The Role of Parts in the System Behaviour

Davide Di Ruscio[1], Ivano Malavolta[2], Patrizio Pelliccione[3]

[1]University of L'Aquila,
Department of Information Engineering, Computer Science and Mathematics, Italy
[2]Gran Sasso Science Institute, L'Aquila (Italy)
[3]Chalmers University of Technology | University of Gothenburg,
Department of Computer Science and Engineering, Sweden
davide.diruscio@univaq.it, ivano.malavolta@gssi.infn.it, patrizio.pelliccione@gu.se

Abstract. In today's world, we are surrounded by software-based systems that control so many critical activities. Every few years we experiment dramatic software failures and this asks for software that gives evidence of resilience and continuity. Moreover, we are observing an unavoidable shift from stand-alone systems to systems of systems, to ecosystems, to cyber-physical systems and in general to systems that are composed of various independent parts that collaborate and cooperate to realise the desired goal.
Our thesis is that the resilience of such systems should be constructed compositionally and incrementally out of the resilience of system parts. Understanding the role of parts in the system behaviour will (i) promote a "divide-and-conquer strategy" on the verification of systems, (ii) enable the verification of systems that continuously evolve during their life-time, (iii) allow the detection and isolation of faults, and (iv) facilitate the definition of suitable reaction strategies. In this paper we propose a methodology that integrates needs of flexibility and agility with needs of resilience. We instantiate the methodology in the domain of a swarm of autonomous quadrotors that cooperate in order to achieve a given goal.

1 Introduction

Increasingly, we are surrounded by software-based systems that provide services that are increasingly becoming ineluctable elements of everyday life. Examples of the served domains are transportation, telecommunication, health-care, etc. According to Marc Andreessen[1] "*Software is eating the world*"[2]: for instance, the major music companies are software companies: Apple's iTunes, Spotify and Pandora. As another example, today, the world's largest bookseller is a software company: Amazon.

Software controls so many critical activities, and thus, at societal level, software is required to provide evidence of resilience and continuity. Every few years we experiment dramatic software failures, e.g.: (i) the Knight Capital Group announced on

[1] Mr. Andreessen is co-founder and general partner of the venture capital firm Andreessen-Horowitz, which has invested in Facebook, Groupon, Skype, Twitter, Zynga, and Foursquare, among others. He is also an investor in LinkedIn and co-founded Netscape, one of the first browser companies.

[2] `http://goo.gl/FCGord`

August 2, 2012 that it lost $440 million when it sold all the stocks it accidentally bought the day before due to a software bug; (ii) in 10 years, about 10,000,000 cars have been recalled due to software-related problems by Ford, Cadillac, General Motors, Jaguar, Nissan, Pontiac, Volvo, Chrysler Group, Honda, Lexus, Toyota and others.

This motivates a growing interest, both industrial and academic, in techniques that can provide evidence about the software resilience. Modern systems are no more stand-alone; they are composed of several sub-systems, often independent each other but that collaborate so to realize the system goal. Examples of these systems are systems of systems, ecosystems, and cyber-physical systems characterized by high dynamicity that might affect each subsystem, the way their are interconnected and the environment in which they live. Therefore, methods that aim at assuring the resilience of such systems cannot neglect the system structure. This means that the resilience of systems that are composed of various independent parts that collaborate and cooperate to realise the desired goal should be constructed compositionally out of the resilience of system parts. Each single part of the system should understand under which conditions it can ensure a correct behaviour and should be robust to failures and to malicious attacks.

This paper proposes a methodology that builds on lessons learned in integrating flexible and agile development processes with the need of resilience. The methodology starts from the dichotomy that exists between building systems and expressing system specifications. On one hand, systems are built out of existing components; they are suitably composed and integrated so to realize a system goal [1]. On the other hand, the specification of the system is often defined only at the system level without providing details of what behaviour it is expected from each single part composing the system [2, 3]. A fundamental aspect is to decompose the system goal in a set of sub-system specifications that describe the role of the system parts in the system behaviour: this enables incremental and compositional verification and permits to maintain the desired degree of resilience despite of system evolution and adaptation to a continuous changing environment. These concepts are better explained by focusing on the domain of a swarm of autonomous quadrotors that collaborate to realize a common goal. In this example each quadrotor is an autonomous node that behaves independently by each other; however, each node realises specific tasks that concur to the realization of the mission, which is the system's goal.

Roadmap of the paper. The remainder of the paper is structured as follows: Section 2 describes development processes that integrate agility and flexibility with the need of resilience. Section 3 discusses the importance of understanding the contribution of single parts for ensuring the resilience of the entire system. Section 4 illustrates the development process in the context of a swarm of autonomous quadrotors. Related works in goal decomposition and compositional verification techniques are described in Section 5. Section 6 concludes the paper and outlines some perspective work.

2 Agility and Resilience

Modern software systems are composed of several different parts that are independent but that collaborate to realize the goal of the sysyem. Such systems are inherently dynamic since they need to operate in a continuously changing environment and must be

able to evolve and quickly adapt to different types of changes, even unanticipated, while guaranteeing the resilience today's users expect [1]. As highlighted in [4–6], uncertainty is becoming a first-class concern of today's software systems. Uncertainty calls for strategies able to determine under what conditions desired goals will be achieved and behavioral invariance will be preserved [4–6].

