Skill-based design of dependable robotic architectures

Alexandre Albore, David Doose, Christophe Grand, Jérémie Guiochet, Charles Lesire, Augustin Manecy

Skill-based design of dependable robotic architectures

Alexandre Albore · David Doose · Christophe Grand · Jérémie Guiochet · Charles Lesire* · Augustin Manecy

the date of receipt and acceptance should be inserted later

Abstract Software architectures for autonomous systems are generally structured with 3 layers: a decisional layer managing autonomous reasoning, a functional layer managing reactive tasks and processing, and an executive layer bridging the gap between both. The executive layer plays a central role, as it links high-level tasks with low-level processing, and is generally responsible for the robustness or the fault-tolerance of the overall system. In this paper, we propose a development process for such an executive layer that emphasizes on the dependability of this layer. To do so, we structure the executive layer using *skills*, that are formally defined using a specific language, and we then provide some tools to verify these models, generate some code, and a methodology to assess the fault-tolerance of the resulting architecture.

Keywords Skill-based architecture · Dependability · Development Process · Model-Checking · Fault-Tree Analysis

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 [31], and more precisely 3-layers (functional, executive, and decisional layers) as shown in Fig. 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's state), and chooses the appropriate termination mode when its execution is finished, in order to let the decisional layer

A. Albore · D. Doose · C. Grand · C. Lesire · A. Manecy
ONERA/DTIS, University of Toulouse, France
E-mail: firstname.lastname@onera.fr

J. Guiochet
LAAS-CNRS, University of Toulouse, France
E-mail: jeremie.guiochet@laas.fr

* corresponding author

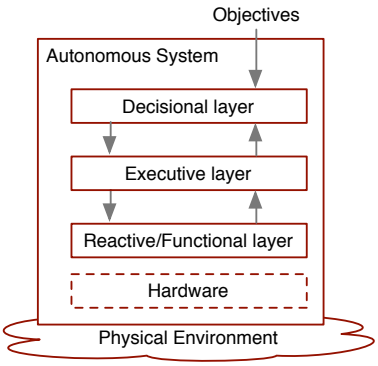

Fig. 1 A three layer architecture for autonomy

determine the 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 [1] where back-up plans are selected by the decisional layer in response to some hazardous event or failure.

In critical applications in which autonomous systems are used, decisional capabilities must settle on a robust and safe executive (and consequently functional) layer. When developing such a layer, the designer must provide evidences that the executive layer has been correctly designed, and that possible hazards are correctly dealt with, either at the functional or at the executive level.

A recent and interesting step in that direction comes from the adoption of a skill-based architecture, where the modularity of robotic capacities allows a more robust control of robotic functions, and increased safety. In fact, as in other domains, fault prevention in the software of autonomous system is mainly carried out through the modularity of software components and the tools adapted to the heterogeneity of the used models [53, 28, 40].

In this paper, we propose a process to develop dependable software architectures based on such *skill* concepts to populate the executive layer. To do so, we provide (1) a formal modelling of the executive layer through a Domain Specific Language, (2) tools to support the development of this layer in relation to the other layers, (3) tools to assess the dependability of the skill-layer in relation with the functional layer. The paper is structured as follows. Section 2 presents the state-of-the-art related to skill concepts in robotics, their formalization and verification. Then, a general overview of the development process we propose is presented in Section 3, along with an illustrative application. This process is further detailed in the upcoming sections: Section 4 presents the domain specific language, Section 5 the associated model-checking tools, Section 6 the code generation toolchain, and Section 7 the fault tolerance analysis. We conclude this paper by presenting a concrete application in Section 8 and discussing future works in Section 9.

2 Related Works

This section presents main works on skill definition, formalization, implementation in a decisional architecture, and verification.

2.1 Skill-based architectures in robotics

The conceptual model of robotic skills is born to enhance the modularity of robotic capacities, integrating both sensing and acting within a small set of "robot skills" [53]. Skills are implemented such that a robot can easily be reconfigured or repurposed. The aim is to allow users that are not expert in control or robotic field to easily use advanced robot systems, through a minimal parametrization of the skills which are in charge of implementing basic functionalities. Ideally, a skill can be easily composed with other skills because it already integrates all information, resources access rules, behavioral logic, recovery strategies, low level controls, etc., which are needed to execute a certain task. In robotic literature, the concepts of *task*, *primitive action*, and *robot skill* are commonly used, and sometimes they overlap. The confusion arises because task-level programming (a well-known direction taken by the robotics community) [11] makes use of low level entities, usually named "robot skills", that instantiate primitive actions, usually centered on motion [3, 26, 43, 39, 8, 9][1] Brooks writes about "task achieving behaviors" [11], which is a break from the purely functional architectures previously developed and focuses the robotic architecture on layers of behaviors that integrate sensors and actuators. Such layered architecture has been widely used since [12, 71, 33, 69, 68]. In this sense, primitive actions are basic functionalities or atomic motions of a robot. Their combination can produce tasks, which are aimed at achieving complex goals [9].

The skills, as they are intended in this paper, are components of a modular robotic architecture which differs from macros or tasks in their generality, composability, and reconfigurability [53]. They are basic functionalities that participate to design the complete task. Thus, their combinations are called *tasks*, which are the mission-level implementation of a robotic system [53, 63, 60]. This approach mainly leads to multi-layers architectures. The skills-based approaches to robot programming vary in different fields of robotic research, not all adopting the 3-layered approach described in this paper. For instance, in [60], the authors perform task-level programming using manually reprogrammable skills in a pyramidal architecture based on three layers of abstraction: tasks, skills, and device primitives. These three layers are decoupled from the hardware, implementing the functionalities of the robot at three levels of abstraction. In SkiROS (Skills-ROS) [57, 56], the architecture is organized into four layers of abstraction: skills are separated from the executive layer, which provides an interface to embed and coordinate motion primitives, and services. The decision layer can be either a user-interface or a automated task planner. A set of standardized device interfaces allows an easy link between layers, and extend the portability of all code.

[1] However, the expression *robot skills* refers, in more recent years, to skills chaining and skills learning in Deep Reinforcement Learning [38, 21, 48, 42].

2.2 Skills and decision layer

It is expected that Robotics 4.0 evolves towards "more advanced features in terms of motion, computing, perception, and cognition" [24]. The improved autonomy of robotics systems depends on the advances of Artificial Intelligence (AI) algorithms.

The decisional layer encodes the mission description using a variety of approaches and languages. The underlying formalism of mission description languages may vary from logical languages such as Linear-Temporal Logic (LTL) or Computation Tree Logic (CTL) [22,46,29], to discrete event dynamic system specification as Petri Nets [74,20] or finite state machines [10,66], and Behavior Trees [27,14,25].

Automated Planning has been used for instance to combine high level tasks in order to solve a problem [52,53,30,19,18], based on the current state of the robot, and the desired goal/final situation. The concatenation of tasks provided by an automated planner brings the robot to its goal. Automated Planning spouses well the skill formalism as it defines transition in terms of pre- and post-conditions using STRIPS [23] or a STRIPS derived modeling language [57]. In particular, a temporal description of planning actions, which correspond in the model to the robot skills, has been developed to ensure concurrency at the task planning level [51].

A declarative way to chain skills in a modular manner, in a fully or partially modeled mission, is done through Behavior Trees (BTs). This formalism allows describing to a higher level of abstraction than most of the skills-related programmings, and to model more complex skills chaining and coordination, including skills executions conditioned on system or environment variables. Behavior Trees have gained attention among roboticists for this reason [14,58,25]. Several robotic skill applications delegate BTs for the control flow of the mission. Generally, BTs are specified by the programmers [37, 15,14,73,6,1], but they can also be generated by Reinforcement Learning (RL) [35,36, 5,44] or Evolutionary Algorithms [59] to adapt to a specific task. Automated Planing is also a technology of choice to generate robotics mission scenarios, and several works go towards this direction to produce BTs in the shape of a plan [62,13,72,51].

2.3 Skill formalism

Skill characterization and parametrization are generally performed via an ad hoc specification language. A relational model representing the interaction between skills (or elementary behaviors) and resources and data (provided by the robot or external) has been proposed in [54]. This model however neither represents the behavior of resources, nor possible terminal modes of the skill executions. In [67] skills are represented with preconditions and device resources requirements in an OWL ontology, along with mechanisms to define synchronization of skill execution. This extension of *ROSETTA*, a set of ontologies of robot skills aimed at supporting the reconfiguration and adaptation of robot-based manufacturing cells [64,65], deals with a rather poor skill model, e.g., with no information about failure modes or rates.

A graph-based skill formalism to describe robot manipulation skills and bridge the gap between high-level specification and low-level adaptive interaction control is introduced in [34]. The transitions between manipulation primitives during a skill execution are based on pre, post, and fault conditions. However, even if the system is interfaced with an adaptive controller, the transition system, and skills concatenation remain relatively simple.

LightRocks [66] is a domain specific language (DSL) for robot programming. It defines a transition system between elemental actions, which result in a FSM. Contrary to the usual skill-based architecture, which fundamentals will be described in the years following this publication, skills are supposed to be coded by robot experts as net of the Primitive actions; higher level tasks can be implemented by combining skills, and do not require expert knowledge. The different layers of the architecture implemented with LightRocks can then be compiled in a code using a single Generic Action Component, which interfaces eventually with the API of the robot.

2.4 Implementing skills

Designing skills is supposed to be accessible to any user. A common drawback is, however, the need for laborious and complex parametrization, resulting in a manual tuning phase that is deemed necessary to find satisfactory parameters for a specific skill [61,60]. Such task can be made easier by a representation language [43]. Automated approaches have been used, like Reinforcement Learning (RL), for acquiring new motor skills from demonstration [8]. RL is employed to learn motor primitives that represent a skill [7,2]. In [49], supervised learning by demonstration is used in combination with dynamic movement primitives to learn bipedal walking in simulation.

In the field of Industry 4.0, the need for adaptation of robotic platforms for a wider product variation has raised the attention on skill-based architecture [2]. To facilitate different products manufacturing and consequently providing decreasing production costs, "flexibility" is searched in robotic applications, ideally embedding decisional capacities, and the ability to adapt to different tasks. Fully autonomous robots are then supposed to make their own decisions to perform tasks in constantly changeable environments without operator's interaction [7,72,51].

The Human-Robot Interaction (HRI) is central in many skills related works [53, 63], because of the need to overcome the requirement of an expert supervision to reprogram robots to accomplish certain tasks interacting with humans [70]. Functional User Interfaces allow skills to be easily concatenated, through the use of simple task-level programming methods, to program a variety of tasks that can then be deployed and executed by the autonomous robot.

2.5 Skill verification

One challenge when deploying skill-based architectures is to guarantee properties regarding the execution of the skills. Two main approaches may be cited: static verification, focusing on verifying properties on the model of the skill, and dynamic verification to check properties when the system is running. Static verification and testing have been widely studied, but there are few works on run-time verification, i.e. mechanisms able to detect properties violation and engage recovery mechanisms to keep the system in an acceptable state. This is part of the overall concept of fault tolerance coming from the dependability community [4]. Many works are focusing on the functional level of architectures, to implement fault detection and recovery, but few are focusing on the skill level. Some previous examples, like the work in [55], was actually running run-time verification at the skill level, but without any formalization of the skill concept.

