Skill-Based Architecture Development for Online Mission Reconfiguration and Failure Management

Alexandre Albore*, David Doose*, Christophe Grand*, Charles Lesire* and Augustin Manecy*
*ONERA/DTIS, University of Toulouse
2 av. Edouard Belin,31400, Toulouse, France
Email: {firstname.lastname}@onera.fr

***Abstract*—The development of software architectures that ensure both a high-level of autonomy in the mission, and the robustness to possible failures, is a challenging task. In this paper, we propose to structure the software architecture around a *skill management layer*, based on formal skill models. This skill management layer helps to structure and test the underlying functional layer, while it provides a simple abstraction to the decision layer. This architecture has been used to support the implementation of resilient behaviours, using Behaviour Trees, in autonomous UAV missions, when facing sensor failures or communication losses.**

***Index Terms*—Software Engineering, Error handling and recovery, Robotics**

1. Introduction

Autonomous robotic systems are going to be a common tool to perform observation missions, such as inspection of infrastructures, monitoring of the environment, or post-disaster situation assessment. For the operator to be confident and interested in the use of autonomous systems, these must guarantee a robust behaviour when confronted with hazardous situations, such as failures of sensors or processing. Developing a complete functional architecture that implements intelligent behaviours resilient to failures is heavy work. These behaviours can be provided by the robot manufacturer, however they often require a costly development process, which is most of the time not affordable by small enterprises. The manufacturer itself is generally unaware of the final user requirements, being not adapted to the actual situation in which the robot is deployed. If the robust behaviour is brought to the system by the robot users, then they are required to have a precise understanding of the robot's internal functions, which is not always the case.

In this paper, we are interested in providing models and tools to implement adaptive recovery strategies in case of robot failures. To do so, we propose a layered approach. Functional software architectures are indeed complex software systems, made by several processes or nodes, often specifically designed to fit the robot platform's peculiar configuration. The robotic community has taken a large benefit in using ROS [1], as it helps to structure the architecture in nodes, and gives modularity and flexibility in the design of functional layers. We argue that implementing resilient behaviours for autonomous systems requires to reason on a more abstract level, by manipulating resources or skills [2], [3] instead of nodes and topics needed for internal communication. The robot user would then not need to know how the functions are internally implemented but only what are the capabilities (skills) and resources provided by the robot.

In the following, we present the development process of a robotic software architecture based on the skill formal model from [3]. Skills are basic functionalities that participate, in a modular manner, to design the complete task or mission of a robotic system. We have specifically used resource models and how these resources are used in the skill models to implement failure detection, redundant sensors or processing, and alternatives skills to perform the mission. The development process then uses on one side code generation to organize the functional architecture around *manager* nodes, which will interact with the nodes of the functional architecture, and answer to skill execution requests. On the other side, a standardized *skill interface* is used to implement the autonomous robot behaviour to perform the mission, and the existence of a formal skill model is exploited to identify faulty situations and to implement recovery behaviours.

We have used this process to develop the skill-based architecture of an autonomous UAV, and implemented the behaviours to perform observation missions. Task level programming, which encompasses the mission outline and the system recovery strategies, is then implemented using Behaviour Trees (BTs) – a formalism used to encode the control architecture of a Hybrid Dynamical System [4], [5]. The structure of a BT is close to function calls in programming, with subtrees accomplishing specific behaviours of more complex programming, combining the different skill executions.

In Section 2, we present two scenarios in which failures occur, and the layered skill-based architecture we eventually deployed for one of them. Some works related to skill model and skill management are presented in Section 3. In Section 4 we present the skills modeling language, and how we used the associated tools in the development process. Then we present the BT models in Section 5, and some experimental results on the presented scenarios in Section 6.

2. Use Cases / Scenarios

Here we describe two different scenarios to illustrate how a skill-based architecture can be used to deal with hazardous events during robotic missions.

2.1. Communication Loss during BVLOS Flight

This use case aims at inspecting a building by turning all around it 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 FPS and a minimum image quality criteria. During the inspection, we assume the UAV travels through different communication zones featuring more or less degraded bandwidth until entering a critical bandwidth zone (i.e., for which FPS criteria cannot be respected anymore). When this occurs, the `stream` skill terminates in failure and the mission BT switches to another branch, that stops the inspection, makes the UAV hover a few seconds, then land at its current location.

2.2. Obstacle Avoidance during 3D Mapping

This use case aims at building a map of an unknown environment using an hexarotor UAV. The cartography is done by a SLAM algorithm relaying on IMU and LiDAR data. To maximize LiDAR scan area, the LiDAR was strapped to the drone in order to perform scans orthogonal to the displacement direction. A frontal ZED camera produces a depth-map used to detect obstacles. In case of a failure of the frontal depth-camera, the displacement strategy is reconfigured by performing "crab" displacement in order to use the LiDAR to detect obstacles, as described in Figure 1. As a result, the cartography performances are degraded, but the obstacle avoidance function can be maintained.

