psi}\ (\mathrm{sf}_\rightsquigarrow)$$

Whenever it is satisfied, ϕ inevitably leads to traces along which a is certainly infinitely often enabled (premises 2 and 3). Since $\mathcal{M}$ is strongly fair for a and firing a inevitably leads to ψ (premise 1), then $\phi \rightsquigarrow \psi$ holds.

The formula $\phi \rightsquigarrow \psi$ holds under a weakly fair $\mathcal{M}$ for an action $a \in \Sigma$ if the rule $(\mathrm{wf}_\rightsquigarrow)$ is backward satisfied.

$$\frac{\begin{array}{ll} 1) & \mathcal{M} \models \mathsf{Fired}(a) \rightsquigarrow \psi \\ 2) & \mathcal{M} \models \phi \Rightarrow \mathsf{Enabled}(a) \\ 3) & \mathcal{M} \models \mathbf{E}\Diamond\Box\mathsf{Enabled}(a) \end{array}}{\mathcal{M} \models \mathrm{WF}(a) \Rightarrow \mathcal{M} \models \phi \rightsquigarrow \psi}\ (\mathrm{wf}_\rightsquigarrow)$$

Whenever ϕ is satisfied, then the action a is enabled, which means that in the traces where a is eventually always enabled, ϕ holds permanently from the states at which a is enabled for the first time along these traces (premises 2 and 3) *i.e.,* $\rightsquigarrow$ in the second premise of Rule $(\mathrm{sf}_\rightsquigarrow)$ becomes $\Rightarrow$. Since $\mathcal{M}$ is weakly fair for a and firing a inevitably leads to ψ (premise 1), then $\phi \rightsquigarrow \psi$ holds.

The formula $\phi \rightsquigarrow \psi$ holds under an eventually fair $\mathcal{M}$ for an action $a \in \Sigma$ if the rule $(\mathrm{ef}_\rightsquigarrow)$ is backward satisfied.

$$\frac{\begin{array}{ll} 1) & \mathcal{M} \models \mathsf{Fired}(a) \rightsquigarrow \psi \\ 2) & \mathcal{M} \models \phi \rightsquigarrow \mathsf{Enabled}(a) \\ 3) & \mathcal{M} \models \mathbf{A}\Diamond\mathsf{Enabled}(a) \end{array}}{\mathcal{M} \models \mathrm{EF}(a) \Rightarrow \mathcal{M} \models \phi \rightsquigarrow \psi}\ (\mathrm{ef}_\rightsquigarrow)$$

If the action a is eventually enabled in some fragments of all the traces (premise 3), then ϕ when satisfied will lead inevitably to those fragments (premise 2). Since a is also fired from those fragments and inevitably leads to ψ (premise 1), then $\phi \rightsquigarrow \psi$ holds.

We can also define rules for liveness formulae of the form $\mathbf{A}\Diamond\phi$. The rule $(\mathrm{sf}_{\mathbf{A}\Diamond})$ states that $\mathbf{A}\Diamond\phi$ holds under a strongly fair $\mathcal{M}$ for an action $a \in \Sigma$ if a is infinitely often enabled in all the traces (premise 2). Since firing a inevitably leads to ϕ (premise 1), then $\mathbf{A}\Diamond\phi$ holds.

$$\frac{\begin{array}{ll} 1) & \mathcal{M} \models \mathsf{Fired}(a) \rightsquigarrow \phi \\ 2) & \mathcal{M} \models \mathbf{A}\Box\Diamond\mathsf{Enabled}(a) \end{array}}{\mathcal{M} \models \mathrm{SF}(a) \Rightarrow \mathcal{M} \models \mathbf{A}\Diamond\phi}\ (\mathrm{sf}_{\mathbf{A}\Diamond})$$

The rule ($\text{wf}_{\mathbf{A}\Diamond}$) states that a formula $\mathbf{A}\Diamond\phi$ holds under a weakly fair $\mathcal{M}$ for an action $a \in \Sigma$ if a is eventually always enabled in all the traces (premise 1). Since firing a inevitably leads to ϕ (premise 2), then $\mathbf{A}\Diamond\phi$ holds.

$$\frac{\begin{array}{ll}1) & \mathcal{M} \models \text{Fired}(a) \rightsquigarrow \phi \\ 2) & \mathcal{M} \models \mathbf{A}\Diamond\Box\text{Enabled}(a)\end{array}}{\mathcal{M} \models \text{WF}(a) \Rightarrow \mathcal{M} \models \mathbf{A}\Diamond\phi}\ (\text{wf}_{\mathbf{A}\Diamond})$$

The rule ($\text{ef}_{\mathbf{A}\Diamond}$) states that a formula $\mathbf{A}\Diamond\phi$ holds under a eventually fair $\mathcal{M}$ for an action $a \in \Sigma$ if a is eventually enabled in all traces over finite fragments (premise 1). Since a will be inevitably fired from those fragments and leads to ϕ (premise 2), then $\mathbf{A}\Diamond\phi$ holds.