On the contrary, authors of [40] focused on safety of the skill-based architecture adding a monitoring model, such that the safe execution of each task is guaranteed by adding a new safety skill to each task. The role of such skill is to monitor, and eventually stop the task execution if a faulty behavior is detected in any of the skills composing the task. Each skill has then the capacity of triggering an alarm to a monitoring process that runs continuously in parallel with the execution of the functional primitives of the skill. Pre and post conditions skill checks were also introduced by [53] to identify faults at execution-time.

2.6 Conclusion

As discussed, several skills-based formalism and architectures exist in robotics, as the approach is not entirely novel and is gaining momentum in robotic applications. However, a complete development framework that allows to describe complex interactions between skills and resources, data, and the environment, is rare to find. In this work, we propose a development process for skills-based robotic architecture using a novel skill-specification language, from which it is possible to generate the code for platform-specific implementation. This allows users that are not expert programmers to adapt their implementation by modifying the generated code from a high level description of the skills and their relational model. In that way, specific low-level functions can be composed in a more complex framework using skills. The verification-oriented approach we tackle also ensures that some basic execution properties of the skills are guaranteed: skills execution can then be verified, in order to detect that no undesired states would be visited at run-time. Dynamic verification is performed on the transition function of the skills, as it is expressed by the robot language described in this paper.

3 Skills-Based Development Process Overview

3.1 Contribution

This paper presents a development process for dependable robotic architectures that is settled on the concepts of *skills*. These skills, grouped into *skillsets*, are on one side an abstraction of the functional architecture of robotic systems, and therefore give to the decision layer a high-level description of nominal executions and failure modes of the system, necessary to define higher level behaviors. On the other side, skills can act as controllers of the functional layer, by orchestrating the use of sensors/processing/actuators while satisfying properties. In the development process proposed, the central element is the *skillset*. The objective of this development process is to define the skill-layer of an autonomous robot, to help the development by generating components that manage the skills execution, and to propose verification and analysis tools that make the resulting architecture trustworthy in its management of failure conditions.

The development process is presented in Fig. 2. The first step of the process consists in defining the skillsets of our robotic system. Skillsets are mainly composed of resources and skills, both formally specified. A Domain Specific Language (DSL) then allows

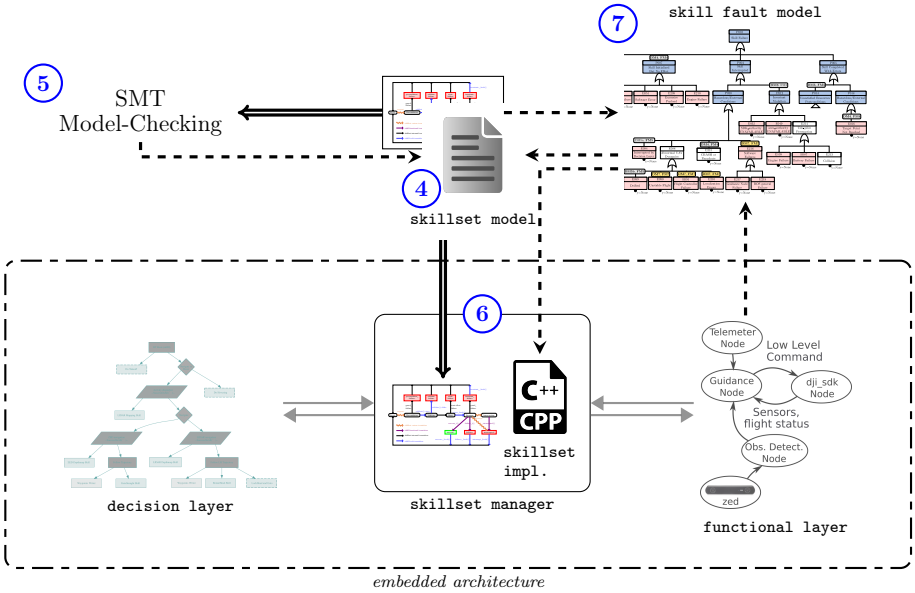

Fig. 2 Overview of the proposed development process. Legend: double links: model or code generation; dashed links: manual modelling or updates; plain links: layer interactions.

to specialize these resources and skills for a specific system. The skillsets elements, formalization, and the DSL, are presented in Section 4.

The next step of the process is to verify the correctness of the skillset models specified by the developer. The model written using the DSL and the semantics of skillsets are translated into a Satisfaction Modulo Theory (SMT) problem, on which the solver Z3 [47] is applied to check for inconsistencies. In case some inconsistencies are raised, the general process calls for improving the skillset model to avoid inconsistent behaviors. This verification step is presented in Section 5.

Once a correct model of a skillset is designed, the next step of the development process is to generate an execution managing component. This component, called the *skillset manager*, implements the execution semantics of the skills. It also provides a standardized interface to trigger skill execution from a decision layer. Finally, the link with the functional layer must be implemented by the user as a specific `C++` class, that extends the general skillset semantics. The part of the skillset manager that corresponds to a system specific code is called the *skillset implementation*. The resulting skillset manager takes the form of a ROS2 node, managing the execution of the skillset according to its semantics, making the link with the functional layer, and providing an API to be controllable from the decision layer. The description of the skillset manager is presented in Section 6.

Finally, as the skillset verification is only based on skillset models, we need an extra step to analyze the complete behavior of both the skillset and functional layers, in order to check that the potential failure conditions expressed in the skillset model actually cover (possibly unambiguously) the potential failures of the functional layer or the robotic system itself. To do so, the next step of the development process follows a Fault-Tree Analysis (FTA) methodology, in which the failure of each skill is analyzed

as an independent event. The top part of each fault tree directly comes from the skillset models, by applying a specific pattern. The fault-tree is then manually extended by analyzing the skillset implementation, the functional layer, and the robotic system. This FTA can result in recommendations for updating either an existing skillset model or skillset implementation to better manage possible failures. It can also be used during the skillset conception process itself, to help to identify the different success/failure termination modes and the corresponding termination conditions. This FTA step is further described in Section 7.

The work presented here has been partly published in the following papers:

- in [41], we proposed a first version of the skillset definition language;
- in [1], we mainly described how to use skillset models to implement a robust decision layer using Behavior Trees (BTs) for a UAV application;
- in [45], we presented the fault tree analysis of skill-based architectures.

The present paper improves these contributions to the following points:

- a new version of the skillset description language is presented, and the execution semantics of skillsets is detailed;
- the verification of skillset models is a new contribution;
- all the steps are presented in a unified manner as a complete development process for autonomous and dependable robotic systems.

3.2 BVLOS Infrastructure Inspection

In order to illustrate the development process, we use, in the next sections, a BVLOS Infrastructure Inspection example. This use case describes the inspection of a building with a UAV. The inspection is executed by turning all around the building in a BVLOS scenario (UAV Beyond the Visual Line of Sight of the safety telepilot). Even if the flight plan is executed automatically, the aerial regulation requires that the safety telepilot can take back manual control at any moment, which assumes permanent video feedback. This feature is provided by a `stream` skill which adjusts the video compression rate to guarantee a minimum frame rate and a minimum image quality criteria. During the inspection, we assume that the UAV travels through different communication zones featuring more or less degraded bandwidth, until entering a critical bandwidth zone (i.e., for which the frame rate criteria cannot be respected anymore). The implemented architecture must then be able to cope with such a situation. Also, to fulfill aerial regulations, some evidences on the dependability of the UAV architecture must be provided. The application of the several steps of the development process proposed in this paper is presented in Section 8.

4 Skills Modeling

The modeling relies on a language that we want to be unambiguous and that allows describing the capabilities of different robots. It is called *robot-language*. This language supports the modeling, the various analyses, and the generation of the code that is then embedded and deployed on different robots. It is important to notice that the runtime semantics of a skillset relies on three distinct elements: its model defined using

robot-language; its functional definition (the skillset implementation) implemented by `C++` code placed in specific locations (called hook), and its use by the decision layer which produces (so-called external) requests on the skillset.

4.1 Skillset

A skillset, as the name suggests, represents the different capabilities of a robot and the accessible data and resources it uses. A skillset has a name and contains a set of data (denoted by $\mathcal{D}$), a set of resources (denoted by $\mathcal{R}$), a set of events (denoted by $\mathcal{V}$), and a set of skills (denoted by $\mathcal{S}$). Listing 3 shows the definition pattern of a skillset and its different elements.

```
skillset uav {
    data     { ... } // skillset data definition
    resource { ... } // skillset resources definition
    event    { ... } // skillset events definition
    skill    { ... } // skillset skills definition
}
```

Fig. 3 Skillset robot-language elements.
A skillset definition can contain: data, resources, events and skills.

In the remainder of this section, we will use the model of the UAV performing the mission described in Section 3.2 to illustrate these different elements of a skillset.

4.2 Skillset Data

Each skillset has `data` that represents the data it owns and makes available to other elements of the architecture. This data is characterized by a name and a type, and may have a period. The data can be retrieved by three different means: (i) when one of the data is modified within the skillset, it is then automatically published; (ii) by queries; (iii) periodically if the period is specified in the *robot-language* model; in that case, the period is specified in seconds. In the example of Listing 4, all the data are published at each modification and on request.

```
    data {
        battery  : Battery
        position : GeoPoint period 1.0
        home     : GeoPoint
    }
```

Fig. 4 Skillset robot-language data definition example.
The UAV skillset has three pieces of data that it makes available to its users: its current position (*position*), its home position (*homepoint*), and its battery level (*battery*). In this model, only the current position is published periodically, every second.

4.3 Skillset Resource

As the name suggests, `resource` are intended to represent resources, whether they are hardware (e.g. sensors) or software/conceptual (e.g. authority). Using the robot-language specification, a resource is a state-machine defined by a name, an initial state, and a set of transitions between states. Listing 5 shows the definition of the UAV resources. The resource is an important concept of the execution model of the skillset architecture because it allows specifying the conditions of operation, and the impact of the robots' actions on the system. Consequently, and we will detail it later, resources are used in the preconditions, invariant, and different effects of the robots' skills. The state of the different resources is both automatically published at each modification of the internal state of the skillset and also on request.

```
resource {
    authority {
        state { Free Pilot Software }
        initial Pilot
        transition {
            Free     -> Software
            Free     -> Pilot
            Software -> Free
            Software -> Pilot
            Pilot    -> Free
        }
    }
    flight_status {
        state { NotReady OnGround InAir }
        initial NotReady
        transition all
    }
    motion {
        state { Available Used }
        initial Available
        transition all
    }
    battery {
        state { Good Low Critical }
        initial Good
        transition {
            Good -> Low
            Good -> Critical
            Low  -> Critical
        }
    }
    ...
```

Fig. 5 Skillset robot-language resource definition example.
The UAV has different resources. The *authority* resource indicates which entity can interact with the UAV. The pilot can regain control of the UAV at any time. However, the UAV can only regain control if the authority is *Free*. Resource *flight_status* indicates the status of the UAV.

Each resource $r \in \mathcal{R}$ is composed of a list of n states $S^r = \{S^r_0 \dots S^r_{n-1}\}$ (with S^r_0 the initial state) and a list of transitions $T^r \subset (S^r \times S^r)$. By convenience, we also

use the following functional notations: $states(r) = S^r$, $transitions(r) = T^r$ and for each transition $t \in T^r$, $src(t)$ represents the source state of the transition and $dst(t)$ its destination.

