ProSkill: A formal skill language for acting in robotics

Félix Ingrand

ProSkill: A formal skill language for acting in robotics

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

Abstract

Acting is an important decisional function for autonomous robots. Acting relies on skills to implement and to model the activities it oversees: refinement, local recovery, temporal dispatching, external asynchronous events, and commands execution, all done online. While sitting between planning and the robotic platform, acting often relies on programming primitives and an interpreter which executes these skills. Following our experience in providing a formal framework to program the functional components of our robots, we propose a new language, to program the acting skills. This language maps unequivocally into a formal model which can then be used to check properties offline or execute the skills, or more precisely their formal equivalent, and perform runtime verification. We illustrate with a real example how we can program a survey mission for a drone in this new language, prove some formal properties on the program and directly execute the formal model on the drone to perform the mission.

1 Introduction and motivation

Acting is an important decisional function to ensure proper deliberation on an autonomous system [Ingrand and Ghallab, 2017]. It often sits between *planning* and the platform, but unlike *planning* it is an online process, which must stay reactive to the dynamic of the environment and the platform and cannot devote resources to long computations and complex searches. *Acting* often relies on models, called *skills*, which specify how to perform actions (as an operational model), while the *action* models used for *planning* are more what is abstractly needed to perform the action (as a descriptive model) [Ghallab et al., 2016].

The most basic skills need to connect to the commands made available by the functional level to the *acting* component, call them asynchronously, get execution status and result, but it also needs means to receive exogenous events as they occur in the environment. This action/command dispatching may also rely on preconditions and invariants checking, interruptions, temporal constraints, etc. Above the basic skills one often finds more complex skills, similar to programs with control structures to allow for local choices and local recoveries with test, branching, looping, parallel and asynchronous execution.

Considering the expected functionalities of an acting component, its skill language/framework should provide the following features:

- Support for Validation and Verification (V&V). Notwithstanding the other functionalities, this is the feature the work presented in this paper focuses on.

- One cannot only rely on basic skills connecting to the robot commands, one also needs some programming primitives (e.g., test, branching, loop).
- Controlling and managing multiple commands sent to functional components require some primitives to handle parallelism, threads, resources.
- In embedded real-time systems, one needs an explicit time representation to wait, synchronize, check temporal constraints, etc.
- The language should reflect the tight coupling and necessary consistency between *planning* and *acting*.

There are other less critical features which are now expected from an acting component: human awareness (how does the acting component manage the human presence and activities), link with motion/manipulation (most robots move and solving their motion planning problems often require an elaborate coupling with acting). Acting skills can be programmed by hand, but learning them or their parameters must also be investigated.

Nevertheless, all these considered, the work presented here focuses on the formal V&V aspects of acting.

The paper is organized as follows. After introducing above the *acting* decisional functionality and its skills, a state of the art of the various approaches and formalisms, with respect to V&V, is discussed in the next section concluding with a justification of the proposed approach and its originality. Section 3 presents the ProSkill acting language and its primitives. The Fiacre formal framework we use is presented in Section 4. Section 5 presents how ProSkill primitives are mapped in Fiacre and then the type of formal properties one can show online but also at runtime. In Section 6, we introduce a real robotic example on which we successfully deployed our approach. A discussion in Section 7 reassesses the pros and cons of the ProSkill/Fiacre formal acting language, leading to possible future work and then to the conclusion of the paper.

2 Acting: state of the art and proposed approaches

Over the years, the *planning* field has managed to define and agree on a number of common languages and formalisms: PDDL [Ghallab et al., 1998]; NDDL [Frank and Jónsson, 2003]; ANML [Smith et al., 2008]; HDDL [Höller et al., 2020]; chronicle [Ghallab, 1996]; etc. Similarly, for robotic functional components (nodes), ROS [Quigley et al., 2009] has emerged as a de facto standard, at least to enable data and code sharing among roboticists. In both domains, many would argue that it is good for the field as it allows systems interoperability, comparison, competition, sharing, etc. while others would regret that it restricts and constrains the development of new original approaches, as researchers may remain too focused on improving the associated algorithms. Meanwhile, the *acting* field remains mostly organized along one system — one language tandems. Yet, we will see in the following sections that there are large categories of approaches to model and execute skills to perform *acting*, and we will stress how these approaches may support V&V.

2.1 Reactive planners and procedural languages

Acting must cope with the real-world contingencies, although it cannot really plan, it can reuse some plans parts, or choose among a set of "reactive plans" (e.g. RAP [Firby, 1987]), and be able to synchronize and perform task decomposition (e.g., TDL [Simmons and Apfelbaum, 1998]). These principles led to the development of many systems. Among them, PRS (Procedural Reasoning System) first implementations (late eighties) were mostly used for procedural maintenance [Georgeff and Ingrand, 1989]. It is only later that at LAAS, PRS skills (called Operational Procedure) were used as an acting component in the LAAS architecture [Alami et al., 1998, Ingrand et al., 2007]. The Propice [Despouys and Ingrand, 1999] fork is an attempt to infuse some planning in the acting functionality of the robots. Overall, PRS was deployed in many robotic experiments. From a V&V point of view, there are some attempts to transform PRS skills in Colored Petri Net [De Araújo and de Medeiros, 2004] and later to give a formal semantics to the PRS skill language [de Silva et al., 2018]. But both studies remain theoretical and were never applied, nor deployed to verify skills. Similarly, some work was conducted to show how some formal properties on TDL [Simmons and Apfelbaum, 1998] skills could be proven using NuSMV [Simmons and Pecheur, 2000]. But again, this study remained mostly theoretical.

More recently, PLP [Brafman et al., 2016] proposes an acting language which borrows some of the PDDL constructs and provides means to monitor the behavior of functional components. [Kovalchuk et al., 2021] introduces a stochastic extension to their model based on UPPAAL-SMC [David et al., 2015]. With respect to V&V, the goal to build a monitor of the controller is interesting, but it did not go as far as deploying a formal V&V.

Behavior Trees [Colledanchise and Ögren, 2018] are becoming quite popular in the robotic community. Initially developed for the video game industry, their simple yet powerful execution tree mechanism (skills) based on *sequence*, *fallback* and *parallel* nodes has been quite successful in deploying reactive modular acting systems. Yet, they lack an explicit time representation and even if the authors propose some mechanisms to check efficiency, safety, and robustness, those mostly rely on ad hoc procedures, not a fully equivalent formal model (but we will see in the discussion (Section 7) that this could be seriously considered).

RAE [Ghallab et al., 2016] introduces the basic algorithms which can drive a procedural reactive acting engine. Inspired by the PRS [Ingrand et al., 2007] engine, it mostly differs on its semantics of task (RAE) vs. goal (PRS) and on its handling of event-based skills execution. The later implementations of RAE introduce a look ahead planning mechanism to evaluate future outcomes and help make better choices while acting.

