A formal implementation of Behavior Trees to act in robotics

Félix Ingrand

A formal implementation of Behavior Trees to act in robotics

Félix Ingrand
felix@laas.fr
LAAS-CNRS, Université de Toulouse
Toulouse, France

Abstract

Behavior Trees (BT) are becoming quite popular as an *Acting* component of autonomous robotic systems. We propose to define a formal semantics to BT by translating them to a formal language which enables us to perform verification of programs written with BT, as well as runtime verification while these BT execute. This allows us to formally verify BT correctness without requiring BT programmers to master formal languages and without compromising BT most valuable features: modularity, flexibility and reusability. We present the formal framework we use: FIACRE, its language and the produced TTS model; TINA, its model checking tools and HIPPO, its runtime verification engine. We then show how the translation from BT to FIACRE is automatically done, the type of formal LTL and CTL properties we can check offline and how to execute the formal model online in place of a regular BT engine. We illustrate our approach on two robotics applications, and show how BT can be extended with state variables, *eval* nodes, node evaluation results and benefit of other features available in the FIACRE formal framework (e.g., time).

1 Introduction and motivation

Behavior Trees (BT) were initially developed and deployed to program Non-Player Characters (NPCs) in video games. BT are a powerful and modular framework for designing and implementing decision-making and control systems. They provide a structured way to define complex behaviors by combining smaller, reusable behavior modules into a hierarchical tree structure. BT have gained popularity in robotics due to their flexibility, readability, and ability to handle dynamic environments (e.g., Nav2, a popular navigation stack used in ROS2, is now deployed using BT [Macenski et al., 2024], similarly, BT are a keystone of projects such as [Street et al., 2024] where they aim at verifying the whole development toolchain).

Why yet another BT implementation? To provide a formal representation of BT which can then be used to formally verify interesting properties of the BT (can this node succeed, or fail, is this node reachable or not, can we prove that if we reach this state, eventually we will reach this one within some time interval, etc), and also to execute the formal model which implements the BT in lieu of a regular BT engine.

A behavior tree consists of nodes arranged hierarchically. Parent nodes tick their children to pass them the execution "token". Children return either `success`, `running` or `failure` to their parent when the current execution tick is done, and the parent pursues the execution following a specified behavior.

Root node The entry point of the tree that initiates the behavior evaluation and generates the successive ticks.

***Leaf* nodes** The terminal nodes that perform specific functions when ticked:

- ***Condition* nodes** Evaluate conditions, such as checking sensor data, battery level, or environmental states. They return `success` or `failure`.
- ***Action* nodes** Trigger actions in the environment, such as moving a joint, picking up an object, or sending a command. Action, on top of `success` or `failure`, may also return `running` (e.g., when it takes *some* time to complete the action such as a robot motion).

***Control* nodes** These modes specify the execution logic (behavior) of their children nodes (i.e., how to execute them):

- ***Sequence*** Executes its children from left to right. If any child fails, the *Sequence* fails, halting further execution. If the last one succeeds, the *Sequence* succeeds. If a child returns `running`, so does the *Sequence*, which will, when ticked again, either call the last running child or restart the *Sequence*, depending on its type: *Sequence* or *ReactiveSequence*.
- ***Fallback*** (a mirror of *Sequence*) Executes its children from left to right. If any child succeeds, the *Fallback* succeeds, skipping the rest. If a child returns `failure`, we tick the next child, unless it was the last one, in which case the *Fallback* returns `failure`. When a child returns `running`, the *Fallback* returns `running`, and, when ticked again, will tick either the last ticked child or the first one depending on its type: *Fallback* or *ReactiveFallback*.
- ***Parallel*** Runs/ticks multiple children simultaneously and succeeds or fails based on some success threshold (e.g., either all or some of the children must succeed for `success`).

***Decorator* nodes** perform a specific operation on a single child (e.g., the *Invert* node returns `success` when its child returns `failure`, and vice versa and returns `running` when the child returns `running`). Here is a non-exhaustive list of *Decorator* nodes: *Repeat*, *ForceFailure*, *ForceSuccess*, *RetryUntilSuccessful*, etc.

This is a very quick and shallow description of BT, and we invite the reader to check books (e.g., [Colledanchise and Ögren, 2018]) and online tutorials (e.g., `https://www.behaviortree.dev/`, which introduces the popular `BehaviorTree.CPP`), to get a complete picture of the BT "programming" ecosystem,

Note that the BT specifications are not "closed". If most implementations propose the BT nodes described above, some offer additional *Control* nodes which implement more "specific" control algorithms (e.g., Nav2 [Macenski et al., 2024] proposes *Recovery*, *PipelineSequence*, *RoundRobin*, ...) or additional *Decorator* nodes (e.g., Nav2 proposes *RateController*, etc). Overall, BT are making explicit the *control* of the execution of the nodes. However, for the leaves of the tree (*Action* and *Condition* nodes), the specification remains minimal and silent on some features (e.g. can one pass arguments to nodes? returning values? asynchronous calls? time taken by the real execution? etc). Nevertheless, most implementations specify

how these features are handled (e.g., with black board for variables, or in the C/C++ code called to implement them, multi-threading, etc).

Considering that more and more robotics applications, using BT, may be deployed in critical applications (autonomous drones or vehicles, etc) or in an environment with human (service robots), we need to be able to prove some safety properties on the BT, and to trust their execution will remain faithful to the programmer's intentions. For this we believe we need to harness some formal models to the BT language and its execution engine to enable some formal offline and online validation and verification of BT.

2 State of the art and proposed approach

We split the state of the art in two parts, on one side, the approaches and the papers concerned with BT in robotic applications, on the other side, the ones which study formal models jointly with BT.

2.1 BT in robotics

BT are praised for their modularity, readability, scalability, flexibility, robustness, and supposedly being easy to debug and test. Even if these are questionable and somewhat subjective, one cannot deny their rising success and interest in robotics for autonomous navigation, human-robot interaction, manipulation tasks and multi-agent systems.

The book [Colledanchise and Ögren, 2018] covers most, if not all, aspects of BT in robotics. They make an extensive presentation of BT, how they compare to FSM, how they can be linked to the planning activity, etc. They mention the importance of safety and formalism, although not much is said on formal proof and verification.

In [Iovino et al., 2022] the authors make a comprehensive and large survey of the topic of BT in AI and robotic applications. The existing literature is described and categorized based on methods, application areas and contributions, and the paper concludes with a list of open research challenges: explainable AI, human–robot interaction, safe AI, and the combination of learning and BT.

The work presented in [Marzinotto et al., 2014] shows the equivalence between BT and Controlled Hybrid Dynamical Systems. Similarly the authors of [Ögren and Sprague, 2022] study how to deploy Behavior Trees in Robot Control Systems, and they propose an interesting formal analysis regarding convergence and regions of attraction.

The authors of [Schulz-Rosengarten et al., 2024] address one of the BT "shortsight" and propose to add a cleaner communication extension, but lack formalism and proof on the LF. The input/output mechanism is inspiring.

As for evaluating BT, the authors of [Gugliermo et al., 2024] propose a set of metrics (some static, some gathered from real runs), to evaluate some BT properties, as to evaluate and analyze them.

Implementation considerations Deploying BT in robotic applications requires addressing implementation issues which may not be present in Video Game programming. In [Colledanchise and Natale, 2018] the authors present an original approach to handle parallelism and concurrency in BT (CBT) with execution progress and resources management. In [Colledanchise and Natale, 2021] , the same authors point out the issues on memory nodes (to avoid reevaluating), asynchronous action calls, parameters, halt (blocking or not), etc.

BT and planning In many robotics architectures, BT is considered as the "acting" component of the decisional layer [Ingrand and Ghallab, 2017]. This is the case for example in PlanSys2 [Martín et al., 2021] (now part of the AIPlan4EU platform [Micheli et al., 2025]) where the planner produces plans as BT which can be deployed for plan execution. It is also interesting to study how BT may also be extended to perform some planning. In [Colledanchise et al., 2019] the authors propose a dynamic modification of BT for planning (planning with back chaining), so they can perform robust acting, without resorting to replanning. On another yet different type of planning/BT interaction, in [Köckemann et al., 2023] planning is used to produce testing plans for BT, whose testing participates to increase the trust we put in these BT.

2.2 BT and formal models

This paper main subject is about BT and formal V&V, so we now examine the state of the art in this area.

In [Klöckner, 2013] the authors propose to interface BT mission plans and a simulation of the world using the description logic ($\mathcal{ALC}(\mathcal{D})$). So the description logic formal model acts as a safety check between the plan execution, and the simulation. Even if this does not provide a formal proof of the mission plans BT, it improves the trust we can have in these plans by formally validating their execution (in simulation) before deploying them in the real world.

The authors of [Colledanchise et al., 2017] propose to synthesize correct by construction BT from an environment specification along the agent model and an objective expressed in LTL. From a standpoint, the approach is clearly sound and synthesizes correct BT, but requires the programmer to write LTL goal specifications to get started which may be seen as a deterrent to non formal "programmers".

The last four approaches we present here have strong similarities with the one we propose.

In [Biggar and Zamani, 2020] the authors propose to synthesize LTL from BT and then show that the obtained model satisfies some LTL specifications. The paper goes in depth to explain the translation process, although it is not clear it can be automated, and a priori, the produced formal model cannot be directly executed in place of a regular BT engine.

The authors of [Colledanchise et al., 2021] focus on the formalisation of the execution context of BT to be able to perform runtime verification. They propose *channel systems* to model the "surroundings" of the BT and then to check at runtime that some specifications, written in the SCOPE language, are satisfied while the BT executes. So the approach, which is not limited to BT, provides a very strong and formal execution framework sitting between the BT and the robot, behaving like a safety bag. The battery example they present has some similarity with the one we deploy on our UAV in section 6.1.

Similarly, the authors of [Serbinowska et al., 2024] focus primarily on runtime verification of BT with contingency monitors (BTM) written with a DSL: BehaVerify. These monitors can be used to correct an undesirable behavior when it is detected and can handle LTL specifications. Yet, they can also check the BT at design time, by checking these BTM with model checking.

In [Wang et al., 2024], the authors present an approach where they use the BIP formal framework to model BT and propose an implementation of their tool: xml2bip. They then use model checking (not D-Finder as the original BIP implementation did) to check for formal properties. Although some versions of BIP come with a runtime engine (e.g., the one used

in [Bensalem et al., 2011]), they do not yet propose a "fornal" execution of the BT with the BIP engine.

2.3 Proposed approach

Our approach has one main goal: to provide a formal semantics for BT, by translating it to a formal model, which can then be used offline to check formal properties, but also online to implement and enforce this semantics.

We propose to achieve this objective by following these steps.

- Define a clear complete and unequivocal translation of all BT to a formal model in FIACRE [Berthomieu et al., 2007, 2008].
- The obtained BT formal model can be checked and analyzed to prove logical and temporal properties (LTL and CTL).
- The *same* BT formal model can be linked to actual code and executed like other BT framework engines (e.g. `Behaviortree.CPP`, `BT.py`) do. This shows that the operational semantic of the BT formal model is the expected one, while guaranteeing the property proven offline.

Moreover, implementing this approach leads to some interesting side effects and features. It clarifies the BT semantics when needed, e.g., the wait/halt semantics when `running` nodes must be halted. It also enables time representation extensions and enriches the BT language with state variables and functions evaluation.

The rest of the paper is organized as follows. After introducing above the BT, the state of the art and our approach, we first present in Section 3 the FIACRE/TINA/HIPPO formal suite we use. Section 4 introduces how each BT node is mapped in FIACRE (this is implemented in our BT2FIACRE tool[1]). Then Section 5 presents how these FIACRE BT nodes are put together to build the complete formal model of the whole BT, and we then show what are the type of formal properties one can prove offline but also at runtime. Two examples are presented: in Section 6.1, we introduce a drone controller written in BT for which we successfully deployed our approach, and; in Section 6.2, we show how the Nav2 BT [Macenski et al., 2024] can be deployed with BT2FIACRE. A discussion in Section 7 reassesses the pros and cons of the BT2FIACRE tool and the use of FIACRE as an underlying formal language to provide a formal model and a formal semantics to BT, followed by future work section and the conclusion of the paper.

3 A Formal Framework for Offline and Runtime Verification: The FIACRE Language, Models, and Tools

While this paper does not aim to exhaustively present the formal framework we use, some terminology and explanations are essential for clarity and make the paper self contained.

[1]The BT2FIACRE tool developed for this study can be downloaded from the repository: https://redmine.laas.fr/projects/bt2fiacre/pages/index.

Readers interested in more details may consult the specific papers and websites cited below. [2]

3.1 Terminology, Models, Languages, and Tools

We define the following terms:

Time Petri Nets [Berthomieu and Diaz, 1991] are an extension of traditional *Petri nets* where each transition has an associated time interval (typically $[0, \infty)$) specifying the time range within which an enabled transition can be fired.

TTS Time Transition Systems extend *Time Petri nets* by adding data-handling capabilities, allowing transitions to invoke data processing functions.

TINA (short for "TIme Petri Net Analyzer") is a toolkit for editing, simulating, and analyzing *Petri nets*, *Time Petri nets*, and *TTS*. Within this toolkit, `sift` and `selt` enable the construction of reachable state sets and the verification of LTL properties.[3]

FIACRE stands for "Intermediate Format for Embedded Distributed Component Architectures" (in French). It is a formally defined language designed to represent the behavioral and timing aspects of embedded and distributed systems for purposes of formal verification and simulation. FIACRE specifications can be compiled into a *TTS* using the `frac` compiler.[4]

H-FIACRE adds *Event Ports* and *Tasks* linked to C/C++ functions, to make the FIACRE models "executable".

HIPPO is an engine for executing *TTS* resulting from the H-FIACRE specifications compilation [Hladik et al., 2021].[5]

This framework has been applied across various projects and applications,[6] including the validation and verification of functional components in our robotics experiments [Dal Zilio et al., 2023], but also to the validation and verification of robotic skills programmed in PROSKILL [Ingrand, 2024a].

3.2 FIACRE Semantics

Although the formal model and tools are detailed in specific papers and websites (see above), we include a brief example to illustrate the semantics of the FIACRE language. The example, a triple-click detector for a mouse, is shown in Listing 1p7[7] and illustrated in Figure 1. It defines three FIACRE processes, each represented by an automaton. The first process, **clicker**,

[2] Note that a similar FIACRE presentation can be found in this paper [Ingrand, 2024a] (from the same author). We include it almost as is in this paper as to make the paper self contained, nevertheless, If the reviewers believe this section should be shortened and replaced by a pointer to the other paper, this is perfectly fine for us.

[3] https://projects.laas.fr/tina/index.php

[4] https://projects.laas.fr/fiacre/index.php

[5] https://projects.laas.fr/hippo/index.php

[6] https://projects.laas.fr/fiacre/papers.php

[7] All floating Listing and Figures numbers are given with the page number. In this case, Listing 1p7 is Listing 1, page 7.

generates a *click* at any time, waiting between 0 and ∞, then synchronizes on the click port with **detect_triple_click**. This second process has four states, waiting for synchronization on click or until the maximum allowed time between clicks (0.2 sec) has passed. Note the `select` option in *wait_second* and *wait_third* states, introducing a non-deterministic choice for exploration by the model checker. Upon reaching *detected*, a synchronization on triple_click enables the transition of the **triple_click_receiver** process to *received_tc*.

