A skill fault model for autonomous systems

Gabriela Catalán Medina
gamilesca@gmail.com
LAAS-CNRS, ONERA, University of Toulouse
Toulouse, France

Jérémie Guiochet
jeremie.guiochet@laas.fr
LAAS-CNRS, UPS, University of Toulouse
Toulouse, France

Charles Lesire
charles.lesire@onera.fr
ONERA/DTIS, University of Toulouse
Toulouse, France

Augustin Manecy
augustin.manecy@onera.fr
ONERA/DTIS, University of Toulouse
Toulouse, France

ABSTRACT

Autonomous systems are now deployed for many applications to perform more and more complex tasks in open environments. To manage complexity of their control software architecture, a current trend is to use a 3-layers approach, with a decisional layer (able to formulate decisions), a functional layer (low level control actions), and between them a skill layer. This layer is dedicated to convert high level plan objectives into low level atomic actions, sent to the functional layer. In order to deal with failures that may happen at runtime, detection mechanisms and reaction strategies may be implemented in these layers, or even in external devices. However, no generic technique is available to guarantee that all these mechanisms will be consistent. We present in this paper an approach that focus on the skill layer, with a proposal of a generic skill fault model used to design and analyze failure detection and reactions mechanisms. This approach has been successfully applied to a real drone application, and we present an extract of the resulting fault analysis models.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Software and its engineering** → *Fault tree analysis*; *Software fault tolerance*.

KEYWORDS

Robot Skills, Fault tree, Fault Tolerance

ACM Reference Format:
Gabriela Catalán Medina, Jérémie Guiochet, Charles Lesire, and Augustin Manecy. 2022. A skill fault model for autonomous systems. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

One main challenge in developing autonomous systems is to define software architectures integrating low-level functions (e.g., control) and decisional-level features (e.g., task planner). A popular approach is to deploy layered architectures [12], and more precisely 3-layers (functional, executive, and decisional layers) as shown in Figure 1. The executive layer is an abstraction layer, usually in charge of splitting the high level tasks, commanded by the decisional layer, into atomic actions that should be realized by the functional layer. This intermediate layer checks if a commanded task can be realized (according to the current system state), and choose the appropriate termination mode when it is finished, to let the decisional layer determine following actions according to the mission objectives. This kind of architecture is particularly suitable to manage mission reconfiguration at decisional level, as done for example in [2] where back-up plans are selected by the decisional layer in response to some hazardous events or failures. However, it may become complex to manage all (or part of) failures when the architecture exhibits a significant set of entities at the executive and functional layer. Indeed, in such complex architectures, the treatment of failures that may occur at runtime is usually performed at different levels of abstraction (e.g., at the functional level, or at the decisional level). In this context, it is thus difficult for the developers to guarantee that all the failure detection and treatment mechanisms will be consistent. Moreover, no systematic analysis technique is proposed today to specify such mechanisms and implement them in a 3-layer architecture.

We propose in this paper a methodology to analyze such a 3-layer architecture facing a given set of potential failures. We assume that the intermediate layer, or *skill layer*, relies on a model-based skill description framework as proposed in [15]. For that we develop a generic skill fault model, based on the fault tree analysis (FTA) technique, extracted from our expertise on skill failures. We propose to apply this skill fault model to each skill, in order to specify failure detection and reaction strategies in an architecture design step. This contribution provides a technique that is generic, model-based, and that may be used for the skill design or verification steps.

The paper is structured as follows. Section 2 presents related works on autonomous architectures with skill layers, and how failures are treated in such architectures. Then, section 3 introduces more in details what is a skill, how it is modelled, implemented and executed. The fault tree pattern of a skill which is the starting point of the fault analysis is then presented in section 4. Consequently, we

apply our methodology in section 5: we present in details the analysis of one skill. Finally we will discuss the proposed methodology and propose future works in section 6.

2 SKILLS AND FAULTS IN AUTONOMOUS ARCHITECTURES

As stated in the survey on deliberative systems [12], hierarchical architectures are probably the most used in autonomous robotics. These are usually composed of several layers, from hardware level to decisional (or deliberative) level. While some architectures tend to a two layer approach, one for elementary tasks (like sensing), and another for decisional aspects (e.g., ORCCAD [4], CLARATY [22]) others propose to use a three layer architecture [1, 10, 13], as represented in Figure 1.

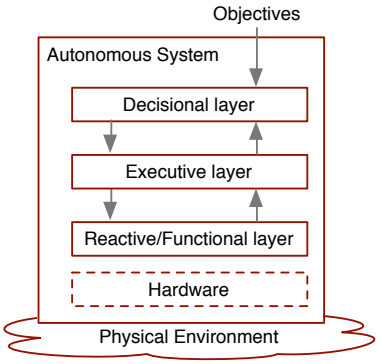

Figure 1: A three layer architecture for autonomy

In this latter case, the proposed software layers are:

Decisional layer is the highest abstracted level of the architecture, receiving objectives (from another system, or an operator) and generating some plans according to an abstract representation of the system and its environment;

Executive/skill layer checks that plans sent by the decisional layer can be realized and converts them into primitive functions for the functional under layer;

Functional layer is in charge of feedback control loops coupling sensors to actuators, perception facilities and trajectory computation.