$$\frac{\begin{array}{ll}1) & \mathcal{M} \models \text{Fired}(a) \rightsquigarrow \phi \\ 2) & \mathcal{M} \models \mathbf{A}\Diamond\text{Enabled}(a)\end{array}}{\mathcal{M} \models \text{EF}(a) \Rightarrow \mathcal{M} \models \mathbf{A}\Diamond\phi}\ (\text{ef}_{\mathbf{A}\Diamond})$$

We would like to mention that our method to define these rules were mainly inspired from [50] and the studies about fairness conditions in the Temporal Logic of Actions (TLA), the logic behind the TLA$^+$ language [17], [51].

Expressing fairness in UPPAAL models

After this theoretical body, we show now how the previous fairness conditions could be expressed in UPPAAL models.

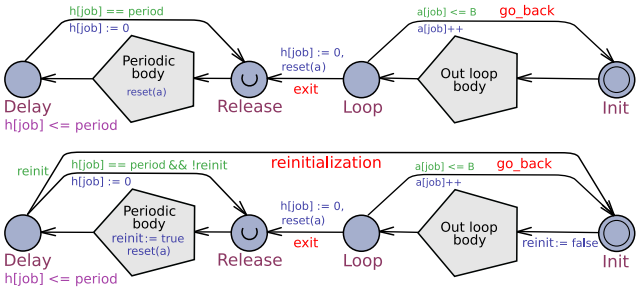

FIGURE 14. Patterns *Eventual fair* (top) and *Strong fair* (bottom) *periodic jobs* with "Out loop" bodies: updates may applied in "Periodic body". Labels "exit", "go_back" and "reinitialization" are action names.

If the pattern periodic job of Fig. 14 (top) is instantiated in more then one process without the guard a[job] <= B of go_back, a starvation problem occurs by infinitely running Out loop body of job while preventing the others to resume. Therefore, $\mathbf{A}\Diamond\phi$ and $\phi \rightsquigarrow \psi$ where ϕ and ψ are state predicates of being resp. in the locations Init and Delay cannot be satisfied for all the processes. To bypass this problem, we should constrain the model by an eventual fairness condition $\text{EF}(\text{exit})$ on the action exit. We use for that an integer array a (for *active*). Init cannot be reached if a[job] exceed a bound B after firing repeatedly go_back. At some point, exit is fired and a is reset to 0 by some process. Consequently, execution is alternated fairly between all processes. If Init is reached from Delay infinitely often for reinitialization reasons ($\mathbf{A}\Box\Diamond$ reinit holds), then exit is infinitely often enabled and fired, and the periodic job is constrained implicitly by the strong fairness condition $\text{SF}(\text{exit})$ (see Fig. 14, bottom).

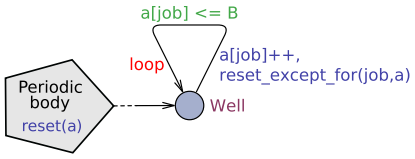

FIGURE 15. Pattern *Weak fair loop job*; in some periodic activities one might expect a "Well" state that keeps the process active while being in a default state.

If the pattern loop behavior of Fig. 15 is instantiated in more then one process without the guard a[job] <= B on the loop, a starvation problem occurs by reaching the state Well in a given process, and firing always the action loop. In this case, the model should be constrained by weak fairness $\text{WF}(\text{loop})$ on the action loop. When the bound B is reached, the current process is leaved and a is reset to 0 by another process. The update reset_except_for(job,a) prevents resetting a[job] by the current process job each time loop is fired.

Finally, we impart that UPPAAL ensures timed locations (with clock invariants) to be fairly visited. This induces unconditionally fair timed transitions in the resulted trace tree, which is compliant with axioms (ufe_τ) and (uff_τ). Fairness conditions are only used to constrain the activation of channel actions from untimed locations.

D. CASE STUDY: DESIGN PROCESS AND MEASURES

The UPPAAL models of the periodic jobs *Control* and *Flight* resp. of *GCS* and *Drone*, are available for download[2]. The models are not exhaustive but sufficiently representative to put into practice the previous design patterns, and to verify endogenous contracts. Our design was divided to two submodels `Drone_Flight.xml` and `GCS_Control.xml` and constrained by urgent and commited locations (when possible) in order to reduce the state space, prevent false counterexamples, and speedup model-checking.