Following these specifications, a component is defined by placing three process instances in parallel (line 1.56)[8] and linking them through two ports (line 1.54). This example is simple by design, though the FIACRE language supports complex data types, bidirectional ports, local and global variables, conditions, switch/case statements, transition guards, and function calls (internal to FIACRE or external in C/C++ code) for advanced computation. More complex FIACRE specifications can be found in A and B.

Listing 1: FIACRE specification for a triple click detector (FIACRE offline version).

```
1  process clicker [click:sync] is // synthesize clicks and sync them on its port at any time
2  states wait_click, make_click
3
4  from wait_click
5     wait [0, ...[; // wait any time from zero to infinity
6     to make_click
7
8  from make_click
9     click; // issue a click sync on the Fiacre port
10    to wait_click
11
12 process detect_triple_click [click:sync,triple_click:sync] is
13 states wait_first, wait_second, wait_third, detected
14
15 from wait_first
16    click;        // first click
17    to wait_second
18
19 from wait_second
20    select       // we wait either
21      wait [0.2,0.2]; // exactly 0.2 second
22      to wait_first // then reset the detector
23    []
24      click;      // or for the second click
25      to wait_third // whichever comes first
26    end
27
28 from wait_third
29    select       // again for the third click
30      wait [0.2,0.2];
31      to wait_first
32    []
33      click;      // third
34      to detected
35    end
36
37 from detected
38    triple_click; // sync on the triple_click port
39    to wait_first
40
41 process triple_click_receiver[triple_click:sync] is
42 states waiting_tc, received_tc
43
44 from waiting_tc
45    triple_click; // just wait for a sync on this port
```

[8] Listing lines are referenced with the <listing number>.<line number>, example: 1.56 is Listing 1, line: 56.

```
46     to received_tc
47
48   from received_tc
49     /* do what needs to be done when a TC has been detected */
50     to waiting_tc
51
52   component comp_tc is //we now specifiy the component
53
54   port click:sync in [0,0], triple_click:sync in [0,0] // two ports
55
56   par * in // 3 processes composed in parallel
57     detect_triple_click[click, triple_click] // process 1
58   || clicker[click] // process 2
59   || triple_click_receiver[triple_click] //process 3
60   end
61
62   comp_tc // this instantiates the component
63
64   // some properties to check
65   property ddlf is deadlockfree // deadlock free (TRUE)
66   assert ddlf
67   // in the next property comp_tc/3/state designates the state in the 3rd process of
68   property cannot_receveice_tc is absent comp_tc/3/state received_tc // the comp_tc component
69   assert cannot_receveice_tc // we cannot detect a triple click (FALSE)
```

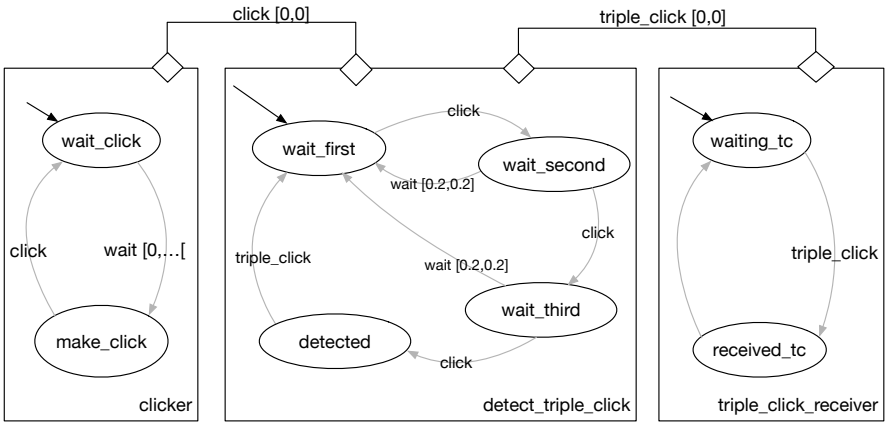

Figure 1: The FIACRE processes modeling the FIACRE specification on Listing 1p7.

3.3 Offline Formal Verification

The frac compiler is used to compile the FIACRE specifications shown in Listing 1p7 into an equivalent TTS. With the sift tool from the TINA toolbox, one can then construct the system's set of reachable states, and by using selt, the properties outlined in the initial FIACRE specifications, as well as any additional properties, can be verified. The TINA toolbox offers a variety of other tools that readers may explore for further analysis.

Listing 1p7 suggests several properties to check for this specification: Is the model deadlock-free (line 1.65)? This is confirmed to be TRUE. Can the model successfully detect a triple click (line 1.68)? By verifying that reaching the *received_tc* state is possible, the model confirms that it can indeed detect a triple click. Additional complex properties could be added, such as ensuring there is at most 0.4 sec between the first and last clicks.

The verification approach used by the TINA tools relies on model checking, which can be affected by state explosion [Clarke et al., 2012], potentially limiting its effectiveness. However, as demonstrated in section 6.1.1, the results from our example remain both insightful and non-trivial.

3.4 H-FIACRE Runtime Extensions

Although FIACRE was originally designed for offline verification, it has been extended with two primitives that enable runtime verification [Hladik et al., 2021]. These extensions allow the model to connect with C/C++ functions that send events or execute commands, forming what we call H-FIACRE, to distinguish it from the base FIACRE language.

The purpose of the H-FIACRE runtime version is to make the model "executable" in connection with real-world interactions.

Listing 2p9 (along with Figure 2p10) presents the executable version of the specification given in Listing 1p7.

Event Ports are defined in the specification's preamble (see line 2.1), linking a C function to a FIACRE port. In this case, the event *click* is linked to the `c_click` function in C/C++. When this port is one of the possible transitions (lines 1.16, 1.24, and 1.33), the C/C++ function is called, and the port becomes active upon the function's return. These C/C++ functions can accept and return values typed in FIACRE.

Tasks are also defined in the preamble (see line 2.3), associating a task (in this case, *report_triplec*) with a C/C++ function (here `c_report_triple_click`), which is called asynchronously upon a `start` (see line 2.18). This enables the corresponding `sync` (line 2.22) once the C/C++ function completes. Values can be passed to the task at call time and returned when it completes.

Listing 2: H-FIACRE processes implementing a triple click detector.

```
1  event click : sync is c_click // declare the Fiacre event port which transmits click
2  task report_triplec () : nat is // declare the task and
3    c_report_triple_click // the C/C++ function called by this task
4
5  process detect_triple_click [triple_click:sync] is
6  // this process is exactly the same than in the regular Fiacre version
7  // only the click port is now an event port
8
9  process triple_click_receiver[triple_click:sync] is
10 states waiting_tc, received_tc, sync_report
11 var ignore : nat
12
13 from waiting_tc
14    triple_click;
15    to received_tc
16
17 from received_tc // show an example of an external call
18    start report_triplec();
19    to sync_report
20
21 from sync_report
22    sync report_triplec ignore; // wait until the call return
23    to waiting_tc
24
25 component comp_tc is
26 port triple_click:sync
```

```
27
28  par * in
29     detect_triple_click[triple_click]
30  || triple_click_receiver[triple_click]
31  end
32
33  comp_tc
```

In this example, we replace the **clicker** process, which previously synchronized with click at any moment, with the *click* event port (highlighted in purple). Additionally, we introduce a task (*report_triplec* shown in light blue) to execute when synchronizing with a *triple_click* in the **triple_click_receiver** process. The remainder of the model remains unchanged, transforming it from a model specifying a triple-click detector to an actual program or controller that implements it. In this way, the specification itself becomes an executable program.

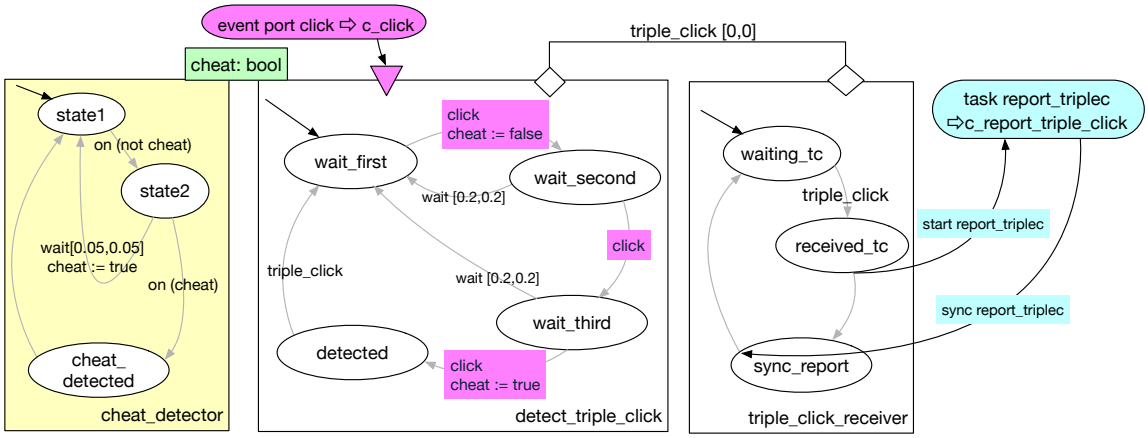

Figure 2: Illustration of the H-Fiacre program on Listing 2p9.

3.5 Runtime (Online) Verification

The H-Fiacre model, once compiled into TTS format using frac, is linked with the Hippo engine and the C/C++ functions needed to run it (e.g., `c_click` and `c_report_triple_click`). The Hippo engine executes the TTS model in real-time and initiates the appropriate C/C++ function calls (in separate threads) connected to event ports and tasks. Note that the properties checked offline also hold in the online version (as the reachable states set of the former include the one form the latter).

Additionally, the model can be extended with runtime verification properties by incorporating a monitoring process. For instance, if this controller is applied in a video game where rapid sequences of triple clicks are suspicious (indicating potential use of a cheating device), a new **cheat_detector** process could be added (shown on the left in Figure 2p10 and in Listing 3p11). This process could have three states and use a shared Boolean variable, `cheat`. This variable would be set to 'false' upon transitioning to the *waiting_second* state in the **detect_triple_click** process, and switched to 'true' upon reaching the *detected* state if the click sequence timing suggests non-human activity.

Listing 3: **cheat_detector** process detecting a cheating device by monitoring the `cheat` Boolean variable.

```
process cheat_detector(&cheat:bool) is

states state1, state2, cheat_detected

from state1
  on (not cheat); // guard on (not cheat)
  to state2

from state2 // cheat was set to false
  select // either
     wait [0.05,0.05]; // 50 ms elapsed
     cheat := true; // reset the cheat variable
     to state1 // go back to monitoring
  []
     on (cheat); // cheat became true again before the 50ms above.
     to cheat_detected //caught cheating
  end

from cheat_detected
  // the player is cheating, do what needs to be done.
  to state1
```

Within the **cheat_detector** process, in *state1*, the system sets a guard on `(not cheat)` before transitioning to *state2*, where it then waits for either 50 ms or until `cheat` becomes true. If `cheat` becomes true before 50 ms has elapsed (indicating a suspiciously fast triple-click), it transitions to *cheat_detected* and flags this unusual activity. If 50 ms passes without the `cheat` variable being set to true, the system sets `cheat` to true and returns to *state1*.

This approach allows us to synthesize a controller that directly runs the specification. This dual capability is a major strength of the Fiacre framework: the same formal model can be verified offline and executed online. In practical terms, this means that the controller's real-time behavior aligns with the initial model specifications, validating that the offline-verifiable properties are applied consistently in the live system. While observing expected behavior is a necessary, though not entirely sufficient, indicator of correctness, it significantly strengthens the link between specification and execution.

Moreover, if runtime behavior diverges from expectations, you can debug it as you would with any programs. From a formal perspective, the possible traces of the H-Fiacre version (also called the Hippo version) are contained within those of the Fiacre model (also called the Tina version), ensuring consistency between the runtime model and its offline counterpart.

4 The mapping of BT in Fiacre

Before getting into the details of the produced formal models, we present on Figure 3p12 the overall workflow from BT to the formal executable version (top part in green), and the formal verifiable version and its analyzed properties report (bottom part in purple). One should keep in mind, that the BT programmers only provide the various BT in `.btf` format (like the one on Listing 4p11), the C/C++ codes which glue *Action* and *Condition* BT to the real robot commands and perception primitives (all in blue) and, optionally, LTL properties to verify, and monitors written in Fiacre (in slanted blue). The rest is fully synthesized and automatically compiled.

Listing 4: A simple drone survey BT. Note that for historical reasons and as we are reusing some of our existing tools, we do not use an XML syntax but rather a Lisp like syntax (called

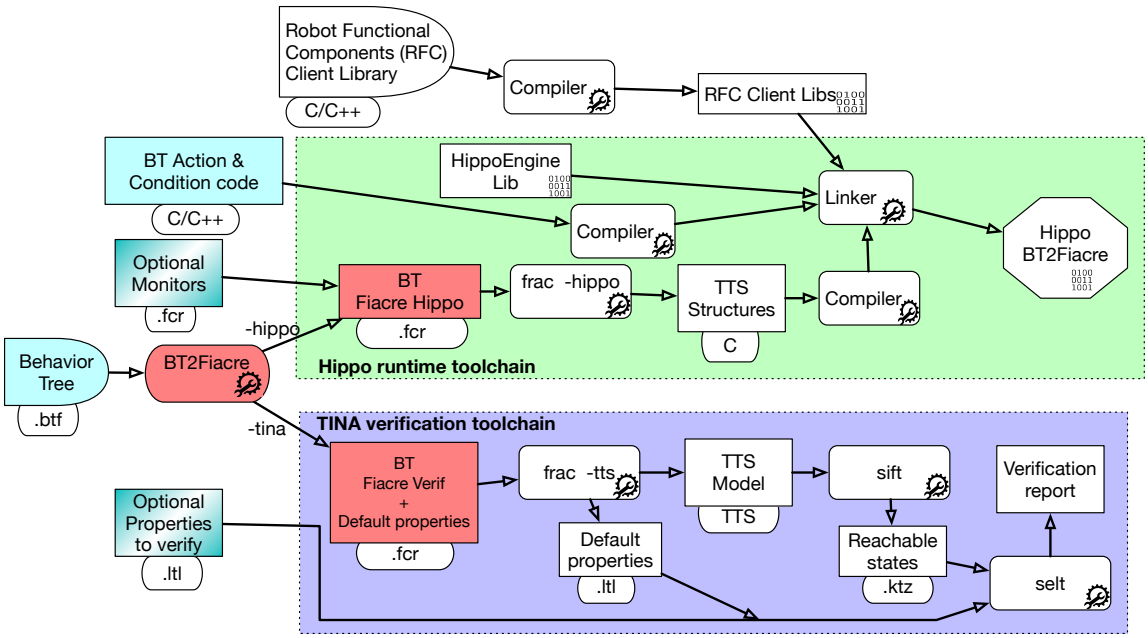

Figure 3: The BT2Fiacre (Hippo/Tina) workflow. Only the data in the blue boxes need to be provided by the programmer. The BT2Fiacre tool (in light red) synthesizes the two models, the rest is fully automated. In light green, the workflow for the Hippo runtime verification version, and in light purple, the workflow for the Tina offline verification version.

the `.btf` format), but this has no consequence on the content and the interpretation of BT (See Appendix E, Listing 10p40 and 11p41 for an example in both formats).

```
((BehaviorTree :name drone
  (Sequence
   (ParallelAll :wait 1 :halt 0 ; if wait is 1, will wait the running branch if one fails
    (Action :ID start_drone)
    (Action :ID start_camera))
   (ReactiveSequence
    (Fallback
     (Condition :ID battery_ok) ; check if the battery is OK
     (ForceFailure :ID fail ; if not, just land and fail
      (Action :ID land)))
    (Fallback
     (Condition :ID localization_ok) ;same for localization
     (ForceFailure :ID fail
      (Action :ID land)))
    (Sequence
     (Action :ID takeoff :args (height 1.0 duration 0))
     (Parallel :success 1 :wait 0 :halt 1 ; If the survey or the nav succeed, we are done.
      (Action :ID camera_survey)
      (Sequence
       (Action :ID goto_waypoint :args (x -3 y -3 z 5))
       (Action :ID goto_waypoint :args (x -1.5 y 3 z 5))
       (Action :ID goto_waypoint :args (x 0 y -3 z 5))
       (Action :ID goto_waypoint :args (x 1.5 y 3 z 5))
       (Action :ID goto_waypoint :args (x 3 y -3 z 5))
       (Action :ID goto_waypoint :args (x 3 y -3 z 5))))
     (Action :ID goto_waypoint :args (x 0 y 0 z 5))
     (Action :ID land)
     (Action :ID shutdown_drone))))))
```

As mentioned above, the semantics of BT is not formally defined. By translating BT to FIACRE, we define a formal semantics, hopefully consistent with the operational semantics people expect from BT.