A current trend is to deploy the executive layer with a skill-based approach, such as it is proposed in [2]. The *skills* correspond to the robot's functions or capabilities which are abstracted from the functional layer. For instance, as presented later in Figure 4, skills of an Unmanned Aerial Vehicle (UAV) may be Take-off, Goto (i.e., reach a position), Land, etc. Similar works have been proposed to use a skill model to abstract high-level functions available in the functional layer [5, 6, 19].

In such layered-architectures, most of the work regarding fault detection and treatment is covered by the fault tolerance concept coming from the dependability community [3]. It is basically composed of an error detection mechanism and a recovery mechanism, in order to keep the system in an acceptable state. Many works focus on the functional level [7, 23], and perform timing or reasonableness checks of functional software modules, and activate a request towards the decisional layer in case of detection. It is then up to the decisional layer to engage a reaction strategy. A similar approach is deployed in [8] for the detection, but with a drastic reaction to disconnect the decisional level in order to switch into a tele-operated mode (i.e. the system is no longer autonomous). These works are not based on explicit models, and are thus not so generic to be applied in different contexts and technologies. On the opposite, the work in [9] is based on formal models of the components, and can automatically generate monitors, but is effective at the functional level of an autonomous architecture. They are actually very few works explicitly focusing on fault tolerance at the decisional level like the one on redundant planning [16]. Some contributions for fault tolerance at decisional level may also come from the "execution monitoring" community [18, 20] or fault detection and diagnosis in robotics [14], but few are focusing on using a model at the decisional level. Another direction is to develop independent devices (external to the 3 layers) to monitor architecture based on a safety model of the system (e.g., [17]). However, such approaches use their own model, and not the ones available in the autonomous architecture and used to deliberate and control the system. Managing fault detection and recovery at the executive layer is actually quite rare. For instance, in [21], the executive layer is completely dedicated to the verification of requests between the decisional and functional layer, but does not manipulate skills or intermediate abstractions. Deploying a framework for analysing faults propagation at skill level (or executive level) based on models of the skills, including a fault model, is thus an original work.

3 BACKGROUND ON SKILL MODELS AND IMPLEMENTATION

In this paper, we settle on the skill models proposed in [15], as the proposed skill modeling language contains interesting features for failure analysis (several terminal states, execution conditions, ...) In this section, we remind the important parts of the modeling and implementation process necessary to present our contribution on the skill fault model and the associated analysis process.

In the language proposed in [15], the top-level container is a **skill-set** model: it contains a set of consistent *resources* and *skills* model describing a part of the skill-based architecture. For instance, in the use-case considered later, one skill-set describes all the features related to the "motion management" of the UAV, while another skill-set manages "perception" skills (mapping skills).

Resources are modeled as Finite State-Machines (FSMs) that represent the status of actual devices, or logical states. For instance, the current flight status of the UAV is modeled using the resource presented in Listing 1; the status is initially unknown, and the transitions are marked as *extern*, meaning that they can be triggered by the functional layer (or the environment).

Every **skill** has the same execution model, also defined by a common FSM, depicted in Figure 2. A *client* (typically implemented in the decision layer) can `start` the skill execution, providing some inputs at the same time. These inputs are checked by a user-defined

```
resource flight_status {
    initial UNKNOWN
    extern UNKNOWN -> ROTORS_NOT_READY
    extern UNKNOWN -> ON_GROUND
    extern UNKNOWN -> IN_AIR
    extern ROTORS_NOT_READY -> ON_GROUND
    extern ON_GROUND        -> ROTORS_NOT_READY
    extern ON_GROUND        -> IN_AIR
    extern IN_AIR           -> ON_GROUND
}
```

Listing 1: Model of the `flight_status` resource using the DSL proposed in [15]

callback (checking for example actual robot capabilities or safety limitations), and if considered invalid, the skill execution is rejected (final state NV). Otherwise, some resource preconditions defined by the skill model are checked (state CR). Again, if these conditions are not met, the skill execution is rejected (final state NR). Otherwise, the execution is effectively started (transition `dispatch` calling another user-defined callback – Rg stands for Running). During the execution, some modelled invariants must hold. If an invariant is violated, the skill execution is interrupted and ends in the RI state. The skill execution can also be interrupted by the client (`interrupt` transition – Ig stands for Interrupting). Finally, the skill can end on a set of possible terminal states, or results, represented by the states M_i (when result post-conditions are satisfied) and $\overline{M_i}$ (when post-conditions are not satisfied). Some of these results can actually be interpreted as *failure modes* of the skill.