We consider a model with two drones and one GCS. The GCS model is *multi-task* and *multi-call*: the job *Control* is instantiated twice one per connected drone; several method calls can be made simultaneously by drones in the node GCS. However, the drone model is *mono-task* and *mono-call*: the job *Flight* is instantiated once in a drone node, and a method cannot be invoked simultaneously several times since only the GCS calls methods from drones, and calls are synchronous. The network communication band is composed of three half-duplex tracks: messages cannot transit simultaneously on a track regardless of their directions (*GCS* to *Drone* or *Drone* to *GCS*). The middleware Rx receives data from each subsystem in a circular FIFO buffer.

Using these models, we analyzed by prediction the safety behavioral requirements of the GCS and drones when remote communication timeouts, malicious attacks, and functional errors occur. The properties were defined according to the verification process detailed in Subsection VI-E. The model's properties took 45 mn to be verified. The model-checking were performed using the 64 bits version 4.1.19 of UPPAAL running on a Core i7-4710MQ machine at 2.5 GHz.

[2] `https://github.com/mouelhis/uppaals`

E. VERIFICATION PROCESS

We opted for a verification-by-contract process making use of the text-based BCL language [41] to define the endogenous resilience contracts. A BCL contract $\mathrm{C} \triangleq (\mathsf{A} : \bar{\mathrm{a}} \mid \mathsf{G} : \bar{\mathrm{g}})$ is an assume/guarantee statement where $\bar{\mathrm{a}}$ is a vector of assumptions and $\bar{\mathrm{g}}$ is a vector of the guarantees. Assumptions constrains whether a specification meets the guarantees. The BCL rationals should be as simple as possible to reason about requirements and their dependencies. They are very useful to decompose, make easier hard verification processes, and to exhaustively define the formal properties.

For sake of genericity, we present BCL contracts and CTL properties as patterns like in Subsection VI-B. We start by a first contract on the worst-case call blocking time (WCBT) allowed for remote method invocations.

$\mathrm{WCBT} \triangleq (\mathsf{A} : \mathrm{awn} \mid \mathsf{G} : \mathrm{mcr})$
$\mathrm{awn} \triangleq$ Always [wireless connection is reliable]
$\mathrm{mcr} \triangleq$ Everytime [a job calls a method remotely]
Then [a response is received]
Within [x tu (time unit)]

Contract WCBT stipulates that the guarantee on the remote call responsiveness (mcr) is relative to the availability and reliability of the wireless network (assumption awn). Since awn cannot be specified in UPPAAL, it is assumed to be always true. According to Pattern 7 and Fig. 12, this contract can be specified for a periodic job by the following three CTL formulae (typewritten in the query language of UPPAAL):

```
WCBT1 ≜ A[] Time_In imply h[job] <= MAX_WAIT
WCBT2 ≜ A[] Time_Out imply h[job] > MAX_WAIT
WCBT3 ≜ Decision --> Delay
```

where **WCBT1** is a safety property stating that always in every trace of the job, being in the location Time_In means that a return value is received within MAX_WAIT (h[job] is reset to 0 before the method call). Otherwise, the job response is time-out (h[job] $>$ MAX_WAIT): Time_Out is reached (**WCBT2**), and timeout_count is decremented. The liveness property **WCBT3** ensures that waiting is not infinitely blocking and the periodic activity always happen (see Fig. 5). `-->` represents $\rightsquigarrow$.

$\mathrm{WCET} \triangleq (\mathsf{A} : \mathrm{hao}, \mathrm{wcbt} \mid \mathsf{G} : \mathrm{et})$
$\mathrm{hao} \triangleq$ Always [hardware is reliable]
$\mathrm{wcbt} \triangleq$ Always [call blocking time satisfies WCBT]
$\mathrm{et} \triangleq$ Everytime [job periodic cycle is released]
Then [job terminates its periodic activity]
Within [p tu]

Contract WCET is about the worst-case execution time of periodic jobs. It stipulates that timing predictability (guarantee et) of periodic executions relies on the reliability of the embedded platform and components (battery, sensors and actuators, etc) of the drone (assumption hao) and the guarantee of Contract WCBT (assumption wcbt). Since hao cannot be specified, it is assumed to be always true. Pattern 7 (Fig. 12) natively supports Contract WCET: periodic jobs have timed locations Delay (with a clock invariant h[job] <= period) and guard h[job] == period to exit Delay when period (the WCET upper bound) is elapsed.