The implementation of this resilient behaviour has been achieved by the development of the 3-layer architecture presented in Figure 2, whose central element is the skill management layer. The *functional layer* implements the "low-level" functions of the robotic system; This set of functions is generally implemented by several components with a set of parameters to manage configurations and the components usually provide an API to tune parameters, receive setpoints/references and deliver running status. In our example, the functional layer consists in several ROS nodes, communicating through topics, providing dynamic parameters for configuration and implementing guidance control loops, obstacle detection and sensor readings and fusion. The *skill-based layer* is composed of one or several skillsets which provide a formal access to robot's capabilities (see Sec. 4.1). A skillset is composed of skill managers and resources managers which are components automatically generated from the skills' model. The skill managers offer an abstraction layer to easily start/configure a subset

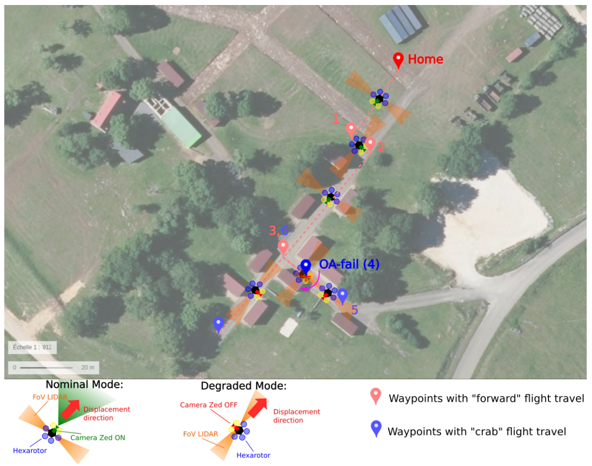

Figure 1. Recovery behaviour after failure of the depth-camera: the initial trajectory (red waypoints) is performed in nominal mode; after the failure of the camera, the LiDAR is used to detect obstacles, and the following waypoints (in blue) are performed in degraded mode.

of functional layer components and automatically change resource states accordingly to the skill model (via internal transitions). Interactions with the functional layer can be added by the developer via customizable callbacks of skill managers (see Sec. 4.2) and external transitions of resources can also be triggered via a *SkillSet Interface API* to reflect functional layer states. The *decision layer*, implemented here as a Behaviour Tree (BT), calls sequentially (or in parallel) several skills through a standardized skillset interface, also automatically generated, in order to perform the mission. It monitors skill progress and also checks their termination results to detect failures and then switch to alternative branches which activate other subsets of skills to continue the mission –if possible–, or to adopt a suitable behaviour. A mission operator can start the mission by triggering the BT execution and can follow the execution of the mission thanks to a BT-viewer, and a skill timeline to visualize skill execution states.

3. Related Works

Skill models have been used in several works in the field of robotic manufacturing. Descriptive models of skills have been proposed in [6]. These models include input parameters, preconditions, and predictive effects, which are used to plan skill sequences. The authors have proposed a methodology to translate skill models into PDDL planning models. They do not consider possible failures of skills. In [7], they also define skill monitors that check online the preconditions and effects of skills to detect failures. These monitors are external processes that observe some elements of the architecture, and are not a structuring feature in the architecture development process: the interaction with the robot platform has still to be programmed manually, and failure management based on models is not possible.

Other works have proposed a formal model of skills. A relational model representing the relations between skills