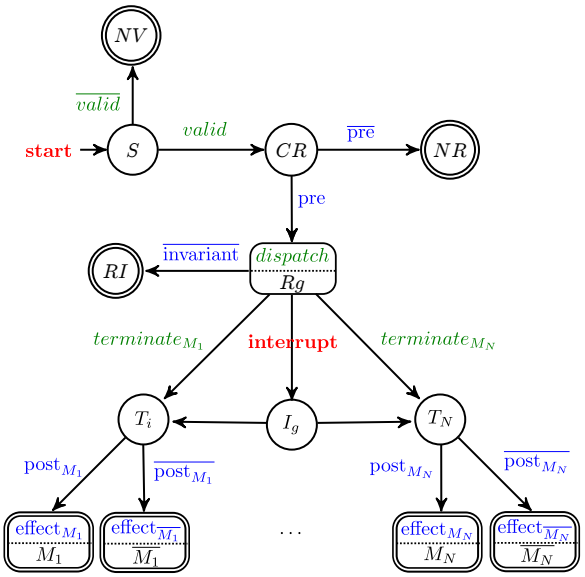

Figure 2: Skill FSM from [15]. Double circled states are terminal states and rounded-boxes states have an entry effect. Bold red labels correspond to transitions triggered from a client. Italic green labels correspond to functions available in the skill implementation part. Blue labels correspond to automatic tests and effects.

Every skill can then be defined using the skills *domain specific language (DSL)*, that allows to specialize some parts of the skill FSM. This is done by writting a skill-set model, using the skill DSL, which allows to define for each skill:

- its *inputs*, that are further passed to the FSM in the `start` transition;
- its resource *preconditions*: to be executed, the skill requires that some resource states meet these preconditions;
- its resource *invariants*: during execution, the skill requires that these conditions on resource states hold;
- some *effects* on resources, that will change the resource states either at the beginning of the execution (during the `dispatch` transition) or at the end of the execution (when reaching one of the M_i states);
- the possible *results* of the skills, i.e. the set of M_i states.

This specification is called the **skill model**.

The UAV `takeoff` skill model is given in Listing 2. This skill

```
1  skill takeoff {
2      input {
3          height: float64 // validate can fail if h>h_geo_fence
4          speed:  float64 // maximum ascending speed
5      }
6      effect {
7          take_control: axes_authority -> USED
8          release_control: axes_authority -> AVAILABLE
9      }
10     precondition {
11         sdk_authority: resource=(SDK_authority==AVAILABLE)
12         not_moving: resource=(axes_authority==AVAILABLE)
13         on_ground: resource=(flight_status==ON_GROUND)
14         home_valid: resource=(homepoint_status==VALID)
15         success take_control
16     }
17     invariant {
18         keep_sdk_authority: resource=(SDK_authority==AVAILABLE)
           ↪  violation=release_control
19         in_control: resource=(axes_authority==USED)
20     }
21     result {
22         AT_ALTITUDE: apply=release_control
23         BLOCKED: apply=release_control
24         ABORTED: apply=release_control
25     }
26 }
```

Listing 2: Model of the `takeoff` skill using the DSL proposed in [15]

has two inputs (lines 2 to 5): the height to reach when taking off, and the maximum allowed vertical speed. The takeoff preconditions are: authorities must be available (`SDK_authority` is given by the security tele-pilot – line 11 –, `axes_authority` by the skillset manager – line 12), the UAV must be on ground (line 13) and the home point must have been defined (line 14). In that case, the effect applied on dispatch (line 15) is `take_control`, that asks to change resource `axes_authority` to USED (line 7, to prevent other skills to command simultaneous displacements). During the take off execution, two invariants must hold: the `axes_authority` must still be "owned" by the takeoff skill (line 19), and the tele-pilot should not have taken back the manual control (line 18). Finally, three possible results are specified, a nominal one, `AT_ALTITUDE`, when the

target altitude is reached (line 22), and two failure results: BLOCKED (line 23), triggered when the UAV cannot move up, and ABORTED (line 24), triggered when the UAV moves hazardously, in which case the maneuver is interrupted by the functional layer.

Then, a code generator produces a ROS node corresponding to a *skill manager*, that implements the FSM of Figure 2 specialized by the skill model of Listing 2. The FSM can then be further specialized by the user, who can implement methods in the generated code corresponding to the following parts of the FSM (in green on Figure 2):

- the *validation* function, that checks if the inputs are correct (e.g., the target height of the take off must be positive and below the maximal authorized altitude), and returns either $valid$ (transition to CR) or $\overline{valid}$ (transition to NV);
- the *dispatch* function, that is responsible of effectively starting the skill execution by sending commands to the functional layer;
- the *terminate* functions, that must be called when receiving the corresponding information from the functional layer.

These functions implemented by the user within the generated FSM are called the **skill implementation**.

4 SKILL FAULT MODEL AND ANALYSIS

As presented before, the skill layer is a core component of the architecture, and we explore in this study how some faults (internal or external) may be detected and treated at this level. We base our study on the fault/error/failure definitions from dependability concepts [3]: when activated, a fault in a unit becomes an error, and if it propagates to the boundary of the considered unit, it becomes a failure. Here we consider a skill as a unit, and our basic idea is to analyse all failure modes of a skill, with a deductive method, to identify errors and then faults that led to each failure of the skills.