$\mathrm{Endogenous} \triangleq (\mathsf{A} : \mathrm{hao}, \mathrm{mcr} \mid \mathsf{G} : \mathrm{atr}, \mathrm{cer}, \mathrm{ttr})$
$\mathrm{atr} \triangleq$ Everytime [intrusion is detected]
Then [retrograded mode is activated]
Immediately
$\mathrm{cer} \triangleq$ Everytime [critical error happens]
Then [retrograded mode is activated]
Immediately
$\mathrm{ttr} \triangleq$ Everytime [timeout happens several times]
Then [retrograded mode is activated]
Immediately

Contract Endogenous stipulates that whatever a malicious attack, a critical error, or recurrent timeouts happen, then the retrograde mode is activated (resp. defined by the guarantees atr, cer and ttr). This contract can be specified for Pattern 7 (Fig. 12) by the following CTL properties.

```
Malicious_Attack ≜ E<> intrusion
Critical_Error   ≜ E<> critical_error
Max_Timeouts     ≜ E<> timeout_count == 0
Endogenous       ≜ (intrusion ||
                    critical_error ||
                    timeout_count == 0) -->
                    Retrograded_Mode
```

The first three properties should be checked to ensure that there is some traces where **intrusion**, **critical_error**, and **timeout_count == 0** hold eventually so as **Endogenous** is guaranteed to be checked on real traces. If only some of them hold, the liveness property **Endogenous** may hold with no trace to check for the others which is not representative.

Jobs *Flight* and *Control* resp. of the components *Drone* and *GCS* were checked to be deadlock-free (**A[] not deadlock**), and safe according to the verification process herein. The *GCS* model was verified with respect to eventual and weak fairness constraints (see Subsection VI-C).

VII. ANALYSIS OF EXOGENOUS RESILIENCE

Exogenous resilience is relative to the drone's presence and operation in its ambient urban environment. Concretely, it should be checked by simulation or by target tests on the signals acquired from sensors and/or applied on actuators. In order to analyze the exogenous resilience in offline, we need ideally a virtualization of the drone in the urban environment to have a better understanding of safety issues, and rigorously reason about them. A 3D model of an imaginary city was constructed for that purpose using the toolbox Blender.

This model describes a navigation scenario as depicted in Section III and Fig. 1. A scenario is an ordered sequence of *frames*, the Blender's fundamental time unit, that can reach fractions of a second depending on the frame rate. Since it is practically impossible to meticulously define the animations frame per frame, *interpolation* is applied to solve this issue. Blender allows the definition of pertinent positions or *keys* (3D coordinates) for objects at specific frames along which it creates the 3D animation using interpolation algorithms. Three different interpolations are possible: *constant*, *linear* and *Bézier*. We use the Bézier interpolation since it produces smooth and more realistic animations.

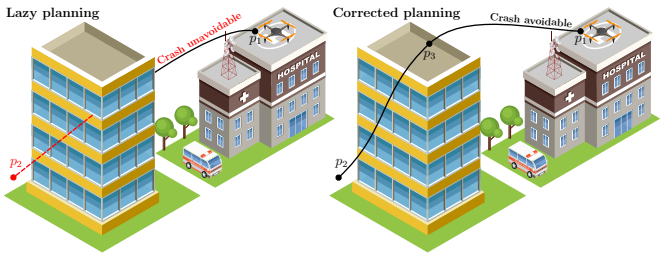

FIGURE 16. Path planning; *lazy planning* (left): the first planned path p_1p_2 of the drone between p_1 and the sub-target position p_2 is computed in a lazy way and leads to a crash with an obstacle building; *corrected planning* (right): GCS recompute a new path $p_1p_3p_2$ that passes over the building to avoid the crash.

As stated in Subsection V-C, GCS has the role to control drone missions remotely, and to performs all heavy computations likely to consume energy, vital for flights. Since it has a better awareness of the urban environment, GCS performs the important operation of *path planning*. By considering the current position of the drone, the destination and the urban cartography, GCS computes by iterative correction the path that the drone needs to follow until the destination. Fig. 16 illustrates the planning process on a sub-path between two positions. After finding the next position p_2 to be reached, GCS acts first lazily and considers the simplest path p_1p_2 (Fig. 16, left). Next, it discovers (by anticipated verification) that p_1p_2 intersects with a building (only permanent obstacles are considered). Finally, it recomputes the path by adding an intermediate position p_3 above the building (Fig. 16, right).

$$
\begin{array}{lcl}
\text{Exogenous} & \triangleq & (\mathsf{A}: \text{ok}, \text{oaw} \mid \mathsf{G}: \text{cf}, \text{da}) \\
\text{ok} & \triangleq & \mathsf{Always}\ [\ \text{obstacles are buildings}\ ] \\
\text{oaw} & \triangleq & \mathsf{Always}\ [\ \text{GCS is aware of obstacles}\ ] \\
\text{cf} & \triangleq & \mathsf{Always}\ [\ \text{collisions are avoided}\ ] \\
\text{da} & \triangleq & \mathsf{Always}\ [\ \text{drone altitude} < \text{MA meters}\ ]
\end{array}
$$

Contract Exogenous states that a flight path is collision-free (guarantee cf), and the altitude coordinates of the path positions are always bounded by a maximum altitude MA fixed in meters (guarantee da). We assume that obstacles are restricted to buildings, and GCS is aware of their positions and dimensions (assumptions ok and oaw). The Blender built-in plugin CAT is used to specify and verify the LTL properties of the iterative path planning process. We use LTL since the 3D animations on which the properties are checked are one-line frame sequences.