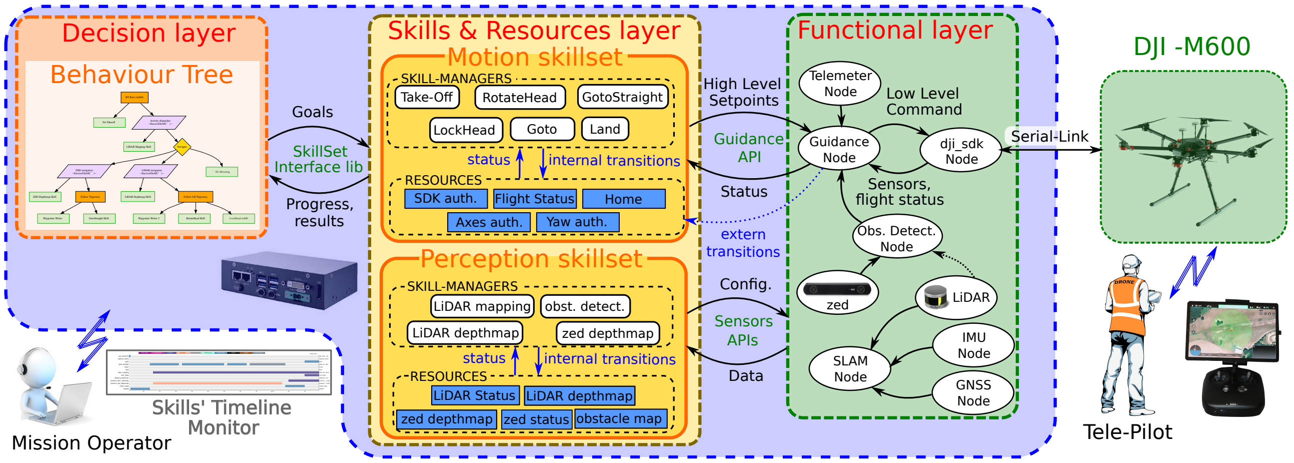

Figure 2. The 3 layer based architecture: decision (Behaviour Tree), skills-resources and functional layers

and resources and data has been proposed in [8]. This model however neither represents the behaviour of resources, nor possible terminal states of the skill executions. In [9], skills are represented with preconditions and device resources requirements in an OWL ontology, including synchronization of skill execution. Again, the skill models do not integrate information about possible failures. Performance Level Profiles (PLP) [10] use as a semi-formal language (based on XML Schema) to represent modules available on a robot platform. The module's description is rich, including preconditions/effects, resources, modes, and rates. Moreover, some tools are provided to generate PDDL models or runtime monitors. The behaviour of the modules is however not defined, preventing the use of verification methods or the definition of sound protocols.

In a previous work [3], we have proposed a specification language for skills, that has deep links with the previously mentioned models. Its specificity is that, first, it includes a rich model of resources through state-machines, and links the skill execution with constraints on these resources; second, the execution of these skills is formally defined, easing code template generation. In the next section, we remind the features of the skill language, and present the use of the associated models and tools to develop a skill-management architecture that emphasizes the failure management process. In Section 5, we present how we controlled the mission, including recovery strategies, using BTs.

4. Skill Management Architecture

4.1. Skill and Resource Models

The skills specification language proposed in [3] groups elements into *skillsets*, containing skills related to one subsystem: navigation functions, some driver and associated processing, etc. A skillset is made of three kinds of elements:

- *data*, that specify the data that are made available to the decision-making layer (internal data of the functional architecture need not to be described);
- *resources*, that specify elements of the architecture that will enable and constrain the execution of skills;
- *skills*, that represent available robot capabilities.

We detail and illustrate resource and skill models.

4.1.1. Resources. A resource is modeled as a state-machine. A resource can represent either the status of a device of the platform (typically, a sensor), or some logical condition for skill execution. For instance, the resource modeled in Listing 1 (with corresponding state-machine depicted in Figure 3) represents the fact that the control system of the robotic platform gave control authority to the autonomous architecture. Transitions of this resource are defined as *extern*, meaning that the skills will not be able to trigger these transitions, only the functional layer can.

```
resource SDK_authority {
  initial UNAVAILABLE
  extern  AVAILABLE -> UNAVAILABLE
  extern  UNAVAILABLE -> AVAILABLE
}
```

Listing 1. Authority resource model.

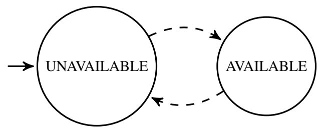

Figure 3. Authority resource state-machine (dashed transitions are *extern*).

Listing 2 and Figure 4 represent a resource modelling the availability of the axes control. This resource is purely *internal*: it is used to represent mutual exclusion between skills, but has no link with the functional layer.

4.1.2. Skills. Skills represent the capabilities exposed by the functional layer. Skills are defined by several elements:

- *inputs*, that define the parameters of the skill;

```
resource axes_authority {
  initial AVAILABLE
  AVAILABLE -> USED
  USED -> AVAILABLE
}
```

Listing 2. Axes authority resource model.

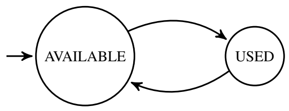

Figure 4. Axes authority resource state-machine.

- *preconditions*, over resource states or data predicates;
- *invariants*, that must hold during skill execution;
- *effects*, applied when the execution ends;
- *results*, corresponding to the possible terminal states of the skill;
- a *progress* field, giving the period at which the skill will give some feedback on skill execution progress.