2.2 State machines and programming language

Finite state machines are often proposed as acting models. Yet, one should consider whose states are considered? The states of the world and its components, or the states of the acting program execution? In the former, they capture the acceptable states and state transitions of the components, while in the later, they capture the possible execution states which can be more difficult to maintain (some argue that they are then just a bunch of `GoTo` instructions).

Yet, there are acting systems which mostly rely on them. SMACH [Bohren and Cousins, 2010] is an acting system deployed along ROS which models skills with hierarchical state machines. The states in SMACH are execution states, each with the possible outcomes and one can have multiple state machines active at once. Similarly, RAFCON [Brunner et al.,

2016] also proposes hierarchical state machines to program robotics systems and allow the creation of concurrent flow controls. rFSM statecharts [Klotzbücher and Bruyninckx, 2012] stresses the coordination aspect of the acting system and proposes to use Harel statechart to implement it.

Interestingly, RMPL [Ingham et al., 2001, Williams and Ingham, 2003] mixes hierarchical finite states machines, to represent the state of some devices used in the experiment and, a programming language inspired by Esterel [Boussinot and de Simone, 1991] to program the skills which need to be deployed and executed. This is an original combination, and we will further discuss it in Section 7. More recently, Proteus [McClelland et al., 2021] also relies on similar hierarchical finite states machines for components but proposes a more classical programming language to deploy the system.

2.3 Acting/planning framework

We must also consider systems using skills which are common (or share a significant part) between the planning (action model) and the acting language and could benefit from their action model "part" to improve the verifiability of their skills. Indeed, most planners are performing some kind of model verification, using states exploration, constraints satisfaction, etc.

Cypress (i.e., SIPE/PRS) proposes the Act formalism [Wilkins and Myers, 1995] to unify both functionalities, but this common skill/action model remains mostly syntactic, and both engines pick the [art they need from the Act representation.

The IDEA [Finzi et al., 2004] and T-ReX [McGann et al., 2008] also proposes to merge the planning and acting representation, arguing, with some truth, that they are similar processes, with just a different horizon and response time. Organizing the state variables of the problem along different planners/reactors, they use constraints to specify the acceptable state variable values and value transitions. But writing the proper constraints to perform acting and planning ends up being quite tedious and error prone.

OMPAS [Turi et al., 2023, Turi, 2024] proposes an acting language whose skills can be automatically transformed in temporal chronicles to be used by a temporal planner for some limited horizon planning and help the acting component to make better informed choices. This is an interesting approach as it bridges skill models and action models.

2.4 Programming skills within a formal framework

Last, there are also many *acting* approaches which explicitly rely on some well-founded formal frameworks:

Situation calculus Many systems propose Situation Calculus as an underlying model for planning and acting: GOLEX [Hähnel et al., 1998], YAGI [Eckstein and Steinbauer, 2020], Golog++ [Mataré et al., 2021] to name a few. Yet despite the formal underlying framework, their deployment in real systems remains confidential.

Petri net is a well-known formalism to model concurrent systems and is supported by many formal tools. It is used in many acting systems: PROCOSA [Barbier et al., 2006], ASPIC [Lesire and Pommereau, 2018], Petri net [Costelha and Lima, 2012] and Hierarchical Petri Nets [Figat and Zieliński, 2022], etc. [Albore et al., 2023, Pelletier et al., 2023] propose SkiNet, a skill language which automatically maps in Time Transition

Systems (Time Petri Net with data)[1]. This approach has some commonality with the one we propose, and we discuss their difference in Section 7.

Synchronous language Many approaches rely on a synchronous language "hypothesis" (communication and computation in no time). Historically MAESTRO/ORCCAD [Coste-Maniere et al., 1992, Espiau et al., 1996, Kim and Kang, 2005], relying on Esterel, already nailed down the idea of an acting skill language able to map in a formal model. ReX [Kaelbling and Wilson, 1988]/GAPPS [Kaelbling, 1988] can also be seen as a synchronous approach skill implementation. Plexil [Verma et al., 2006] executes skills modeled with different types of nodes along some control structures. A formal, but adhoc, extension is presented in [Dowek et al., 2007].

Robot Chart The Robot Chart [Cavalcanti, 2017] framework is also deeply grounded in formal models, and numerous extensions have been proposed. But the language does not seem to grab much popularity or use in the robotics community which probably discards it as too complex or cumbersome to use.

2.5 Our approach

On one hand, most of the approaches presented in Section 2 (Subsection 2.4 apart) lack support for V&V with an automatic and systematic translation of their skills in a formal framework. On the other hand, the ones presented in section 2.4 have a strong potential to provide formal V&V, yet they often remain difficult or cumbersome to use for roboticists as most of them require some knowledge of the underlying formalism. Moreover, none of them provide a model which can be used both for offline *and* runtime verification.

In the survey [Bjørner and Havelund, 2014], the authors write:

> *"We will argue that we are moving towards a point of singularity, where specification and programming will be done within the same language and verification tooling framework. This will help break down the barrier for programmers to write specifications."*

Similarly, in [Nordmann et al., 2016] the authors survey robotic DSL, and they argue that:

> *"Both communities should foster collaboration in order to make formal methods more practicable and accepted in robotic software development and to make DS(M)L approaches more well-founded in theory to foster work in the field of model validation and verification."*

Following these advices again (we have already presented a formal framework for functional components [Dal Zilio et al., 2023]), we now want to extend the use of formal models toward the *acting* component. Hence, the proposed approach presents the following characteristics:

- Programming the acting component using our proposed skill language does not require any knowledge in formal models and languages,
- The language can be automatically, fully, and unequivocally translated in FIACRE [Berthomieu et al., 2008, 2020], a preexisting formal language, with a clear semantics,

[1] We borrow some of SkiNet semantics in PROSKILL.

- The obtained FIACRE formal model can be used both offline with model checking to prove some properties of the *acting* program, but also online to execute the formal model on the robot.
- The skill programs *are* equivalent to a formal specification, and the formal specification *is* verifiable and executable on the robot.

As we will see, providing a skill language, whose equivalent formal model is directly executing on the robot, has several advantages: it increases the credibility that the skills are doing what they are intended too, and it improves the acceptability by roboticist as no extra step is needed to run the skills within a formal framework, so there is no need to develop a new "skill" execution engine.

3 The PROSKILL language

The PROSKILL language and specifications rely on four basic primitives: state variables, events, basic skills, and composite skills. The first three are inspired by similar objects in the SkiNet skill language [Albore et al., 2023].