The above contract can be simply translated to the following LTL property pattern:

$$\mathsf{Safe}(\bar{p}) \triangleq \Box\left[\bigwedge_{p_i \in \bar{p}, B \in \mathcal{B}} (p_i \notin B \wedge p_i.altitude < \text{MA})\right]$$

where $\bar{p}$ is the path, and $\mathcal{B}$ is the set of building obstacles B_i defined in the 3D city model as parallelepiped objects. Video animations of lazy and corrected flight paths are available under the following links along with CAT windows to show whether the LTL properties are met or not:

- lazy planning: `https://youtu.be/MdaZhvlz_l8`;
- corrected planning: `https://youtu.be/5cW6PBzoIj8`.

VIII. CONCLUSION DISCUSSIONS AND PERSPECTIVES

We are currently facing a growth in systems complexity, with increasingly advanced technologies. CPSs, as part of Industry 4.0, are subject to this technological evolution. Embedded computers in CPSs perform complex tasks to control sophisticated physical processes in environments, that are becoming more and more ambient, open and hazardous. In addition, CPSs are required to be resilient to errors and disturbances, and able to recover with the minimum costs. Modeling such systems is difficult especially in critical contexts with regards to their hardware, software and networking architectures, and event unpredictability of their environments.

The contributions of this article suit this context. They are articulated around a predictive approach to analyze resilience in an urban drone rescue system with ground remote control. It is based on a distributed object-oriented component-based software architecture. The structure of an object-oriented component, from our viewpoint, is new compared to the CCM specifications [42] (resembling most to ours) and other definitions [52], in which periodic jobs, data listeners, and references to component instances are implicit features. We dealt with endogenous and exogenous resilience aspects of the case study at both design and verification. Endogenous resilience is reflected in the system capability of processing internal functional and timing faults, and resisting against cyberattacks. Exogenous resilience relies on the system's ability to safely operate in its ambient environment.

To this effect, we have defined a formal methodology to predict the system's behavior by abstraction, and to verify its resilience properties. We used UPPAAL networks of timed automata to model the distributed interoperability between subsystems, and to analyze its endogenous resilience under an abstract networking model. In order to show the UPPAAL modeling approach in an elegant way, and to avoid hindering the body of the article with the final big models, we opted for the presentation of generic design patterns for the main software and middleware units of the system's component architecture (see Fig. 7). Model-checking in UPPAAL led us to also reason about fairness conditions to validate liveness properties. Under this particular scope, we have considered a subset of the CTL* [18] syntax as an extension of CTL to express strong, weak, and eventual fairness. Strong and weak fairness conditions were well studied for different kinds of temporal logics [17], [18], [50]. The novelty in this article includes the introduction of the eventual fairness condition, the rules to prove liveness properties, and the style by which the formal material was presented.

Our design was mainly shaped to properly tackle a key problem of CPSs which comes down to this question: *how to explicitly analyze the endogenous interactive behavior of the system parts early during the design phase to ensure high integrity in safety-critical scenarios*? By focusing on this key issue, we are solving one of the major problems of classic reactive approaches widely adopted in critical systems industry, which to our knowledge still has no complete and rigorous tooled solutions (see Section I).

Regarding the choice of UPPAAL, we claim that this tool is the best suited to our time constrained predictive analysis of CPSs endogenous resilience for the following reasons: 1) its specification language, based on TAs, is GUI-powered and adequate to design time-aware models; 2) it supports a subset of the C language to specify state transition updates on clocks and discrete variables, and to pre-chew implementation; 3) its query language used to define properties is based on CTL, which is a branching time logic quantifying over computation trees rather than individual traces; 4) it also provides a GUI for simulation and counterexample analysis; 5) it supports statistical model-checking (UPPAAL-SCM) [53].

Other free toolboxes comparable to UPPAAL are available. To the best of our knowledge, none of them combine all the above features. In this paragraph, we quote two tools that we think are the closest to UPPAAL. The first tool is PAT (Pattern Analysis Toolkit)[3]: an enhanced model-checker for real-time and concurrent systems. Model-checking under PAT is based on LTL, which is less expressive than CTL. Counterexamples can be graphically visualized, but unfolded in long traces if the model is complex. Under UPPAAL, counterexamples simulation is more user-friendly since traces can be animated directly on TAs. The second is TAPAAL[4]: an efficient model-checker of timed-arc Petri nets. Like UPPAAL, it offers a GUI for design, simulation, and CTL-based model-checking. However, besides the fact that design using timed-arc Petri nets is less intuitive compared to TAs, TAPPAL has no background language to specify transition behaviors over discrete variables, which is useful for implementation purposes.