4.1 Skill Fault Model

In that purpose, we propose a generic skill fault model, aiding the designers to identify failure modes, detection means, and recovery actions (actions to keep the system in an acceptable state regarding the safety and mission objectives). This skill fault model is based on fault tree analysis (FTA) [11]. FTA is a well-known risk analysis technique, used for years in many domains (from nuclear power plants to aeronautics). It is a top-down approach starting with an undesirable event called a *top event*, and then determining how this top event may be caused by individual or combined lower level failures. In FTA, failures from different fields are combined. Logical relations between them are represented by logic symbols (AND and OR gates). In a fault tree analysis, the top event is a hazard that must have been foreseen and thus identified previously. The leaves of the fault tree are called *basic events*, and all events between the top event and the leaves are called *intermediate events*. Each event can be interpreted as a failure from the components viewpoint, or as a fault or an error from the overall system viewpoint. We will refer as skill failure for the top events. We then propose to use such a fault tree to help the designers to identify failure causes, and specify detection and recovery mechanisms. To make this analysis, we propose a generic skill fault tree pattern to guide the designer in this task.

Based on the concepts of **skill model** and **skill implementation** described in the previous section, we propose to decompose the top event corresponding to the skill failure as depicted in Figure 3. A skill can then fail if:

- it cannot be started (node P002) due to invalid inputs (node P010) or resource precondition issues (node P008);
- it is started but no effect is observed (node P007); this error must then be detailed by the designer based on the relation between each skill and the functional layer;
- the skill execution is interrupted (node P004), either due to the violation of resource invariants (node P012) or to external conditions (node P020), for which we propose standard errors, linked to elements of the functional layer (nodes E024, E025 and E026);
- the skill completes with an error (node P005), which can be caused by external conditions (node P018), that again must be detailed by the designer, or by resource constraints (node P014).

This fault tree pattern, also called our **skill fault model**, is aimed at being instantiated according to the skill under investigation. Some branches can then be irrelevant for a specific skill and removed from the resulting tree. The skill fault model instantiated and specialized for a specific skill is called the **skill fault tree**. We can notice that all the events in this pattern are combined with *OR* gates, which implies that if one of these events happens, the whole skill execution will fail.

In this fault tree, a Detection Mechanism (DM) and a Failure Mode (FM) are assigned to some nodes (from DM1 to DM7 and FM1 to FM7). Figure 3 exhibits for each DM/FM if it is supposed to be covered by the skill model and by which mechanism (invariant, pre, post, etc.), or if it is supposed to be handled in the skill implementation. For instance, FM1 is equivalent to state NR of Figure 2 and DM3 corresponds to the $\overline{invariant}$ transition. These DM and FM are to be considered as guidelines: depending on the actual skills or the actual considered failures, they can be changed or placed elsewhere in the model.

4.2 Analysis Process

Based on the **skill fault model** presented in the previous section, we have defined an analysis process that we systematically apply to all the skills contained in a skill-set. This analysis process is composed of several steps:

(1) Listing of all the events that may impact correct skill execution,
(2) Design of each **skill fault tree** based on the **skill fault model** pattern (Figure 3), using the following steps:
   (a) Connection of each event listed in (1) with each **skill fault tree**,
   (b) Determination of all relative failure modes (FM_i) and potential detection mechanisms (DM_i) for each branch of the fault tree;
(3) Verification, for each skill, that the **skill fault tree** is consistent with the **skill model** and **skill implementation**:
   (a) Checking that each DM_i and FM_i is covered by the skill model or the skill implementation