The development of safety-critical systems is traditionally associated to waterfall and V-lifecycles that involve huge effort in planning, specification, and documentation. These development processes are typically rigorous and enable software certification; however, the adoption of these processes makes it difficult to manage requirements volatility and the introduction of new technologies. Moreover, these processes can lead to substantial costs in producing and maintaining documentation.

Agile methods are very attractive to software companies since they promise flexibility and speed. Agile methods focus more on continuous process management and code-level quality than classic software engineering process models [7], however, their adoption in safety-critical domains poses difficulties and challenges. Research on the use of agile methods in safety-critical domains is still in its infancy. However, initial works demonstrate that agile methods are not inherently at odds with the requirements of safety critical development [8]. At the same time, it is evident that agile methods cannot be directly applied as they are in a safety-critical domain that requires certification. Agile methodologies might define as necessary some activities that are enablers of resilience for safety-critical systems [7]. These are some examples:

Up-front design and incremental development of safety arguments [8]: the rationale of the up-front design is to enable hazard analysis, safety analysis, and certification. Moreover, iterative and incremental development should construct not only software, but should be devoted also to the construction of the arguments that the software is acceptably safe: each release should produce software acceptably safe.

Safety-by-Design [7]: strongly typed interfaces that accept request events and returns request responses where a state machine internal to the component determines whether the request can be satisfied in the component's current state [7]. This supports the notion of intrinsic safety [9], i.e., no component can be in an unexpected state. This would enable also simulation and verification, to see if the composition of all components works properly.

Lightweight traceability of requirements at development time [7]: requirements management in lockstep with code management. Developers write potential requirements in a wiki. A collaborative review process analyses, refines, and decides which requirements are rejected (stored for potential revision in the future), and which requirements are accepted. Accepted requirements are introduced into the product backlog; the team selects the requirements suitable for implementation in the current sprint. Synchronization between requirements management and code management is ensured.

Identify high-risk system properties that need special handling [10]: integration of formal specification into agile methods for high-risk system properties, before they are implemented in the next sprint. Less critical parts of the system are implemented in parallel without any formality.

3 Understanding the role of parts to ensure resilience

A key point for ensuring resilience is having a precise description of the system specification. As said by Leslie Lamport: "*A specification is a written description of what a system is supposed to do. Specifying a system helps us understand it. It's a good idea to understand a system before building it, so it's a good idea to write a specification of a system before implementing it.*" [11]. A specification is an abstraction of the system aimed at describing some aspects and at ignoring others. Specifications describe statements of intents whose satisfaction requires the cooperation of parts of the system [12]; unfortunately, often specifications do not state explicitly the role of the system's parts. The importance of having a precise specification of system's parts is well recognized and it is still a challenging open research direction [13, 14]. These sub-specifications represent the technical requirements on the system sub-parts and assumptions on their environment. We call these sub-specifications *operational requirements*, according to [3].

Figure 1.a highlights the importance of decomposing the system specification in a set of sub-specifications. The figure is completed by observing that the way of producing software is changing: it is increasingly produced according to a certain goal and by integrating existing software [1]. In other words, the focus while producing software, is moving towards integrator means (e.g., architectures, connectors, integration patterns) that ease the collaboration and integration of existing services in a resilient way [1].

Figure 1.b shows an overview of the proposed methodology that builds on lessons learned in integrating agility and flexibility with the need of resilience. The methodology consists of a number of tasks and artefacts as described below.

Upfront specification: As suggested by the work presented in [8], an important aspect to ensure resilience is having an upfront design that represents the specification of the high-level goal of the system. Having a full upfront design of a complex system is often difficult and impractical due to the high degree of uncertainty and to the overhead to be invested to maintain the design during the system lifetime. Thus, the upfront specification can also be defined incrementally as new information is available or as new requirements are considered.

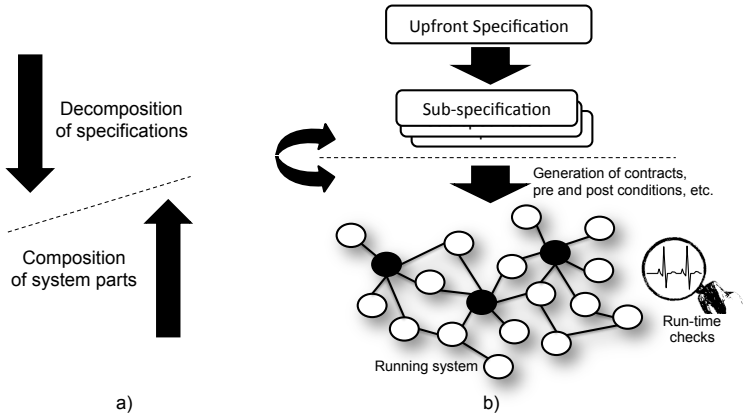


Fig. 1. a) Decomposition and Composition; b) Agility and Resilience

Sub-specification: The next step is to refine and map the high-level goals into precise specifications of software behaviour [15], called sub-specification in Figure 1.b. Such a specification consists of operational requirements, contracts or pre- and post- conditions so to enable proofs at run-time. Depending on the system and on the considered development process, pre- and post- conditions or contracts might be chosen. Pre- and post-conditions can be interpreted as the assumptions that the component/sub-system makes on its environment in order to guarantee its properties [16]. According to the design-by-contract paradigm [13], contracts instead encode operational requirements in target programming languages, i.e., pre-, post-conditions and invariants are written by using the syntax of the used programming language, thus they are specified at a lower level of abstraction. It is important to remark that generating sub-specifications from the upfront one is a complex task that cannot be always automated. In the domain of choreographed systems we managed to automate the generation of sub-specifications [17], even though this is not always the case since it depends on the considered application domain and on the complexity of the modeled software system.