Powerful commercial proof tools are also available for industrial safety-critical applications: Prover Certifier (Prover Technology AB), Frama-C (CEA), Modeling Rules Verifier and Optimizer (SafeRiver), Systerel Smart Solver (Systerel), etc. These tools make use of inductive proof engines based on efficient symbolic SAT/SMT solvers: MathSAT[5], CVC4[6], Glucose[7], Lingeling[8], etc. This work could be extended by introducing a complete formal design and verification framework for resilience analysis based on the above solvers.

Our predictive formal analysis of exogenous resilience is embodied within a virtual modern city 3D (Blender) model. To meet the exogenous contracts, flight paths are synthesized by repetitive LTL model-checking using the CAT tool (a Blender plugin). The goal behind this virtualization is that the GCS follows a similar approach when computing flight paths for drones on real city 3D maps. We deliberately did not give more materials about this part in Section VII because what was left is restricted to useless implementation details. Unfortunately, we are not able to share the source code of the CAT plugin and the Blender 3D models due to the intellectual property clauses promulgated by CEA.

[3] https://pat.comp.nus.edu.sg
[4] http://www.tapaal.net
[5] http://mathsat.fbk.eu/
[6] http://cvc4.cs.stanford.edu/web/
[7] http://www.labri.fr/perso/lsimon/glucose/
[8] http://fmv.jku.at/lingeling/

We see three main directions for future works. The first is to introduce a top-down component-based theory of our approach. One possible trail is to revisit and question the theory of timed I/O automata in [54], particularly the "input-enabled" hypothesis (all input actions are enabled from any location in an I/O timed automaton). This assumption is contradictory with our approach and quite inconsistent, from our perspective, with object-oriented design. The second is to compare, with statistical model-checking, existing network protocols and UAV fleet models using UPPAAL-SCM, using our predictive analysis approach. The third is prototyping real drones based on the software architecture of Fig. 7 and the implementation approach discussed in [10].

ACKNOWLEDGMENT

We thank Manolo Dulva Hina (ECE Paris.Lyon, INSEEC U.) for his help to improve the quality of the manuscript.