4.3.1 Resource Guard

A resource guard $\mathcal{G}$ is a logic formula on the state of the resources. The guard is used in the events, the preconditions, and the invariants of the skills, and is defined in the following equation:

$$\mathcal{G} := \texttt{true} \mid \texttt{false} \mid r \texttt{ == } S_i^r \mid r \texttt{ != } S_i^r \mid \texttt{not } \mathcal{G} \mid \mathcal{G} \texttt{ and } \mathcal{G} \mid \mathcal{G} \texttt{ or } \mathcal{G} \tag{1}$$

$eval()$ is the evaluation function of the guard formula on the current state of the system, and is defined as follows:

$$\begin{aligned} eval(\texttt{true}) &\equiv \top \\ eval(\texttt{false}) &\equiv \bot \\ eval(r \texttt{ == } s) &\equiv \text{iff } state(r) = s \\ eval(r \texttt{ != } s) &\equiv \text{iff } state(r) \neq s \\ eval(\texttt{not } x) &\equiv \text{iff } eval(x) = \bot \\ eval(x \texttt{ and } y) &\equiv \text{iff } eval(x) \wedge eval(y) \\ eval(x \texttt{ or } y) &\equiv \text{iff } eval(x) \vee eval(y) \end{aligned} \tag{2}$$

where $state(r)$ represents the current state of resource r.

4.3.2 Resource Arcs and Effects

A resource effect aims to change the state of a specific resource. It is defined by the name of the resource and its next state, and is called an arc:

$$\mathcal{A} := r \rightarrow S_i^r \tag{3}$$

Note that the current state of the resource is not specified.

An effect is a set of arcs: $\mathcal{E} = \{a \in \mathcal{A}\}$. The effects are used in the events and the skills (if a precondition fails, at start, if an invariant fails, and to terminate). In order to ease the reading, we defined for each arc $a \in \mathcal{A}$ the functional notations: $resource(a)$ the corresponding resource and $next(a)$ the next state of the arc. An effect is valid if and only if it contains at most one arc for each resource, i.e.:

$$\forall_{a_1,a_2 \in \mathcal{A}} \left(resource(a_1) = resource(a_2)\right) \implies (a_1 = a_2) \tag{4}$$

In the robot-language specification, our tool ensures that all effects are valid. It is important to notice that even if an effect is validated, it is not guaranteed to be applied. Indeed, if there is no transition from the current state of the resource to the next state, then the effect is impossible. The function $check(e)$ checks if an effect $e \in \mathcal{E}$ is possible, and is defined as follows:

$$\begin{aligned} check(e) \equiv \forall_{a \in e} \exists_{t \in T^r} \mid & \left(src(t) = state(r)\right) \wedge \left(dst(t) = next(a)\right) \\ \text{with } & r = resource(a) \end{aligned} \tag{5}$$

4.4 Skillset Event

The purpose of events is to enable state changes on one or more resources, from outside the skillset manager. An event is defined by its name, a *guard* (optional), the $effects$ (optional), and a *hook*. The *guard* and the $effect$ are defined in the robot-language specification. The event *hook* is a piece of code defined by the developer of the skillset. The result of an event can be *success*, $guard\_failure$ if the guard is not satisfied when the event is called, or $effets\_failure$ if the effects cannot be applied. Listing 6 shows the model of event *take_ authority*.

```
event {
    take_authority {
        guard  authority != Pilot
        effect authority -> Pilot
    }
    ...
```

Fig. 6 Skillset robot-language event definition example.
This effect aims to give the authority of the drone to the pilot if the pilot does not have it already.

Algorithm 1 describes event processing. At first, the skillset is "locked" to ensure the consistency of the inner elements of the skillset (data, resources, status, ...). Then the guard is evaluated, if the guard is satisfied then we check if the effects can be applied. If they can all be applied, then the event hook is executed, and then the effects are applied. It is important to notice that the event hook is applied **only** if the guard is satisfied and **all** the effects are allowed. If at least one part of the effects is not possible, none are applied to maintain consistency of the system and the correctness of the specification with the real execution. The $invariant\_loop()$ function aims to propagate the effects into the running skills in a deterministic way and will be detailed later in Section 4.5.

4.5 Skillset Skill

The robot-language allows skills to be defined using the following elements:

- **inputs** are used in the functional algorithm of the skill. Each input is defined as a data and thus has a name and a type.
- **outputs** are data produced at the end of the execution of the skill.
- **preconditions** are used to check if the skill can start its execution properly. Each skill can have zero, one, or more preconditions.
- **start** is used to define some effects on the resources of the skillset before starting.
- **invariant** is used to specify mandatory properties to the execution of the skill. If at least one invariant is violated, then the skill is stopped.
- **progress** is used to monitor the execution of the skill. It produces monitoring data periodically during the execution of the skill.
- **interrupt** is used to specify the effects applied after the skill is interrupted. Any skill can be interrupted.

Algorithm 1: Event algorithm

```
Input: guard, effects, hook
Output: result
begin
    lock_skillset()
    if eval(guard) then
        if check(effects) then
            hook()
            apply(effects)
            invariants_loop()
            result ← success
        else
            result ← effects_failure
        end
    else
        result ← guard_failure
    end
    unlock_skillset()
    return result
end
```

- **success** is used to specify the different execution success states of the skill and their corresponding effects on the resources after its completion.
- **failure** is used to specify the different execution failure states of the skill and their corresponding effects on the resources after its completion.
- **postconditions** are used to specify the states of the resources at the end of the execution of the skill. Contrary to the effects, postconditions is a resource formula and the real effect is done by the functional layer. Invariants, success, and failure states can have postconditions.

The internal behavior of a skill follows the state-machine depicted in Fig. 7. When a skill is called then the first step is to check each precondition, if one fails then the skill stops and returns an error, if all the preconditions are satisfied then the skill continues its internal state-machine to the *validation* step. The validation step consists in running the validation hook. If the validation hook fails, then the request is terminated, otherwise, the skill continues its internal state-machine and the effects specified in the start block proceed by checking if all its effects are possible. If at least one effect is not possible, then the skill returns a "start error". If all are possible, then they are applied. Once the start is executed without error, the skill is running and then the invariant loop function is called to check all the invariants of the skillset. The invariant loop can lead other skills to terminate with an invariant failure, and the effects of those invariant failures can also imply other invariant failures, and so on. The invariant loop terminates because a stopped skill cannot re-enter in the running state. While running, the skill can finish its execution with an invariant failure or terminate its execution: by being interrupted or can succeed or fail. The interrupt is an external request as the start of the skill is. The success or the failure of a skill is an internal call from the functional part of the skillset that also specifies the success (resp. failure) state.

Moreover, each skill also has several hooks. Each hook is linked to a specific element of the complete behavior of the skill execution:

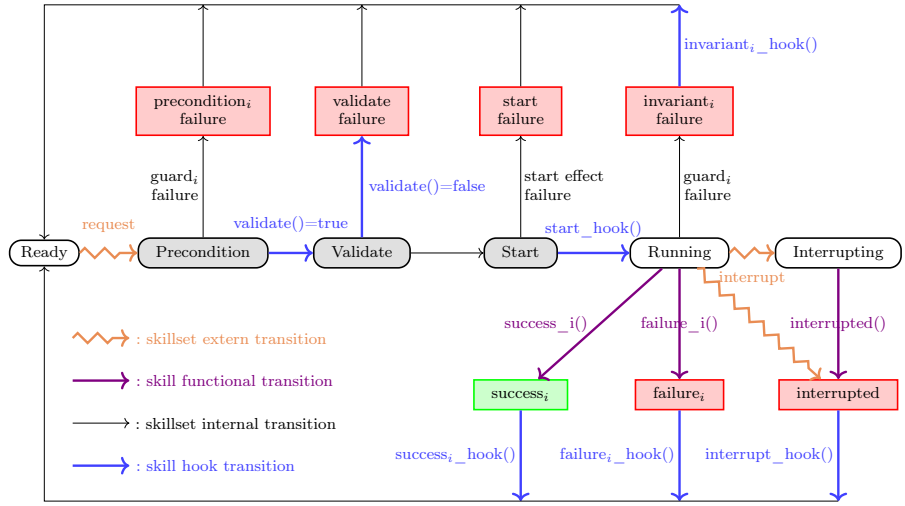

Fig. 7 Skill state-machine.
In this figure, the gray rounded rectangles represent the internal states of the skill state machine (precondition, validate, start). The white rounded rectangles represent the observable internal states of the skill. The rectangles represent the output states (red for failures, and green for success).
The orange arrows represent requests from outside the skill: start or interrupt the skill. The blue arrows represent the different hook calls. The purple arrows represent the functional calls that interact with the skill. Note that to terminates a skill, the functional part must explicitly call the `C++` function corresponding to the desired output state.

- **validate** hook is used to define whether a skill request can start. This hook is only related to the functional elements of the skill and the inputs of the request (i.e. not the resources).
- **start** hook is executed when the skill is started.
- **invariant** hooks are executed if an invariant is violated. Each invariant has its own hook.
- **progress** hook is executed when the skill is running at the period of the progress. This hook also aims to produce the outputs specified in the progress block (if any).
- **interrupt** hook is executed when the skill is interrupted.
- **success** hook is executed at the end of the execution of the skill if it terminates with a success. Each success state has its own hook.
- **failure** hook is executed at the end of the execution of the skill if it terminates with a failure. Each failure state has its own hook.

The execution of a skill is described in Algorithm 2. It follows the state machine presented earlier. First, when a skill is called, it checks if it is already running. If it is, it instantly returns an error. If the skill is available, then it starts the evaluation of its state-machine step by step: evaluation of preconditions, start, invariant. If those steps succeed, then the skill is running. It is important to notice that the effects are either all applied or none is applied. Moreover, the corresponding hooks are executed only if all the effects have been applied.

The `invariant_loop()` function consists in checking the invariants of all skills in the skillset. If at least one invariant failed, the corresponding skill is stopped and the

Algorithm 2: Skill main algorithm

```
Input: inputs, validate_hook, preconditions, start
Output: result
begin
    lock_skillset()
    if is_running then
        result ← already_running
        unlock_skillset()
        return result
    end
    foreach precond ∈ preconditions do
        if not eval(precond.guard) then
            result ← precondition_failure
            unlock_skillset()
            return result
        end
    end
    if not validate_hook(inputs) then
        result ← validate_failure
        unlock_skillset()
        return result
    end
    if not check(start.effects) then
        result ← start_failure
        unlock_skillset()
        return result
    end
    start.hook()
    apply(start.effects)
    is_running ← true
    invariants_loop()
    unlock_skillset()
end
```

corresponding effects are applied and its hook is executed. Because the invariant failure of a skill can have effects, it can invalidate the invariants of another running skill, and so on. It is important to notice that the invariant loop function is executed inside the skillset lock. Indeed, having the invariant loop outside the lock could lead to a configuration in which a skill can be seen as running with an invariant not satisfied.

4.5.1 Inputs and outputs

The skill inputs and outputs are defined with a name and a type, like the data are. Their types must have been defined. Figure 8 shows the inputs and the outputs of the `takeoff` skill.