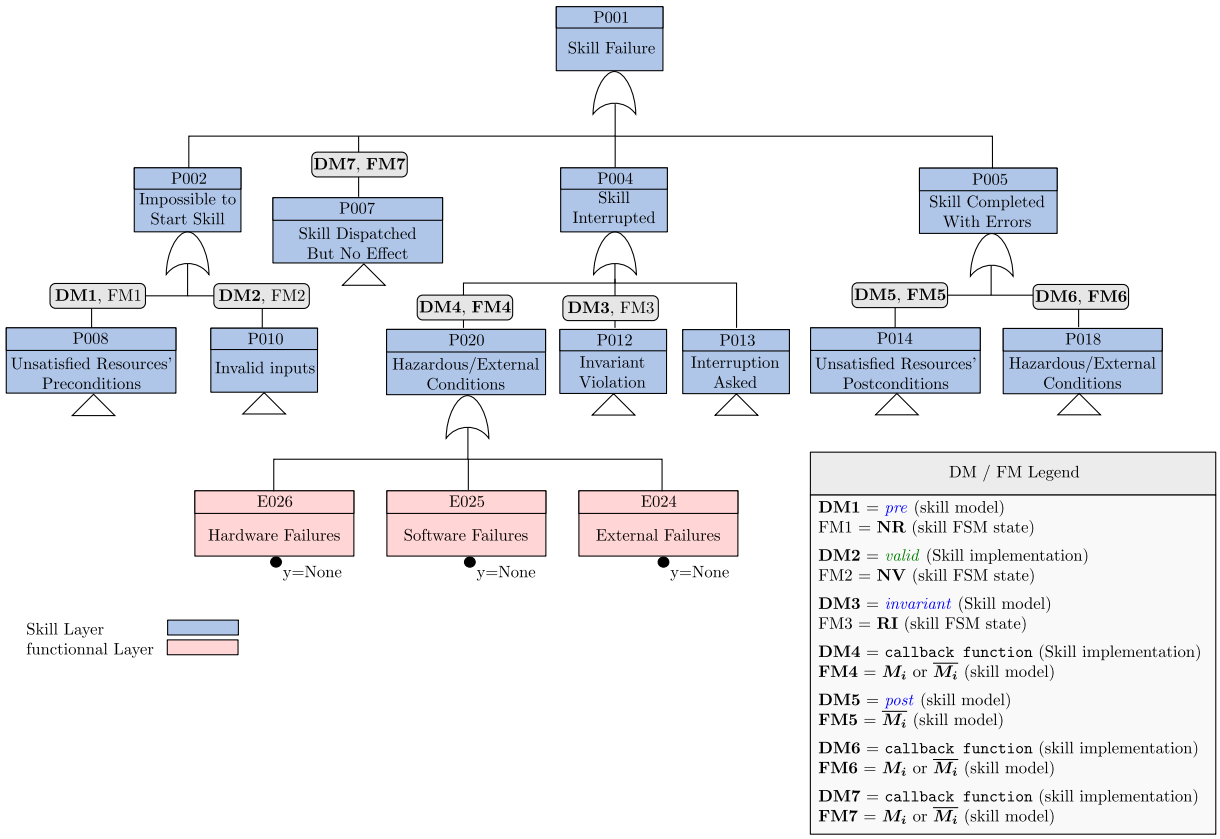

Figure 3: Skill Fault Model: Pxxx represent fault tree nodes linked to skill layer, Exxx represent fault tree nodes linked to functional layer. DM_i and FM_i are respectively Detection Mechanisms and Failure Modes. Normal font DM/FM are supposed to be managed in the skill FSM (e.g., FM1 corresponds to NR state of the skill FSM). Bold DM/FM are supposed to be managed either in the skill implementation (e.g., DM2 has to be implemented in the *valid* function) or in the skill model (e.g., FM4 has to be described as result mode in the skill model).

(b) Modification of the skill model or implementation to add or correct a missing or incomplete DM_i or FM_i.

Step (3) of the proposed process is an iterative procedure which aims at modifying the **skill model** and the **skill implementation** until all DM_i and FM_i identified by FTA are covered. Classically a missing DM_i can be fixed by 1) adding a resource and/or a resource condition (`invariant`, `pre`, `post`) to the **skill model**, or 2) adding in the **skill implementation** a call to a *terminate* function in reaction of some events or data coming from the functional layer. A missing FM_i will generally lead to the addition of a new terminal state M_i (and its appropriate management in the model and in the implementation) or to the modification of resource `post` conditions.

This **skill fault model** and the corresponding analysis process can be used in two different development processes. First, it can be used to analyse an existing model and its implementation, in order to validate what is already developed for a system. The FTA based on the skill fault model can then lead to changes in the model and implementation, as presented above and shown in the next section on a specific use-case. Second, it can be used earlier in the development process, before the actual development of a skill. In that context, this prior analysis would give or complete the specification of the skill model and what must be carried out in the implementation.

5 CASE STUDY

In this section, we describe how we apply the methodology based on the generic **skill fault model** to analyse the skill-layer of a robotic system. This system is an UAV performing an automatic inspection of a building in BVLOS conditions (Beyond the Visual Line Of Sight of the supervising tele-pilot), see [2] for details about this scenario. The 3-layer software architecture of the UAV is depicted in Figure 4. The skill-layer contains two skill-set managers, based on the formalization proposed in [15].

To ensure the correctness of this skill-layer with respect to the possible failures that could arise in the BVLOS scenario, we applied the analysis process presented earlier in section 4.2 to all the skills of this layer. As discussed before, all the skills were already modeled and implemented before applying this process. In this section, we will only present the analysis process applied to the `takeoff` skill, and formulate recommendations to modify the skill model and implementation in order to make it correct with respect to the FTA. The complexity of the fault trees of the complete analysis is really

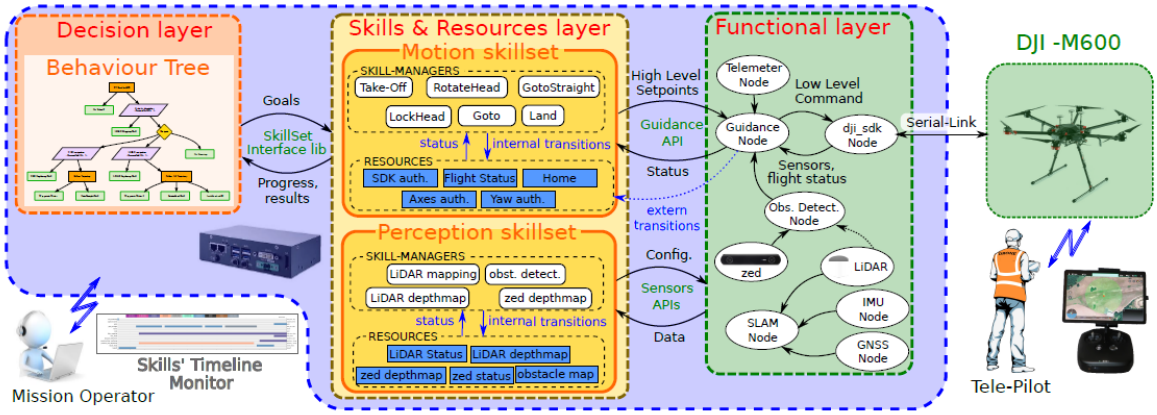

Figure 4: 3-layer architecture from [2]

close to the complexity of the fault tree presented here, i.e. it is manageable with light tooling.