Another goal we pursue is to model as much BT "additional information" as possible in the FIACRE model (for example if the variable used in BT can be modelled in FIACRE, the better).

We shall first present the overall mapping and then we will point out where the semantics had to be clarified and what are the "additional" information we gather in the FIACRE model.

4.1 The general mapping

Each BT node is automatically mapped in a FIACRE process (See Section 3.2), whose automata is modeled in accordance to its node type (*control*, *decorator*). Then, all these FIACRE processes are instantiated and composed in a component which provides a shared array of FIACRE records. This array `BTnode[]` has one element for each BT and is a shared variable between all the FIACRE processes. In the following figures `BTnode[self]` is the record for the current BT node. Each `BTnode[]` record has two fields of interest: `caller` and `rstatus`. `caller` is initialized at `None` and will be set to the BT node which ticks/calls it. It is set back to `None` when its execution (for the current tick) is finished. `rstatus` contains the last returned status (`success`, `failure` or `running`) of the BT node. `rstatus` may also be set to `halt_me` (by its caller), when a `running` BT node must be halted.

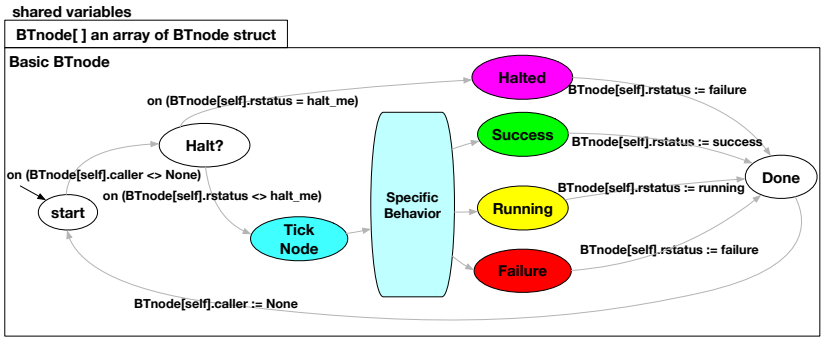

Figure 4: Preamble and postamble of BT nodes in FIACRE.

As shown on Figure 4p13, a BT node FIACRE process starts with a preamble (on Listing 8p34 or 9p35 from *start_* to *tick_node*) with a guard on its activation on `BTnode[self].caller` (i.e. presumably its parent node has ticked its by setting its `caller` field). Follow a test to check if it has been asked to halt (in case it had previously returned `running` and now its parent asks it to halt). If it needs to run, it transitions to the *tick_node* state where the particular of this node type is handled.

Similarly, the postamble mostly consists of the four automata states (`success`, `failure`, `running` and `halted`), all returning the proper return status `rstatus` and then transitioning to a *done* state, in which the control is relinquished by setting `caller` to `None` (the parent node has a guard on this to check that the node has finished this tick).

In the following *Action* and *Condition* nodes in FIACRE are presented in their TINA version (i.e., the offline verification) but also HIPPO version, (i.e. the runtime version). All other BT

nodes FIACRE models are strictly the same between the two versions.

4.2 *Condition* Node

The *Condition* node is the simplest node, it only returns `success` or `failure`. The TINA version will just return either values (See Figure 5p14), while the HIPPO version will call the *external* C/C++ function which performs the test and return its value (See Figure 6p14). The C/C++ function is expected to be fast and should not delay execution unnecessarily. Note that in our formalism, one can pass arguments to the C/C++ call. This is very much welcome to make the BT language more expressive and to some extent to have these values available in the formal model.

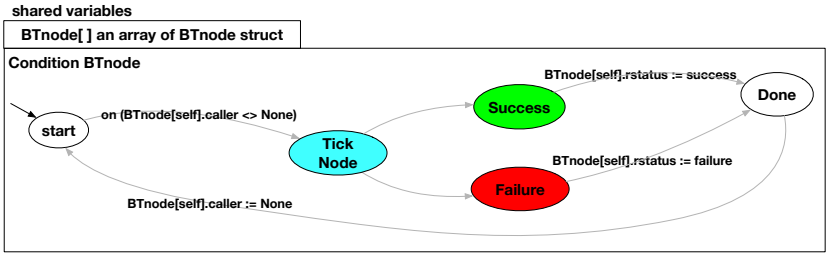

Figure 5: The FIACRE process modeling the *Condition* node for the verification (TINA) model.

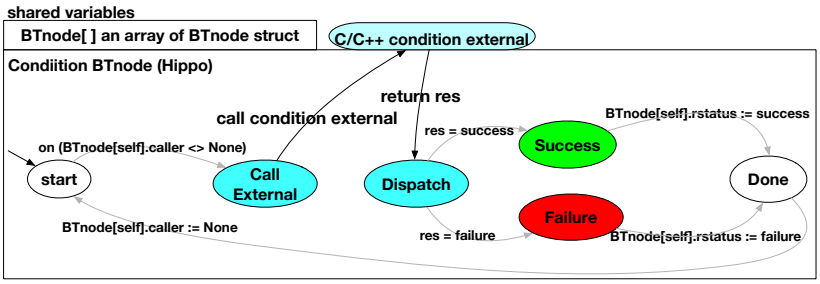

Figure 6: The FIACRE process modeling the *Condition* node for the runtime (HIPPO) model.

4.3 *Action* Node

The *Action* node is slightly more complex, as it can also return `running`. Indeed, actions may take some time and not return `success` or `failure` right away. To preserve the reactivity of the overall execution, and keep the tick short, it is often advised to return `running` while the action is still executing in its own thread. As a consequence, it means that an **Action** node may be *halted* (i.e. at some point, it is in a `running` state but its parent node wants to halt it). In the TINA version (See Figure 7p15), all the return values are possible, while in the HIPPO version (See Figure 8p15), a FIACRE *task* is started calling the C/C++ `action_task`, and subsequent ticks will return `running` until the C/C++ `action_task` finishes and returns `success` or `failure`. An *external* C/C++ `action_halt` is also defined and is called if the action

must be halted (it then returns `failure`). While the `action_task` may take some time in its own thread, the `action_halt` is expected to be fast. In an *Action* node too, one may define and pass some arguments to the C/C++ function. Listing 8p34 in Appendix A shows the FIACRE code for the *takeoff* action.

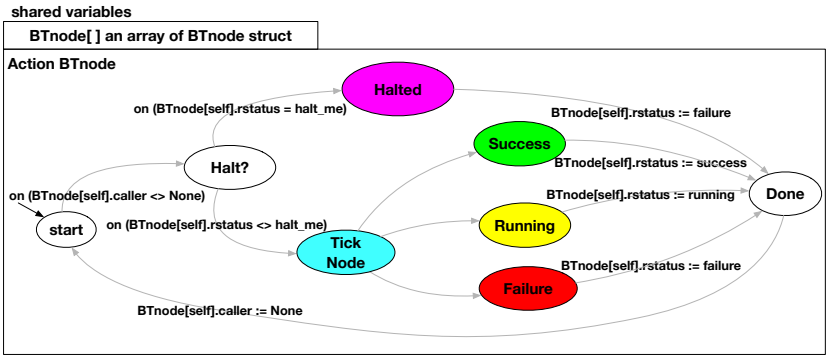

Figure 7: The FIACRE process modeling the *Action* node for the verification (TINA) model.

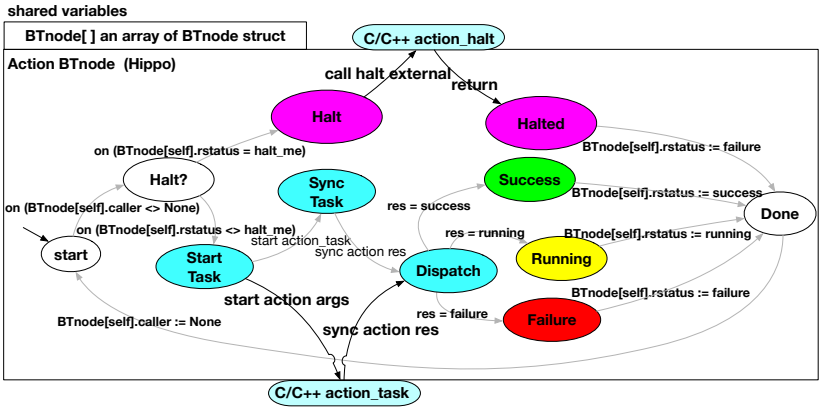

Figure 8: The FIACRE process modeling the *Action* node for the runtime (HIPPO) model.

Note that only the *Action* and *Condition* nodes have a slightly different TINA (offline) and HIPPO (online) versions. For all the other BT nodes, the model is strictly the same.

4.4 *Sequence* Nodes

The *Sequence* node ticks/executes its child in sequence as they succeed, until one fails. It then returns `failure`. If all succeed, it returns `success`. If one child returns `running`, the *Sequence* returns `running`. Upon return of the tick this particular child is ticked/called again.

Listing 9p35, shows the complete FIACRE code of a *Sequence* node with three children. The FIACRE process has a local variable `next_seq` initialized at 1, which holds which node will be ticked/called when the current node is ticked/called again. From the *Tick Node* state (See Figure 9p16), one proceeds to the $Child_{\texttt{next\_seq}}$ state, which ticks/calls the proper child. We wait until this child is done `caller=None` and check its returned status `rstatus`:

- `success`, if it is the last child to succeed, we return `success`, otherwise, we proceed to the next child $Child_{\texttt{next\_seq+1}}$.
- `failure`, we return `failure`.
- `running`, we set `next_seq` to the proper value and return `running`.

There exist variants of *Sequence: ReactiveSequence*, *SequenceWithMemory*. They alter the way the sequence is ticked after `failure` or `running` are returned. For example *ReactiveSequence* always restarts the sequence from the beginning after one child has returned `running`. The goal here is not to list all the variants, just to give a hint on how these are transformed in FIACRE. Of course, the generated FIACRE code implements the proper semantics for each variant by properly setting the `next_seq` value.

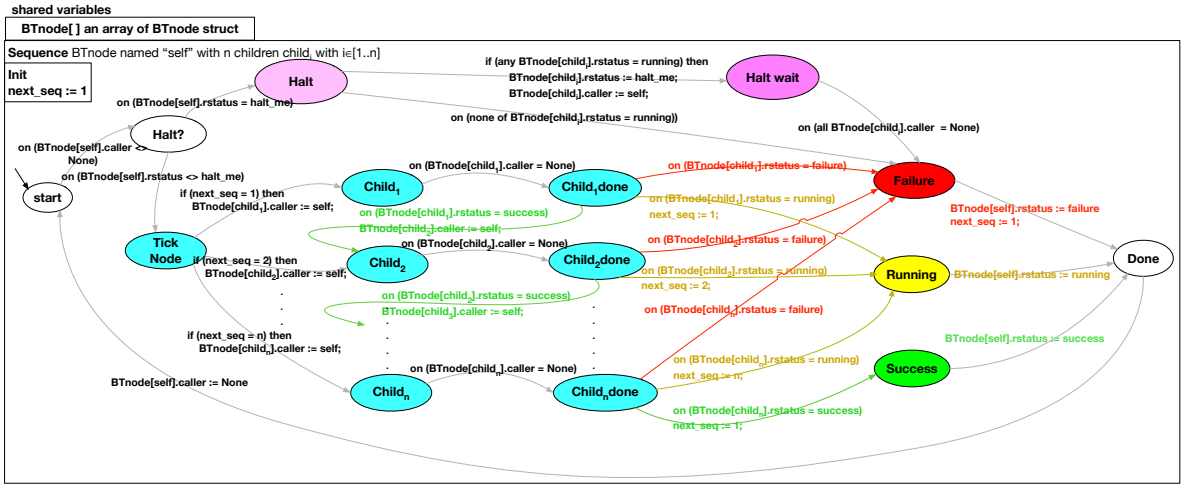

Figure 9: The FIACRE process modeling the *Sequence* node.

4.5 *Fallback* Nodes

The *Fallback* node executes/ticks its child in sequence as they fail, until one succeeds. It then returns `success`. If all fail, it returns `failure`. If one child returns `running`, it returns `running`. Upon return of the tick this child is again ticked/called. As shown on the Figure 14p38, it is the symmetric of the *Sequence* node, thus its behavior is just the mirror of the *Sequence* one described above. There is a variant of *Fallback*: *ReactiveFallback*. It alters the way the *Fallback* is restarted/reticked after `running` is returned. *ReactiveFallback* always restarts from the beginning after one child has returned `running`. This is achieved in FIACRE by properly setting the `next_fb` value.

4.6 *Parallel* Nodes

The *Parallel* node specifies how many children `m` out of all `n` children must succeed for the *Parallel* node to succeed (with *ParallelAll* they must all succeed: `n = m`). From this, we see that the implementation just ticks all the node, and then keep track of how many return `failure`,

success or running (See Figure 15p39). If any final condition for success or failure is met (enough BTnode[child1].rstatus = success)[9] or (enough BTnode[childn].rstatus = failure)[10], then it proceeds to the corresponding state, otherwise, this node returns running. Note that *Parallel* nodes can lead to children being halted (if the condition for success or failure are met while some children are still running).