4.5.2 Preconditions and start

A skill can have several preconditions and only one start block. Each precondition is defined by its name, a guard, and can have effects. The start block simply has effects. Figure 9 shows the preconditions and the start of the `takeoff` skill.

```
skill takeoff {
    input {
        height: Float // [m] validate can fail if h>h_geo_fence
        speed : Float // [m/s] maximum ascending velocity
    }
    output {
        height: Float // [m] validate can fail if h>h_geo_fence
    }
```

Fig. 8 Robot-language Skill inputs and outputs definitions.
The `takeoff` skill has two inputs: the altitude to be reached and the maximum ascent speed. It also has an output: the altitude reached at the end of the take off.

```
    precondition {
        has_authority: authority == Software
        on_ground    : flight_status == OnGround
        motion_avail : motion == Available
        battery_good : battery == Good
    }
    start motion -> Used
```

Fig. 9 Robot-language skill preconditions and start definition for the skill `takeoff`.
To take off, the drone must (preconditions): have authority, be on the ground, and no other skill control the drone (`motion` is `Available`). When the take-off begins (start effect), the skill takes control of the drone.

4.5.3 Invariant

Each skill invariant is defined by a name, a guard and its effects applied when the invariant is violated. Figure 10 shows the invariant definition of the `takeoff` skill.

```
    invariant {
        in_control {
            guard motion == Used
        }
        has_authority {
            guard  authority == Software
            effect motion -> Available
        }
        battery {
            guard  battery != Critical
            effect motion -> Available
        }
    }
```

Fig. 10 Robot-language skill invariant definition for the skill `takeoff`.
During the entire take off phase, the drone must maintain authority and motion control. If either the authority or the motion control is loss, then motion control is set to `Available`.

4.5.4 Running

While running, it may be useful to get information about the completion of the skill. This is achieved with the progress block, which has a period in seconds and outputs. During the execution, the skill will produce the specified outputs with the rate according to the period. Figure 11 show the progress and update definitions of the `takeoff` skill.

```
progress {
    period 1.0
    output height: Float
}
```

Fig. 11 Robot-language skill progress definition for the skill `takeoff`.
During its execution, the `takeoff` skill periodically, every second, publishes its current altitude.

4.5.5 Terminate

While running, a skill can finish its execution with an invariant failure or can terminate its execution by being interrupted, or can succeed or can fail. A skill can succeed (resp. fail) in different states, and each state has a specific hook and effects. Figure 12 shows the interrupt definition, the unique success definition, and the different failures of the `takeoff` skill. Sometimes a skill cannot be interrupted instantaneously. The purpose of the parameter `interrupting` is to specify the fact that the interrupt is instantaneous (set to `false`) or can take time (set to `true`). In this case, while receiving the interrupt, the skill switches to the "interrupting' state and waits that the functional layer indicates the interrupt completion (interrupted). The postconditions are properties that represent the normal behavior of the skill. A postcondition can also be seen as part of the specification for the functional layer. The evaluation of the postconditions is returned at the end of the skill execution. Thus, if one fails, then the functional layer can react properly.

4.6 Conclusion

In this section, we introduced the specification language *robot-language*. This language allows the definition of skillsets that contain, among others, the available data, the resources, the events, and the skills. This language is the cornerstone of both the different verifications and the `C++` code generation.

5 Skillset model Verification

In this section, we present the skillset model verification. The main objective is to independently check the different elements of the skillset. We seek to show by this verification that the different elements of the model are correctly defined. The notion of resource is a cornerstone of the semantics of skills. Indeed, it constrains the guards,

```
interrupt {
    interrupting true
    effect motion -> Available
}
success at_altitude {
    effect motion -> Available
    postcondition flight_status == InAir
}
failure {
    grounded {
        effect motion -> Available
        postcondition flight_status == OnGround
    }
    emergency {
        effect motion -> Available
        postcondition flight_status == InAir
    }
}
```

Fig. 12 Robot-language skill terminate definitions for the skill `takeoff`.
When the skill is interrupted, the motion control is released. The skill has a success state (*at_altitute*) which, if reached, makes motion control available and indicates that the drone is in the air. The `takeoff` skill has two failure states: one in which the drone is on the ground and one in which it has begun its take off, but has failed to reach the requested altitude. The different postconditions indicate the states of the resources, once the skill terminates.

the effects present in the events, and the skills and consequently impacts the whole behavior of the skillset. Thus, the skillset model verification consists in checking that the guards make sense, and checking that the effects are possible. It is important to note that, even if these checks fail, the system will run without crashing and the execution has a predictable and correctly defined behavior. But it is likely that this behavior is not the desired one and can be associated with an "erroneous" specification.

5.1 Guard Verification

The first check is on the guards. We formally check that the guards make sense. That is, they can be true or false. More precisely, since the guards relate to the state of the resources, we seek to verify that there is at least one configuration of the resources in which the logical formula of the guard is evaluated as true **and** at least one configuration in which it is evaluated as false. The guards are present in the events, preconditions and invariants of the skills.

To solve this problem, we use the Sat Modulo Theory formalism, as it allows us to represent problems involving logical formulas and some structured data useful for our modeling. Each resource is represented with a Sort DataType (Enum) which contains all the states of the resource. The transitions of the state machine are represented by a function that indicates whether there is a transition between two states. The current state of each resource is translated by a corresponding spell constant. The guard formulas are simply translated following the semantics given in Eq. (2). To check if the guard holds, its corresponding constraint is added to the solver. Then we can ask the solver if the system has a solution. If a solution exists, we can ask for the state

of the variables and thus find the state of the resources that lead to the satisfaction of the model.

5.1.1 Event

The method presented above is applied to the guard of each event, in order to determine whether it can be evaluated as true. Listing 13 highlights the SMT model of the *take_ authority* event previously defined in Listing 6.

```
(declare-datatypes () ((authority Free Software Pilot)))
(declare-const authority_current authority)
(assert (not (= authority_current Pilot)))

(check-sat)
; returns sat
(get-model)
;   (define-fun authority_current () authority Free)
(eval authority_current)
; Free
```

Fig. 13 SMT-lib-2 model for take_authority event guard verification.
The first line is used to declare the resource 'authority' and its states. The second line declares the current state of the resource. The third one represents the guard constraint. Finally, the last lines are used to call the solver and get the results. The comments (in green) show the result of the evaluation.

5.1.2 Skill precondition and invariant

In the skills, the guards are present in the preconditions and invariants. However, unlike those present in events, they are linked together. Indeed, the preconditions (resp. invariants) are evaluated in the order of their declaration in the model. Consequently, to test the usefulness of a precondition (resp. invariant), all the previous preconditions (resp. invariant) must be satisfied. Thus, a guard of a precondition is useful if it can be both true and false:

$$\text{can-be-true}_i = (\forall_{j \in 1..i-1}\ guard_j) \wedge guard_i \tag{6}$$

$$\text{can-be-false}_i = (\forall_{j \in 1..i-1}\ guard_j) \wedge \neg guard_i \tag{7}$$

5.2 Effects Verification

The second check is on the effects. We have shown in the previous section that the execution semantics of the specification language is well-defined, whether the effects can be realized or not. However, in the general case, when effects are specified, it is important that they are realized. Therefore, it is essential to determine whether effects will always be realized or whether there are configurations in which they may fail.

To analyze the effects, we use the same method as for the guard study. We create an SMT model of the resources, the guards, the effects, and using the solver, we look for a counter-example that shows a configuration in which the analyzed effect can fail.

5.2.1 Event

The effect of an event can fail if there is a solution to the SMT problem in which the guard is satisfied, and there is at least one arc of an effect that is not feasible.

$$\text{event-effect-can-fail}_i = guard_i \wedge \exists_{r \in resources(\mathcal{E}_i)} \neg\, transition(current(r), next(r)) \quad (8)$$

To illustrate effect failure detection, let's consider the *take_authority* event we defined earlier. We will voluntarily introduce an error in the effects by slightly modifying the state machine of the "authority" resource by no longer allowing the pilot to take control when the drone has the commands. Listing 14 shows the new state machine of the resource.

```
resource {
    authority {
        state { Pilot Free Software }
        initial Pilot
        transition {
            Free     -> Software
            Free     -> Pilot
            Software -> Free
            Pilot    -> Free
        }
    }
```

Fig. 14 Robot-language skillset resource definition (incomplete).

Listing 15 shows the SMT model of the event effect failure check.

The function *authority_transition* represents the state-machine of resource *authority*. In this example, the verification highlights an effect error. Indeed, if the resource authority is *Software*, the effect cannot be applied because there is no transition from the state *Software* to the state *Pilot*.

5.2.2 Skill

Precondition

In order to study the failure of an effect of a precondition, we assume that all the previous ones hold and that the current one doesn't. We then check the precondition effect by applying the same method used above for the events. The same principle is applied to the invariants.

$$\text{precondition-effect-can-fail}_i = (\forall_{j \in 1..i-1}\ guard_j) \wedge \neg\ guard_i \wedge \exists_{r \in resources(\mathcal{E}_i)} \neg\, transition(current(r), next(r)) \quad (9)$$

Start

A start effect can fail only if all the preconditions are stratified and the effect fails.

$$\text{start-effect-can-fail}_i = \left(\forall_{p \in preconditions}\ guard_p\right) \wedge \exists_{r \in resources(\mathcal{E}_i)} \neg\, transition(current(r), next(r)) \quad (10)$$

```
; Resource res definition
(declare-datatypes () ((authority Free Software Pilot)))
(define-fun authority_transition ((x authority) (y authority)) Bool
    (or (= x y)
        (and (= x Free)  (or (= y Software) (= y Pilot)))
        (and (= x Software) (= y Free))
        (and (= x Pilot) (= y Free))
    )
)
(declare-const authority_current authority)
(declare-const authority_next    authority)
; guard: authority != Pilot
(assert (not (= authority_current Pilot)))
; effect result: authority -> Pilot
(assert (= authority_next Pilot))
; no transition
(assert (not (authority_transition authority_current
    authority_next)))

(check-sat)
; returns sat
(get-model)
;   (define-fun authority_current () authority Software)
;   (define-fun authority_next () authority Pilot)
(eval authority_current)
; Software
```

Fig. 15 SMT-lib-2 model for take_authority event effects verification.

Invariant
The error analysis of invariant effects is similar to that of preconditions. The effects of an invariant can fail if and only if all invariants preceding it are satisfied and its guard is false.

$$\text{invariant-effect-can-fail}_i = (\forall_{j \in 1..i-1}\ guard_j) \wedge \neg\ guard_i \\ \wedge \exists_{r \in resources(\mathcal{E}_i)} \neg\ transition(current(r), next(r)) \quad (11)$$

Terminate
The error analysis of terminate effects follows the same principle as above. An effect can fail if all invariants are satisfied and the effect fails.

$$\text{terminate-effect-can-fail}_i = (\forall_{inv \in invariant}\ guard_{inv}) \wedge \neg\ guard_i \\ \wedge \exists_{r \in resources(\mathcal{E}_i)} \neg\ transition(current(r), next(r)) \quad (12)$$

5.3 Conclusion

In this section, we presented the skillset verification process. It allows checking if the model defined with robot-language is well-defined. To achieve this goal, the tool checks if the different guards of the event, the precondition, and the invariants are useful (they can be either evaluated to be true or false). Moreover, the tool allows verifying if the effects can always be applied (never fail). Those two formal verification steps are

important because if satisfied they highlight the fact that the model is correct and also that the execution of the skillset will always satisfy its specification.