5.1 Takeoff fault model

In this paper, we present the FTA applied to the `takeoff` skill, in which we considered the set of possible events that can affect the skill execution (step (1) of the analysis process presented in section 4.2).

Figure 5 presents the **takeoff fault tree**, obtained from the generic **skill fault model** once all the considered hazardous event have been added. To build this tree, we first applied step (2a) of the analysis process: for each node of the **skill fault model**, we analysed which errors could lead to this failure and selected some of them: battery failure (i.e., battery level too low), software errors, engine failure, excessive payload, etc. Some errors can be linked to several nodes, and then appear multiple times in the fault tree. For instance, when the UAV is on ground, a battery failure leads to the impossibility to take off because thrust is insufficient (Node E010), while during the flight, a battery failure yields to an alarm, and the tele-pilot should take control back (node E087).

Once all errors where considered and added at relevant places in the fault tree, we inspected the existing skill model and implementation and positioned the corresponding DM_i and FM_i on the fault tree (step (2b) of the analysis process).

5.2 Takeoff recommendations and improvements

The next step is then to identify inconsistencies or unmanaged events in the fault tree of Figure 5 (step (3a)) and then to modify the skill model or skill implementation to fix these inconsistencies (step (3b)).

A first observation is that considered errors can propagate up to top of the fault tree, i.e. lead to a skill failure, without being handled by a DM/FM mechanism. From this observation, we identified missing detection mechanisms and failure modes, which are represented in yellow in Figure 5. One of the modification we have made to solve some of these inconsistencies was to improve the implementation of the *valid* function.

Another observation was that some errors of different nature could lead to the same failure modes of the skill, but with different detection mechanisms. For instance, we consider that wind can make the UAV drift w.r.t its objective point. Drifting conditions (node E089) are detected by tracking the performance of the control law (in the functional layer), by a function (`large_track_error`) denoted DM5b. This function, part of the skill implementation, triggers FM5 (terminal state ABORTED). This terminal state is also used in other parts of the Takeoff fault tree, but are related to other errors. This situation may make the failure mode ambiguous, and therefore make unclear the kind of reactions the decisional layer should implement. In that specific case for instance, FM5 activated when the UAV is on ground or in air may have different meanings, that have very different risk consequences on the UAV.

These situations have been tackled by adding to the model some post conditions about termination modes. Using this simple mechanism, it is possible to clearly identify the failure modes of the skill over nominal termination modes. As a consequence, if the post conditions are correctly specified, failures modes correspond to $\overline{M_i}$ termination states, when M_i are most of time success final modes.

A last observation was about same DMs used at different places of the fault tree. For instance, DM1a is present in the skill model precondition and treated by the skill FSM, but is was also treated by DM1b in the *valid* function. The analysis allowed to find redundant checks and we choose to keep the one done by preconditions as the skill model can be used by the decisional layer.

Based on other observations and recommendations, we have further improved the skill model and implementation. Based on these modifications, the Takeoff fault tree has been updated, and the same analysis (identify inconsistencies, making recommendations, modifying the skill) has been performed again until we have a satisfying fault tree model. The resulting takeoff skill model is given in Listing 3.