4.7 *Decorator* Nodes

Decorator nodes have only one child. When this child is *done* the transitions to success, failure or running depend on the particular *decorator* type (*Inverter*, *ForceFailure*, *ForceSuccess*, *KeepRunningUntilFailure*, *RetryUntilSuccessful*, *RateController*, *Repeat*, etc). For example, the *Inverter* (See Figure 10p17) one will swap the success and failure transitions, while leaving untouched the running one, *Repeat* will call its child a number of times, etc.

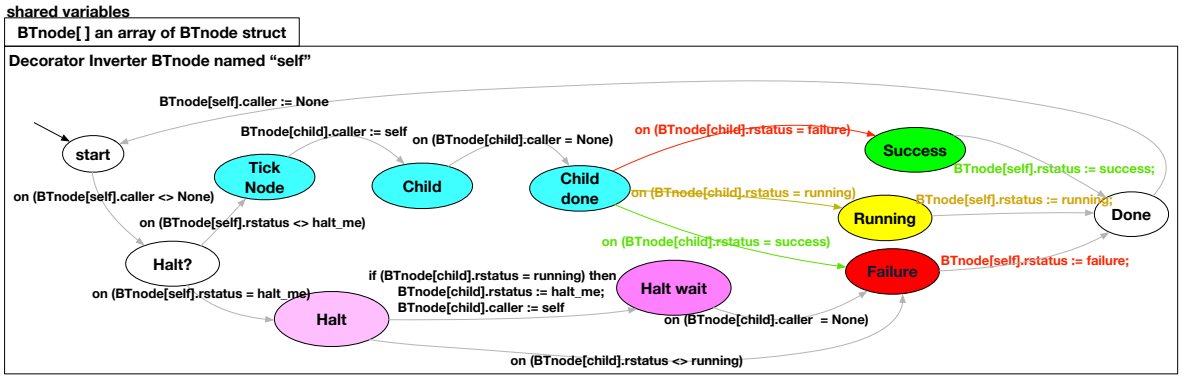

Figure 10: The FIACRE process modeling a *Decorator* (*Inverter*) node.

4.8 Semantics clarification and added Features

As mentioned earlier, the translation to FIACRE defines a formal semantics of BT. But there are particular choices which need to be clarified and specified. Many of these choices have already been identified [Ghiorzi and Tacchella, 2024], we try here to settle them with a formal proposition.

4.8.1 Halting running BT nodes

We have seen there are a number of situations where one may have to handle "orphan" running branches. They are not really orphans, but the situation is such that their parents have already decided the success or failure, independently of their own final outcome other than running. This happens when a *Parallel* fails (or succeeds) and there are still running children, similarly when a *ReactiveSequence* fails or a *ReactiveFallback* succeeds, one child may still be running. So for these nodes we added 2 keywords :wait and :halt to their specification. If :halt is true, we explicitly halt the still running children. Note that this is

[9] (enough... success) is true when success >= m
[10] (enough... failure) is true when failure > n - m

propagated all the way to leaves nodes, in particular *Action* nodes will explicitly halt whatever they are doing by calling their `action_halt` C/C++ function (see Section 4.3). If `:wait` is true, the node waits until all the children return something else than `running`. These choices can be discussed and modified, but at least, the semantics is clarified, defined and implemented in the formal model.

4.8.2 Ticks and BT root

FIACRE supports time, and so does the produced TTS and the TINA verification tools. We consider that a BT tick takes one FIACRE unit of time[11]. By default, only the BT root generates ticks, one after another, independently of "how long" the tick traversal took. So the default BT model produced in FIACRE has just one transition $[1, 1]$ in the root BT, and all other transitions are timeless: $[0, 0]$. This is a perfectly "fine" assumption if one considers the tick more as an execution "token" traversing the BT in no time, but if one wants to perform more advanced validation and verification, it may be a good idea to propose different tick semantics. For now we propose two other semantics, one is to have all the *Action* and *Condition* BT nodes to have a $[1, 1]$ transition on their *tick_node* state, and the other one is to have all BT nodes with such $[1, 1]$ transition. Of course, the chosen semantics among the three possibilities can be specified when producing the FIACRE model. Note that from a temporal model checking point of view, the last semantics is preferred, as it minimizes simultaneous transitions interleaving, hence the branching factor to the reachable states exploration.

The BT root is also responsible for keeping the BT alive. What should it do when its child returns `running`, `success` or `failure`? Some implementations keep running on `success`, others do not. To clarify the model we produce, the root BT keeps `running` while its child returns `running`, and stops when it returns `success` or `failure` (e.g., in the drone experiment we present in Section 6.1, this is the expected behavior).

4.8.3 State Variables, Expression Evaluation and Node Status

BT tend to rely on "external" *Actions* and *Conditions* to compute values, tests them, etc. One of our objectives is to get as much as possible of the BT and its associated *Condition* and *Action* nodes modelled in FIACRE. The more we get, the more properties we can prove on the reachable state of the BT, and the more we control execution at runtime.

Hence we introduce *state variables* which can be used in the BT model and will end up in the FIACRE model as FIACRE variables.

For now, state variables can hold a natural number or an enumeration. See the example below with flight_level which can take an `int` between 0 and 3, or battery which can take three values (*Good*, *Low* and *Critical*) (note that for an enumeration, one can specify the acceptable transitions from one value to others).

We also added two new leaves nodes to the BT specification:

SetSV nodes can be used to set a state variable passed with the :SV keyword, by calling the :ID function (defined in FIACRE). *SetSV* nodes only return `success`.

Eval nodes evaluate the condition they hold and return `success` or `failure`.

[11] In the regular FIACRE verification language, time is "unitless", but in the HIPPO engine, we set the "tick" (100 ms in the experiments described in this paper).

Last, we also make available for evaluation the resulting status of any BT, e.g. `(Eval (= camera_track.rstatus success))` will return `success` if the execution of the `camera_track` *Action* node was a `success`.

5 Deploying, Model Checking and Running the FIACRE BT

The final FIACRE model of the BT is produced by instantiating all the BT nodes FIACRE processes and combining them in parallel.

As shown on Figure 3p12 both FIACRE models (offline and online) are compiled in their respective TTS (i.e., a Time Petri Net with data) with the `frac` compiler.

5.1 Offline formal verification of BT

The TINA toolbox used here mostly rely on LTL (SE-LTL state/event LTL [Chaki et al., 2004] to be more accurate). LTL already offers a rich language and a lot of flexibility to write and prove some logical and temporal properties on the formal BT. Using model checking, one can either check properties on the fly (reachability of a state), or show the absence of a state (but for this, the whole reachable state set has to be built). By default, the `sift` tool computes the reachable states set of the BT. The resulting `.ktz` file (binary kripke transition system) can then be analyzed with default or additional properties with the `selt` tools[12].

For each BT node, we define and check some default properties: can the BT execute and complete, can it succeed, fail, or return running? Can it be halted? Most of these properties correspond to either a particular *state* in the fiacre model, or to some explicit value in the `btnode[self].rstatus` record. So the properties can be synthesized automatically as follow (by checking if their negation is reachable, i.e. we try to show that the state cannot be reached, and expect `selt` to return a counter example):

```
// For the Action_takeoff btnode
prove absent drone/12/state done
prove absent drone/12/value (btnode[takeoff].rstatus=success)
prove absent drone/12/value (btnode[takeoff].rstatus=failure)
prove absent drone/12/state halted
// the 12 is the index of the Action_takeoff btnode process instance in all the processes
// combined in parallel to build the drone component
```

One can also check more complex properties. For example, if we reach a particular state, then we will eventually reach another one.

We also modelled some examples found in some papers presented in Section 2.1.

In [Biggar and Zamani, 2020] the authors present an example of a Mars rover using unfolded solar panels to charge its battery, but should fold them when there is a storm. The same example is studied in [Wang et al., 2024] with the BT transformed to BIP on which they check the same properties. The equivalent BT in the `.btf` formalism can be found in Appendix F, Listing 12p42. After translation to FIACRE, we can build the set of reachable states:

```
sift -stats mars_rover.tts -rsd mars_rover.tts/mars_rover.ktz
20783 classe(s), 20275 marking(s), 72 domain(s), 103111 transition(s)
0.520s
```

and to prove the property (check that the robot cannot be hit by a storm while its solar panels are unfolded):

[12] The TINA toolbox provides many tools, we just focus here on `selt` and `sift`.

```
prove absent (mars_rover/3/state Unfolded and mars_rover/1/state Storm)
```

we get (as they do):

```
never (sv__panel__automata_1_sUnfolded /\ sv__meteo__automata_1_sStorm)
FALSE
```

`selt` finds a counter example, i.e., a state where this can happen (which is a problem and need to be fixed).

Similarly, in [Wang et al., 2024] they develop an example with a train having a non null speed while a door is open, we modelled it and reach the same formal proof results.

5.2 Online (or runtime) verification

The formal executable model is obtained by producing the TTS with the `frac` compiler, and then linking the result with the HIPPO engine library and the C/C++ code implementing the *Action* and *Condition* FIACRE tasks and FIACRE externals (Section 4.3 and 4.2). In most case these C/C++ code use the client library which allows to call actions, services or check topics on ROS nodes [Quigley et al., 2009]; or to make requests to or read ports from GenoM modules [Dal Zilio et al., 2023].

As mentioned earlier, our translation from `.btf` BT to FIACRE defines the formal semantics of the language. We have seen above that we can prove properties on this formal model. But executing the formal model with HIPPO is also a way to "validate" that the formal semantics we defined is correct with respect to the operational semantics one expects from BT. In other words, the execution of the formal model produces what is expected by the original BT programmer, as if he/she were using a regular BT execution engine.

As seen on Figures 16p40 and 11p21, our BT show changing color while executing: white, the node has not yet been visited; dark blue, the tick is currently in this node; light blue, the tick has been passed to child(ren) node(s) below and the node is waiting for the children to return; yellow, the node has returned `running`; green, the node has returned `success`; red, the node has returned `failure`; purple, the node was running and has been halted by one of its parent nodes; pink, the node has been instructed to halt itself (and its running branches).

The white square node close to the root of the BT indicates the HIPPO tick which is also the BT tick.

6 Some illustrating examples

We now illustrate our approach with two examples, a drone controller and the Nav2 navigation ROS2 stack.

6.1 An UAV surveying an area

We program an UAV to perform a survey mission with a BT. The functional layer of this experiment (Figure 11p21) has already been presented in [Dal Zilio et al., 2023], but suffices to say that it provides robust localization, navigation, flight control and allows us to command the drone and its camera. It is deployed using the GenoM specification language (which also maps in the same formal framework to validate and verify the functional components [Dal Zilio et al., 2023]).

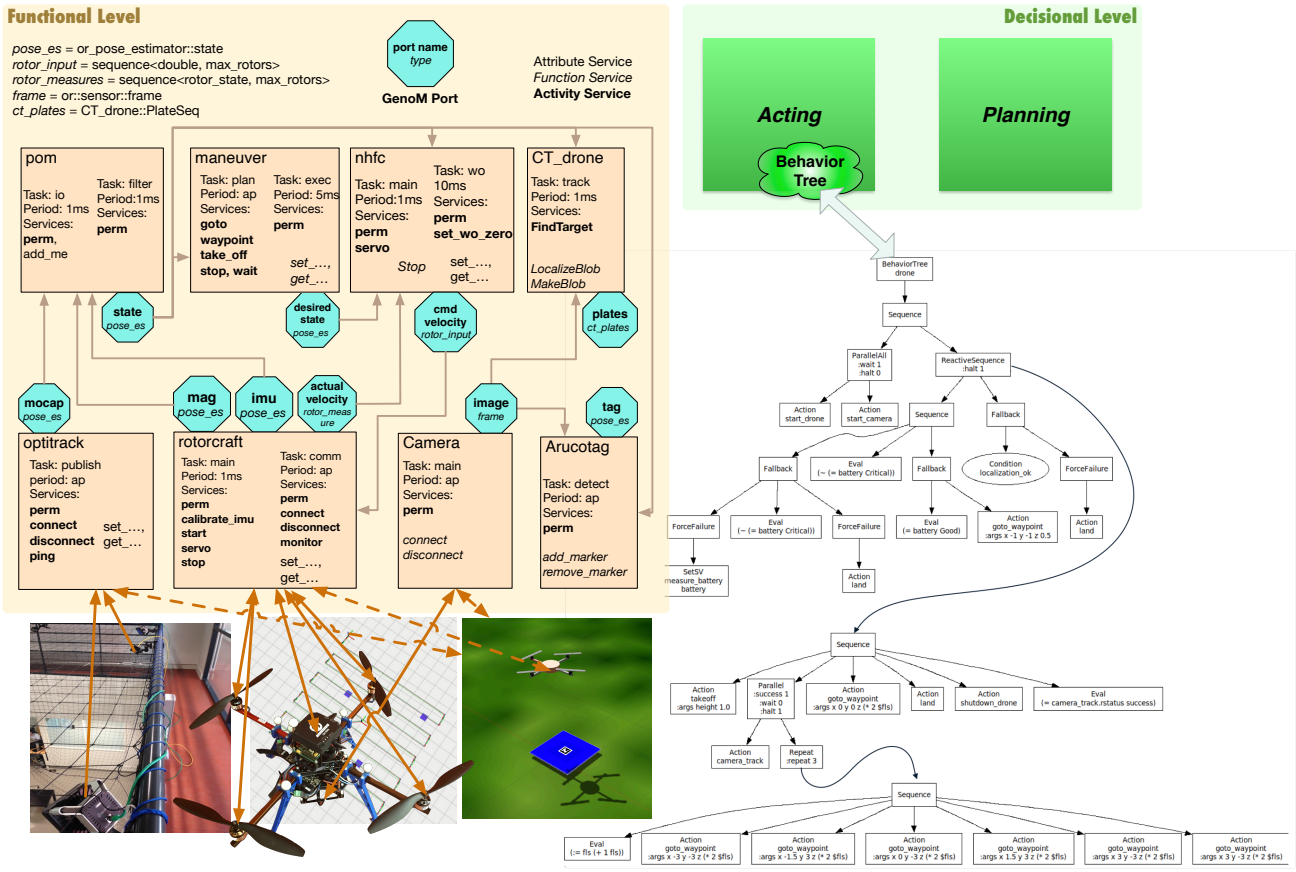

Figure 11: Architecture of the drone experiment. On the left, all the functional components involved, on the right a Behavior Tree in charge of the overall mission.

Eight functional components are deployed for this experiment. The set of primitive commands available for the *acting* component are: **start_drone**, **start_camera**, **shutdown_drone**, **takeoff**, **land**, **goto_waypoint**, and **camera_survey**. Each of them has a corresponding *Action* node in the larger BT **drone** on listing 5p21 (e.g, **takeoff** (line 33), **goto_waypoint** (lines 39-45), etc),

The state variables handled by the model are battery: *Good*, *Low* and *Critical* (line 6); and flight_level: a natural number between 0 and 3 (line 1).