REFERENCES

[1] W. Sangiovanni-Vincentelli, A. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems," European Journal of Control, vol. 18, no. 3, pp. 217–238, 2012.
[2] Y. Y. Haimes, "On the definition of resilience in systems," Risk Analysis, vol. 29, no. 4, pp. 498–501, 2018.
[3] S. Amin, G. Schwartz, and S. S. Sastry, "Challenges for Control Research: Resilient Cyber-Physical Systems," IEEE CSS, The Impact of Control Technology, 2nd ed. (report), 2014.
[4] N. Halbwachs, Synchronous Programming of Reactive Systems. Springer Science+Business Media, Springer US, 1993.
[5] A. Gamati, Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification, 1st ed. Springer-Verlag, New York, 2009.
[6] J.-L. Boulanger, F.-X. Fornari, J.-L. Camus, and B. Dion, SCADE: Language and Applications, 1st ed. Wiley-IEEE Press, 2015.
[7] Radio Technical Commission for Aeronautics (RTCA), "DO-178C: Software Considerations in Airborne Systems and Equipment Certification," 2012, Standard.
[8] European Committee for Electrotechnical Standardization (CENELEC), "EN 50128: Railway applications – Communications, signalling and processing systems," 2001, Standard (rev. 2011).
[9] International Organization for Standardization, "ISO 26262: Road vehicles – Functional safety," 2004, Standard (rev. 2012).
[10] S. Mouelhi, D. Cancila, and A. Ramdane-Cherif, "Distributed object-oriented design of autonomous control systems for connected vehicle platoons," in Proc. of 22nd Int. Conf. on Engineering of Complex Computer Systems (ICECCS). IEEE, 2017, pp. 40–49.
[11] C. Szyperski, J. Bosch, and W. Weck, "Component-oriented programming," in Proc. of the European Conference on Object-Oriented Programming (ECOOP). Springer, Berlin, Heidelberg, 1999, pp. 184–192.
[12] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in Proc. of 4th Int. Schl. on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT, ser. LNCS, no. 3185. Springer, Berlin, Heidelberg, 2004, pp. 200–236.
[13] R. Alur and D. L. Dill, "A theory of timed automata," Theoretical Computer Science, vol. 126, no. 2, pp. 183–235, 1994.
[14] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in Proc. of the Works. on Logic of Programs, ser. LNCS. Springer, Berlin, Heidelberg, 1982, pp. 52–71.
[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," ACM Trans. Program. Lang. Syst., vol. 8, no. 2, pp. 244–263, 1986.
[16] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," Information and Computation, vol. 111, no. 2, pp. 193–244, 1994.
[17] L. Lamport, Specifying Systems: The TLA$^+$ language and tools for hardware and software engineers. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[18] E. A. Emerson and J. Y. Halpern, "Sometimes and Not Never Revisited: On branching versus linear time temporal logic," J. ACM, vol. 33, no. 1, pp. 151–178, 1986.
[19] A. Pnueli, "The temporal logic of programs," in Proc. of 18th Symp. on Foundations of Computer Science (SFCS). IEEE, 1977, pp. 46–57.
[20] P. Pettersson, "Formal methods applied in industry-on the commercialisation of the UPPAAL tool," in Proc. of 35th Computer Software and Applications Conf. (COMPSAC). IEEE, 2011, pp. 450–451.
[21] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL implementation secrets," in Proc. of Int. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, ser. LNCS. Springer, Berlin, Heidelberg, 2002, pp. 3–22.
[22] H. E. Jensen, K. G. Larsen, and A. Skou, "Scaling up UPPAAL," in Proc. of Int. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, ser. LNCS. Springer, Berlin, Heidelberg, 2000, pp. 19–30.
[23] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," Electronic Notes in Theoretical Computer Science, vol. 89, no. 4, pp. 543–560, 2003.
[24] K. R. M. Leino, "Automating induction with an SMT solver," in Porc. of Int. Wshp. on Verification, Model Checking, and Abstract Interpretation, ser. LNCS. Springer, Berlin, Heidelberg, 2012, pp. 315–331.
[25] E. Hollnagel, R. L. Wears, and J. Braithwaite, "From Safety-I to Safety-II: a white paper," 2015, The Resilient Health Care Network.
[26] E. Hollnagel, Safety-I and Safety-II: The past and future of safety management. CRC Press, 2018.
[27] A. Bagnato, R. K. Bíró, D. Bonino, C. Pastrone, W. Elmenreich, R. Reiners, M. Schranz, and E. Arnautovic, "Designing swarms of cyber-physical systems: the h2020 cpswarm project," in Proc. of the Computing Frontiers Conf. ACM, 2017, pp. 305–312.
[28] D. Ratasich, O. Höftberger, H. Isakovic, M. Shafique, and R. Grosu, "A self-healing framework for building resilient cyber-physical systems," in Proc. of 20th IEEE Symp. on Real-Time Distributed Computing (ISORC). IEEE, 2017, pp. 133–140.
[29] Q. Zhu and T. Başar, "Robust and resilient control design for cyber-physical systems with an application to power systems," in Proc. of 50th IEEE Conf. on Decision and Control and European Control Conf. (CDC-ECC). IEEE, 2011, pp. 4066–4071.
[30] G. Denker, N. Dutt, S. Mehrotra, M.-O. Stehr, C. Talcott, and N. Venkatasubramanian, "Resilient dependable cyber-physical systems: a middleware perspective," Journal of Internet Services and Applications, vol. 3, no. 1, pp. 41–49, 2012.
[31] I. Jovanov and M. Pajic, "Relaxing integrity requirements for attack-resilient cyber-physical systems," arXiv preprint arXiv:1707.02950, 2017.
[32] A. Cardenas, S. Amin, and S. Sastry, "Secure control: Towards survivable cyber-physical systems," in Proc. of 28th Int. Conf. on Distributed Computing Systems (workshops) ICDCS. IEEE, 2008, pp. 495–500.
[33] B. Vogel-Heuser, C. Diedrich, D. Pantförder, and P. Göhner, "Coupling heterogeneous production systems by a multi-agent based cyber-physical production system," in Proc. of 12th IEEE Int. Conf. Industrial Informatics (INDIN'14). IEEE, 2014, pp. 713–719.
[34] M. Lezoche and H. Panetto, "Cyber-physical systems, a new formal paradigm to model redundancy and resiliency," Enterprise Information Systems, vol. 0, no. 0, pp. 1–22, 2018.
[35] M. Lezoche, E. Yahia, A. Aubry, H. Panetto, and M. Zdravković, "Conceptualising and structuring semantics in cooperative enterprise information systems models," Computers in Industry, vol. 63, no. 8, pp. 775–787, 2012.
[36] A. Claesson, D. Fredman, L. Svensson, M. Ringh, J. Hollenberg, P. Nordberg, M. Rosenqvist, T. Djarv, S. Österberg, J. Lennartsson et al., "Unmanned Aerial Vehicles (drones) in out-of-hospital-cardiac-arrest," Scandinavian journal of trauma, resuscitation and emergency medicine, vol. 24, no. 1, p. 124, 2016.
[37] Object Management Group (OMG), "OMG Systems Modeling Language SysML," 2017.
[38] The LML Steering Committee Charter, "Lifecycle modeling Language (LML), Specification," 2015.
[39] B. Meyer, "Applying Design by Contract," Computer, vol. 25, no. 10, pp. 40–51, 1992.
[40] C. A. R. Hoare, "An axiomatic basis for computer programming," Communications of the ACM, vol. 12, no. 10, pp. 576–580, 1969.
[41] O. Ferrante, R. Passerone, A. Ferrari, L. Mangeruca, and C. Sofronis, "BCL: A compositional contract language for embedded systems," in Proc. of the IEEE Int. Conf. on Emerging Technology and Factory Automation (ETFA). IEEE, 2014, pp. 1–6.
[42] International Organization for Standardization, "ISO/IEC 19500: Information technology – Object Management Group – Common Object Request Broker Architecture (CORBA)," 2003, Standard (rev. 2012).
[43] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," IEEE Trans. on Computers, vol. 44, no. 1, pp. 73–91, 1995.
[44] AdaCore, "High-integrity object-oriented programming in Ada," 2016, v1.4.
[45] ——, "SPARK 2014 Reference manual," 2014.
[46] AdaCore, "PolyORB User's guide," 2003.
[47] Y. Zeng, R. Zhang, and T. J. Lim, "Wireless communications with Unmanned Aerial Vehicles: opportunities and challenges," IEEE Communications Magazine, vol. 54, no. 5, pp. 36–42, 2016.
[48] IEEE Standards Association, "802.11p: Amendment 6 to the IEEE 802.11 for Wireless Access in Vehicular Environments (WAVE)," 2010, Standard.
[49] European Telecommunications Standards Institute, "EN 302 663: Access layer specification for Intelligent Transport Systems operating in the 5 GHz frequency band," 2012, Standard.
[50] C. Baier and J.-P. Katoen, Principles of Model-Checking (Representation and Mind Series). The MIT Press, 2008.
[51] L. Lamport, "The TLA$^+$ Hyperbook," 2015, Hyperlink book.
[52] C. Szyperski, Component Software: Beyond Object-Oriented Programming, 2nd ed. Boston, MA, USA: Addison-Wesley Professional, 2002.
[53] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "UPPAAL-SMC tutorial," Int. Journal on Software Tools for Technology Transfer, vol. 17, no. 4, pp. 397–415, 2015.
[54] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, "Timed I/O Automata: A complete specification theory for real-time systems," in Proc. of the 13th ACM Int. Conf. on Hybrid Systems: Comp. and Cont. (HSCC'10). New York, NY, USA: ACM, 2010, pp. 91–100.