Listing 3 defines the *takeoff* skill of our drone. Its inputs are the height to reach and the ascending speed. Skill preconditions are that the authority must be available (checking states of resources `SDK_authority` – Listing 1 – and `axes_authority` – Listing 2), and that the drone is on ground (modeled in a `flight_status` resource) with its home-point valid (`homepoint_status` resource). If these preconditions do hold, then the `take_control` effect is applied, taking control over the axes resource. During

```
skill takeoff {
  progress=0.5
  input {
    height: float64 // < h_geo_fence
    speed:  float64
  }
  effect {
    take_control: axes_authority -> USED
    release_control: axes_authority -> AVAILABLE
    reset {}
  }
  precondition {
    sdk_authority:
     resource=(SDK_authority==AVAILABLE)
    not_moving: resource=(axes_authority==AVAILABLE)
    on_ground: resource=(flight_status==ON_GROUND)
    home_valid: resource=(homepoint_status==VALID)
    success take_control
  }
  invariant {
    keep_sdk_authority:
     resource=(SDK_authority==AVAILABLE)
     violation=reset
    in_control: resource=(axes_authority==USED)
     violation=reset
  }
  result {
   AT_ALTITUDE: apply=release_control
   BLOCKED: apply=release_control
   ABORTED: apply=release_control // too big drift
  }
}
```

Listing 3. Takeoff skill model.

this skill execution, the several resource states must not change (see the `invariant` definition block). In case one invariant is violated, the `reset` effect is applied. This effect is empty, meaning that it will not change the state of any resource. Finally, the *takeoff* skill execution can `result` in several states: `AT_ALTITUDE`, meaning that the takeoff succeeded, `BLOCKED`, if the drone could not take off, and `ABORTED`, in case of an error of the localisation or the control processing. In all cases, the axes resource is released.

Every skill has an execution semantics defined by the state-machine depicted in Fig. 5. When the skill execution is started, a first validation step is processed. This validation must be implemented by the developer, that can accept the skill execution based on some checking of the skill input. Then if the skill request is accepted, the resource preconditions are tested. If some resource preconditions are not met, the execution fails (*NR* state). Otherwise, the skill is actually started: the *dispatch* transition is triggered. The skill is then running (*Rg* state), periodically reporting some progress, and checking the resource invariant. If an invariant is violated, the skill execution ends in the *RI* state, and the corresponding effect is applied. Otherwise, the skill continues its execution until either an interruption request is made, or a termination function is called. In that case, post-conditions or effects of the result are applied (in states T_i, with $0 < i \leq k$), resulting in ending the skill execution in the corresponding result state, successfully (for states S_i), or not (states N_i).

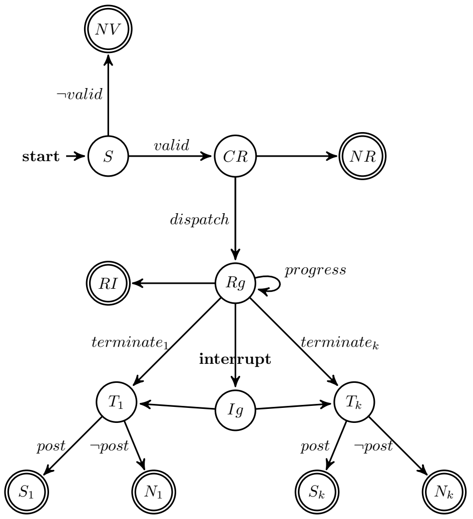

Figure 5. Skill execution state-machine. Double-circled states are terminal execution states.

4.2. Skill Management Architecture

In order to practically manage the execution of skills, we have proposed a skill management architecture, of which an instantiation is shown in Fig. 2. The toolchain associated to the skillset definition language includes a manager generator which produces, based on a model of the skillset, a ROS architecture (available in both ROS and ROS2) that includes the following nodes:

- a *Data Manager*, that subscribes to topics where the functional layer data are published, and provides an interface to access these data from the decision layer;
- a *Resource Manager*, that manages the current state of each resource, and ensures that the triggered transition are feasible, both through services available either to the functional layer (for *extern* transitions), or to skill managers (for internal transitions);
- for each skill, a *Skill Manager*, that implements the state-machine of Figure 5, interacts with the resource manager to manage preconditions, invariants and effects, and in which the skill developer can implement some specific methods: inputs *validation*, skill *dispatching*, and the call to *terminate* transitions.

The skillset toolchain also provides:

- a *Skillset Interface* library (in Python), that allows to interact with the skillset (get data or resource states, start skills, interrupt them or get their result);
- ROS/ROS2 bridges, in case the functional layer is implemented in ROS and the decision layer in ROS2.