For each of these objects, we explain their role, and how they relate to each other, the platform, and the environment.

3.1 State variables

State variables are variables which take their value in either a bounded natural number or among several enumerated values. They always have an initial value, and for the latter, one can allow transition to any other values, or more restrictively to some limited number of transitions.

In PROSKILL they are defined with statements such as the ones on Listing 1.

Listing 1: Example of State Variables definition

```
(defsv flight_levels
  :init 1
  :min 1
  :max 3)

(defsv battery
  :states (Good Low Critical) ; the possible values the state variable can take
  :init Good ; the initial value
  :transitions ; use the :all keyword if all transitions are allowed,
    ((Good Low)(Low Critical) ; list them otherwise.
     (Critical Low)(Low Good)))
```

In this example the state variable *battery* can change from any values to any other values except from Critical to Good and vice versa.

3.2 Events

Events correspond to external events which can occur at any time. The effects of an event are to produce some state variables value change.

In the following example, some sensors will issue the *battery_to_critical* event, which leads to updating the *battery* state variable (see Listing 2).

Listing 2: Example of an *event* definition

```
1  (defevent battery_to_critical
2    :effects (battery Critical))
```

3.3 Basic skills

The basic skills are the lowest level skills. They are the ones which act on the underlying robotic system, i.e., they launch commands with arguments, and when the command completes, the basic skill gets the status (`success`, `failure`, `failed_inv` or `interrupted`) and the results, if any. The underlying commands can be programmed in ROS ROS [Quigley et al., 2009] (e.g., using ROS Actions), or other robotic frameworks, such as GenoM [Dal Zilio et al., 2023].

Although the syntax is slightly different, the basic skills semantics is equivalent to the one described in [Albore et al., 2023, Section 4.5, Figure 7]. Listing 3 shows an example, withdrawn from the drone use case we present in section 6, of a basic skill specifying and commanding the **takeoff** of the drone.

Listing 3: Example of a Basic Skill **takeoff**

```
1  (defskill takeoff
2    :input ($height float $duration float) ; The expected arguments
3    :precondition (not_moving (motion Free) ; a list of tag (state_variable value)
4            on_ground (flight_status OnGround) ; all four preconditions need to be satisfied
5            battery_good (battery Good)
6            origin_valid (localization_status Valid))
7    :start (motion Controlled) ; state_variable to set upon starting this skill
8    :invariant (in_control (:guard (motion Controlled)) ; all the guards are monitored during
9              battery (:guard (~ (battery Critical)) ; the execution of this action, if one
10                      :effects (motion Free))) ; fails its effects, if any, are set
11                                               ; and the action is interrupted
12   :time_interval [1,3] ; this is the interval of time in seconds this action should take
13   :action (takeoff) ; this is linked to the C/C++ code which will be invoked
14   :interrupt (:effects (motion Free)) ; when interrupted, set this state_variable
15   :success at_altitude ; success, and their effects on state_variable
16          (:effects (motion Free) ; postcondition are just checked for
17           :postcondition (flight_status InAir)) ; model consistency
18   :failure (grounded (:effects (motion Free) ; they are not enforced
19             :postcondition (flight_status OnGround))
20            emergency (:effects (motion Free)
21             :postcondition (flight_status InAirUnsafe))))
```

Most of the basic skill fields are self-explanatory, state variable values are checked (in `:precondition`, `:guard`, `:postcondition`) and set (in `:start` and `:effect`). Note that all `:guard` and `:effect` are enforced, but `:postcondition` (lines 3.17,[2] 3.19 and 3.21) are just checked (they are expected as an effect of the success or failure, but not enforced). `:time_interval` (line 3.12) is a specification of the duration the action should take, and the `:action` field (line 3.13) specifies the command to call on the robot when this skill runs.

3.4 Composite skills

Composite skills are hierarchically above basic skills and act more like a programming language where one can call other skills (basic or composite), test the returned status and values,

[2]Listing lines are referenced with the <listing number>.<line number>, example: 3.17, Listing 3, line: 17.

test state variable values, branch according to these test, loop, wait for some time or conditions, and even execute multiple branches in parallel. Composite skills are inspired by PRS procedures [Ingrand et al., 1996].

Listing 4 shows an example of composite skill withdrawn from the experiment presented in section 6. The fields common to the basic skills have the same semantics and are treated alike. The most important and new field here is the `:body` field (line 4.8), instead of the `:action` field (line 3.13), which lists the program instructions to execute when the skill is called.

Listing 4: Example of a Composite Skill: **uav_mission**

```
1  (defskill uav_mission
2    :time_interval [60 , 120] ; expected min and max time to perform this skill.
3    :success mission_accomplished ; success and
4      (:effects (mission_status Succeeded)) ; its effect
5    :failure mission_failed ; failure and
6      (:effects (mission_status Failed)) ; its effect
7    :start (mission_status Ongoing) ; setting the mission status SV upon starting
8    :body
9      ((start_drone) ; This will call the start_drone basic skill
10      (^ (localization_status Valid)) ; wait for the localization to be Valid
11      (takeoff height 3.0 duration 0) ; call the takeoff basic skill (3 meters)
12      (if (= takeoff.status success) ; if the takeoff status is success
13        (// ((camera_survey)) ; execute two branches in parallel camera_survey on one
14            ((goto_waypoint x 1 y 2 z 3 yaw 0 duration 0) ; navigation on the other one
15             (^ 2) ; wait 2 seconds
16             (goto_waypoint x -3 y -2 z 4 yaw 1.4 duration 0) ; navigating to a new position
17             (camera_survey.interrupt))) ; interrupt the camera_survey skill
18      (if (= goto_waypoint.status success) ;the last goto_waypoint was a success
19        (landing) ; call the landing skill
20        (if (= landing.status success) ; if successful
21          (shutdown_drone) ; call the shutdown_drone skill
22          (printf "Mission Accomplished") ; print a message
23          (success mission_accomplished)))) ; return the mission_accomplished success
24     (printf "Mission failed") ; otherwise, print and report failure.
25     (failure mission_failed)))
```

Note that for any **skill**, one can access `skill.status` (lines 4.12, 4.18 and 4.20) and `skill.res` to respectively check the status (`success`, `failure`, `failed_inv` or `interrupted`) and the result of the **skill**'s last execution. Moreover, one can also call `skill.interrupt` to interrupt an executing **skill** (line 4.17).

Monitor skills

Monitor skills are a particular type of composite skill which are called immediately upon starting the execution of the ProSkill program (like the main composite skill). Their first instruction is usually to wait for a condition they monitor, then they execute the rest of the body like any other composite skill.