SEBTI MOUELHI received his M.S. degree in computer science, in 2007, from the University of Lorraine, Nancy, France, and the Ph.D. degree in computer science, in 2011, from the University of Franche-Comté, Besançon, France. He was a Post-doctoral Researcher at INRIA, Grenoble, France, in 2011. He was an R&D Engineer at SafeRiver, Montrouge, France, for about three years since late 2012. In Summer 2015, he was an Engineer in safety assurance at ALSTOM Transport, Saint-Ouen, France. Since late 2015, he is an Associate Professor in Embedded Systems at ECE Paris.Lyon, member of INSEEC U., Paris, France. His research interests are in the topics of software engineering, formal methods, embedded systems, real-time, and automatic control.

MOHAMED-EMINE LAAROUCHI received the "Diplôme d'Ingénieur" from ENICarthage (École Nationale d'Ingénieurs de Carthage), Tunisia, in 2015, and the M.S. degree in computer science (specialty: Design of Industrial Complex Systems) from École Polytechnique, Palaiseau, France. He is currently pursuing the Ph.D. degree with the French Alternative Energies and Atomic Energy Commission (CEA), Gif-sur-Yvette, France, and Télécom SudParis, Evry, France, since late 2016. His research interests include software engineering, embedded systems, and the safety analysis of autonomous cyber-physical systems.

DANIELA CANCILA received the Laurea M.S. degree in philosophy from the University of Roma (La Sapienza), Rome, Italy, and the "Diplôme d'études Approfondies" (DEA), in discrete mathematics and theoretical computer science, from, Institut de Mathématiques de Luminy, Marseille, France. She received the Ph.D. degree in theoretical computer science, in 2004, from the University of Udine, Italy. Since 2008, she is a Research Engineer at the French Alternative Energies and Atomic Energy Commission (CEA), Gif-sur-Yvette, France. Her research interests are mainly in the topics of model-based design and safety analysis of cyber-physical systems.

HAKIMA CHAOUCHI received the Ph.D. degree from Sorbonne University, Paris, France, and the King's College of London, UK, in 2004. She is a Full Professor at Télécom SudParis, member of Institut Mines-Télécom, Evry, France, where she is leading a research group on emerging networks and services. She has authored and co-authored more than 100 international scientific publications. Her research activities are in the topics of wireless and mobile communication, emerging 5G and LTE narrow band, IoT technologies, service discovery and composition, network security and privacy. She is actually member of WG2 Innovation Ecosystem of the European AIOTI Alliance, initiated by the European Commission in 2015, and a scientific advisor on Research, and Innovation Strategy at the French Ministry of High Education and Research.