Generation of contracts, pre- and post- conditions: As discussed in [7], contracts and pre- and post- conditions might assume the form of state machines and they can be used to perform verification at run-time but also to regulate the behaviour of the system during its execution according to the specification. In particular, the idea is to associate a state machine encoding the (sub-)specification to its corresponding component/sub-system.

Run-time checks: According to the performed actions and to the sent and received messages, the obtained state machine is animated and the computational state of the real system is thus explicitly represented. Thus, once a message that differs from what expected is received, the message is blocked and suitable recovery strategies might be adopted. This promotes the intrinsic safety [9] notion that aims at protecting each system part from strange behaviours. This will permit to isolate potential problems, or prevent from malicious attacks. A similar approach has been proposed in [18] where the encoded state machine was used to ensure that the final code behave according to a checked and verified software architecture behavioural model. Moreover, each system part should explicitly deal with exceptional situations, for instance by implementing the idealized component model [19, 20]. According to this model, when the *normal behavior* of a component raises an exception, called local exception, its exception handling part is automatically invoked. If the exception is successfully handled, the component resumes its normal behaviour, otherwise an *external exception* is signalled to the enclosing context. Two are the possible types of exceptions caused by external invocations: *failure exceptions* due to a failure in processing a valid request and *interface exceptions* due to an invalid service request.

It is important to remark that *running system* and *specifications* in Fig, 1 need to be kept aligned. This is of crucial importance since as previously explained the specification becomes part of the implementation; therefore coherence and consistency between specification and running system must be ensured. It is important to note that here we are not promoting to have the system code aligned with models as typically done in the case of code generation and round-trip engineering. Here we are proposing to have requirements and system properties properly identified, made explicit, and also encoded

in the running system. The encoded properties should be considered as an implementation of a monitor, which is executed in parallel with the system, and which controls both the (sub-)system behaviour and its interactions with the environment. The property, encoded as a state machine, should be defined at a proper abstraction level so to not compromise the performance of the system, and the usability with unnecessary details.

4 Ensuring resilience in a swarm of autonomous quadrotors

In this section we discuss the application of the methodology shown in Fig. 1 in the domain of civilian missions executed by swarms of autonomous quadrotors. In particular, Section 4.1 introduces the domain of interest and outline the different kinds of resilience that it is possible to have when using swarms of autonomous quadrotors. Section 4.2 introduces the FLYAQ [21] software platform consisting of languages and tools that have been conceived to manage resilient quadrocopters according to the methodology shown in Fig. 1. Section 4.3 discusses a concrete application of FLYAQ.

4.1 Overview and resilience

A quadrotor, also called a quadrotor helicopter or quadcopter, is a multicopter that is lifted and propelled by four rotors. Quadrotors are classified in the family of UAV (Unmanned aerial vehicle), e.g., drones without a human pilot on board that can be either controlled autonomously by computers in the vehicle, or under the remote control of a pilot on the ground or in another vehicle. Quadrotors are programmed with a very low level language or providing very basic primitives; this issue introduces an "error-prone" process even for experienced users and asks developers strong expertise about the dynamics and the technical characteristics of the used quadrotor. It also makes the specification of missions unaffordable for a non-technical user, which has typically a very poor (if any) experience in software programming. Therefore, the specification of a mission is already difficult when considering a single quadrotor and it becomes even more complex when dealing with missions involving a swarm of quadrotors. Then, it emerges the need for software engineering approaches and methodologies especially tailored to develop and maintain a swarm of quadrotors. We will refer in this paper to a platform that has been developed to eliminate this technological barrier. The platform is called FLYAQ [21] and it allows non-technical operators to straightforwardly define civilian missions of swarms of flying drones at a high level of abstraction, thus hiding the complexity of the low-level and flight dynamics-related information of the drones.

The overall system is composed of a swarm of independent quadrotors, a ground station that is the hub to perform reconfigurations and that receives information from the run-time execution, and laptops or tablet that are used by the end-user to define and monitor the mission. Currently the ground station is controlling everything, then the mission is centrally coordinated by the ground station that sends instructions to the quadrotors composing the swarm. However, in this paper we move towards a decentralization of the mission with the idea of distributing the computation to quadrotors; in this way resilience is distributed on each quadrotor, thus we are not relying on the

communication channels anymore, rather the resilience of the entire system is built on top of the resilience of single parts.

Before discussing strategies adopted to ensure resilience, we further analyse and describe resilience in the domain of a swarm of quadrotors. Specifically, we distinguish between resilience of a single quadrotor, and resilience of the entire system.