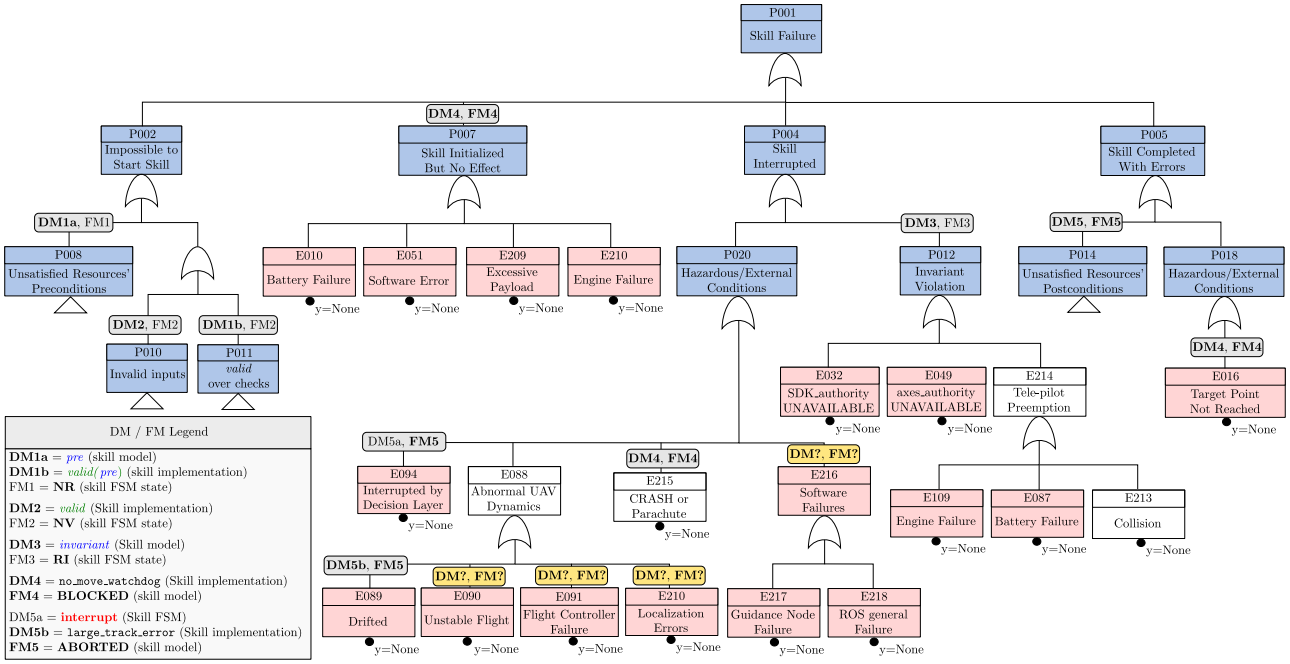

Figure 5: Takeoff fault tree

6 CONCLUSION

In this paper, we have proposed an analysis process to validate the skill-layer of an autonomous system architecture. This process relies on the Fault Tree Analysis methodology. To perform this FTA and benefit from the formalization of the several skills of this skill-layer, we have first proposed a **skill fault model**. The proposed FTA then uses this skill fault model as a pattern to perform the analysis of each individual skill of the architecture.

The FTA process applied to each skill is then basically composed of three steps: specialization of the skill fault model for this particular skill, identification of failure modes and detection mechanisms, and verification of the consistency of the resulting tree. This leads to the definition of recommendations to update or improve either the skill model or the skill implementation.

We have applied this FTA process to the skill-layer of an autonomous UAV performing an inspection mission in BVLOS conditions. We have shown the application of the process to the `takeoff` skill, by building the skill fault tree, and then identifying inconsistencies and modifying the `takeoff` skill model and implementation.

We are convinced that applying this process to our architecture leads to a more systematic and rigorous definition of our skill-layer, and also to more precise fault management mechanisms. This would also help the design of the decision layer.

We investigate two future work directions. First, we are applying the same process for the skill layer of new robotic systems that has not been developed, then using the FTA process as a way to specify the skill model and what must be implemented with respect to failure management.

Second, we would like to use the FTA of a skill (either already developed or just specified) in order to derive test cases to have a more complete validation approach. These test cases will then be used to verify that the actual execution is compliant with the fault tree model that has been made only by looking at the model or the implementation source code.

We also identified several limitations for this approach. First, our skill fault model need to be refined while applied to other case studies. For now, we have a set of skills dedicated to drone control, but we expect that the fault model will be efficient for autonomous mobile robots for instance. Second, the reaction strategies (after detection and switch the skill into a failure mode) are not part of our fault trees and we believe that integrating them would bring a more consistent design of revovery mechanisms. Third, it is possible that some failures may be not detected at the skill layer, and even trig some hazardous situations. We still need to explore what other mechanisms (e.g., external safety monitors) could be designed to complete our approach. Finally, it will be important to treat the situations when several detections are activated at the same time. In such situation, the strategy may be implemented in the skill, or event in a dedicated skill. This aspects needs to be investigate to be part of our proposal.

REFERENCES

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. 1998. An Architecture for Autonomy. *The International Journal of Robotics Research (IJRR)* 17, 4 (1998), 315–337. https://doi.org/10.1177/027836499801700402

[2] A. Albore, D. Doose, C. Grand, C. Lesire, and A. Manecy. 2021. Skill-Based Architecture Development for Online Mission Reconfiguration and Failure Management. In *International Workshop on Robotics Software Engineering (RoSE@ICSE)*. Madrid, Spain. https://doi.org/10.1109/RoSE52553.2021.00015

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on*

```
skill takeoff {
    input {
        height: float64 // validate can fail if h>h_geo_fence
        speed:  float64 // maximum ascending speed
    }
    effect {
        take_control: axes_authority -> USED
        release_control: axes_authority -> AVAILABLE
        reset {}
    }
    precondition {
        sdk_authority: resource=(SDK_authority==AVAILABLE)
        not_moving: resource=(axes_authority==AVAILABLE)
        on_ground: resource=(flight_status==ON_GROUND)
        home_valid: resource=(homepoint_status==VALID)
        battery_good: resource=(battery==GOOD)
        success take_control
    }
    invariant {
        keep_sdk_authority: resource=(SDK_authority==AVAILABLE)
        ↪ violation=release_control
        in_control: resource=(axes_authority==USED)
        ↪ violation=reset
    }
    result {
        AT_ALTITUDE: post=(flight_status==IN_AIR)
        ↪ apply=release_control
        BLOCKED: post=(flight_status==IN_AIR)
        ↪ apply=release_control
        INTERRUPTED: post=(flight_status==ON_GROUND)
        ↪ apply=release_control
        DRIFTED: post=(flight_status==IN_AIR)
        ↪ apply=release_control
    }
}
```

Listing 3: Model of the `takeoff` skill updated after the analysis

Dependable and Secure Computing 1, 1 (2004), 11–33. https://doi.org/10.1109/TDSC.2004.2

[4] J. J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. 1998. The ORCCAD Architecture. *The International Journal of Robotics Research (IJRR)* 17, 4 (1998), 338–359.