Listing 5: A more complex drone survey BT in the `.btf` format, with *state variables* declaration, *SetSV* and *Eval* nodes. The := is the assignment operator, and ∼ is the logical negation (See also Figure 12p23).

```
1  ((defsv fls   ; the flight level of the drone
2    :init 0
3    :min 0
4    :max 3)
5
6  (defsv battery   ; the battery level of the drone
7    :states (Good Low Critical)    ;Self explanatory
8    :init Good
9    :transitions :all)
```

```
10
11  (BehaviorTree :name drone
12      (Sequence
13        (ParallelAll :wait 1 :halt 0
14          (Action :ID start_drone)
15          (Action :ID start_camera))
16        (ReactiveSequence :halt 1
17          (Sequence
18            (Fallback
19              (ForceFailure :ID fail
20                (SetSV :ID measure_battery :sv battery))
21              (Eval (~ (= battery critical)))
22              (ForceFailure :ID fail_mission
23                (Action :ID land)))
24            (Eval (~ (= battery critical)))
25            (Fallback
26              (Eval (= battery good))
27              (Action :ID goto_waypoint :args (x -1 y -1 z 0.5)))) ; goto charging station
28          (Fallback
29            (Condition :ID localization_ok)
30            (ForceFailure :ID fail
31              (Action :ID land)))
32          (Sequence
33            (Action :ID takeoff :args (height 1.0 duration 0))
34            (Parallel :success 1 :wait 0 :halt 1 ; if the tracking or the nav success... we are done.
35              (Action :ID camera_tracking :name camera_track)
36              (Repeat :repeat 3
37                (Sequence
38                  (Eval (:= fls (+ 1 fls)))
39                  (Action :ID goto_waypoint :args (x -3 y -3 z (* 2 $fls)))
40                  (Action :ID goto_waypoint :args (x -1.5 y 3 z (* 2 $fls)))
41                  (Action :ID goto_waypoint :args (x 0 y -3 z (* 2 $fls)))
42                  (Action :ID goto_waypoint :args (x 1.5 y 3 z (* 2 $fls)))
43                  (Action :ID goto_waypoint :args (x 3 y -3 z (* 2 $fls)))
44                  (Action :ID goto_waypoint :args (x 3 y -3 z (* 2 $fls))))))
45            (Action :ID goto_waypoint :args (x 0 y 0 z 5))
46            (Action :ID land)
47            (Action :ID shutdown_drone)
48            (Eval (= camera_track.rstatus success)))))))
```

Calling BT2Fiacre results in a Fiacre model with two processes for the state variables, and thirty eight processes for the BT nodes.[13]

6.1.1 Offline verification results

To compute the reachable states set of the drone BT (Listing 5p21) we call the `sift` tool:

```
sift -stats drone.tts drone.tts/drone.ktz
49 196 302 classe(s), 49 145 735 marking(s), 152 domain(s), 267 702 880 transition(s)
4552.704s
```

which takes around 1 hour and 16 minutes to build on an Intel(R) Xeon(R) Silver 4110 CPU @ 2.10 GHz with 256 GB of memory. The resulting `.ktz` has 49196302 classes, 49145735 markings, 152 domains, 267702880 transitions. In this context, a marking is a particular set of states and values for all the processes and variables in the system. A class is a state extended with timing information on the enabled transitions (therefore we can have several classes with the same marking).

All the default properties presented in Section 5.1 have been checked and all the results are the expected ones. Most of them take between less than a second and 300 seconds to

[13] The resulting code can be found in the `examples` subdirectory of https://redmine.laas.fr/projects/bt2fiacre/repository.

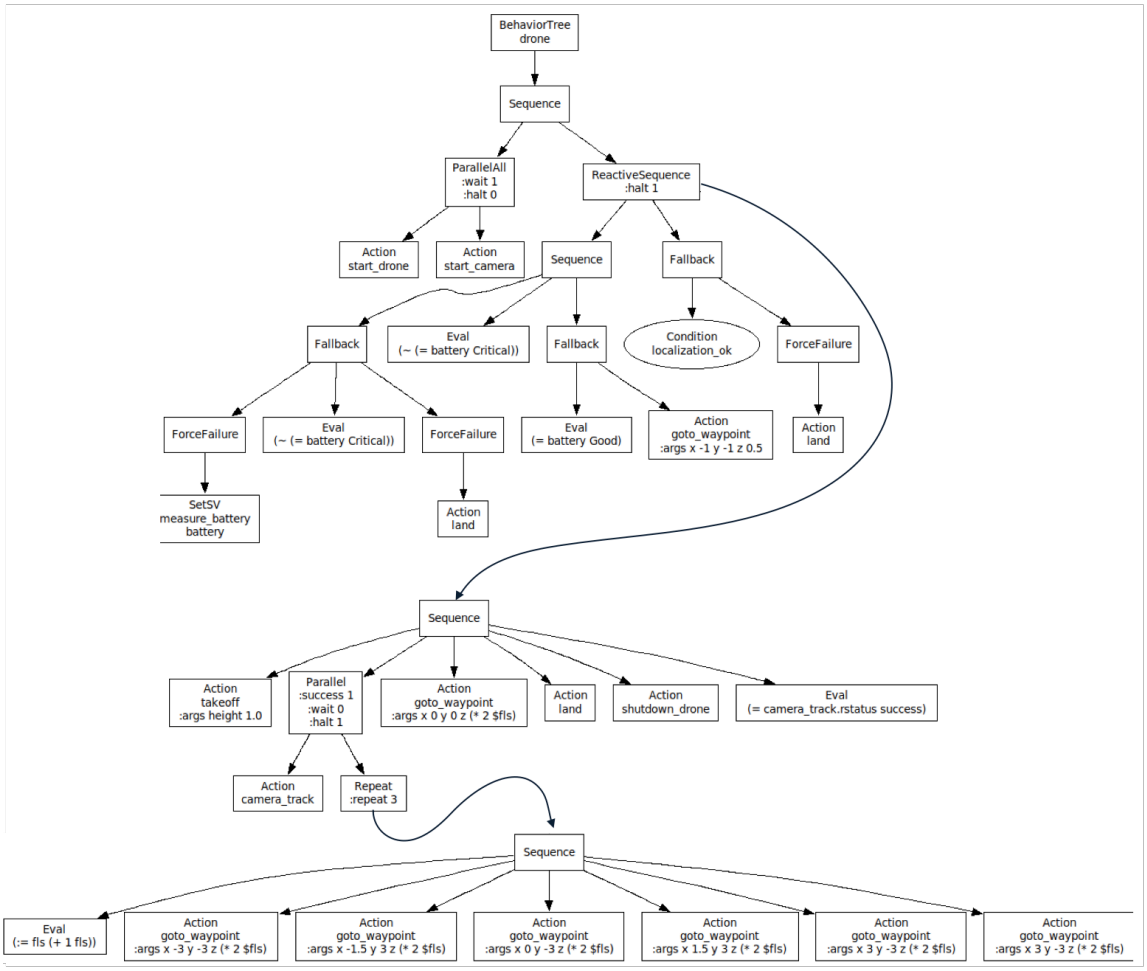

Figure 12: The graphical representation of the BT Listing 5p21.

be verified. As expected, the ones for which the model checker finds a counter example are usually quick to be verified while the ones which require to explore the whole set take more time.

Among the default proven properties, some are still interesting to comment on. For example, despite the *ParallelAll* on the *start_drone* and the *start_camera*, they cannot be halted (because the *ParallelAll* has `:halt 0`). Unlike the BT children of the *ReactiveSequence* which has `:halt 1`, they can all be halted (except the first branch)[14].

But on the top of the default properties, we can add new ones specific to this particular experiment. Can we prove that if the battery is *Critical* then the drone will attempt to land, and if it is *Low* it will return to the charging station? Can we show that if the camera survey fails then the whole drone mission will fail? Can the drone fly higher than 6 meters?

These properties can easily be written in FIACRE pointing to states of BT FIACRE nodes (e.g., *tick_node*, *failure*), values of state variable (e.g. fls, battery), or even returned status

[14]As only a branch returning `running` on the previous tick may be halted if one branch on its left fails, hence, never the first branch.

from executed BT (e.g., `btnode[localization_ok_btn18].rstatus = failure`)[15]:

```
property attempt_to_land_if_battery_Critical is
ltl (□ ((drone/5/value (battery=Critical)) and (drone/5/state done)) ⇒
     ♢(drone/8/state tick_node)) // land_btn12

property attempt_to_go_to_charging_station_if_battery_Low is
ltl (□ ((drone/12/value (battery=Low)) and (drone/5/state done)) ⇒
     ♢(drone/13/state tick_node)) // goto_waypoint_btn16

property attempt_to_land_if_localization_failure is
ltl (□ ((drone/16/value (btnode[localization_ok_btn18].rstatus = failure) and (drone/16/state done))) ⇒
     ♢(drone/17/state tick_node)) // land_btn20

property camera_fails_implies_mission_fails is
ltl (□ (drone/21/state failure) ⇒ // camera_tracking_camera_track
     ♢(drone/39/state failure))    // BehaviorTree1_drone

property fly_not_higher_than_6m is absent (drone/35/value (fls > 3))
```

The result are proven true for all these properties:

```
operator attempt_to_land_if_battery_Critical : prop
TRUE
0.001s
operator attempt_to_go_to_charging_station_if_battery_Low : prop
TRUE
0.001s
operator attempt_to_land_if_localization_failure : prop
TRUE
0.001s
operator camera_fails_implies_mission_fails : prop
TRUE
0.001s
operator fly_not_higher_than_6m : prop
TRUE
242.462s
```

More interestingly, we can also take advantage of FIACRE/TINA timed models. For example, if one considers the one tick per node semantics (See Section 4.8.2), we can prove that the a critical battery will always leads to a landing within a $[0, 2]$ ticks interval (everything else considered in the BT):

```
property attempt_to_land_if_battery_Critical_timed is
((drone/5/value (battery=Critical))and (drone/5/state done)) leadsto (drone/8/state tick_node) within [0,2]
```

which results in:

```
operator attempt_to_land_if_battery_Critical_timed : prop
TRUE
49.170s
```

6.1.2 Runtime verification results

The **Drone** BT execution by HIPPO runs as expected, the survey mission is executed, and by adding random fault (on battery level, or failing the *localization_ok Condition* node), we show that the drone behaves as expected (perform land to prevent further problems). If the survey finds the object, it returns `success` and the navigation is halted (because the *Parallel* is `:success 1 :wait 0 :halt 1`. Note that at the end, we check with an *Eval* mode, that the returned status of the `camera_track` *Action* node is `success`. So when the overall *Drone* BT completes, it will return `success` if and only if the target was found, `failure` otherwise.

[15] The $\Box, \Rightarrow, \Diamond$ operators have the usual LTL semantics.

Again, the fact that the BT formal model execution exhibit what is expected from the original programmer is a good sign that the formal semantics is consistent with the operational one, and that the offline proofs we did on the very same model hold for the original BT.[16]

From a performance point of view, the overhead of the HIPPO execution is negligible. After all, the HIPPO engine is a Petri Net (TTS) "execution" engine, and has 497 places and 1113 transitions and manages up to 7 task execution threads (one for each *Action* BT node).

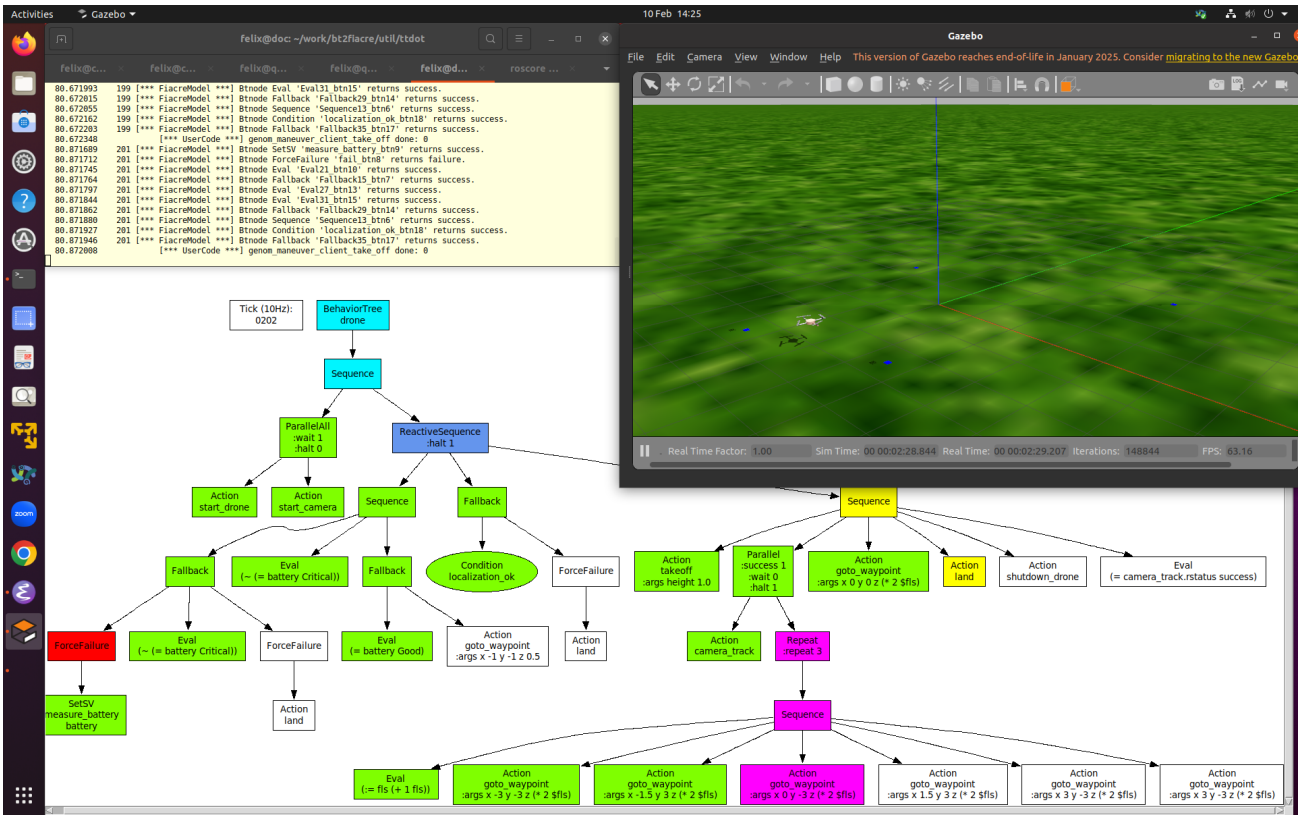

Figure 13: The screen dump of the drone BT mission 5p21 executing in HIPPO. This shows that the top *BehaviorTree* and its child *Sequence* child have passed the tick to the *ReactiveSequence* currently executing while its *Sequence* has returned `running` from its *Land* child, while the *camera_track* has returned `success`, as a result, the *Parallel* has halted the *Repeat* and its children. Note the tick counter near the root which indicates the mission is in its 20th second of execution.