Resilience of a single quadrotor: A quadrotor should be resilient and tolerant to both software and hardware faults. Quadrotors will be more and more required to adhere to standards, rules, and regulations just like an aircraft. Certification affects software, hardware, as well as those platforms that are used to produce, test, or devices and sensors to be installed on quadrotors.
Hardware: this involves (i) redundancy of engines, rotors, transmission and communication system, and each other critical part, (ii) emergency system able to execute a safe emergency landing in case of malfunctioning quadrotors, and (iii) using parachutes.
Software: this involves strategies to make the quadrotor resilient to (i) errors in the code, such as wrong mission design (impossible to realize the mission with the available resources), or wrong code, (ii) inaccurate or unpredictable context - automatic obstacles avoidance or reaction to strong wind, (iii) attacks from malicious entities, e.g., spoofing attack - a program might masquerade as another by falsifying data and thereby trying to gain an illegitimate advantage in terms of communication.

Resilience of the entire system: The overall system has the objective of realizing the mission even when facing changes and unforeseen failures at run-time. Static checks can be made at design time, once the mission has been designed (e.g., to check the feasibility of the mission according to the available drones, weather conditions, quadrotors equipments, etc.) [21]. However, the challenge here is to construct a reconfiguration engine which is able to react to changes and problems at run-time, and then maintaining a desired degree of resilience at run-time. More details about the resilience of the entire system might be found in [21]. Here in the following we focus on the resilience of a single quadrotor from the software perspective, while we rely on the hardware resilience provided by the constructor of drones.

4.2 Building a resilient quadrocopter: software perspective

By starting from an upfront specification, which is the goal of the mission, FLYAQ generates the sub-specifications for each actor of the system. In FLYAQ, the upfront specification is provided by means of the Monitoring Mission Language (MML) and the Context Language (CL) [22]. MML is a domain-specific language for representing a monitoring mission as an ordered sequence of tasks to be performed by the swarm. Basically, an MML task is a predefined sub-mission which can be executed by a set of drones, it can be performed until completion or it can be interrupted by an event. Examples of tasks provided by MML include: covering a specific area by acquiring images on a grid with a given graphic resolution, looking for some visual marker within a specific area, keeping a certain position on a given geographical point, coming back to a geographical point defined as the home of the mission, etc. CL models contains the specification of contextual information of the mission. For example, in FLYAQ context models contain information about obstacles, no fly zones, emergency places where to

land in case of emergency. Moreover, the FLYAQ platform uses an intermediate language called Quadrotor Behaviour Language (QBL); this language represents detailed instructions for each single quadrotor of the swarm, and it is used at run-time for instructing each single quadrotor. From an high-level point of view, the beahaviour of each drone is composed of a set of movements (e.g., take off, land, go to a specific geographical point), each of them having a set of pre- and post-actions, which are executed by the drone before (or after) the movement. Examples of actions include: taking a picture, starting or stopping a video streaming activity, sending a message to the ground station, sending a message to another drone, etc.

Sub-specifications are instead described as state machines, in which each state represents an internal computational state of the quadrotor, while a transition from one state to another can represent either (i) a message that is sent or received by the quadrotor, or (ii) an operation that is internal to the quadrotor, such as moving to a precise geographical point, hovering for two seconds, taking a picture, etc. A formal definition of sub-specification is given in Definition. 1.

Definition 1 (Sub-specification). *Let Msg be a finite set of messages that a quadrotor can exchange, let Act be a finite set of internal actions that a quadrotor can perform, like take off, landing, movements, taking pictures, etc. A sub-specification is a tuple $Spec{=}(N,S,s_0,A,T)$ where:*

- *N is the state machine name.*
- *S is a finite set of states.*
- *$s_0{\in}S$ is the initial state.*
- *A is a finite set of transition labels.*
- *T is a finite set of transitions. A transition $t{\in}T$ is a tuple $t{=}(s_s,s_t,label,type)$ where:*
  - *$s_s{\in}S$ is the source state;*
  - *$s_t{\in}S$ is the target state;*
  - *$label{\in}A$ is the transition label;*
  - *$type{=}\{!m_1;?m_2;\tau\}$ is the transition type: send, receive, or internal, respectively, where $m_1,m_2{\in}Msg$, $\tau{\in}\ Act$;*

The (sub-)specification is contained in the software component the resides in each quadrotor. As shown in Figure 2 the software component is composed of four main parts: *Controller*, *(Sub-)Specification*, *Normal behaviour*, and *Exceptional behaviour*. To explain how these components will exploit the specification at run-time we distinguish between four different situations:
− *Received message*: when a message is received from the context (which can be the ground station or another quadrotor), before interpreting the message the controller checks the specification to check if the message is expected. If the message is allowed in the current computational state, then the message is forwarded to the normal part of the component, otherwise the message is forwarded to the exceptional part.
− *Sent message*: before authorizing the sending of a message the controller checks the conformity of the behaviour with the specification.
− *Performing an action*: before performing an internal action, the controller checks the status of the specification to see if the action is allowed in the actual computational