Listing 4 shows a very partial extract from the specialization of the `TakeoffSkillManager`, where the developer makes the actual link with the functional layer (done here thanks to the *dji_guidance* object of our guidance API). We can notice that, in the `validate` method, the user checks the input values, and can also access some functions in the functional layer to monitor the possible execution of the skill. The `on_dispatch` method actually starts the skill execution. The developer has also registered a callback to some updates in the functional layer, and depending on the received information, can terminate the execution in one of the possible results.

4.3. Skill-based Development Process

We have used the skill models and the generation of the manager architecture to implement an incremental development and test methodology. First, the definition of resource and skill models has led to better structure the functional layer, emphasizing which data and control flows are internal to the functional architecture, and which are meant to be exposed to a decision layer. We have then used the skills manager and interface generator to set up a symmetrical testing process:

- unit-testing of skill managers is implemented using the skillset interface library, allowing to first setup some context (interacting with resources or the functional layer), then to start skills, and check their result;
- testing of the decision-layer without deploying the actual functional layer; the manager generator also produces *mock* managers, that instead of being linked with the functional layer, respond to skill execution request according to their configuration, which consists of adding delays to the responses, and defining the result.

```
class TakeoffSkillManager(AbstractTakeoffSkillManager):

  def __init__(self):
    AbstractTakeoffSkillManager.__init__(self)
    self.max_speed = rospy.get_param('~max_speed', 3.0)
    self.dtg_h = 0.0
    self.dtg_v = 0.0

  def validate(self, goal_id, goal):
    # check take-off parameters
    if goal.speed > self.max_speed:
      return False

    elif not dji_guidance.check_home_fence(goal.target):
      rospy.logwarn('Point out of geo-fence!')
      return False
    else:
      return True

  def on_dispatch(self, goal_id, goal):
    self.goal_id = goal_id
    # build setpoint and publish it to m600_guidance
    dji_guidance.send_position_ref(goal.target)

  def progress(self, goal_id, goal):
    return 1.0 - (self.dtg_v / goal.height)

  def _CB_new_position(self):
    if self.dtg_v < self.vert_validation_dist:
      self.terminated_AT_ALTITUDE(self.goal_id)
    elif self.dtg_h > 2.0:
      rospy.logwarn('Too much horizontal drift!')
      self.terminated_ABORTED(self.goal_id)
    elif (tvel - self._last_t_moving > 5.0):
      rospy.logwarn('BLOCKED for more than 5.0 sec!')
      self.terminated_BLOCKED(self.goal_id)
```

Listing 4. Extract of the *takeoff* manager. A skeleton of this code is provided to the developer, who only modifies this part of the generated managers.

These two testing steps, based on the tools provided by automatic generation from the formal skill models, allow having a first validation of both the skill manager implementation and the behaviour implemented in the decision-layer. To go further in the testing process of the decision-layer, we set up two complementary testing suites:

- skill managers connected to a simulator, instead of the actual robot functional layer;
- skill managers connected to the functional layer running on the robot platform, performing Hardware-In-the-Loop simulations (using vendor simulation[1]).

These last validation steps allow not only to validate the behaviour of the overall architecture, but also the integration of the decision-making and its impact on the on-board processing. Note that these several testing and validation steps are supported by the fact that the skillset interface is strictly the same, the skill managers being mocks, simulation managers, or functional managers; and that a great part of this architecture and interface is automatically generated from models.

1. The HIL simulation tool is provided by the *DJI Assistant 2* configuration tool (https://www.dji.com/fr/downloads/softwares/assistant-dji-2-for-matrice). A computer, connected to the drone's flight controller, emulates the sensor outputs, but the flight simulattion is performed in the embedded flight controller (DJI-A3 Pro). During HIL simulations, all the flight functionalities are available, i.e., interactions with telepilot's remote controller, Onboard-SDK features (to send low-level orders to the flight controller from the functional layer), etc.

5. Implementing Recovery Strategies with BTs

We use BTs to develop a control architecture for high-level mission programming. 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 tree breaks down the complex task of coding a robotic mission into smaller, independent behaviours, from the root node. While a subtree implements and abstracts several actions and calculations, on a finer scale, the single behaviours 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 *ticking* of the BT), the tree structure is traversed from left to right, visiting in order each branch of the tree, launching behaviours when needed, or just processing the return status of every behaviour. The return status is then progressed back towards the root node of the tree, and modified according to the type of each node.