6.2 ROS2 Nav2 BT

One of the most popular navigation stacks in robotics is the Nav2 one. It is now part of ROS2 and is proposed with a number of BT to implement it [Macenski et al., 2024]. So to also properly test our approach we implemented the Nav2 BT in our framework. Note

[16] Some videos of these runs and the BT animation are available here: https://redmine.laas.fr/projects/bt2fiacre/pages/index#yet-another-more-complex-example-drone3.

that these BT define some new *Decorator* and *Control* nodes: *Recovery*, *PipelineSequence*, *RateController*, *RoundRobin*, etc. They were implemented in our BT2Fiacre tool and are being translated to Fiacre along the regular ones. One interesting aspect of our approach is that the operational semantics of these new nodes can be proven in our formal semantic of Fiacre using the Tina toolbox. For example, one can formally prove that our implementation of *Recovery* is correct by showing that the second node can only be called when the first one has returned `failure`, and that the first one can only be called again when the second has returned `success`, etc.

We are able to check the reachable state of the Nav2 BT presented in Appendix, Listing 11p41 and Figure 17p41.

```
sift -stats ros2-nav2.tts ros2-nav2.tts/ros2-nav2.ktz
171 656 098 classe(s), 171 656 098 marking(s), 66 domain(s), 422 970 818 transition(s)
5776.872s

selt ros2-nav2.tts/ros2-nav2.ktz ros2-nav2.tts/ros2-nav2.ltl -b
Selt version 3.8.0 -- 05/02/24 -- LAAS/CNRS
ktz loaded, 171 656 098 states, 422 970 818 transitions
2212.781s
```

Building the set of reachable states of this BT takes 1 h 46 min on the same CPU as the one used in Section 6.1.1 and results in reachable state sets larger than the one we had with the drone experiment. Similarly, all the default formal properties were checked (taking between 0 and 85 seconds for each property) without any unexpected results.

More interestingly is to check that our implementation of the added BT (e.g., *Recovery*, *RoundRobin*, etc) is correct.

Recovery The *Recovery* node[17] is a control flow node with two children. It returns `success` if and only if the first child returns `success`. The second child will be executed only if the first child returns `failure`. If the second child succeeds, then the first child will be executed again. The user can specify how many `:retry` times the recovery actions should be taken before returning `failure`.

We define a simple generic example:

Listing 6: A simple *Recovery* node.

```
((BehaviorTree :name bt_recovery
    (Recovery :num_retries 1 :name recovery
        (Action :name action)
        (Action :name recov))))
```

for which the model checker selt is able to prove the following properties:

```
// Action_action  // 1
// Action_recov   // 2
// Recovery_recovery  // 3
// BehaviorTree_bt_recovery // 4

property failure_recov_leads_to_failure is // TRUE if the recovery action fails, than the recovery fails
(bt_recovery/2/state failure) leadsto (bt_recovery/3/state failure) within [0,0]

property failure_action_leads_to_recovery_failure is // FALSE because retry is still 0 (not retried yet)
(bt_recovery/1/state failure and bt_recovery/3/value (retry = 0)) leadsto (bt_recovery/3/state failure) within
    [0,0]

property failure_action_leads_to_recovery_failure is // TRUE because we already retied
```

[17] https://docs.nav2.org/behavior_trees/overview/nav2_specific_nodes.html

```
(bt_recovery/1/state failure and bt_recovery/3/value (retry = 1)) leadsto (bt_recovery/3/state failure) within
     [0,0]

property success_action_leads_to_success is // TRUE
(bt_recovery/1/state success) leadsto (bt_recovery/3/state success) within [0,0]
```

We present here a proof on a simple instance, but similar proofs can be made on an instance of *Recovery* embedded in a larger BT, for example the ones in the Nav2 BT in Appendix, Listing 11p41 and Figure 17p41.

RoundRobin The *RoundRobin* node[18] is a control node which ticks its children in a round robin fashion until a child returns `success`, in which case the parent node will also return `success`. If all children return `failure` so will the parent *RoundRobin*.

Similarly, we define a simple generic example:

Listing 7: A simple *RoundRobin* node.

```
((BehaviorTree :name bt_roundrobin
   (KeepRunningUntilFailure :name kr
     (RoundRobin :name RR
        (Action :name A1)
        (Action :name A2)
        (Action :name A3)
        (Action :name A4)))))
```

for which the model checker is able to prove the following properties:

```
// Action_A1 // 1
// Action_A2 // 2
// Action_A3 // 3
// Action_A4 // 4
// RoundRobin_RR // 5

property failure_a1_leads_to_failure is // FALSE a failure of one child does not necessary leads to rr failure
(bt_roundrobin/1/state failure) leadsto (bt_roundrobin/5/state failure) within [0,1]

property failure_a4_leads_to_a1_ticked is // TRUE a failure of a child with less than 3 failure so far leads
     to the next child to be ticked
((bt_roundrobin/4/state failure ) and (bt_roundrobin/5/value (failed < 3))) leadsto (bt_roundrobin/1/state
     tick_node) within [0,0]

property success_a2_leads_a3_ticked_exp_true is // TRUE success of a child leads to the next one to be ticked
     upon rr reticked
(bt_roundrobin/2/state success) leadsto (bt_roundrobin/3/state tick_node) within [0,3] // 3 because the tick
     has to go all the way up

property success_a2_leads_rr_success_exp_true is // TRUE
(bt_roundrobin/2/state success) leadsto (bt_roundrobin/5/state success) within [0,0]
```

Similar proofs can be made with any BT nodes to show that our FIACRE implementation satisfies the expected formal properties defining the BT node considered.

6.2.1 Runtime

All the Nav2 BT have been translated to `.btf` and run with Hippo and simulated ROS2 actions. One of the features we added and we believe goes beyond the "nice graphical touch" aspect, is to dynamically color the BT nodes while they execute. Of course, there are text traces of the Hippo engine running the BT, but being able to follow the execution ticks as well as the last returned status for each BT is rather informative. Some videos of these runs and the BT

[18] https://docs.nav2.org/behavior_trees/overview/nav2_specific_nodes.html

animation are available here: https://redmine.laas.fr/projects/bt2fiacre/pages/index#playing-with-ros2-nav2-bt. Figure 17p41 shows the Nav2 BT (See Listing 11p41) executed by Hippo.

7 Limits, discussion, future work, and conclusion

Before concluding, we propose to examine the limits of our work, to discuss its pros and cons and consider future work.

7.1 Limits

Some limits of our work are mostly due to choices currently made, which could be reconsidered, if needed. For example, we do not implement a black board to handle variables. Instead we propose to handle BT variables in Fiacre directly. This has the advantage of being able to include these variables in the formal model, hence in the proof and the runtime verification (e.g. the fls flight level in the drone experiment). The disadvantage is that only Fiacre supported types can be used.

An area where our approach would suffer, is when BT are dynamically modified, or transformed. For example in Section 2.1 we consider BT used for planning. In these approaches, BT are being modified on the fly, clearly, we could not perform model checking on the fly, yet assuming the dynamic of the planner remains slow (below one second) we could still synthesize and compile the model (both are almost instantaneous), and jump in its execution on the fly.

Although all our examples are based on one BT, absolutely nothing prevents us from having multiple BT executing together. Yet from a verification point of view, this would probably lead to an intractable model as models running in parallel tend to multiply the size of their reachable states set, to account for all the possible transition interleavings.

This brings us to the main limiting factor of our approach. Offline verification with model checking may lead to large intractable reachable states set. This is a well known limit of these approaches and we do not have any magic bullet. It is hard to predict what will be the size of the reachable states set for a given BT. Some very large BT may still produce a rather small reachable states set, while a simple BT (e.g., with a lot of parallelism, loops, etc) are untractable. Yet we have seen in our examples that we can verify reasonably complex robotics BT skills, that we can prove "individual" BT node behavior (e.g. the new control nodes added in Nav2), and that we can also handle complex properties on missions written with BT. For the models too complex to be verified offline, we can still use the complete online version with some added monitors, or work on an abstracted model for offline verification (but we would then loose the offline/online equivalence).

7.2 Discussion

Our approach to improve the trust one can put in BT completely relies on a on a well established formal language and framework (Fiacre and TTS), a formal verification toolbox (Tina to check LTL, CTL, properties, patterns [Abid et al., 2014][19]) and Hippo, a runtime engine able to execute the formal model. We have seen in Section 2.2 that there are other

[19]Patterns allow the verification of properties with explicit time (e.g. to prove that at most x units of time will elapse between two states.).

approaches which transform BT in a formal framework, or harness a formal model around BT, but to our knowledge, none of them support the *automatic* translation of BT to an equivalent formal model *and* the execution of the same formal model in place of the original BT. This is a very strong argument as the same formal model, which clarifies and specifies the formal semantics of BT, can be used to prove properties and also execute and show, while running, that it properly implements the expected operational semantic, moreover knowing that the proof made offline also holds online.

Despite the formal and proven tools deployed, our translation tool BT2Fiacre may still contain errors and bugs, but again, the fact that we can use theses tools to prove the resulting translated model against specifications done on the BT is reassuring. One could take every single BT node type, write its formal specification and prove that the Fiacre translation satisfies them with Tina (as we did for the ones added in the Nav2 experiment).

Furthermore, Fiacre offers some features which could be valuable to be added to BT. Explicit time representation is the foremost feature which could come handy if added to the BT. Indeed, robot planning, acting and control is usually handling explicit time representation. Control loops have frequency, plans have duration and deadline, actions take some time to execute, etc. So it would be perfectly logical to add explicit time representation to BT. As a consequence, this would also have an impact on the tick semantic which would probably become more an execution token than a time elapsing counter.

Another domain where our approach and Fiacre could offer some valuable addition is with environment modelling. BT are intrinsically embedded operational models, they really exist to be executed in the real world (or in simulation). But now that we can consider model checking them, one needs to take into account (when possible) the outcomes of actions or conditions in the environment. Without any particular information, the model checker consider all possibilities (`success`, `failure` or `running`), but the real world often prove to be more "constrained", and one could consider a more accurate and explicit model of the environment (e.g., with external asynchronous events, state variable values changes, etc).

Parallelism is clearly allowed and properly modelled in our framework. Moreover the model checker can verify properties even if it needs to consider all the possible execution interleavings of parallel nodes (but at some cost). Meanwhile, the semantic of transition in Fiacre is such that resource checking and reservation on a transition are atomic. As a consequence, resources management, even considering parallelism comes for free with Fiacre.

Last, our approach has been implemented in a software suite [Ingrand, 2024b] which has been applied to many BT examples (available here: `https://redmine.laas.fr/projects/bt2fiacre/pages/index`). Moreover, all the examples presented here are available in the `examples` directory and we invite other formal approach to test their systems on these examples and report their results for comparison.

7.3 Future work

Considering the tight links between the Fiacre framework and our approach/tool BT2Fiacre, it is clear that any improvement in the former may also improve the latter. The Fiacre developers are considering extending the data types handled in the language by adding rational numbers and strings. Both types would still allow model checking and enable the deployment of richer state variables in our `.btf` format.

Similarly, there are many existing features in Fiacre which could be used in `.btf` BT (some have already been used in ProSkill [Ingrand, 2024a], another robotics acting language

mapping to FIACRE). As already mentioned in Section 7.2 above, time would be a valuable addition to model, with time interval $[min, max]$, how long an *Action* is expected to take, or to wait a given amount of time before returning from a *Wait* node, etc. Not only would this enrich the `.btf` format and language, but it would also be taken into account by the TINA model checking tools, and enforced by the HIPPO engine.

External asynchronous events and state variables asynchronous value changes are features also available in FIACRE and its tools. This again could enrich the `.btf` format and allow the programmer to account for "uncontrollable" state transitions.

7.4 Conclusion

BT are more and more popular in robotics. To encourage their deployment and improve the trust one has in robotic applications using BT, we propose an approach and an automatic tool to transform any BT in a formal model with a formal semantics. The resulting models can then be used offline with model checking to prove some properties of the BT, but also linked to the real actions and perceptions of the robots and executed online on the robot. Of course, this can be deployed in BT applications outside of robotics, and also participates to define extensions to BT and to better formalize them.

Acknowledgement

We thank Bernard Berthomieu, Silvano Dal Zilio and Pierre-Emmanuel Hladik for their help while developing and deploying the work presented here.

References

N. Abid, S. Dal Zilio, and D. Le Botlan. A formal framework to specify and verify real-time properties on critical systems. *International Journal of Critical Computer-Based Systems*, 5(1-2):4–30, 2014. doi:10.1504/IJCCBS.2014.059593. Cited on page 28.

S. Bensalem, L. de Silva, F. Ingrand, and R. Yan. A Verifiable and Correct-by-Construction Controller for Robot Functional Levels. *Journal of Software Engineering for Robotics*, 1 (2):1–19, Sept. 2011. URL `http://arxiv.org/abs/0908.0221v1`. Cited on page 5.

B. Berthomieu and M. Diaz. Modeling and Verification of Time-Dependent Systems Using Time Petri Nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, Mar. 1991. doi:10.1109/32.75415. Cited on page 6.

B. Berthomieu, J.-P. Bodeveix, M. Filali, H. Garavel, F. Lang, D. Le Botlan, F. Vernadat, and S. Dal Zilio. The syntax and semantics of fiacre. Technical Report 7264, LAAS, 2007. URL `https://projects.laas.fr/fiacre/doc/fiacre.pdf`. Cited on page 5.

B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *Embedded Real-Time Software and Systems*, Toulouse, 2008. URL `https://hal.laas.fr/inria-00262442`. Cited on page 5.

O. Biggar and M. Zamani. A framework for formal verification of behavior trees with linear temporal logic. *IEEE Robotics and Automation Letters*, 5(2):2341–2348, 2020. doi:10.1109/LRA.2020.2970634. Cited on pages 4, 19, and 42.

S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods*, pages 128–147, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24756-2. doi:10.1007/978-3-540-2. Cited on page 19.

E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer, Berlin, Heidelberg, 2012. ISBN 978-3-642-35746-6. doi:10.1007/978-3-642-35746-6_1. Cited on page 9.

M. Colledanchise and L. Natale. Improving the Parallel Execution of Behavior Trees. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sept. 2018. doi:10.1109/IROS.2018.8593504. Cited on page 3.

M. Colledanchise and L. Natale. On the implementation of behavior trees in robotics. *IEEE Robotics and Automation Letters*, 6(3):5929–5936, 2021. doi:10.1109/LRA.2021.3087442. Cited on page 3.

M. Colledanchise and P. Ögren. *Behavior Trees in Robotics and AI*. CRC Press, jul 2018. doi:10.1201/9780429489105. Cited on pages 2 and 3.

M. Colledanchise, R. M. Murray, and P. Ögren. Synthesis of correct-by-construction behavior trees. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6039–6046, 2017. doi:10.1109/IROS.2017.8206502. Cited on page 4.