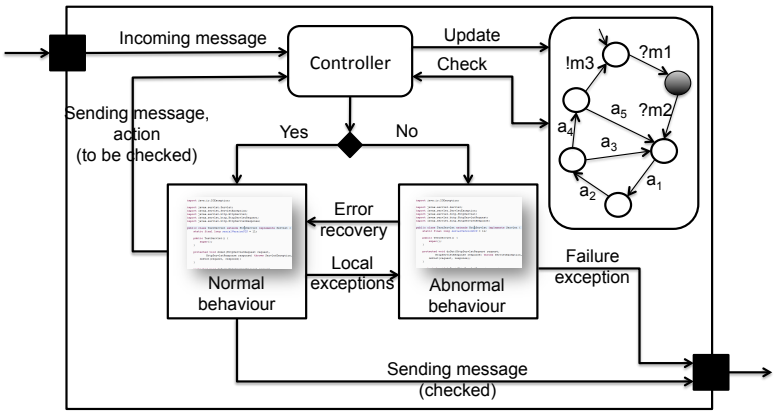


Fig. 2. Quadrotor software controller

state. In the case of positive answer the specification is updated by changing the current state to the state target of the transition that fires, in the case of negative answer, the exception part of the node is called into action.
– *Performed action*: when an action is performed a transition is fired in the specification and a target state is reached. Then the configuration of the quadrotors is analysed to identify possible failures and to check whether actual status of sensors and devices (like battery consumption) diverge from what expected. In case of problems the exceptional part of the node has to manage and possibly solve the problem.

4.3 Example

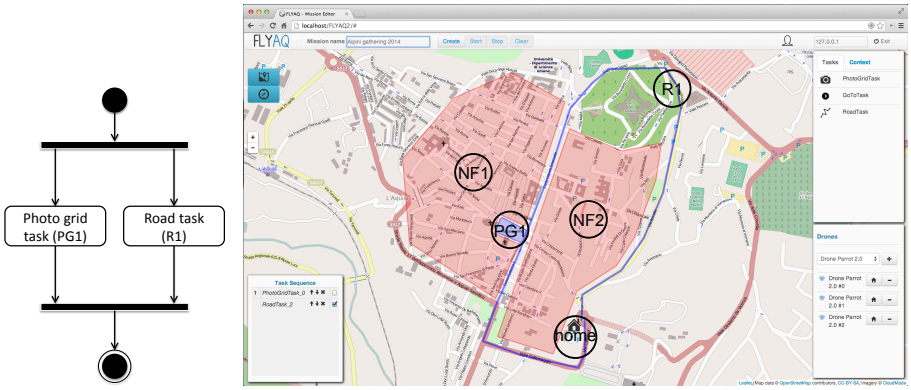
In this section we put in practice the approach on a concrete example. It describes a mission for monitoring a large public event for security reasons. The event (`http://www.laquila2015.it/`) is the 88th national gathering of the Alpini, an elite mountain warfare military corps of the Italian Army. The event will take place in L'Aquila (Italy) the 15th, 16, and 17th of May 2015. A crowd of more than 70,000 participants took part of the past edition of the Alpini's national gathering, and the same number of participants is expected to come to the city of L'Aquila in 2015. Clearly, having such a large number of participants in a three-days event poses many challenges to the organizers, mainly about security and safety issues. In this context, FLYAQ will play a relevant role since it will enable organizers to have a swarm of autonomous quadrotors that continuously monitor the city center during the event. In the following sections we describe the upfront specification of the monitoring mission that will be executed every hour, and its corresponding sub-specifications.

Upfront specification: Figure 3 shows the *MML mission model* for monitoring of the annual Alpini gathering event. More specifically, Figure 3(a) shows the tasks of the mission as a UML activity diagram, and Figure 3(b) shows the same tasks of the UML activity diagram as an overlay to a geographical map of the city. The mission is composed of two tasks that will be performed in parallel, namely:

– *Photo grid task (PG1)*: this task is performed above the main square of the city (see the blue rectangle in Figure 3(b)), where there will be the main talks from the institutions, musical concerts, etc. The photo grid task identifies a virtual grid within

the area, each cell of the grid having a size of ten meters. The drones executing the task will fly over each cell of the grid at an altitude of 25 meters, and then will take a picture of the area directly below them; thus, the result of the photo grid task is a set of high-definition pictures fully covering the main square of the city. The photo grid task can be assigned to multiple drones.

– *Road task (R1)*: this task refers to a polyline corresponding to the main streets in the central area of the city (see the blue polyline in Figure 3(b)). This task can be assigned to a single drone, that will (i) fly along the polyline at an altitude of 25 meters, and (ii) take a picture every 200 meters along the polyline.



(a) Activity diagram (b) Geographical areas and waypoints defined in the FLYAQ editor

Fig. 3. The Alpini event monitoring mission

The mission will be realized by three drones that will be positioned in a large parking lot close to the city center (see the *home* circle in Figure 3(b).b): two drones will take care of executing the photo grid task, whereas a single drone will execute the road task independently. As previously explained, the FLYAQ platform allows the user to define contextual information about the mission. In this example, the *CL context model* contains two *no fly* areas called NF1 and NF2. Those areas correspond to the zones within the city center with residential buildings. Each drone performing the mission cannot fly over no-fly areas under any possible conditions. As discussed in Section 4.2, the FLYAQ platform is able to automatically generate a *QBL behavioural model* from the mission model. Figure 4 graphically shows the behavioural model corresponding to the example mission. The behavioural model is composed of three behavioural traces, each of them representing the behaviour of a single drone. A behavioural trace is composed of a set of atomic movements that the drone will perform to complete its portion of mission[3]. In the following we describe the kinds of the used behavioural actions:

– Start: represents the initial setup of the drone;
– Stop: represents the final movement used to end any sequence of movements;
– TakeOff: represents the take off operation of the drone; in the behavioural model each take off is complete when the drone reaches an altitude of two meters;

[3] Tasks are not part of the behavioural model, we show the mission task representing each sub-behaviour trace in Figure 4 only for the sake of clarity.