6 ROS2 Code Generation and Skill Implementation

The development process we propose comes with a code generation toolchain. This toolchain translates a *robot-language* skillset model into a set of ROS2 packages. To do this, it is necessary to make the abstract types defined in the model concrete; this is done using a separate configuration file. We have chosen to distinguish the model from the concrete types in order to be able to generate, for the same skillset, code for possible different targets, in which the concrete types would be different. For instance, the `Battery` data type of Listing 4 can be mapped to a unique floating point value representing the remaining battery percentage, or to the full `sensor_msgs/Battery` standard ROS2 state, without changing the semantics of the skillset model.

The code generator systematically produces two ROS2 packages for each skillset: one for the interface messages and one for the main node of the skillset. These two packages must not be modified by the user. Then, the user can also generate a specific package to implement the hooks of the skillset manager specifying the links with the robot functional layer. A typical directory structure of the generated elements is presented in Fig. 16. Each directory/package is presented below.

6.1 Interfaces

The communication between the decision layer and the skillset managers is implemented through ROS2 topics, so the "interfaces" package contains only messages. Using bare ROS2 topics (instead of the service or action layers) allows us to better configure the quality of service regarding the desired execution semantics.

The messages in the package interfaces include both messages corresponding to the data used by the skillset and the skills (data, input, output, progress, ...), but also all messages corresponding to the requests and responses of the data, the events and the skills of the skillset. In Fig. 16, this package corresponds to the `uav_skillset_interfaces` package.

6.2 Skillset package

The skillset package (`uav_skillset` in Fig. 16) contains an abstract ROS2 node provided as a `C++` library. This node contains the various data specified in the *robot-language* skillset model, as well as the resources. It declares all topics allowing the dialogue with the skillset manager (from the decision layer) but also its introspection (internal status of resources, data and skills). The generated code of the node implements the logic of the different elements present in the model (resource logic, skills state machine, effects on resources, guard management, ...) that have been described in Section 4.

This node provides an "empty" and abstract (i.e., *virtual* in `C++`) implementation of the different hooks of the model. These hooks will have to be completed later by the developer to add the functional part useful for the execution of the final skillset.

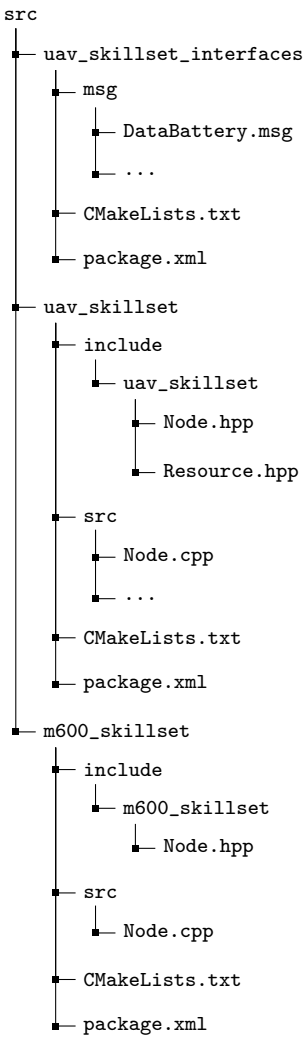

Fig. 16 Typical ROS2 directory structure of the `m600` specialization of the `uav` skillset.

In addition, this node has various methods that allow the functional part (inheriting node) to interact with the skillset internally, i.e. without going through the topics. These methods are declared "protected" to restrict their use to the realization of the final node. These methods allow, for example, a skill to be completed successfully or unsuccessfully.

6.3 Project package

The code generation toolchain also allows generating a ROS2 "project" package for the final implementation of the skillset (the `m600_skillset` in Fig. 16). In this case, a ROS2 node that inherits from the abstract node of the skillset is generated. It is of course

possible to generate different project packages for the same skillset. The developer must then add the `C++` code corresponding to the functional part in the different hooks defined as abstract in the parent skillset node.

7 Analysis of the Skill Layer Fault Tolerance

As presented in Fig. 7, the skill model (including the state-machine) is built around a set of detection mechanisms (validation of inputs, preconditions, invariants, etc.), that may lead to failed states (interrupted, failure, etc.). However, to be able to specify and design these detection mechanisms and failure states for each skill, we propose in this section a process based on a skill fault model. First a generic skill fault model is proposed using fault trees, then an analysis process using this fault model is proposed. As an illustration, this process will be applied to skill `takeoff` in Section 8.5.

7.1 Skill Fault Model

In this subsection, we propose a generic skill fault model. This model will help the designers to identify errors leading to the skill failure, error detection means, and recovery actions (actions to keep the system in an acceptable state regarding high level objectives, e.g. safety or mission).

This skill fault model is based on fault tree analysis (FTA) [17]. 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 with a graphical representation as in Fig. 17, starting on the top with an undesirable event called the *top event*, and then determining how this top event may be caused by individual or combined lower level failures. In FTA, errors from different domains may be combined (e.g., software and hardware errors). Logical relations between them are represented by logic symbols (AND and OR gates). In a fault tree, 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 intermediate or basic event can be interpreted as an error from the overall system viewpoint. We apply this representation by considering skill failures as the top events of our FTA.

Based on the concepts of **skill model** and **skill implementation** described in the previous sections, we propose to decompose the top event corresponding to the skill failure as depicted in Fig. 17.

This generic tree considers that a skill may fail if:

- it cannot be started (node P002) due to validation failure (invalid inputs, unresponsive functional layer, etc.) (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);

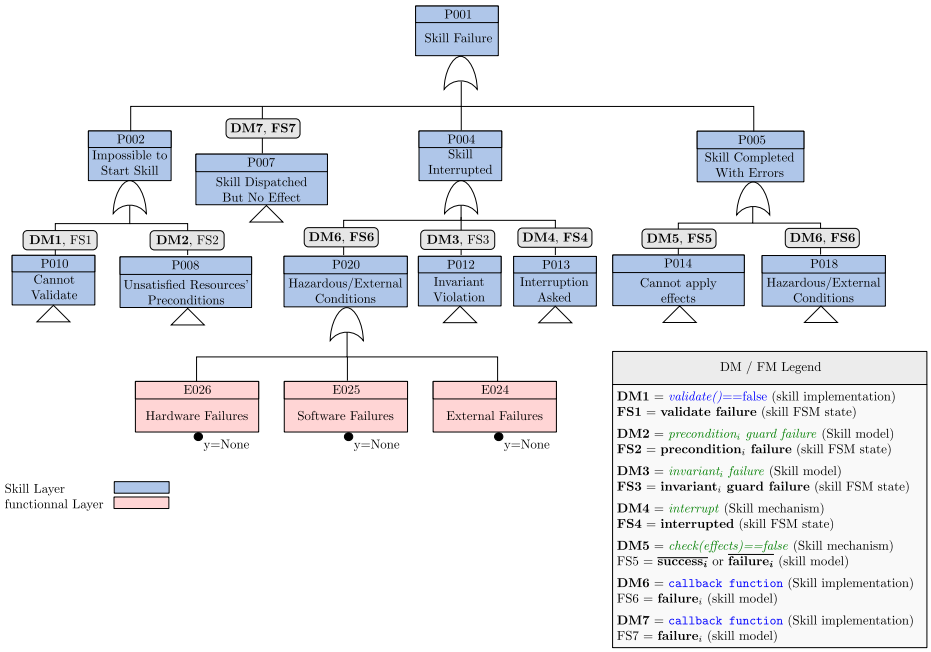

Fig. 17 Skill Fault Model: Pxxx represent fault tree nodes linked to skill layer, Exxx represent fault tree nodes linked to functional layer. DM_i and FS_i are respectively Detection Mechanisms and Failure States. Normal font DM/FS are supposed to be managed in the skill state-machine (e.g., FS1 corresponds to **validate failure** state of the **skill state machine**). Bold DM/FS are supposed to be managed either in the skill implementation (e.g., **DM2** has to be implemented in the *validate* hook) or in the skill model (e.g., **FS4** has to be described as result mode in the **skill model**).

– the skill achieves 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 the basic events happens, the whole skill execution will fail. Such an event is usually called a minimal cut set of order one, and is an identified weakness of the system. Once such a fault tree is determined, it may be then used for quantitative analysis (probabilities calculation of the top event), but in our case we focus on the qualitative analysis, i.e., reducing the minimal cut sets of order one. For that, Detection Mechanisms (DM) and Failure States (FS) are assigned to some nodes (from `DM1` to `DM7` and `FS1` to `FS7`). Figure 17 exhibits for each DM/FS if it is supposed to be covered by the skill model and by which mechanism (invariant, pre, etc.), or if it is supposed to be handled in the skill implementation. For instance, `FS1` is equivalent to state `validate failure` of Fig. 7 and `DM3` corresponds to the *guard failure* transition. This DM and FS 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.

7.2 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 skillset. 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 (Fig. 17), using the following steps:
   (a) Connection of each event listed in step 1 with each **skill fault tree**,
   (b) Determination of all relative failure states (FS_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 FS_i is covered by the skill model or the skill implementation
   (b) Modification of the skill model or implementation to add or correct a missing or incomplete DM_i or FS_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 FS_i identified by FTA are covered. Classically a missing DM_i can be fixed by 1) adding a resource and/or a resource condition (e.g., in `invariant` or `precondition`) 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 FS_i will generally lead to the addition of a new terminal state (and its appropriate management in the model and in the implementation) or to the modification of resource conditions.

8 Application

This section describes how the skill approach has been implemented in a decisional architecture of a drone. The verification activity (presented in section 5) and the fault tolerance analysis (presented in section 7) are partially applied to this case study.

8.1 M600 Architecture and Field Experiments

The complete architecture including the drone functional layer, the skillset layer based on the `uav` skillset model, and the decision layer based on Behavior Trees (BTs) has been implemented on a DJI-M600 platform to perform the BVLOS inspection scenario.

In the experimental setup (Fig. 18), the drone follows a path around a metallic farm composed nominally of three waypoints and should stop the mission if the video stream to the telepilot is lost. A first reduction of the communication bandwidth is artificially simulated when the drone enters the first area in orange. Then a drastic diminution of the bandwidth is simulated when the drone enters the second area in

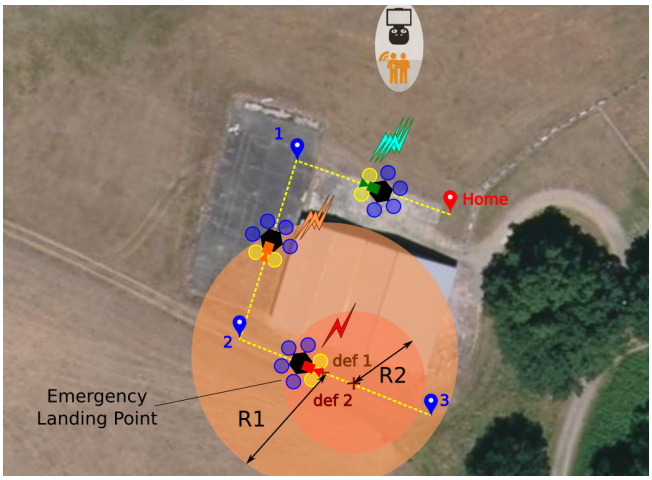

Fig. 18 Illustration of the BVLOS experimental setup. The drone takes off from the home point and has to inspect the building going to waypoints 1, 2, then 3. The communication bandwidth with the safety pilot (located north of the building) decreases along the drone trajectory.

red, implying to immediately stop the mission, start to descent at 3 m ground-height, and then automatically land at the current position.