4 A formal framework for offline and runtime verification: Fiacre, language, models, and tools

Although the goal of this paper is not to present in detail the formal framework we use, it is necessary to clarify some terminology and give some explanations to make it self-contained. The more curious readers can check the specific papers and websites referenced below.

Listing 5: Example of a Monitor Skill to gently land the drone when the *battery* level becomes critical.

```
(defskill monitor_battery_critical
  :monitor t ; Mark this skill as a monitoring one (hence it gets started upon startup)
  :body ((^ (battery Critical)) ; wait until the battery becomes critical
         (printf "We will try to land the drone safely...") ; print a message
         (set_velocity vx 0.0 vy 0.0 vz -0.1) ; force landing by setting -10cm/s vertical speed
        ))
```

4.1 Terminology, models, languages, and tools

We first clarify the following terms:

Time Petri nets [Berthomieu and Diaz, 1991] are an extension from regular *Petri nets* model where each transition has a time interval (by default $[0, \infty)$) which specifies that the transition is sensitized and can be fired only during this time interval.

TTS Time Transition Systems are an extension of *Time Petri nets* with data, and where transitions can call data processing functions.

TINA stands for "TIme petri Net Analyzer", it is a toolbox for the editing, simulating and analysis of *Petri nets*, *Time Petri nets* and *TTS*. Among these tools, sift and selt can be used to respectively build the set of reachable states of the model and check LTL properties on the model.[3]

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

H-FIACRE is an extension of the FIACRE language to make the specified model "executable" by adding *Event Ports* and *Tasks* both linked to C/C++ functions.

HIPPO is an engine to execute *TTS* obtained from H-FIACRE specifications [Hladik et al., 2021].[5]

This framework has been deployed in numerous projects and applications,[6] and, not surprisingly, is also the framework we used to validate and verify the functional components of our robotics experiments [Dal Zilio et al., 2023].

4.2 FIACRE semantics

Although we refer the reader to specific papers and websites (see above) for the formal model presentation and the tools, we think it is important to get an idea of the semantics of the FIACRE language with a small example: the specification of a mouse triple clicks detector.

[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

This example is specified on Listing 6 along the illustration Figure 1. It defines three FIACRE processes, each with its own automata. The first process, **clicker**, produces a *click* at any time. It waits between 0 and ∞ and then synchronizes on the click port with the **detect_triple_click** process. This second process has four states, waiting for synchronization on click, or that the maximum acceptable time between two clicks (0.2 sec) has elapsed. Note the `select` on the *wait_second* and *wait_third* states, which is a non-deterministic choice which will be explored by the model checker. When the *detected* state is reached, a synchronization on the triple_click is made and this allows the transition of the **triple_click_receiver** process to the *received_tc* state.

Following these process specifications, one component is specified by putting three process instances in parallel (line 6.56) and connecting them with two ports (line 6.54). This example is intentionally simple, but the FIACRE language supports complex data type, directional ports exchanging data (in and out), local and global variables, tests, switch/case, guards on transitions and calls to functions (internal to FIACRE or external with C/C++ code) making complex computations. More complex FIACRE specifications will be introduced and can be found in A and B.

Listing 6: 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
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]
58  || clicker[click]
59  || triple_click_receiver[triple_click]
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
68  property cannot_receveice_tc is absent comp_tc/3/state received_tc
69  assert cannot_receveice_tc // we cannot detect a triple click (FALSE)
```

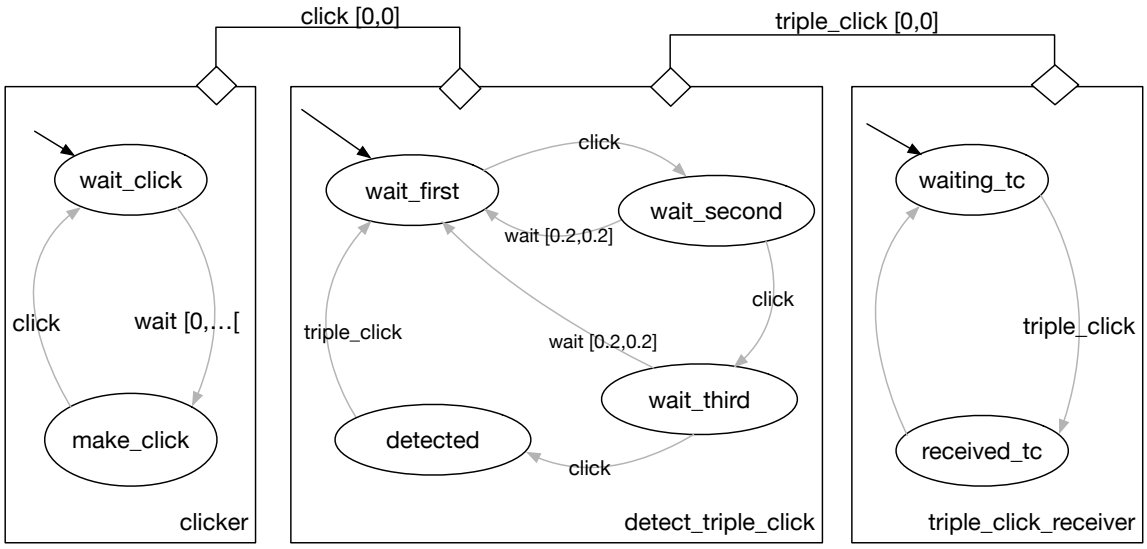

Figure 1: The FIACRE processes modeling the FIACRE specification on Listing 6.

4.3 Offline formal verification

The frac compiler compiles the FIACRE specifications on Listing 6 in an equivalent TTS. Then using the sift tool from the TINA toolbox, one builds the set of reachable states of the system, and then using selt, one checks the properties included in the FIACRE initial specifications as well as new properties, if needed. There are many other tools available in the TINA toolbox, which the interested reader can check.

Listing 6 proposes some properties which will be checked on this specification: is the model deadlock free (line 6.65)? which ends up being TRUE. Can it succeed in detecting a triple click (line 6.68)? (by checking that reaching the *received_tc* is impossible), and this is FALSE, so it means that the model can detect a triple click. More complex properties such as proving that there is at most 0.4 sec between the first and the last click could be added, etc.

Note that the verification approach used by the TINA tools is based on model checking, and as such suffers from state explosion [Clarke et al., 2012] which can jeopardize the usefulness of such an endeavor. Nevertheless, we will see in the results section of the example we present in section 6.1, that we are still able to produce interesting nontrivial results.

4.4 H-FIACRE runtime extensions

Although the FIACRE language was initially designed for offline verification, it has been extended with two primitives which enable it to be used for runtime verification [Hladik et al., 2021], by connecting the model to C/C++ functions which send events or execute some commands. To distinguish it from pure FIACRE we call the extended version H-FIACRE.

The goal of the H-FIACRE runtime version is to make the model "executable" while being connected to the real world.

Listing 7 (along Figure 2) shows the executable version of the specification on Listing 6.

Event ports are declared in the preamble of the specification (see line 7.1) and they associate a C function to a FIACRE port. In this example, the event *click* is associated with the `c_click` C/C++ function. Whenever this port is among the possible transitions (lines 6.16, 6.24 and 6.33), the C/C++ function is called, and the port is activated when the function returns (the C/C++ functions can take and return FIACRE typed arguments).

Tasks are also declared in the preamble (see line 7.2) and they associate a task (here *report_triplec*) to a C/C++ function (here `c_report_triple_click`), which will be called asynchronously upon a `start` (see line 7.17) and will enable the corresponding `sync` (see line 7.21) when the C/C++ function returns. Here also, values can be passed upon calling the task and returned when complete.

Listing 7: 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 c_report_triple_click // The C/C++ function called by this task
3
4  process detect_triple_click [triple_click:sync] is
5  // this process is exactly the same than in the regular Fiacre version
6  // only the click port is now an event port
7
8  process triple_click_receiver[triple_click:sync] is
9  states waiting_tc, received_tc, sync_report
10 var ignore : nat
11
12 from waiting_tc
13    triple_click;
14    to received_tc
15
16 from received_tc // show an example of an external call
17    start report_triplec();
```

```
18     to sync_report
19
20   from sync_report
21     sync report_triplec ignore; // wait until the call return
22     to waiting_tc
23
24   component comp_tc is
25   port triple_click:sync
26
27   par * in
28     detect_triple_click[triple_click]
29   || triple_click_receiver[triple_click]
30   end
31
32   comp_tc
```