The skills, on which our architecture is based, are encoded as single behaviours. These action behaviours, always present as leaves in the tree, call the skills via the *Skillset Interface*. An action behaviour 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 behaviour depends on the result of the related skill. For instance, given the *takeoff* skill of our drone described in Listing 3, we can specify skill results corresponding to *Success*, *Failure* or *Running* statuses of the relative action behaviour in the BT: success when the skill result is $S_{AT\_ALTITUDE}$, failure for other terminal results, and running when the skill is in any non-terminal state (see the skill FSM in Figure 5). As the execution of the skills can be made synchronous or asynchronous, it is when the skill results and the relative behaviour 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, triggering a mission reconfiguration.

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 behaviours guaranteeing the same degree of robustness of the implemented task to the new one [11]. 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.

Behaviours that need to be performed in a predetermined order, are implemented as children of a sequence node (Figure 6). However, BTs allow us to implement more complex action control, and here we use their structure to easily implement both the nominal mission, and the degraded maneuvers. Figure 7 illustrates how a Selector node is used to manage priorities in the execution of subtrees. Selector's children are visited in order from the left, and the first one that returns *Success* or *Running* fixes the status of the parent behaviour for that tick. In Figure 7

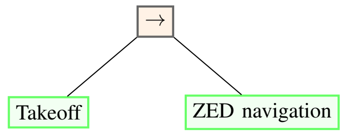

Figure 6. Behaviour tree calling skills sequentially. A Sequence is denoted by a node with an arrow. Action nodes are in green.

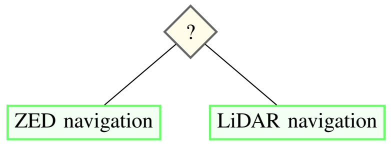

Figure 7. Failure Management with BTs. The Selector node denoted with a question mark returns the status of the leftmost child, but if this one fails, it is the status of the child at its right that is returned.

it is easy to see that if a malfunction occurs when the `ZED navigation` skill is running, then its corresponding behaviour will return *Failure*, and the parent node will run the `LiDAR navigation` behaviour instead. For Selector nodes, the rightmost subtrees *de facto* implement fallback behaviours. The BT of Figure 7 has been used in the second scenario: the subtree managing the skills related to navigating relying on the ZED depth-camera for obstacle avoidance has precedence over the rightmost branch. However, in case of a malfunction of the camera at running-time, the `ZED navigation` branch will return a *Failure* status, and the `LiDAR navigation` will take over using a LiDAR to generate a depth-map.

This transition model, implicitly encoded in the BT structure, ensure that BTs are seen as highly reactive, meaning that more important behaviours interrupt less important ones. This represents, together with their modularity, one of the main advantages of BT-based implementations [12]. In the former example, we have seen how this functioning can be turned to our advantage by programming an execution tree that implements the flexibility of switching between nominal, and failure branches in its own structure.

6. Experiments

6.1. UAV Platform

To perform the two scenarios described in Sec. 2, we have used a DJI-M600: a 14kg-hexacopter equipped with a DJI-A3 Pro flight controller as a low-level avionic and a custom payload, enhanced with perception sensors, and in charge of mission management thanks to the 3-layer architecture as the one of Fig. 2. The payload is composed of an embedded computer, a dedicated image processing unit, an IR telemeter to assist take-off and landing, a stereo-camera (ZED) for vision-based navigation, and a LiDAR and an IMU for SLAM.

6.2. Communication Loss during BVLOS Flight

For this scenario, we performed a set of experiments, in which we defined several inspection trajectories, and simulated the critical bandwidth zones. In this paper, we only describe in detail the 3D mapping scenario; an experiment on the inspection scenario is fully explained in the video available in the *robot skills* website:
http://oara-architecture.gitlab.io/robot-skills/.

6.3. Obstacle Avoidance during 3D Mapping

The BT encoding the resilient behaviour for the 3D mapping mission is depicted in Figure 6: in case of a failure of the ZED depth-camera, the fallback LIDAR-based navigation branch is executed. In each branch, Parallel nodes are used to run concurrently the skill producing the depth-map (ZED-based or LiDAR-based), and the motion trajectory. In the recovery strategy, this trajectory not only control the position of the drone, but also its heading using the *rotate_head* and *lock_head* skills. This behaviour has been tested in a 3D simulator, where skill managers interact with the simulated environment and sensors. Figure 11 shows the timeline of a nominal mission, in which no failure has been reported: the BT has activated a sequence of *goto_straight* skill execution to follow the mapping trajectory, while the LiDAR mapping was active (and finally interrupted by the BT at the end of the mission). Figure 8 shows the 3D map obtained in the nominal navigation mode.