M. Colledanchise, D. Almeida, and P. Ögren. Towards blended reactive planning and acting using behavior trees. In *International Conference on Robotics and Automation (ICRA)*, pages 8839–8845. IEEE, 2019. doi:10.1109/ICRA.2019.8794128. Cited on page 4.

M. Colledanchise, G. Cicala, D. E. Domenichelli, L. Natale, and A. Tacchella. Formalizing the execution context of behavior trees for runtime verification of deliberative policies. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9841–9848, 2021. doi:10.1109/IROS51168.2021.9636129. Cited on page 4.

S. Dal Zilio, P.-E. Hladik, F. Ingrand, and A. Mallet. A formal toolchain for offline and run-time verification of robotic systems. *Robotics and Autonomous Systems*, 159:104301, 2023. doi:10.1016/j.robot.2022.104301. Cited on pages 6 and 20.

E. Ghiorzi and A. Tacchella. Execution semantics of behavior trees in robotic applications. *arXiv preprint arXiv:2408.00090*, 2024. doi:10.48550/arXiv.2408.00090. Cited on page 17.

S. Gugliermo, D. Cáceres Domínguez, M. Iannotta, T. Stoyanov, and E. Schaffernicht. Evaluating behavior trees. *Robotics and Autonomous Systems*, 178:104714, 2024. doi:10.1016/j.robot.2024.104714. Cited on page 3.

P.-E. Hladik, F. Ingrand, S. Dal Zilio, and R. Tekin. Hippo: A formal-model execution engine to control and verify critical real-time systems. *Journal of Systems and Software*, 181:111033, 2021. doi:10.1016/j.jss.2021.111033. Cited on pages 6 and 9.

F. Ingrand. Proskill: A formal skill language for acting in robotics. Technical report, arXiv, 2024a. URL https://arxiv.org/abs/2403.07770. Cited on pages 6 and 29.

F. Ingrand. BT2Fiacre (Behavior Tree 2 Fiacre), Oct. 2024b. URL https://laas.hal.science/hal-04720141. Cited on page 29.

F. Ingrand and M. Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44, June 2017. doi:10.1016/j.artint.2014.11.003. Cited on page 4.

M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith. A survey of behavior trees in robotics and ai. *Robotics and Autonomous Systems*, 154:104096, 2022. ISSN 0921-8890. doi:10.1016/j.robot.2022.104096. Cited on page 3.

A. Klöckner. Interfacing behavior trees with the world using description logic. In *AIAA Guidance, Navigation, and Control (GNC) Conference*, 2013. doi:10.2514/6.2013-4636. Cited on page 4.

U. Köckemann, D. Calisi, G. Gemignani, J. Renoux, and A. Saffiotti. Planning for automated testing of implicit constraints in behavior trees. In *International Conference on Automated Planning and Scheduling*, ICAPS '23. AAAI Press, 2023. doi:10.1609/icaps.v33i1.27247. Cited on page 4.

S. Macenski, R. White, and J. Wallace. Nav2 behavior trees, 2024. URL https://docs.nav2.org/behavior_trees/index.html. Cited on pages 1, 2, 5, 25, and 40.

F. Martín, J. G. Clavero, V. Matellán, and F. J. Rodríguez. Plansys2: A planning system framework for ros2. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9742–9749. IEEE, 2021. doi:10.1109/IROS51168.2021.9636544. Cited on page 4.

A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren. Towards a unified behavior trees framework for robot control. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5420–5427. IEEE, 2014. doi:10.1109/ICRA.2014.6907656. Cited on page 3.

A. Micheli, A. Bit-Monnot, G. Röger, E. Scala, A. Valentini, L. Framba, A. Rovetta, A. Trapasso, L. Bonassi, A. E. Gerevini, L. Iocchi, F. Ingrand, U. Köckemann, F. Patrizi, A. Saetti, I. Serina, and S. Stock. Unified planning: Modeling, manipulating and solving ai planning problems in python. *SoftwareX*, 29:102012, 2025. doi:10.1016/j.softx.2024.102012. Cited on page 4.

P. Ögren and C. I. Sprague. Behavior trees in robot control systems. *Annual Review of Control, Robotics, and Autonomous Systems*, 5(Volume 5, 2022):81–107, 2022. doi:10.1146/annurev-control-042920-095314. Cited on page 3.

M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*. Kobe, Japan, 2009. URL http://robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf. Cited on page 20.

A. Schulz-Rosengarten, A. Ahmad, M. Clement, R. von Hanxleden, B. Asch, M. Lohstroh, E. A. Lee, G. Quiros, and A. Shukla. Behavior trees with dataflow: Coordinating reactive tasks in lingua franca. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, pages 304–305, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3639478.3643093. Cited on page 3.

S. S. Serbinowska, N. Potteiger, A. M. Tumlin, and T. T. Johnson. Verification of behavior trees with contingency monitors. *Electronic Proceedings in Theoretical Computer Science*, 411:56–72, Nov. 2024. doi:10.4204/eptcs.411.4. Cited on page 4.

C. Street, Y. Warsame, M. Mansouri, M. Klauck, C. Henkel, M. Lampacrescia, M. Palmas, R. Lange, E. Ghiorzi, A. Tacchella, R. Azrou, R. Lallement, M. Morelli, G. I. Chen, D. Wallis, S. Bernagozzi, S. Rosa, M. Randazzo, S. Faraci, and L. Natale. Towards a verifiable toolchain for robotics. In *AAAI Fall Symposium Series*, AAAI Symposium Series (Fall). AAAI Press, Aug. 2024. doi:10.1609/aaaiss.v4i1.31823. Cited on page 1.

Q. Wang, H. Dai, Y. Zhao, M. Zhang, and S. Bliudze. Enabling behaviour tree verification via a translation to bip. In D. Marmsoler and M. Sun, editors, *Formal Aspects of Component Software*, pages 3–20, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-71261-6_1. Cited on pages 4, 19, 20, and 42.

A A BT *Action* H-Fiacre process

Listing 8: The H-Fiacre process specification of the *takeoff* BT *Action* node.

```
process btnode_takeoff_btn21 (&btnode: btnode_array, &fls: sv_fls, &battery: sv_battery) is

states start_, tick_node, success, failure, halt, halted, running, error,
   Action_takeoff, dispatch, Action_takeoff_sync, done

var callb: bool,
   report_halted:bool := false, ret_val: ret_status

from start_
   wait [0,0];
   on (btnode[takeoff_btn21].caller <> None); // Wait until we are called
   report_halted := false;
   if (btnode[takeoff_btn21].rstatus = halt_me) then // are we being instructed to halt?
       report_halted := true;
       to halt
   end;
   // Btnode Action 'takeoff_btn21' has been called (:height 1.000000 :duration 0 )
   btnode[takeoff_btn21].rstatus := no_ret_status; // just initializing the rstatus
to tick_node

from halt
   wait [0,0];
   // Btnode Action 'takeoff_btn21' is being halted (:height 1.000000 :duration 0 )
   callb := Fiacre_Action_takeoff_halt (btnode[takeoff_btn21]); //This call the external which
to halted // halts the action

from tick_node
   wait [0,0];
   to Action_takeoff

from Action_takeoff
   // synthesized action arg index 2
   btnode[takeoff_btn21].ArgIndex := 2;
   // Btnode Action 'takeoff_btn21' calling its action (start task)
   start Fiacre_Action_takeoff_task (btnode[takeoff_btn21]); // this call the Fiacre task
to Action_takeoff_sync // which handles this action.

from Action_takeoff_sync
   sync Fiacre_Action_takeoff_task ret_val; // wait until the Fiacre task returns
   // Btnode Action 'takeoff_btn21' returned (sync task)
   to dispatch

from dispatch
   wait [0,0]; // we dispatch to the proper state according to the return values
   if (ret_val = success) then
       to success
   elsif (ret_val = failure) then
       to failure
   elsif (ret_val = running) then
       to running
   else
       to error // a priori unreachable
   end

from success
   wait [0,0];
   // Btnode Action 'takeoff_btn21' (:height 1.000000 :duration 0 ) returns success.
   btnode[takeoff_btn21].rstatus := success;
   to done

from failure
   wait [0,0];
```

```
63      if (report_halted) then // this is mostly for traces
64          // Btnode Action 'takeoff_btn21' (:height 1.000000 :duration 0 ) returns halted failure.
65          null
66      else
67          // Btnode Action 'takeoff_btn21' (:height 1.000000 :duration 0 ) returns failure.
68          null
69      end;
70      btnode[takeoff_btn21].rstatus := failure;
71      to done
72
73  from halted
74      wait [0,0];
75      // Btnode Action 'takeoff_btn21' (:height 1.000000 :duration 0 ) has been halted.
76      report_halted := true;
77      to failure
78
79  from running
80      wait [0,0];
81      // Btnode Action 'takeoff_btn21' (:height 1.000000 :duration 0 ) returns running.
82      btnode[takeoff_btn21].rstatus := running;
83      to done
84
85  from done
86      wait [0,0];
87      // Btnode Action 'takeoff_btn21' returning control to caller and back to 'start_'
88      btnode[takeoff_btn21].caller := None; // we relinquish the tick and go back waiting
89  to start_
```

B A BT *Sequence* H-FIACRE process

Listing 9: The H-FIACRE process specification of a BT *Sequence* node.

```
1   process btnode_Sequence13_btn6 (&btnode: btnode_array) is
2
3   states start_, tick_node, success, failure, halt, halted, halt_wait, running, error,
4       Fallback15_btn7, Fallback15_btn7_done, Eval25_btn12, Eval25_btn12_done,
5       Fallback27_btn13, Fallback27_btn13_done, done
6
7   var next_seq: 1..3 := 1
8
9   from start_
10      wait [0,0];
11      on (btnode[Sequence13_btn6].caller <> None);
12      if (btnode[Sequence13_btn6].rstatus = halt_me) then to halt end;
13      // Btnode Sequence 'Sequence13_btn6' has been called
14      btnode[Sequence13_btn6].rstatus := no_ret_status;
15  to tick_node
16
17  from halt
18      wait [0,0];
19      // Btnode Sequence 'Sequence13_btn6' is being halted
20      if (btnode[Fallback15_btn7].rstatus = running) then
21          // Btnode Sequence 'Sequence13_btn6' halting Fallback 'Fallback15_btn7'
22          btnode[Fallback15_btn7].rstatus := halt_me;
23          btnode[Fallback15_btn7].caller := caller_Sequence13_btn6;
24          to halt_wait
25      end;
26      if (btnode[Eval25_btn12].rstatus = running) then
27          // Btnode Sequence 'Sequence13_btn6' halting Eval 'Eval25_btn12'
28          btnode[Eval25_btn12].rstatus := halt_me;
29          btnode[Eval25_btn12].caller := caller_Sequence13_btn6;
30          to halt_wait
31      end;
32      if (btnode[Fallback27_btn13].rstatus = running) then
```

```
        // Btnode Sequence 'Sequence13_btn6' halting Fallback 'Fallback27_btn13'
        btnode[Fallback27_btn13].rstatus := halt_me;
        btnode[Fallback27_btn13].caller := caller_Sequence13_btn6;
        to halt_wait
    end;
    to halted

from halt_wait
    on ((btnode[Fallback15_btn7].caller = None) and
        (btnode[Eval25_btn12].caller = None) and
        (btnode[Fallback27_btn13].caller = None));
    to halted

from tick_node
    wait [0,0];
    if (next_seq = 1) then to Fallback15_btn7 end;
    if (next_seq = 2) then to Eval25_btn12 end;
    if (next_seq = 3) then to Fallback27_btn13 end;
    to error

from Fallback15_btn7
    wait [0,0];
    // Btnode Sequence 'Sequence13_btn6' calling Fallback 'Fallback15_btn7'
    btnode[Fallback15_btn7].caller := caller_Sequence13_btn6;
    to Fallback15_btn7_done

from Fallback15_btn7_done
    wait [0,0];
    on (btnode[Fallback15_btn7].caller = None);
    // Btnode Sequence 'Sequence13_btn6' getting control back from Fallback 'Fallback15_btn7'
    if (btnode[Fallback15_btn7].rstatus = success) then
        to Eval25_btn12
    elsif (btnode[Fallback15_btn7].rstatus = failure) then
        next_seq := 1;
        to failure
    elsif (btnode[Fallback15_btn7].rstatus = running) then
        next_seq := 1;
        to running
    else
        to error
    end

from Eval25_btn12
    wait [0,0];
    // Btnode Sequence 'Sequence13_btn6' calling Eval 'Eval25_btn12'
    btnode[Eval25_btn12].caller := caller_Sequence13_btn6;
    to Eval25_btn12_done

from Eval25_btn12_done
    wait [0,0];
    on (btnode[Eval25_btn12].caller = None);
    // Btnode Sequence 'Sequence13_btn6' getting control back from Eval 'Eval25_btn12'
    if (btnode[Eval25_btn12].rstatus = success) then
        to Fallback27_btn13
    elsif (btnode[Eval25_btn12].rstatus = failure) then
        next_seq := 1;
        to failure
    elsif (btnode[Eval25_btn12].rstatus = running) then
        next_seq := 2;
        to running
    else
        to error
    end

from Fallback27_btn13
    wait [0,0];
```

```
    // Btnode Sequence 'Sequence13_btn6' calling Fallback 'Fallback27_btn13'
    btnode[Fallback27_btn13].caller := caller_Sequence13_btn6;
    to Fallback27_btn13_done

from Fallback27_btn13_done
    wait [0,0];
    on (btnode[Fallback27_btn13].caller = None);
    // Btnode Sequence 'Sequence13_btn6' getting control back from Fallback 'Fallback27_btn13'
    if (btnode[Fallback27_btn13].rstatus = success) then
        next_seq := 1;
        to success
    elsif (btnode[Fallback27_btn13].rstatus = failure) then
        next_seq := 1;
        to failure
    elsif (btnode[Fallback27_btn13].rstatus = running) then
        next_seq := 3;
        to running
    else
        to error
    end

from success
    wait [0,0];
    // Btnode Sequence 'Sequence13_btn6' returns success.
    btnode[Sequence13_btn6].rstatus := success;
    to done

from failure
    wait [0,0];
    // Btnode Sequence 'Sequence13_btn6' returns failure.
    btnode[Sequence13_btn6].rstatus := failure;
    to done

from halted
    wait [0,0];
    // Btnode Sequence 'Sequence13_btn6' has been halted.
    btnode[Sequence13_btn6].rstatus := failure;
    to done

from running
    wait [0,0];
    // Btnode Sequence 'Sequence13_btn6' returns running.
    btnode[Sequence13_btn6].rstatus := running;
    to done

from done
    wait [0,0];
    // Btnode Sequence 'Sequence13_btn6' returning control to caller and back to '_start'
    btnode[Sequence13_btn6].caller := None;
to start_
```