[5] D. Bozhinoski, E. Aguado, M. G. Oviedo, C. Hernandez, R. Sanz, and A. Wąsowski. 2021. A Modeling Tool for Reconfigurable Skills in ROS. In *International Workshop on Robotics Software Engineering (RoSE@ICSE)*. Madrid, Spain. https://doi.org/10.1109/RoSE52553.2021.00011

[6] R. I. Brafman, M. Bar-Sinai, and M. Ashkenazi. 2016. Performance level profiles: A formal language for describing the expected performance of functional modules. In *International Conference on Intelligent Robots and Systems (IROS)*. Daejeon, South Korea. https://doi.org/10.1109/IROS.2016.7759280

[7] D. Crestani, K. Godary-Dejean, and L. Lapierre. 2015. Enhancing fault tolerance of autonomous mobile robots. *Robotics and Autonomous Systems* 68 (2015), 140–155.

[8] B. Durand, K. Godary-Dejean, L. Lapierre, R. Passama, and D. Crestani. 2010. Fault tolerance enhancement using autonomy adaptation for autonomous mobile robots. In *International Conference on Control and Fault Tolerant Systems (SysTol)*. Nice, France.

[9] M. Foughali, S. Bensalem, J. Combaz, and F. Ingrand. 2020. Runtime Verification of Timed Properties in Autonomous Robots. In *International Conference on Formal Methods and Models for System Design (MEMOCODE)*. Jaipur, India. https://doi.org/10.1109/MEMOCODE51338.2020.9315156

[10] E. Gat. 1998. On Three-Layers Architectures. In *Artificial Intelligence and Mobile Robots*, David Kortenkamp, R. Peter Bonasso, and Robin Murphy (Eds.). MIT Press, Cambridge, MA, USA, 195–210.

[11] IEC61025 2006. *Fault tree analysis (FTA)*. Standard. International Electrotechnical Commission, Geneva, Switzerland.

[12] F. Ingrand and M. Ghallab. 2017. Deliberation for autonomous robots: A survey. *Artificial Intelligence* 247 (2017), 10–44. https://doi.org/10.1016/j.artint.2014.11.003

[13] F. Ingrand, S. Lacroix, S. Lemai-Chenevier, and F. Py. 2007. Decisional autonomy of planetary rovers. *Journal of Field Robotics* 24, 7 (2007), 559–580. https://doi.org/10.1002/rob.20206

[14] Eliahu Khalastchi and Meir Kalech. 2018. On Fault Detection and Diagnosis in Robotic Systems. *ACM Comput. Surv.* 51, 1, Article 9 (jan 2018), 24 pages. https://doi.org/10.1145/3146389

[15] C. Lesire, D. Doose, and C. Grand. 2020. Formalization of Robot Skills with Descriptive and Operational Models. In *International Conference on Intelligent Robots and Systems (IROS)*. Las Vegas, NV, USA. https://doi.org/10.1109/IROS45743.2020.9340698

[16] B. Lussier, M. Gallien, J. Guiochet, F. Ingrand, M.-O. Killijian, and D. Powell. 2007. Planning with Diversified Models for Fault-Tolerant Robots. In *International Conference on Automated Planning and Scheduling (ICAPS)*. Providence, RI, USA.

[17] M. Machin, J. Guiochet, H. Waeselynck, J.-P. Blanquart, M. Roy, and L. Masson. 2018. SMOF - A Safety MOnitoring Framework for Autonomous Systems. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 48, 5 (2018), 702–715.

[18] J. P. Mendoza, M. Veloso, and R. Simmons. 2012. Mobile Robot Fault Detection based on Redundant Information Statistics. In *"Safety in human-robot coexistence and interaction" workshop @IROS*. Vilamoura, Portugal.

[19] A. Nordmann, R. Lange, and F. M. Rico. 2021. System Modes - Digestible System (Re-)Configuration for Robotics. In *International Workshop on Robotics Software Engineering (RoSE@ICSE)*. Madrid, Spain. https://doi.org/10.1109/RoSE52553.2021.00010

[20] Ola Pettersson. 2005. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems* 53, 2 (2005), 73 – 88.

[21] F. Py and F. Ingrand. 2004. Dependable execution control for autonomous robots. In *International Conference on Intelligent Robots and Systems (IROS)*. Sendai, Japan. https://doi.org/10.1109/IROS.2004.1389549

[22] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. 2001. The CLARAty Architecture for Robotic Autonomy. In *IEEE Aerospace Conference*. Big Sky, MT, USA. https://doi.org/10.1109/AERO.2001.931701

[23] S. Zaman, G. Steinbauer, J. Maurer, P. Lepej, and S. Uran. 2013. An integrated model-based diagnosis and repair architecture for ROS-based robot systems. In *International Conference on Robotics and Automation (ICRA)*. Karlsruhe, Germany.