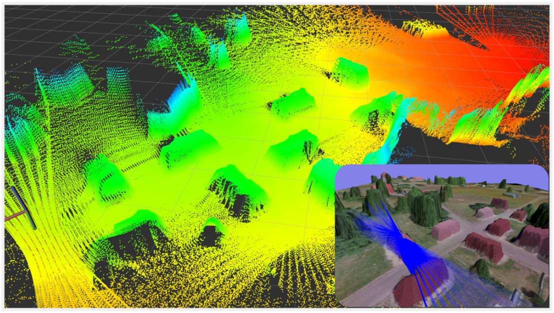
Figure 8. Map built during the nominal mission. Bottom-right frame shows a view of the simulator with the LiDAR scans.

Figure 12 shows the execution of skills in the scenario including the ZED camera failure. The ZED Depthmap skill reports a failure through a *Resource Invariant Violation*: the state of the resource representing the ZED sensor has been changed by the functional layer, invalidating the invariant of the corresponding skill model. The ZED-based navigation branch of the BT was then cancelled, interrupting the ongoing motion (gray box of the *goto_straight* skill), and then executing the fallback LiDAR-based navigation branch, that activates the LiDAR Depthmap skill, and now controls the drone using *rotate_head*, *lock_head* and *goto* skills. Figure 9 shows the map obtained at the end of this scenario. We clearly see the the left part of the area is not as well mapped as in Figure 8, but this behaviour ensured the safety of the drone, while providing a degraded information about the environment.

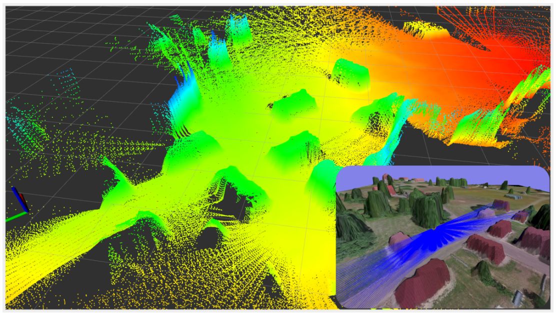
Figure 9. Map built after a ZED failure. Bottom-right frame shows a view of the simulator with the LiDAR scans oriented forward.

7. Conclusion

In this paper, we have presented a 3-layer architecture for autonomous and resilient robot behaviours, that settles on skill and resource models, and their associated execution managers. The skill management layer has a central role, both in the development and testing processes. We also presented how the skill interface can be used to both orchestrate skill executions and detect failures, and how the Behaviour Tree that performs the mission and the recovery strategies can take benefit in using them. We have implemented two specific scenarios with sensor or communication failures, and demonstrated the effectiveness of the approach on experiments and simulations.

We now investigate two further steps in the skill-based development process: (1) taking profit of both the skill formal models and the BT models to analyse the overall behaviour of the drone using model-checking, and (2) using automated planning techniques to help synthesizing automatically some branches of the BT.

References

[1] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[2] M. R. Pedersen, L. Nalpantidis, R. S. Andersen, C. Schou, S. Bogh, V. Krüger, and O. Madsen, "Robot skills for manufacturing: From concept to industrial deployment," *Robotics and Computer-Integrated Manufacturing*, vol. 37, pp. 282 – 291, 2016.

[3] C. Lesire, D. Doose, and C. Grand, "Formalization of robot skills with descriptive and operational models," in *IROS*, Las Vegas, NV, USA (virtual), 2020.

[4] A. Klöckner, " Interfacing Behavior Trees with the World Using Description Logic," in *AIAA GNC Conference*, Boston, MA, USA, 2013.

[5] M. Colledanchise, A. Marzinotto, D. V. Dimarogonas, and P. Oegren, "The advantages of using behavior trees in mult-robot systems," in *ISR*, Munich, Germany, 2016.

[6] F. Rovida and V. Krüger, "Design and development of a software architecture for autonomous mobile manipulators in industrial environments," in *ICIT*, Singapore, 2015.