In this example, we have replaced the **clicker** process, which synced click at any time, by the *click* event port (in purple), and we have added a task (*report_triplec* in light blue) to execute when we synchronize with a *triple_click* in the **triple_click_receiver** process. The rest of the model remains the same, so we moved from a model to specify a triple click detector, to a program/controller which implements it. The specification is now also a program.

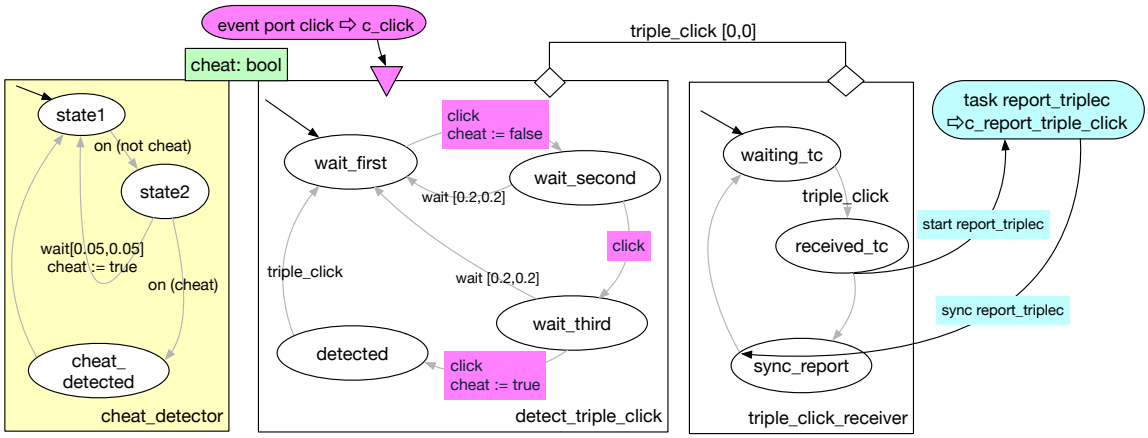

Figure 2: Illustration of the H-Fiacre program on Listing 7.

4.5 Runtime (online) verification

The resulting H-Fiacre model is then compiled (with frac) in TTS and linked with the Hippo engine and the C/C++ functions needed to run the model (i.e., `c_click` and `c_report_triple_click`). The Hippo engine literally runs the resulting TTS model and makes the appropriate calls (in separate threads) to the C/C++ functions associated with *event ports* and *tasks*. Note that we can also enrich the model with properties to check on the fly by adding a monitor process. For example, if this controller is used in a video game where the game programmer wants to detect sequence of triple click produced so fast that they are not humanly possible, and must come from a cheating device, than we could add a new **cheat_detector** process (on the left Figure 2 and Listing 8) with three states, sharing a Boolean variable `cheat` set to false in the transition to *waiting_second* of process **detect_triple_click**, and true in the transition to *detected*.

Listing 8: **cheat_detector** process detecting a cheating device by monitoring the `cheat` Boolean variable.

```
1  process cheat_detector(&cheat:bool) is
2
3  states state1, state2, cheat_detected
4
5  from state1
6    on (not cheat); // guard on (not cheat)
7    to state2
8
9  from state2 // cheat was set to false
10   select // either
11     wait [0.05,0.05]; // 50 ms elapsed
12     cheat := true; // reset the cheat variable
13     to state1 // go back to monitoring
14   []
15     on (cheat); // cheat became true again before the 50ms above.
16     to cheat_detected //caught cheating
17   end
18
19 from cheat_detected
20   // the player is cheating, do what needs to be done.
21   to state1
```

In the **cheat_detector** process *state1*, it would *guard* on `(not cheat)` and then would transition to *state2* and wait either 50 ms or `(cheat)`. If `cheat` becomes true before the 50 ms (i.e., a super-fast triple click has been issued), then it transitions to *cheat_detected* and reports a suspicious behavior, otherwise, it sets `cheat` to true and goes back to *state1*.

So, we synthesize a controller which runs the specification. This is one of the critical advantages of the Fiacre framework: the same ¡formal model can be checked offline and run online. Beyond verification, on one hand, you get the real controller performing what the model specifies, and on the other hand, observing the behavior of the running model confirms that your initial specifications do what you intended it to do, and thus the properties you check offline are indeed, applied to the same online "behaviorally" good model. Of course, observing the proper behavior of the running specified model is a necessary but not sufficient condition. Yet, it is better than having no link between the specification and the execution. Note that if the runtime model is wrong, you can start debugging it (the same way you debug regular programs). Also note that from a formal point of view, for a given model, all the traces of the H-Fiacre version are included in the ones of the Fiacre version.

5 The ProSkill language and its mapping in Fiacre models

We introduced the ProSkill language primitives in Section 3, we now show how each of them translates to Fiacre variables and processes, how they interact (through Fiacre ports) and how we instantiate and compose them in a global Fiacre component.