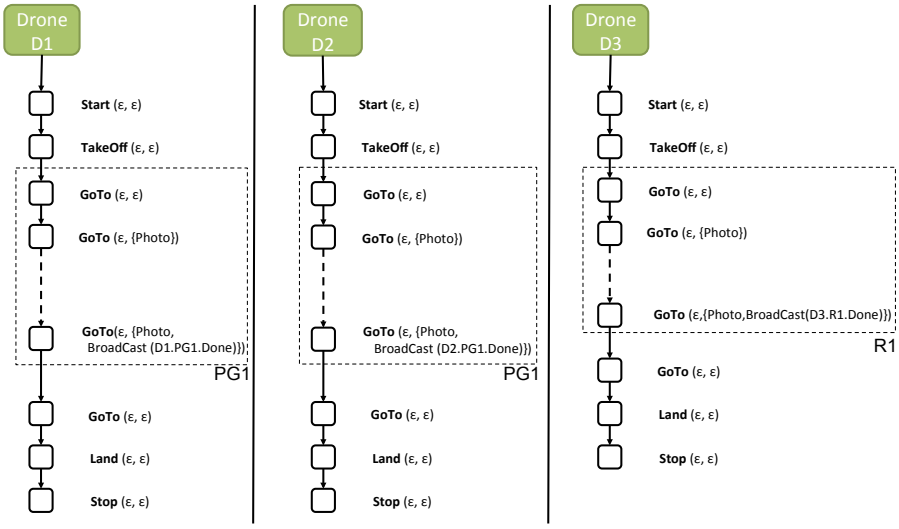


Fig. 4. Behavioural model of the Alpini event monitoring mission

- Land: represents the landing operation of the drone;
- GoTo: is the movement towards a given target geographical coordinate.

Each drone executes the sequence of all the movements of its assigned behavioural trace. In Figure 4 arrows denote the transition between a movement to the next one. A drone can perform a number of actions before or after the execution of a specific movement. Pre- and post-actions of each movement are enclosed with parenthesis on the right of the movement name. The behavioural model of the modelled mission are:

- Photo: represents the action of taking a high-definition photo below the drone;
- BroadCast: represents the action of sending a message to all the drone within the swarm; in the example the broadcast message represents the notification of the completion of a given task for synchronization purposes.

By looking at Figure 4 it is evident that the behavioural trace of each drone follows a specific pattern, that is: the drone (i) starts and takes off, (ii), moves to the initial geographical point of the first task to be performed, (iii) realizes the mission tasks assigned to it, (iv) notifies to all the other drones of the swarm the completion of the assigned task, (v) comes back to its initial position, (vi) lands, and then stops itself. This execution pattern helps us in efficiently maintaining the transformation engine in charge of creating a behavioural model from the upfront specification.

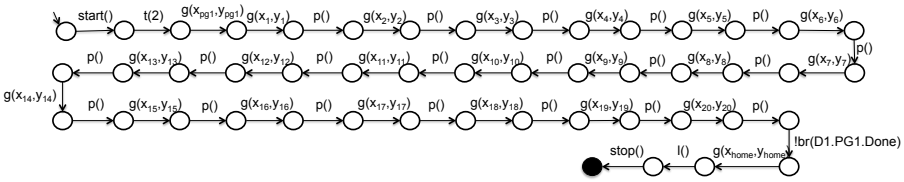
Sub-specifications: Figure 5 shows the sub-specifications of all the drones used in the mission. Drone D1 and drone D2 are assigned to photo grid task PG1, and drone D3 is assigned to road task R1. The state machines in figure will be automatically generated by the behavioural model of the mission, which is in turn automatically generated from the mission model containing the two high-level tasks.

Drones D1 and D2 have a very similar behaviour since they have been assigned to the same high-level task (i.e., PG1), and the behavioural generator equally splits the duties of the drones into two equal portions. Basically, after the preliminary actions of starting up and taking off, each drone assigned to a photo grid task must travel to

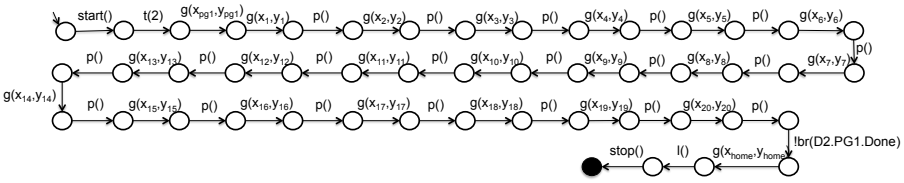
the initial geographical point of the task, i.e., (x_{pg1}, y_{pg1}), and then the state machine contains a chain of transitions representing (i) go to movements, i.e., $g(x_{lat}, y_{lon})$ and (ii) photo actions, i.e., $p()$. In the current example, the photo grid task generated a grid containing 40 points, and thus each drone has been assigned to 20 points in which a photo must be taken. When the last point of the grid has been photographed each drone notifies all the other drones within the mission that it just completed its assigned part, comes back to its initial geographical point (i.e., $home$), lands and stops itself. Drone 3 has been assigned to the road task R1; this task has a reference polyline which is 3550 meters long, and the drone must take a picture every 200 meters. This means that Drone 3 must take a total of 17 pictures, then, similarly to all the other drones, it notifies the completion of its part of mission to all the other drones of the swarm, comes back to its initial geographical point, lands, and stops itself.