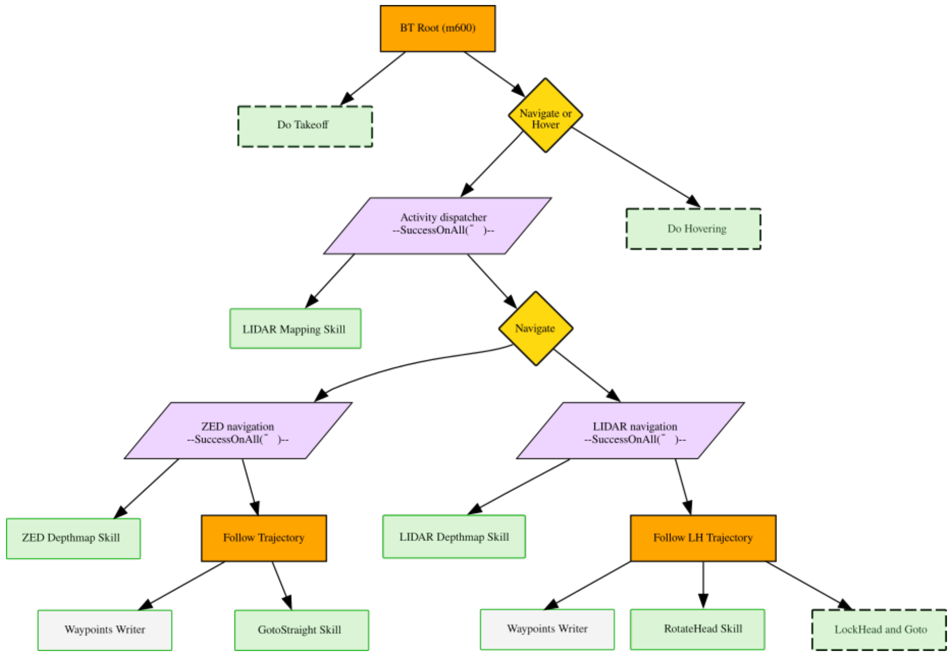

Figure 10. BT for the 3D Mapping scenario. In integrates constructs presented in previous figures. A purple trapezium denotes a Parallel behaviour, grey rectangles are actions internal to the BT (here the `Waypoint Writer` node calculates the next waypoint to reach), and dashed rectangles indicate subtrees that activate different skills (not fully reported here the sake of simplification).

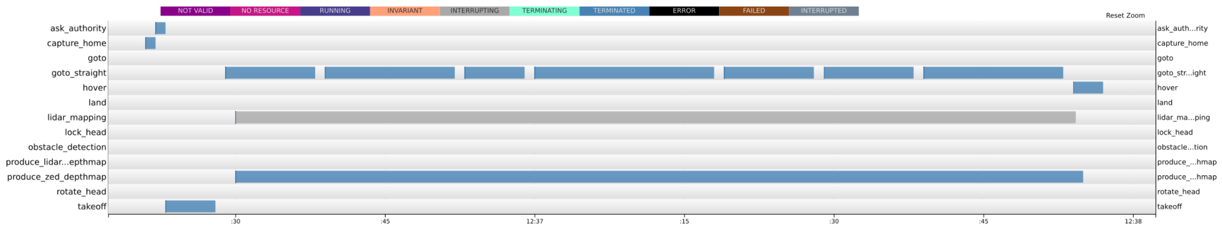

Figure 11. Skill execution timeline of a nominal scenario without failure.

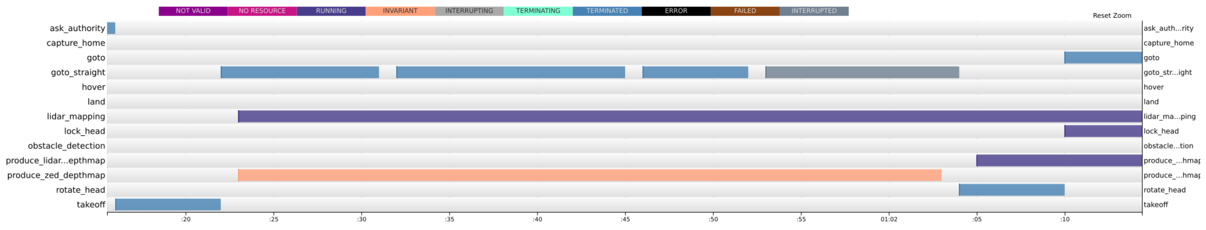

Figure 12. Skill execution timeline showing the reconfiguration behaviour.

[7] M. R. Pedersen and V. Krüger, "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.

[8] L. Pitonakova, R. Crowder, and S. Bullock, "Behaviour-data relations modelling language for multi-robot control algorithms," in *IROS*, Vancouver, BC, Canada, 2017.

[9] E. A. Topp, M. Stenmark, A. Ganslandt, A. Svensson, M. Haage, and J. Malec, "Ontology-based knowledge representation for increased skill reusability in industrial robots," in *IROS*, Madrid, Spain, 2018.

[10] R. I. Brafman, M. Bar-Sinai, and M. Ashkenazi, "Performance level profiles: A formal language for describing the expected performance of functional modules," in *IROS*, Daejeon, South Korea, 2016.

[11] M. Colledanchise and P. Ögren, "How behavior trees modularize robustness and safety in hybrid systems," in *IROS*, Chicago, USA, 2014.

[12] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, "A survey of Behavior Trees in Robotics and AI," *preprint arXiv:2005.05842*, 2020.