5.1 State variables Fiacre version

ProSkill state variables are automatically transformed to Fiacre variables which can only take values among the one specified in the ProSkill specifications. Their allowed value transitions are automatically specified in a Fiacre process. Note that if the state value change is not consistent with the ProSkill specified ones, an error occurs (lines 9.15, 9.32). For natural number state variables, we must specify the accepted values interval. Listing 9 implementing the state variables specified in Listing 1.

Listing 9: FIACRE type and FIACRE process specifying the State Variable acceptable value changes.

```
1
2  type sv_flight_levels is 1..3 /* state variable integer types*/
3
4  process sv_battery_automata (&battery:sv_battery) is
5      // for each sate variable, an automata enforces the allowed transition
6  states Good, Low, Critical, error
7
8  from Good
9      wait [0,0];
10     select
11         on (battery = Low);
12         to Low
13     []
14         on (battery = Critical);
15         to error // forbidden transition
16     end
17
18 from Low
19     wait [0,0];
20     select
21         on (battery = Good);
22         to Good
23     []
24         on (battery = Critical);
25         to Critical
26     end
27
28 from Critical
29     wait [0,0];
30     select
31         on (battery = Good);
32         to error // forbidden transition
33     []
34         on (battery = Low);
35         to Low
36     end
```

5.2 Events FIACRE version

The example introduced in Section 3.2 is automatically translated to the following simple FIACRE process. Each PROSKILL event become a FIACRE sync port which will be connected to an "Environment" FIACRE process in the offline verification and then to the real environment, with event ports[7], in the runtime version (i.e., some C/C++ functions will trigger and synthesize the event accordingly).

Listing 10: FIACRE process specifying how the *battery_to_critical* event is handled.

```
1  process event_battery_to_critical_automata
2         [battery_to_critical : sync] // a Fiacre sync port on which we get the event
3         (&battery: sv_battery) is // the battery state variable
4
5  states start_
6
7  from start_
8      battery_to_critical; // when this port interact
9      battery := Critical; // the battery value is updated to Critical
10 to start_
```

[7] Although PROSKILL events end up mapped in FIACRE event ports in the H-FIACRE version, they should not be confused.

5.3 Basic skills FIACRE version

As expected, the basic skills are the ones "executing" the commands on the robot. The automata FIACRE offline version of the basic skill presented Section 3.3 is illustrated on Figure 3, while the automata H-FIACRE runtime is illustrated on Figure 4 and listed in A, Listing 13.

An important FIACRE variable which appears in the various FIACRE listing is the `skill[]` array, indexed for each skill (basic and composite) and for each parallel branch, whose elements (a FIACRE record) contain for each skill/branch, information as whether it is currently running or not, which skill called it, what are its argument, and what are its last status and result.

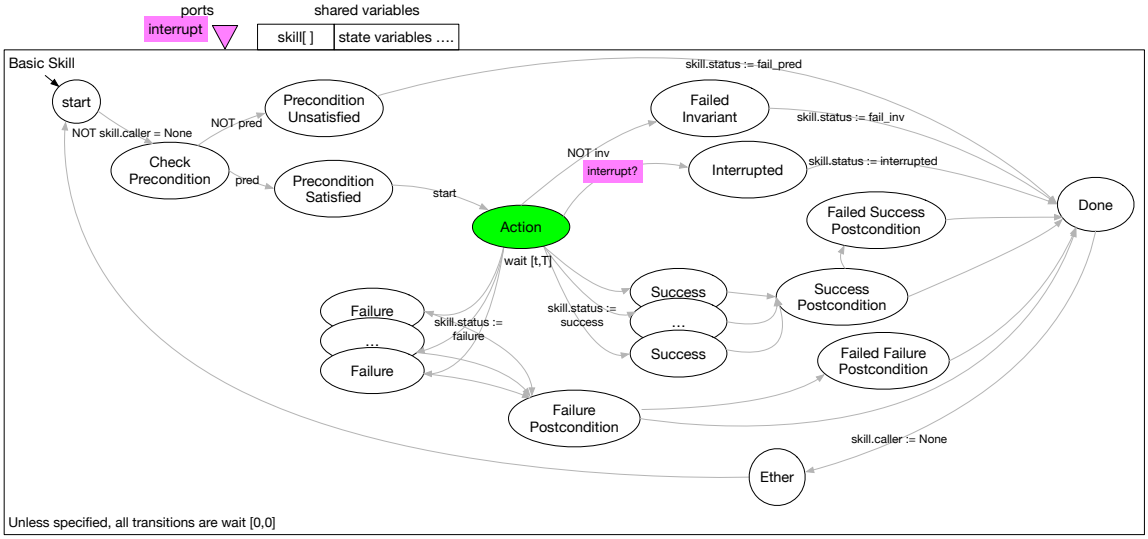

Figure 3: The Offline Verification Basic Skill Automata FIACRE model (TINA).

As expected, the only differences between these two models (FIACRE and H-FIACRE) is how they handle the execution in the *Action* state. In the offline version which will be explored by model checking, the *Action* state has a large `select` (line 11.26 which lists the various possible outcomes of the action execution:

- An external interruption from the interrupt port of the process,
- An invariant guard failure,
- All the possible successes and failures within the $[t, T]$ time interval specified to perform the action (e.g., line 3.12).

In the runtime verification version, the *Action* state leads to the call to the FIACRE task `action_task`, and then waits (`sync`) in the *Action Sync* state one of the possible outcomes:

- An external interruption from the interrupt port,
- An invariant guard failure,
- *Not_undershoot* and *Overshoot* states which will monitor in real-time these timing errors,