The skill-layer has been developed according to the process presented in this paper: defining the `uav` skillset, generating the manager node and implementing the necessary hooks, and analysing the fault tolerance of the skill- and functional- layers.

The decisional layer uses the BT's modularity to describe both the nominal mission and the degraded plan in case the primary mission objectives cannot be achieved. The degraded plan is described as a subtree (a branch of the BT) that triggers an alternative mission execution; in our case, a fallback subtree executing an emergency landing on a safe zone.

In this architecture, we defined two skillsets: the `uav` skillset containing skills and resources related to the motion of a UAV, and a `communication` skillset containing the `stream` skill that monitors the quality of the communication link. The `uav` skillset, that has been used to illustrate the process all along this paper, contains three data, six resources, nine events, and five skills.

The realized experiment can be seen on the video available at: `https://youtu.be/mZxyl6v-tDw` and whose a screenshot is presented in Fig. 19.

8.2 UAV Skillset Model

The UAV Skillset model has been designed considering the capabilities available on the UAV used for this experimentation.

The provided **data** are:

- `battery`: figures the level of battery charge
- `position`: the current position of the UAV in WSG84 coordinate system
- `home`: the internal origin of the UAV in WSG84 coordinate system

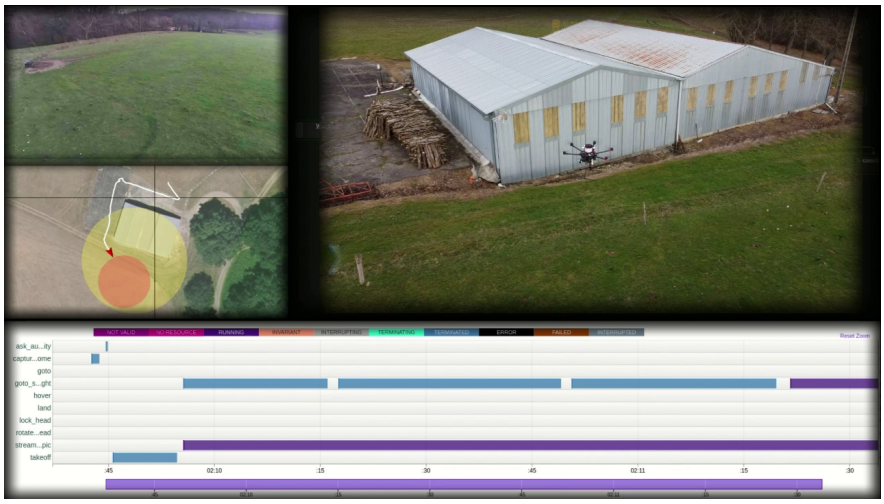

Fig. 19 Screenshot of the experiment video. Top right: external view of the drone inspecting the building. Top left: drone camera view. Middle left: map of the scenario with the drone position. Bottom: timeline showing the active skills and their terminal states.

Each **resource** with its possible states is summarized here (enumeration of transitions are not given for readability reasons):

- `authority {Free, Pilot, Software}`: represents the owner of the UAV control authority
- `home_status {Invalid, Valid}`: indicates if the home point has been acquired since the UAV startup
- `flight_status {NotReady, OnGround, InAir}`: the flight status
- `motion {Available, Used}`: states if the translation motion of the UAV is used by a skill
- `heading {Available, Used}`: states if the heading motion of the UAV is used by a skill
- `battery {Good, Low, Critical}`: provides qualitative state of the battery with respect to its charge level and the UAV specifications

The **events** associated to the UAV **resources** are defined in the Listing 20. Last, the **skills** of the UAV are partially presented by giving only the name of each skill and its inputs if any:

- `ask_authority`: this skill attempts to acquire control authority over the UAV for the other skills
- `capture_home`: set the current UAV position as its home point
- `takeoff {input: (height, speed)}`: requests UAV to take off at a given ground altitude (`height`) with maximum velocity (`speed`)
- `goto {input: (target, speed)}`: requests UAV to reach a position (`target`) with maximum velocity (`speed`)
- `land`: asks the UAV to land on site

```
event {
    authority_to_pilot {
        effect authority -> Pilot
    }
    authority_to_software {
        effect authority -> Software
    }

    home_status_to_valid {
        effect home_status -> Valid
    }
    home_status_to_invalid {
        effect home_status -> Invalid
    }

    flight_status_to_not_ready {
        effect flight_status -> NotReady
    }
    flight_status_to_on_ground {
        effect flight_status -> OnGround
    }
    flight_status_to_in_air {
        effect flight_status -> InAir
    }

    battery_to_low {
        guard battery == Good
        effect battery -> Low
    }
    battery_to_critical {
        effect battery -> Critical
    }
}
```

Fig. 20 Part of the UAV skillset model describing events.

8.3 A BVLOS application using Behavior Trees as decisional layer

The definition of high level behaviors, i.e. mission level specification, are implemented in the decisional layer of our architecture.

The skillset implementation is agnostic with respect to the underlying formalism of mission description languages. We propose in the following to use BTs to implement the control architecture for high-level mission programming, i.e. the decision layer. One of the central advantages of BTs is their modularity, which favors their reusability between different missions. In that way, as subtrees of BTs are still BTs, it is easy to adapt missions or to simply compose already implemented behaviors guaranteeing the same degree of robustness of the implemented task to the new one [15]. Moreover, this very same modularity allows us to describe mission elements while maintaining the flexibility we are aiming at for mission reconfiguration in case of failure.

A BT structure is a directed tree where inner nodes can encode different types of execution models (sequential, parallel, etc.), while leaf nodes can be conditions, calculations, or actions. The BT breaks down the complex task of coding a robotic mission into smaller, independent behaviors, from the root node. While a subtree implements and abstracts several actions and calculations, on a finer scale, the single behaviors

can represent single applications or calls to a function, like a single skill execution, a condition, a logical connective, etc. The leaf nodes execute some computation (calling a skill or performing some calculation), and return their status (*Running* when the execution is ongoing, *Success* or *Failure*). At each control loop (every periodic *tick* of the BT), the tree structure is traversed from left to right, visiting in order each branch of the tree, launching behaviors when needed, or just processing the return status of every behavior. The return status is then progressed back towards the root node of the tree, and modified according to the type of each node. BTs can represent chains of elementary behaviors, where the state transitions are implicitly encoded in the tree structure, instead of explicitly stated in transition tables as for FSM [50, 16]. The skills, on which our architecture is based, are encoded as single behaviors in the BT.

An action behavior is launched when its node is first visited, and its successive execution monitored at every tick via its return status. The return status of an action behavior depends on the result of the related skill. As the execution of the skills can be made synchronous or asynchronous, it is when the skill results and the relative behavior statuses are collected at the ticking of the BT that interruptions can occur. Then, some branch execution can be inhibited by a guard, and other ones consequently activated, possibly triggering a mission reconfiguration.

This transition model, implicitly encoded in the BT structure, ensures that BTs are seen as highly reactive, meaning that more important behaviors interrupt less important ones. This represents, together with their modularity, one of the main advantages of BT-based implementations [32]. We take advantage of these properties to create a full mission specification, where both nominal and fallback subtrees can coexist, in order to perform failure management at execution time. The BT specification of the inspection mission relying on a non-degraded communication of the UAV (in the nominal case) or in an emergency landing when FPS criteria are not respected anymore is illustrated in Fig. 21. After checking that the drone is not in air already, the `takeoff` skill is called (leftmost subtree). The rightmost subtree executes `stream` and `goto` skills in parallel, the latter following tick after tick the waypoints defined in the experimental setup (Fig. 18). If a malfunction occurs when the `stream` skill is running (or during the execution of `goto`), then the fallback rightmost subtree is executed, and runs the `land` behavior instead of streaming, thus terminating the mission.

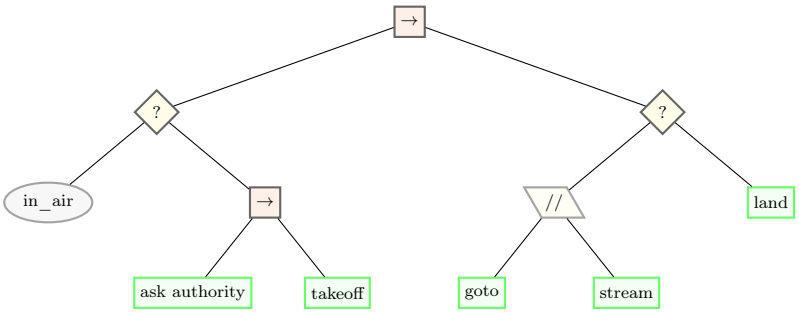

Fig. 21 A (simplified) BT representing the inspection UAV mission.

8.4 Analysing the skillset consistency

Once we designed the `uav` and `communication` skillset, we applied the model-checking step on these models, as described in Section 5. This verification step can return inconsistencies in the model, that lead to change or improve the design of the skillsets.

For instance, let's consider the `goto` model presented in Listing 22. On this example, the verification tool first checks if the preconditions can be satisfied or not. Then, it checks whether the start effect can fail while the preconditions are satisfied. Those checks return `true`.

```
skill goto {
    precondition {
        has_authority: authority != Pilot
        in_air       : flight_status == InAir
        moving_avail : motion == Available
        battery_good : battery != Critical
    }
    start motion -> Used
    invariant {
        in_control {
            guard motion == Used
        }
        has_authority {
            guard  authority == Software
            effect motion -> Available
        }
        in_air {
            guard flight_status == InAir
        }
        battery {
            guard  battery != Critical
            effect motion -> Available
        }
    }
    interrupt {
        interrupting true
        effect motion -> Available
    }
    success arrived {
        effect motion -> Available
    }
    failure emergency {
        effect motion -> Available
    }

}
```

Fig. 22 Model verification example.
The UAV skill `goto` uses the resource definition previously introduced in this paper. This skill can be launched if the pilot doesn't own the authority, the drone is not on the ground, and the battery is charged enough, as it is specified in the preconditions. Once the skill starts, the drone begins to move (cf. definition of start). The skill can keep running if and only if it owns the authority and the drone is not stopped (cf. invariant). Once the `goto` skill terminates, the `motion` authority is set to `Available`.

Then the tool checks if the skill can return an invariant failure just after being started (i.e. without any external modification of the resources). More precisely, the solver try to find an initial configuration of the system in which all the preconditions are satisfied, start succeeds, and at least one invariant is not satisfied. In this case, the solver has found a counter-example in which the initial configuration of the resources is: `authority==Free`, `flight_status==InAir`, `motion==Available`, and `battery==Good`. Indeed, in this configuration the preconditions are satisfied, the start effect can be applied, but the invariant `has_authority` is false, leading to a systematic interruption of the skill once started.

Based on this inconsistency result, we investigated two solutions to fix our model of `goto`: the first one would be to change the precondition if we consider that the drone must have the authority before moving (see Listing 23); the second would be that the drone must take the authority (if not handled by the pilot) when starting to move (see Listing 24). We decided to implement the first solution in our final model.

```
precondition {
    has_authority: authority == Software
    in_air       : flight_status == InAir
    moving_avail : motion == Available
    battery_good : battery != Critical
```

Fig. 23 The drone must have the authority to start the `goto` skill.

```
start {
    motion    -> Used
    authority -> Software
}
```

Fig. 24 The drone takes the authority when starting the `goto` skill.