Notice that each state of the state machines in Figure 5 contains a set of properties that define the operational context and status of the drone when entering it. Examples of properties include: the position of the drone is within a certain range, the battery level is above a certain threshold, the status of its sensors is correct, etc. At run-time, when a drone is entering a specific state, the controller checks if the current operational context and the status of the drone match the values of the properties defined in the state. As seen in Section 4.2, if those checks are satisfied, then the drone can continue the execution of the remaining state machine, otherwise the controller switches to the logic implementing the abnormal part of the drone, thus trying to recover after the error.

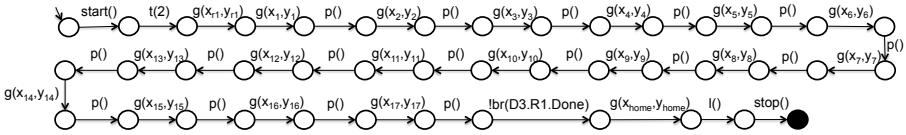
In the following we provide three example scenarios in which the combination of run-time controller and the behavioural state machine may be a good strategy for providing the needed level of resilience to a drone performing a mission on the field. Firstly,



(a) Sub-specification of drone D1



(b) Sub-specification of drone D2



(c) Sub-specification of drone D3

Fig. 5. Drones sub-specifications

assuming that the drone is reaching a certain behavioural state that refers to a specific geographical position (x, y), and suddenly a strong *gust of wind* pushes the drone outside the allowed range with respect to (x, y), then the controller recognizes this unexpected situation and switches its execution to the exceptional behaviour in order to try to bring the drone back to the exact position. Secondly, assuming that an attacker is trying to flood the drone by sending a huge set of messages to it and that the drone is in a state in which there is no outgoing transition with a receive message action, then the controller is able to recognize this dangerous situation and thus it switches its execution to the exceptional behaviour in order to recover from it (e.g., by bringing the drone under attack to its home position). Finally, assuming that an engine of the drone gets broken and that a drone is reaching a certain state with the property $(enginesRPM >= 3000)$, then the controller is able to straightforwardly detect the fault (since the current RPM of the broken engine is equal to zero), and thus it switches its execution to the expectional behaviour in order to, e.g., immediately land the drone in a safe area.

5 Related Work

High-level goals decomposition: Identifying from high-level goals requirements of components, parts of the system or operations is still an open problem. Most of the approaches that use specifications, such as formal methods, assume such operational requirements to be given. However, deriving "correct" operational requirements from high-level goals is challenging and is often delegated to error prone processes.

Letier and Lamwsveerde [3] propose an iterative approach that allows the derivation of operational requirements from high-level goals expressed in real-time linear temporal logic (RT-LTL). The approach is based on operationalisation patterns. Operationalization is a process that maps declarative property specifications to operational specifications satisfying them. The approach produces operational requirements in the form of pre-, post-, and trigger- conditions. The approach is guaranteed to be correct. Informally, here correct means that the conjunction of the operational requirements entails the RT-LTL specification. The approach is limited to a collection of goals and requirement templates provided by the authors. Moreover, the approach necessitates a fully refined, labour-intensive, and error-prone goal model that requires specific expertise.

The tool-supported framework proposed in [2] combines model-checking and Inductive Logic Programming (ILP) to elaborate and refine operational requirements in the form of pre- and trigger- conditions that are correct and complete with respect to a set of system goals. System goals are in the form of LTL formulas. The approach works incrementally by refining an existing partial specification of operational requirements, which is verified with respect to the system goal. The verification is performed by using model checking, which returns a counter-example in the case of the considered property is not valid on the model. The counter-example is exploited to learn and refine the operational requirements. However, the approach does not support learning the operational requirements for an individual component of a system. The work in [23] focuses on the service-oriented computing paradigm and proposes an interesting approach to automatically generate behavioral interfaces of the partner services, by decomposing the requirements specification of a service composition.

Incremental and compositional verification: Authors of [14] present a framework for performing assume-guarantee reasoning in an incremental and fully automatic fashion.

Assume-guarantee theory has been originally introduced in the thesis of Cliff Jones [24] and subsequently developed by many others, including Amir Pnueli [25]. In assume guarantee, the environment is represented as a set of properties that it should satisfy to correctly interact with the component. These properties are called assumptions, which means that they are the assumptions that a component makes on its environment. If these assumptions are satisfied by the environment, then the component that behaves in this environment will satisfy other properties, which are called guarantees. By combining the set of assume/guarantee properties in an appropriate way, it is possible to demonstrate the correctness of the entire system without constructing the complete system. The approach presented in [14] automatically generates, via a learning algorithm, assumptions that the environment needs to satisfy for the property to hold. These assumptions are initially approximate, but become gradually more precise by means of counter-examples obtained by model checking the component and its environment. In [26] authors observe that in reality, a component is only required to satisfy properties in certain specific environments. Moved by these motivations, they generate assumptions that exactly characterise those environments in which the component satisfies its required property.