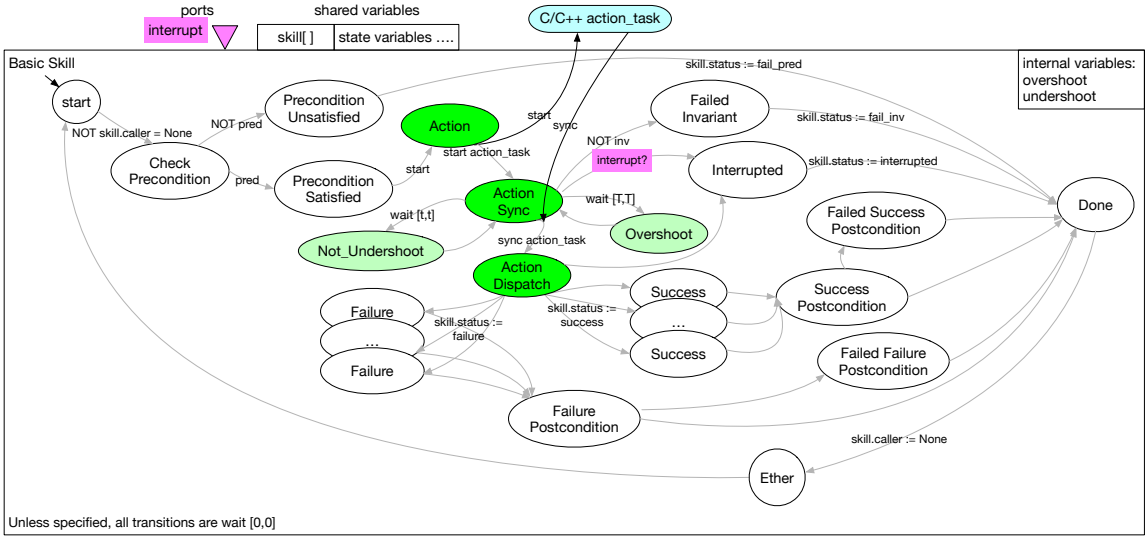

Figure 4: The Online Verification Basic Skill Automata H-FIACRE model (HIPPO).

- The return of the *Action* task call (with the `sync`) which makes a transition to the *Action Dispatch* state, which will dispatch to *Failure* or *Success* states according to the returned status and result.

Listing 11 shows a compact version of the offline **takeoff** basic skill with only two states, *idle* and *run*. Semantically, it is equivalent to the expanded version, but it is better suited for model checking as it removes some null time state transitions interleaving in the model. Note how a skill gets activated by guarding on its `caller skill` field not equal to `None` (line 11.11). Similarly, it gives control back to the caller by setting it back to `None` (line 11.56). Similar FIACRE code can be found in Listing 13 in appendix A which shows the complete H-FIACRE version of the **takeoff** basic skill.

Listing 11: FIACRE process specifying the **takeoff** basic (offline verification compact version) (Figure 5).

```
1  process skill_takeoff
2     [interrupt_takeoff: sync] // interruptible skills get this port
3     (&skill: skill_array, &flight_status: sv_flight_status, &target: sv_target,
4      &mission_status: sv_mission_status, &localization_status: sv_localization_status,
5      &motion: sv_motion, &battery: sv_battery, &camera: sv_camera) is
6
7  states idle, run
8
9  from idle
10    wait [0,0];
11    on (not (skill[takeoff].caller = None)); // a composite skill has called us
12    if (not (invariant_active(skill, flight_status, target, mission_status,
13            localization_status, motion, battery, camera))) and // invariant not propagating
14       ((motion = Free) and (flight_status = OnGround) and (battery = Good) and
15        (localization_status = Valid) and true) then // precondition satisfied
16       motion := Controlled; // start state_variable set
17       skill[takeoff].inv_active := true; // skill is active and its invariant monitored
18       skill[takeoff].status := no_status; // reset status
19       to run // go to the run state
20    else // the precondition is not satisfied
```

```
21          skill[takeoff].caller := None; // the call is not possible
22          to idle
23      end
24
25  from run // Action: (takeoff)
26      select // the wait interval values correspond to the values specified in the time_interval field
27          wait [1, 3]; // at_altitude success
28          skill[takeoff].val := takeoff_ret_val(takeoff_success_at_altitude);
29          motion := Free; // success effect
30          skill[takeoff].status := success // report success
31      []
32          wait [1, 3]; // grounded failure
33          skill[takeoff].val := takeoff_ret_val(takeoff_failure_grounded);
34          motion := Free; // failure effect
35          skill[takeoff].status := failure // report failure
36      []
37          wait [1, 3]; // emergency failure
38          skill[takeoff].val := takeoff_ret_val(takeoff_failure_emergency);
39          motion := Free; // failure effect
40          skill[takeoff].status := failure // report failure
41      []
42          interrupt_takeoff; // the interrupt event port sync
43          motion := Free;
44          skill[takeoff].status := interrupted // report interruption
45      []
46          wait [0,0];
47          on (not (motion = Controlled)); // invariant guard failed
48          skill[takeoff].status := failed_inv // report failed inv.
49      []
50          wait [0,0];
51          on (not (not (battery = Critical))); // invariant guard failed
52          motion := Free; // effects of the invariant failure
53          skill[takeoff].status := failed_inv
54      end;
55      skill[takeoff].inv_active := false;
56      skill[takeoff].caller := None; // return to idle state and inform the caller we are done
57      to idle
```

Figure 5 illustrates the compact version of this FIACRE **takeoff** basic skill process.

5.4 Composite skills FIACRE version

The mapping of the composite skills to FIACRE is also systematic and automatic. It follows the "program to automata" algorithm used in PRS [Ingrand et al., 1996]. We illustrate it with some examples withdrawn from the composite skill in Listing 4.

Skill call The call to **takeoff** on line 4.11 translates to:

```
1  from NS5
2      wait [0,0];
3      skill[takeoff].caller := uav_mission;
4  to NS5_NS4_sync
5
6  from NS5_NS4_sync
7      wait [0,0];
8      on (skill[takeoff].caller = None);
9  to NS4
```

We set the caller `field` of the `skill` array element for the **takeoff** index to the caller (**uav_mission**) (line 3). Note that as explained in the previous section, this will wake up the **takeoff** skill (line 11.11). Then we wait in the next state until **takeoff** is done (line 11.56) which triggers the guard (line 8).