8.5 FTA of skill `takeoff`

In this subsection, we illustrate how the fault management analysis (introduced in Section 7) has been applied to the `takeoff` skill. We obtained the **takeoff fault tree** given by Fig. 25 by adding all the considered hazardous event to the generic **skill fault model** (Fig. 17).

To build this tree, we first applied Step (2a) of the analysis process: for each node of the **skill fault model**, we analyzed 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 (E010), while during the flight, a battery failure yields to an alarm (on telepilot's radio-command), and the telepilot should take control back (E087).

Once all errors were 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 FS_i on the fault tree (Step (2b) of the analysis process). During this analysis, we identified some inconsistencies of the skill or unmanaged events in the fault tree of Fig. 25 (Step (3a)) and then we modified the skill model and/or skill implementation to fix these inconsistencies (Step (3b)).

A first observation was that some considered errors can propagate up to the top of the fault tree, i.e. lead to a skill failure, without being handled by a DM/FS mechanism. From this observation, we identified missing detection mechanisms and failure states, which are represented in yellow in Fig. 25. For most of them (the ones attached to E088), we simply created an additional `emergency` failure state to give back authority to telepilot as UAV's dynamics are abnormal. But this also implied to implement the corresponding DM. For example, drifting conditions (E089) are detected by tracking the performance of the control law (in the functional layer), by a function of the skill implementation (`drifted`) which checks that UAV's position rests in a 2-meter radius of the takeoff point and triggers FS7 (terminal state *emergency*) if needed.

Another observation was that some errors of different nature could lead to the same failure state of the skill, but with different detection mechanisms. As an example, DM6 and DM7 initially yielded both to a same failure state: `blocked`. This situation may make the failure mode ambiguous, and therefore make unclear the kind of reactions the decisional layer should implement. Indeed, the behaviour to adopt should not be the same if we are `OnGround` or not. These situations have been tackled by separating these situations into two failures modes: `grounded` and `emergency`. The drone blocked in the air during takeoff is considered an emergency situation that probably requires the safety pilot to regain control. On the opposite, the `grounded` FS presents no risks for the system, and the decision layer can decide how to react to this situation. These modifications have resulted in the model presented in Listing 12.

Based on these modifications, the Takeoff fault tree has been updated iteratively, and the same analysis (identify inconsistencies, making recommendations, modifying the skill) has been performed again until we have a satisfying fault tree model.

9 Conclusion and Future Works

In this paper, we have presented a development process that guides the developer in the design of an executive layer based on skill models, and in the assessment of the dependability of these skills in relation with the functional layer of the robot.

The proposed development process is based on the following steps and tools:

1. we have proposed a DSL to specify skillsets, i.e. abstractions of the capabilities achieved by the functional layer of the system; this DSL comes with a formal execution semantics;
2. the skillset model can be verified by analysing its correctness using a model-checking engine; this step ensures that the model is valid;
3. a generation toolchain allows generating ROS2 packages in which the developer can implement specific *hooks* to link the skillset manager with the robot functional layer;
4. finally, we have proposed a process based on fault trees, to assess whether fault tolerance at the skills level is correctly implemented in relation with the functional layer.

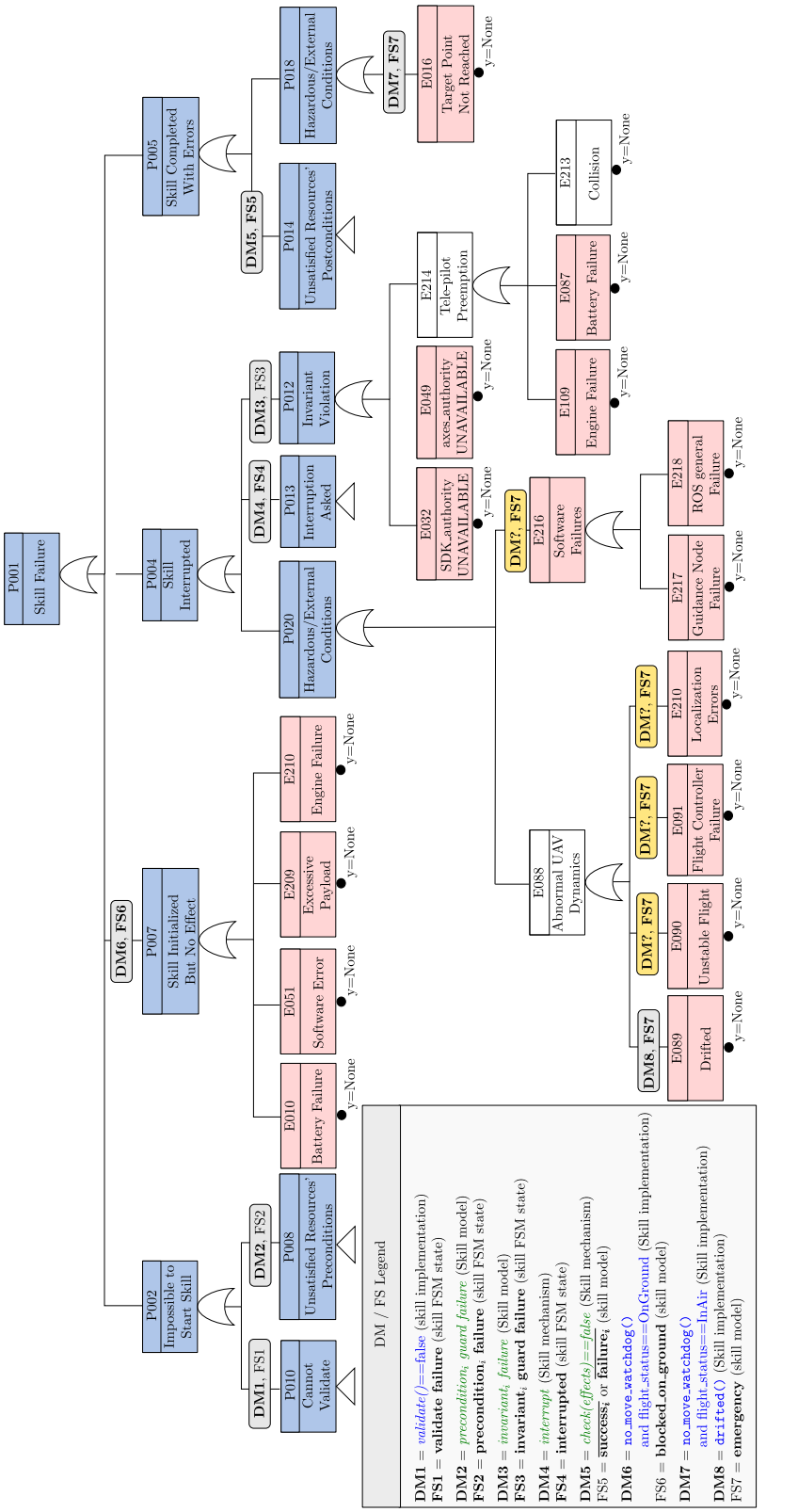

Fig. 25 Takeoff fault tree

We have illustrated this process on an application involving an autonomous UAV performing an inspection mission in BVLOS conditions. The language and the associated tools for model-checking and code generation are available at:

`https://onera-robot-skills.gitlab.io`

Future works are two-fold. First, in order to improve the proposed process and the assessment of the dependability of the architecture, we plan to integrate a testing step in the process. To do so, we are currently investigating how we can take benefit of the skillset model and the Fault-Tree Analysis of each skill to generate test cases that are relevant to cover the fault tree.

Secondly, we aim at extending the analysis towards the decisional layer of the architecture. In the present paper, we have shown a Behavior Tree implementation of the decisional layer of our skill-based architecture. But the development process illustrated in this paper is somehow agnostic of the underlying formalism used to describe the mission of the robots. Given that, the verification process of the skill-based architecture should be extended to the formal mission specification to guarantee its correct execution, given the chaining of the skills, the resources allocated, and the exogenous events.

References

1. Albore, A., Doose, D., Grand, C., Lesire, C., Manecy, A.: Skill-based architecture development for online mission reconfiguration and failure management. In: 2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE), pp. 47–54. IEEE (2021)
2. Alcácer, V., Cruz-Machado, V.: Scanning the industry 4.0: A literature review on technologies for manufacturing systems. Engineering science and technology, an international journal **22**(3), 899–919 (2019)
3. Archibald, C., Petriu, E.: Skills-oriented robot programming. In: Proceedings of the International Conference on Intelligent Autonomous Systems IAS-3, pp. 104–15 (1993)
4. Avizienis, A., Laprie, J., Randell, B., Landwehr, C.E.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing **1**(1), 11–33 (2004). DOI 10.1109/TDSC.2004.2
5. Banerjee, B.: Autonomous acquisition of behavior trees for robot control. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 3460–3467. IEEE (2018)
6. Barbosa, A.S., Plentz, P.D., De Pieri, E.R.: A behavior tree designing tool for online evaluation. In: IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society, pp. 537–542. IEEE (2020)
7. Ben-Ari, M., Mondada, F.: Robots and their applications. In: Elements of robotics, pp. 1–20. Springer (2018)
8. Björkelund, A., Edström, L., Haage, M., Malec, J., Nilsson, K., Nugues, P., Robertz, S.G., Störkle, D., Blomdell, A., Johansson, R., et al.: On the integration of skilled robot motions for productivity in manufacturing. In: 2011 IEEE International Symposium on Assembly and Manufacturing (ISAM), pp. 1–9. IEEE (2011)
9. Bøgh, S., Nielsen, O.S., Pedersen, M.R., Krüger, V., Madsen, O.: Does your robot have skills? In: Proceedings of the 43rd international symposium on robotics. VDE Verlag GMBH (2012)
10. Bohren, J., Cousins, S.: The SMACH high-level executive [ROS news]. IEEE Robotics & Automation Magazine **17**(4), 18–20 (2010)
11. Brooks, R.: A robust layered control system for a mobile robot. IEEE journal on robotics and automation **2**(1), 14–23 (1986)
12. Brooks, R.A.: Intelligence without representation. Artificial intelligence **47**(1-3), 139–159 (1991)