6 Conclusions and Future work

In this paper we discussed how the resilience of critical systems can be constructed compositionally and incrementally out of the resilience of system parts. To this end, a general methodology has been introduced based on upfront specifications of the considered system that are refined into sub-specifications that are in turn amenable to manipulations and run-time monitoring activities. A concrete application of the methodology has been discussed in the domain of civilian monitoring missions that are executed by swarms of quadrotors. By relying on previous experiences of the authors (see [17]), we plan to (i) extend the current version of the FLYAQ platform in order to include the controllers of the drones described in Section 4.2 and the management of normal and abnormal behaviours based on an explicit and abstract representation of the system state, and (ii) to apply the implemented methodology to a real industrial case study.

References

1. Inverardi, P., Autili, M., Di Ruscio, D., Pelliccione, P., Tivoli, M.: Producing software by integration: Challenges and research directions (keynote). In: Procs. of ESEC/FSE 2013, ACM (2013) 2–12
2. Alrajeh, D., Ray, O., Russo, A., Uchitel, S.: Using abduction and induction for operational requirements elaboration. Journal of Applied Logic **7**(3) (2009) 275 – 288
3. Letier, E., van Lamsweerde, A.: Deriving operational software specifications from system goals. In: Procs. of SIGSOFT '02/FSE-10, ACM (2002) 119–128
4. Autili, M., Cortellessa, V., Di Ruscio, D., Inverardi, P., Pelliccione, P., Tivoli, M.: Eagle: Engineering software in the ubiquitous globe by leveraging uncertainty. In: Procs. of ESEC/FSE '11, ACM (2011) 488–491
5. Garlan, D.: Software engineering in an uncertain world. In: Procs. of the FSE/SDP Workshop on Future of Software Engineering Research. FoSER '10, ACM (2010) 125–128
6. Ghezzi, C.: Evolution, adaptation, and the quest for incrementality. In: Procs. of the 17th Monterey Conf. on Large-Scale Complex IT Systems: Development, Operation and Management, Berlin, Heidelberg, Springer-Verlag (2012) 369–379

7. Gary, K., Enquobahrie, A., Ibanez, L., Cheng, P., Yaniv, Z., Cleary, K., Kokoori, S., Muffih, B., Heidenreich, J.: Agile methods for open source safety-critical software. Softw. Pract. Exper. **41**(9) (August 2011) 945–962
8. Ge, X., Paige, R., McDermid, J.: An iterative approach for development of safety-critical software and safety arguments. In: Agile Conf. (AGILE), 2010. (Aug 2010) 35–43
9. Leveson, N.G.: Software safety: Why, what, and how. ACM Comput. Surv. **18**(2) (June 1986) 125–163
10. Wolff, S.: Scrum goes formal: Agile methods for safety-critical systems. In: Software Engineering: Rigorous and Agile Approaches (FormSERA). (June 2012) 23–29
11. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc. (2002)
12. Lamsweerde, A.v.: Goal-oriented requirements enginering: A roundtrip from research to practice. In: Procs. of RE '04, IEEE Comp. Soc. (2004) 4–7
13. Meyer, B.: Object-oriented Software Construction (2Nd Ed.). Prentice-Hall, Inc. (1997)
14. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Procs. of TACAS'03, Springer-Verlag (2003) 331–346
15. van Lamsweerde, A.: Requirements engineering in the year 00: A research perspective. In: Proc. of the 22nd Int. Conf. on Software Engineering. ICSE '00, ACM (2000) 5–19
16. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Procs. of the 9th International Conf. on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'03, Springer-Verlag (2003) 331–346
17. Autili, M., Di Ruscio, D., Di Salle, A., Inverardi, P., Tivoli, M.: A model-based synthesis process for choreography realizability enforcement. In: 16th Int. Conf. on Fundamental Approaches to Software Engineering (FASE). Volume 7793 of LNCS. (2013) 37–52
18. Bucchiarone, A., Di Ruscio, D., Muccini, H., Pelliccione, P.: From Requirements to Java code: an Architecture-centric Approach for producing quality systems. In: Model-Driven Software Development: Integrating Quality Assurance. Information Science Reference (2008)
19. Rubira, C.M.F., de Lemos, R., Ferreira, G.R.M., Castor Filho, F.: Exception handling in the development of dependable component-based systems. Softw. Pract. Exper. **35**(3) (March 2005) 195–236
20. Lee, P.A., Anderson, T.: Fault Tolerance: Principles and Practice. 2nd edn. Springer-Verlag (1990)
21. Di Ruscio, D., Malavolta, I., Pelliccione, P.: Engineering a platform for mission planning of autonomous and resilient quadrotors. In: Procs. of SERENE 2013, LNCS n. 8166 (2013) 33–47
22. Di Ruscio, D., Malavolta, I., Pelliccione, P.: A family of domain-specific languages for specifying civilian missions of multi-robot systems. In: First Workshop on Model-Driven RoboProceedings t Software Engineering - STAF2014. (2014) to appear.
23. Bianculli, D., Giannakopoulou, D., Păsăreanu, C.S.: Interface decomposition for service compositions. In: Procs. of ICSE '11, ACM (2011) 501–510
24. Jones, C.: Development Methods for Computer Programs Including a Notion of Interference. PhD thesis. Oxford University Computing Laboratory, Programming Research Group (1981)
25. Pnueli, A.: Logics and models of concurrent systems. Springer-Verlag (1985) 123–144
26. Giannakopoulou, D., Păsăreanu, C.S., Barringer, H.: Component verification with automatically generated assumptions. Automated Software Engg. **12**(3) (July 2005) 297–320