C *Fallback* and *Parallel* nodes transformation in FIACRE

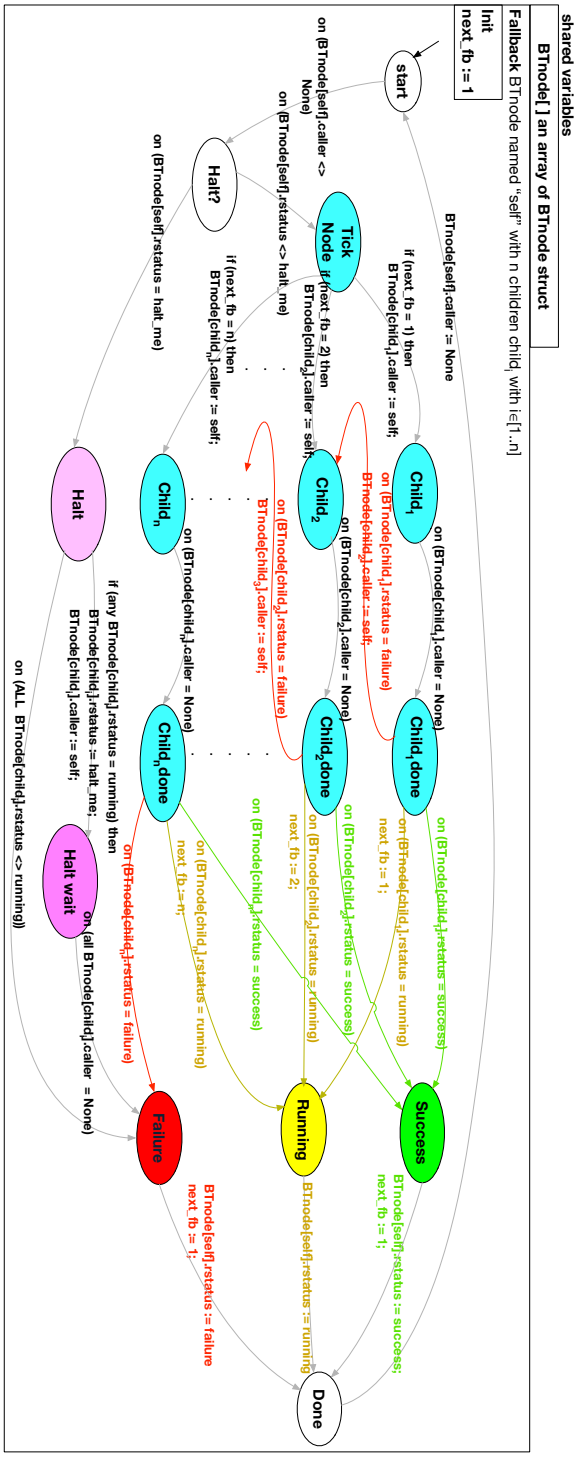

Figure 14: The FIACRE process modeling the *Fallback* node.

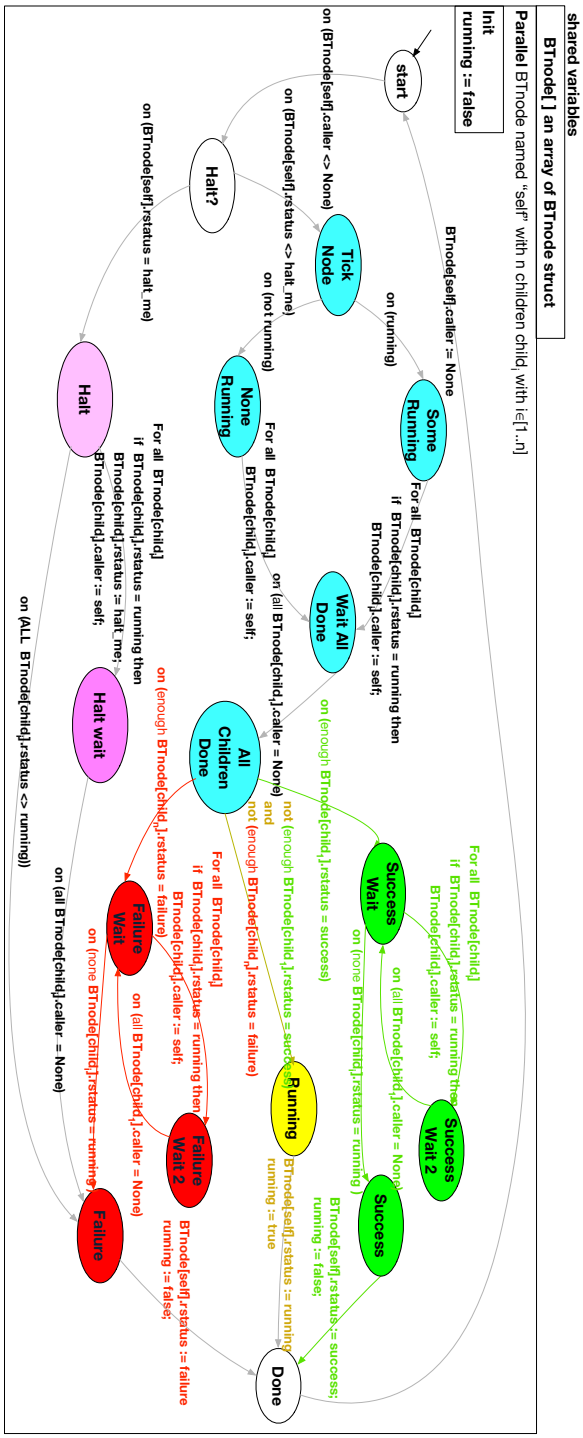

Figure 15: The FIACRE process modeling the *Parallel* node.

D The Drone BT.

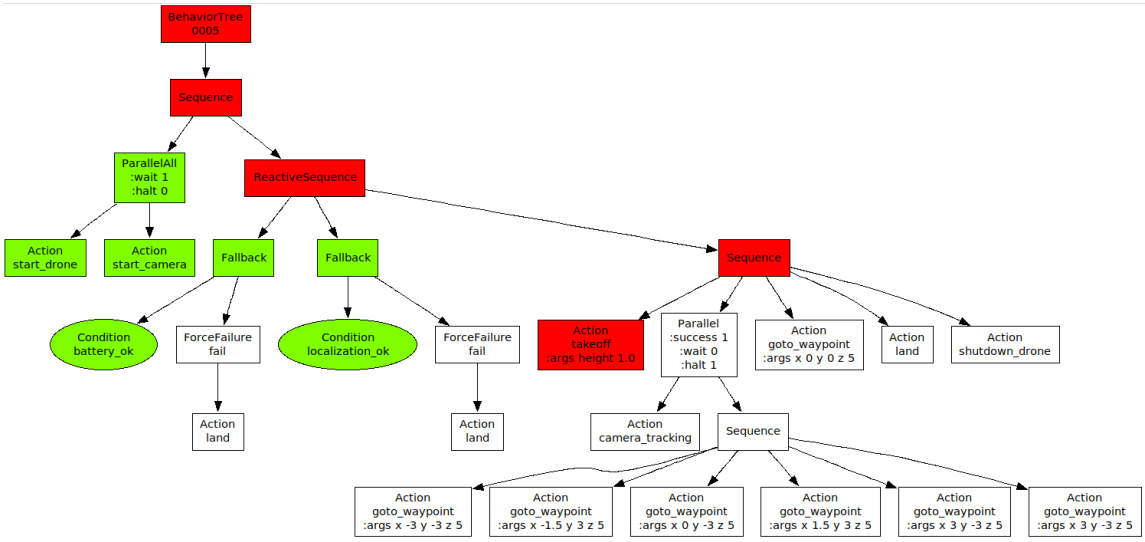

Figure 16: The graphical representation of the BT Listing 4p11.

E A ROS2 Nav2 BT and its BTF equivalent.

Listing 10: One of the ROS2 Nav2 Behavior Tree (Navigate To Pose With Replanning and Recovery) [Macenski et al., 2024].

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <RecoveryNode number_of_retries="6" name="NavigateRecovery">
      <PipelineSequence name="NavigateWithReplanning">
        <RateController hz="1.0">
          <RecoveryNode number_of_retries="1" name="ComputePathToPose">
            <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"/>
            <ReactiveFallback name="ComputePathToPoseRecoveryFallback">
              <GoalUpdated/>
              <ClearEntireCostmap name="ClearGlobalCostmap-Context"
                                  service_name="global_costmap/clear_entirely_global_costmap"/>
            </ReactiveFallback>
          </RecoveryNode>
        </RateController><
        <RecoveryNode number_of_retries="1" name="FollowPath">
          <FollowPath path="{path}" controller_id="FollowPath"/>
          <ReactiveFallback name="FollowPathRecoveryFallback">
            <GoalUpdated/>
            <ClearEntireCostmap name="ClearLocalCostmap-Context"
                                service_name="local_costmap/clear_entirely_local_costmap"/>
          </ReactiveFallback>
        </RecoveryNode>
      </PipelineSequence>
      <ReactiveFallback name="RecoveryFallback">
        <GoalUpdated/>
        <RoundRobin name="RecoveryActions">
          <Sequence name="ClearingActions">
            <ClearEntireCostmap name="ClearLocalCostmap-Subtree"
```

```
                            service_name="local_costmap/clear_entirely_local_costmap"/>
          <ClearEntireCostmap name="ClearGlobalCostmap-Subtree"
                            service_name="global_costmap/clear_entirely_global_costmap"/>
        </Sequence>
        <Spin spin_dist="1.57"/>
        <Wait wait_duration="5"/>
        <BackUp backup_dist="0.15" backup_speed="0.025"/>
      </RoundRobin>
    </ReactiveFallback>
  </RecoveryNode>
 </BehaviorTree>
</root>
```

Listing 11: The `.btf` version of the ROS2 Nav2 BT above (Listing 10p40).

```
((BehaviorTree :ID MainTree ; The top level root node
    (Recovery :num_retries 6 :name NavigateRecovery
       (PipelineSequence :name NavigateWithReplanning
        (RateController :args (hz 1)
           (Recovery :num_retries 1 :name ComputePathToPose
            (Action :ID ComputePathToPose :args (goal $goal path $path planner_id GridBased))
            (ReactiveFallback :name ComputePathToPoseRecoveryFallback
             (Condition :ID GoalUpdated)
             (Action :ID ClearEntireCostmap :name ClearGlobalCostmap_Context1
                     :args (service_name global_costmap/clear_entirely_global_costmap)))))
        (Recovery :num_retries 1 :name FollowPath
           (Action :ID FollowPath :args (path $path controller_id FollowPath))
           (ReactiveFallback :name FollowPathRecoveryFallback
            (Condition :ID GoalUpdated)
            (Action :ID ClearEntireCostmap :name ClearLocalCostmap_Context2
                    :args (service_name local_costmap/clear_entirely_local_costmap)))))
        (ReactiveFallback :name RecoveryFallback
         (Condition :ID GoalUpdated)
         (RoundRobin :name RecoveryActions
           (Sequence :name ClearingActions
            (Action :ID ClearEntireCostmap :name ClearLocalCostmap_Subtree3
                    :args ( service_name local_costmap/clear_entirely_local_costmap))
            (Action :ID ClearEntireCostmap :name ClearGlobalCostmap_Subtree4
                    :args (service_name global_costmap/clear_entirely_global_costmap)))
           (Action :ID Spin :args (spin_dist 1.57))
           (Action :ID Wait :args (wait_duration 5))
           (Action :ID BackUp :args (backup_dist 0.15 backup_speed 0.025)))))))
```

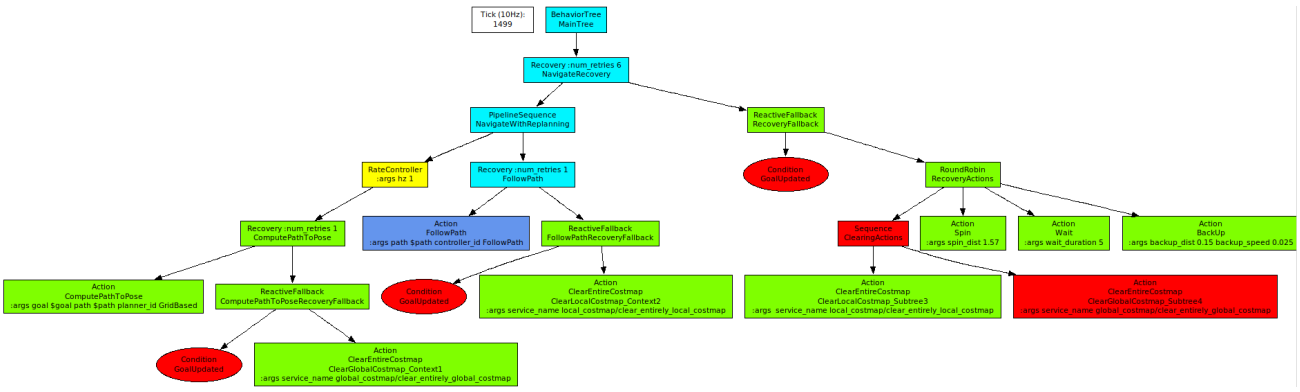

Figure 17: The screen dump of the Nav2 BT (See Listing 11p41 executing.

F The Mars rover example from [Biggar and Zamani, 2020, Wang et al., 2024].

Listing 12: The BTF model of the Mars rover example from [Biggar and Zamani, 2020, Wang et al., 2024].

```
( ;example from
  ; A framework for formal verification of behavior trees with linear temporal logic.(2020)
  ; O. Biggar and M. Zamani.
  ; also presented in
  ; Enabling Behaviour Tree Verification via a Translation to BIP. (2024)
  ; Q Wang, H Dai, Y Zhao, M Zhang, S Bliudze
  (defsv meteo ; this defines the meteo state variable
    :states (MInit Normal Storm) ; MInit an undefined init state
    :init MInit ; the following transitions forbid coming back to MInit
    :transitions ((MInit Normal) (MInit Storm) (Storm Normal) (Normal Storm)))

  (defsv battery
    :states (BInit Good Low) ; BInit just to says that we do not know
    :init BInit ; the following transitions forbid coming back to BInit
    :transitions ((BInit Good) (BInit Low) (Low Good) (Good Low)))

  (defsv panel
    :states (PInit Folded Unfolded) ; PInit just to says that we do not know
    :init PInit ; the following transitions forbid coming back to PInit
    :transitions ((PInit Folded) (PInit Unfolded) (Unfolded Folded) (Folded Unfolded)))

(BehaviorTree :name mars_rover
  (Fallback
   (Sequence
    (Eval (= battery Low)) ; if the battery is low
    (Action :ID UnfoldPanels :name unfold_panels :SF) ; we unfold the panel
    (Eval (:= panel Unfolded))) ; and the panel SV becomes unfolded
   (Sequence
    (Eval (= meteo Storm)) ; if the meteo is a storm
    (Action :ID Hibernate :name hibernate :SF) ; we hibernate
    (Eval (:= panel Folded))) ; and we fold the panel
   (Sequence
    (Action :ID DataReady :name dataready :SF)
    (Action :ID Send :name send :SF)))))

; prove absent ( mars_rover/3/state Unfolded and mars_rover/1/state Storm )
```