13. Colledanchise, M., Almeida, D., Ögren, P.: Towards blended reactive planning and acting using behavior trees. In: 2019 International Conference on Robotics and Automation (ICRA), pp. 8839–8845. IEEE (2019)
14. Colledanchise, M., Marzinotto, A., Dimarogonas, D.V., Oegren, P.: The advantages of using behavior trees in mult-robot systems. In: ISR. Munich, Germany (2016)
15. Colledanchise, M., Ögren, P.: How behavior trees modularize robustness and safety in hybrid systems. In: IROS. Chicago, USA (2014)
16. Colledanchise, M., Ögren, P.: How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. IEEE Transactions on Robotics **33**(2), 372–389 (2016)
17. Commission, I.E., et al.: Fault Tree Analysis (FTA). IEC 61025 (2006)
18. Crosby, M., Petrick, R.P., Rovida, F., Krueger, V.: Integrating mission and task planning in an industrial robotics framework. In: Twenty-Seventh International Conference on Automated Planning and Scheduling (2017)
19. Crosby, M., Rovida, F., Pedersen, M.R., Petrick, R.P., Krüger, V.: Planning for robots with skills. In: 4th ICAPS Workshop on Planning and Robotics 2016, pp. 49–57. ICAPS (2016)
20. Del Duchetto, F., Baxter, P., Hanheide, M.: Lindsey the tour guide robot-usage patterns in a museum long-term deployment. In: 2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN), pp. 1–8. IEEE (2019)
21. Deng, Z., Guan, H., Huang, R., Liang, H., Zhang, L., Zhang, J.: Combining model-based q-learning with structural knowledge transfer for robot skill learning. IEEE Transactions on Cognitive and Developmental Systems **11**(1), 26–35 (2017)
22. Doherty, P., Kvarnström, J., Heintz, F.: A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. Autonomous Agents and Multi-Agent Systems **19**(3), 332–377 (2009)
23. Fikes, R.E., Nilsson, N.J.: Strips: A new approach to the application of theorem proving to problem solving. Artificial intelligence **2**(3-4), 189–208 (1971)
24. Gao, Z., Wanyama, T., Singh, I., Gadhrri, A., Schmidt, R.: From Industry 4.0 to Robotics 4.0 - a conceptual framework for collaborative and intelligent robotic systems. Procedia manufacturing **46**, 591–599 (2020)
25. Ghzouli, R., Berger, T., Johnsen, E.B., Dragule, S., Wąsowski, A.: Behavior trees in action: a study of robotics applications. In: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, pp. 196–209 (2020)
26. Grudic, G.Z., Lawrence, P.D.: Human-to-robot skill transfer using the spore approximation. In: Proceedings of IEEE International Conference on Robotics and Automation, vol. 4, pp. 2962–2967. IEEE (1996)
27. Guerin, K.R., Lea, C., Paxton, C., Hager, G.D.: A framework for end-user instruction of a robot assistant for manufacturing. In: 2015 IEEE international conference on robotics and automation (ICRA), pp. 6167–6174. IEEE (2015)
28. Guiochet, J., Machin, M., Waeselynck, H.: Safety-critical advanced robots: A survey. Robotics and Autonomous Systems **94**, 43–52 (2017)
29. Guo, M., Johansson, K.H., Dimarogonas, D.V.: Revising motion planning under linear temporal logic specifications in partially known workspaces. In: 2013 IEEE International Conference on Robotics and Automation, pp. 5025–5032. IEEE (2013)
30. Heinze, F., Klöckner, M., Wantia, N., Rossmann, J., Kuhlenkötter, B., Deuse, J.: Combining planning and simulation to create human robot cooperative processes with industrial service robots. In: Applied Mechanics and Materials, vol. 840, pp. 91–98. Trans Tech Publ (2016)
31. Ingrand, F., Ghallab, M.: Deliberation for autonomous robots: A survey. Artificial Intelligence **247**, 10–44 (2017). DOI https://doi.org/10.1016/j.artint.2014.11.003
32. Iovino, M., Scukins, E., Styrud, J., Ögren, P., Smith, C.: A survey of Behavior Trees in Robotics and AI. preprint arXiv:2005.05842 (2020)
33. Jennings, N.R., Sycara, K., Wooldridge, M.: A roadmap of agent research and development. Autonomous agents and multi-agent systems **1**(1), 7–38 (1998)
34. Johannsmeier, L., Gerchow, M., Haddadin, S.: A framework for robot manipulation: Skill formalism, meta learning and adaptive control. In: International Conference on Robotics and Automation (ICRA), pp. 5844–5850. IEEE (2019)
35. Jones, S., Studley, M., Hauert, S., Winfield, A.: Evolving behaviour trees for swarm robotics. In: Distributed Autonomous Robotic Systems, pp. 487–501. Springer (2018)

36. Jones, S., Studley, M., Hauert, S., Winfield, A.F.T.: A two teraflop swarm. Frontiers in Robotics and AI **5**, 11 (2018)
37. Klöckner, A.: Interfacing behavior trees with the world using description logic. In: AIAA Guidance, Navigation, and Control (GNC) Conference, p. 4636 (2013)
38. Konidaris, G.D.: Autonomous robot skill acquisition. University of Massachusetts Amherst (2011)
39. Kruger, N., Piater, J., Worgotter, F., Geib, C., Petrick, R., Steedman, M., Asfour, T., Kraft, D., Hommel, B., Agostini, A., et al.: A formal definition of object-action complexes and examples at different levels of the processing hierarchy. Computer and Information Science pp. 1–39 (2009)
40. Leite, A., Pinto, A., Matos, A.: A safety monitoring model for a faulty mobile robot. Robotics **7**(3), 32 (2018)
41. Lesire, C., Doose, D., Grand, C.: Formalization of robot skills with descriptive and operational models. In: IROS. Las Vegas, NV, USA (virtual) (2020)
42. Li, J., Wang, J., Wang, S., Yang, C.: Human–robot skill transmission for mobile robot via learning by demonstration. Neural Computing and Applications pp. 1–11 (2021)
43. Lopes, M., Santos-Victor, J.: A developmental roadmap for learning by imitation in robots. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) **37**(2), 308–321 (2007)
44. Mayr, M., Chatzilygeroudis, K., Ahmad, F., Nardi, L., Krueger, V.: Learning of parameters in behavior trees for movement skills. In: 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 7572–7579. IEEE (2021)
45. Medina, G.C., Guiochet, J., Lesire, C., Manecy, A.: A skill fault model for autonomous systems. In: IEEE/ACM International Workshop on Robotics Software Engineering (RoSE). Pittsburgh, PA, USA (2022)
46. Menghi, C., Garcia, S., Pelliccione, P., Tumova, J.: Multi-robot LTL planning under uncertainty. In: International Symposium on Formal Methods, pp. 399–417. Springer (2018)
47. Moura, L.d., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer (2008)
48. Nematollahi, I., Rosete-Beas, E., Röfer, A., Welschehold, T., Valada, A., Burgard, W.: Robot skill adaptation via soft actor-critic gaussian mixture models. ICRA (2022)
49. Nikolakis, N., Maratos, V., Makris, S.: A cyber physical system (cps) approach for safe human-robot collaboration in a shared workplace. Robotics and Computer-Integrated Manufacturing **56**, 233–243 (2019)
50. Ögren, P.: Increasing modularity of UAV control systems using computer game behavior trees. In: AIAA GNC Conference. Minneapolis, MN, USA (2012)
51. Pane, Y., Mokhtari, V., Aertbeliën, E., De Schutter, J., Decré, W.: Autonomous runtime composition of sensor-based skills using concurrent task planning. IEEE Robotics and Automation Letters **6**(4), 6481–6488 (2021)
52. Pedersen, M.R., Krüger, V.: Automated Planning of Industrial Logistics on a Skill-equipped Robot. In: IROS Workshop on Task Planning for Intelligent Robots in Service and Manufacturing. Hamburg, Germany (2015)
53. Pedersen, M.R., Nalpantidis, L., Andersen, R.S., Schou, C., Bogh, S., Krüger, V., Madsen, O.: Robot skills for manufacturing: From concept to industrial deployment. Robotics and Computer-Integrated Manufacturing **37**, 282 – 291 (2016). DOI 10.1016/j.rcim.2015.04.002
54. Pitonakova, L., Crowder, R., Bullock, S.: Behaviour-data relations modelling language for multi-robot control algorithms. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 727–732. IEEE (2017)
55. Py, F., Ingrand, F.: Dependable execution control for autonomous robots. In: International Conference on Intelligent Robots and Systems (IROS). Sendai, Japan (2004). DOI 10.1109/IROS.2004.1389549
56. Rovida, F., Crosby, M., Holz, D., Polydoros, A.S., Großmann, B., Petrick, R., Krüger, V.: Skiros—a skill-based robot control platform on top of ros. In: Robot operating system (ROS), pp. 121–160. Springer (2017)
57. Rovida, F., Krüger, V.: Design and development of a software architecture for autonomous mobile manipulators in industrial environments. In: IEEE International Conference on Industrial Technology (ICIT), pp. 3288–3295. IEEE (2015)
58. Safronov, E., Colledanchise, M., Natale, L.: Task planning with belief behavior trees. In: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 6870–6877. IEEE (2020)

59. Scheper, K.Y., Tijmons, S., de Visser, C.C., de Croon, G.C.: Behavior trees for evolutionary robotics. Artificial life **22**(1), 23–48 (2016)
60. Schou, C., Andersen, R.S., Chrysostomou, D., Bøgh, S., Madsen, O.: Skill-based instruction of collaborative robots in industrial settings. Robotics and Computer-Integrated Manufacturing **53**, 72–80 (2018). DOI 10.1016/j.rcim.2018.03.008
61. Schou, C., Damgaard, J.S., Bøgh, S., Madsen, O.: Human-robot interface for instructing industrial tasks using kinesthetic teaching. In: IEEE ISR 2013, pp. 1–6. IEEE (2013)
62. Segura-Muros, J.Á., Fernández-Olivares, J.: Integration of an automated hierarchical task planner in ros using behaviour trees. In: 2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT), pp. 20–25. IEEE (2017)
63. Steinmetz, F., Weitschat, R.: Skill parametrization approaches and skill architecture for human-robot interaction. IEEE International Conference on Automation Science and Engineering (CASE) (2016). DOI 10.1109/COASE.2016.7743419
64. Stenmark, M., Malec, J.: Knowledge-based industrial robotics. In: Twelfth Scandinavian Conference on Artificial Intelligence, pp. 265–274. IOS Press (2013)
65. Stenmark, M., Malec, J.: Knowledge-based instruction of manipulation tasks for industrial robotics. Robotics and Computer-Integrated Manufacturing **33**, 56–67 (2015)
66. Thomas, U., Hirzinger, G., Rumpe, B., Schulze, C., Wortmann, A.: A new skill based robot programming language using UML/P statecharts. In: IEEE International Conference on Robotics and Automation, pp. 461–466. IEEE (2013)
67. Topp, E.A., Stenmark, M., Ganslandt, A., Svensson, A., Haage, M., Malec, J.: Ontology-based knowledge representation for increased skill reusability in industrial robots. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 5672–5678. IEEE (2018)
68. Tuci, E., Alkilabi, M.H., Akanyeti, O.: Cooperative object transport in multi-robot systems: A review of the state-of-the-art. Frontiers in Robotics and AI **5**, 59 (2018)
69. Tzafestas, S.G.: Mobile robot control and navigation: A global overview. Journal of Intelligent & Robotic Systems **91**(1), 35–58 (2018)
70. Villani, V., Pini, F., Leali, F., Secchi, C.: Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces and applications. Mechatronics **55**, 248–266 (2018)
71. Wooldridge, M., Jennings, N.R.: Intelligent agents: Theory and practice. The knowledge engineering review **10**(2), 115–152 (1995)
72. Zhou, H., Min, H., Lin, Y.: An autonomous task algorithm based on behavior trees for robot. In: 2019 2nd China Symposium on Cognitive Computing and Hybrid Intelligence (CCHI), pp. 64–70. IEEE (2019)
73. Zhu, X.: Behavior tree design of intelligent behavior of Non-Player Character (NPC) based on UNITY3D. Journal of Intelligent & Fuzzy Systems **37**(5), 6071–6079 (2019)
74. Ziparo, V.A., Iocchi, L., Nardi, D., Palamara, P.F., Costelha, H.: Petri net plans: a formal model for representation and execution of multi-robot plans. In: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1, pp. 79–86 (2008)