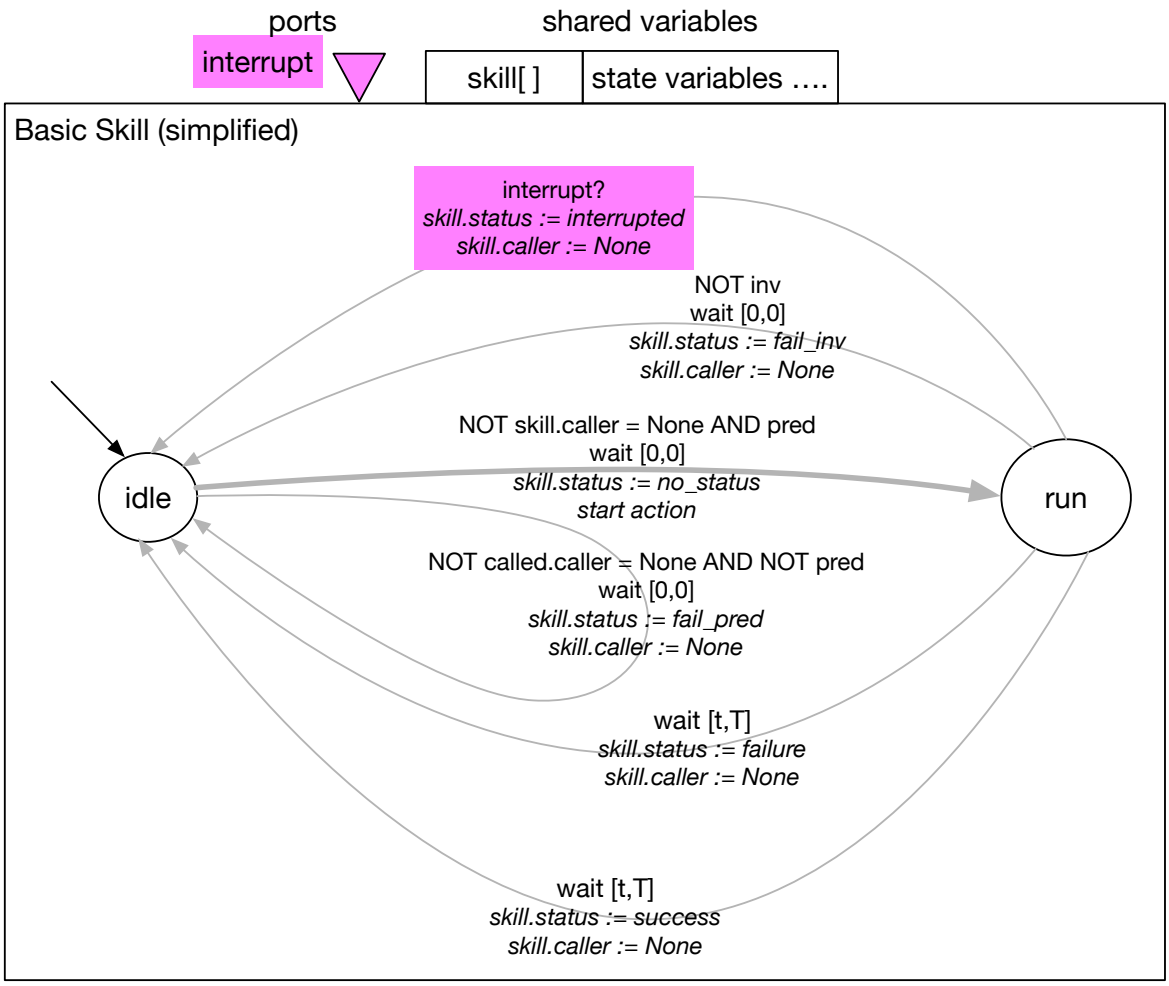

Figure 5: A more compact version of the online verification basic skill FIACRE process (Listing 11).

Sequence Sequences are straightforward, just progress to the next state in the FIACRE automata.

Test and branching The test on the **takeoff** success status on line 4.12 translates to:

```
1   from NS4
2      wait [0,0];
3      if (skill[takeoff].status = success) then
4          to N3_T
5      else
6          to NS8
7      end
```

`N3_T` and `NS8` correspond to the `true` and `false` branches following the test. In this example, there is no `else` branch in the skill, hence it jumps to the end of the `if`.

Parallel branches Parallel branches involve the synthesis of additional FIACRE processes (one for each branch). The initial executing skill or branch relinquishes the control to the parallel branches and waits until they all terminate. Hence the code line 4.13 get translated too:

```
1  from N3_T // Pass the parallel control...
2      wait [0,0];
3      skill[uav_mission_branch_1_0].caller := uav_mission; // ... to the camera_survey branch
4      skill[uav_mission_branch_1_1].caller := uav_the; // ... to the navigation branch
5  to N3_T_NS6_sync
6
7  from N3_T_NS6_sync // wait the control back from all branches
8      wait [0,0];
9      on ((skill[uav_mission_branch_1_0].caller = None) and
10         (skill[uav_mission_branch_1_1].caller = None));
11 to NS6
```

One can see that using the `skill` array element for the considered caller, the control can be passed from the main body to the branches (lines 3 and 4) and then wait for the control to come back to the main body when they are finished (lines 9 and 10).

B lists the four H-FIACRE processes synthesized to model the **uav_mission** (§ 3.4) composite skill. Listing 14 is the main process and handles the execution of the main part of the body, while Listing 15 and Listing 16 handle respectively the two parallel branches (one doing the **camera_survey**, while the other one does the navigation with the two **goto_waypoint**). Last, listing 17 presents the process in charge of checking the temporal over and under shooting of the skill.

5.5 Environment and final component

For the resulting formal model to be properly fully analyzed and run, one needs to connect it to the real environment (or a model of it). For the offline verification version, we synthesize a FIACRE process (e.g., Listing 12) which produces, at any time, all the PROSKILL events and interrupts, both modeled with FIACRE ports, present in the model (like the **clicker** process in the triple click detector example).

Listing 12: The FIACRE process modeling the environment, which synthesizes all the possible PROSKILL events and interrupts of the UAV experiment.

```
1  process basic_environment_drone // ports to produce all possible events and interrupts
2         [localization_status_to_invalid : sync, ..., flight_status_to_in_air_unsafe : sync,
3          battery_to_good : sync, battery_to_low : sync, battery_to_critical : sync,
4          interrupt_takeoff: sync,... ] is
5
6  states start_
7
8  from start_
9      select
10         localization_status_to_invalid
11     []
12         ... // ProSkill event ports sync removed for conciseness
13         flight_status_to_in_air_unsafe
14     []
15         battery_to_good
16     []
17         battery_to_low
18     []
19         battery_to_critical
20     []
21         interrupt_takeoff
```

```
22      []
23          ... // interrupt ports sync removed for conciseness
24
25      end;
26  to start_
```

This is perfect for model checking and to explore all the possible occurrences of asynchronous events and interrupts in the model. Yet, if the application environment follows a more restrictive pattern, then the user can modify this process which hopefully may lead to smaller (but no larger) reachable states set.

For the runtime version, we also provide a simple process handling the FIACRE ports in the model. But in this case, this environment process has two FIACRE event ports, one for PROSKILL events and one for interruptible skills (they can be interrupted by an external "signal"). These two FIACRE event ports are each linked to a C/C++ function which will appropriately trigger when these external events or interrupts are received on the real robot.

5.6 The final FIACRE component

We have presented all the processes which model the various PROSKILL objects. To create a complete FIACRE component, these processes must be instantiated, and their FIACRE ports properly connected. Thus, the final FIACRE component defines:

- The FIACRE variables definition including all the state variables used in the experiment and an array of a `skill` structure which stores the various information needed for each skill instance (section 5.3).
- The FIACRE ports definition for all the possible PROSKILL events and interrupts.
- The FIACRE process instances:
  - State variables automata, which control the allowed transitions for enumerated state variables.
  - Event effects, whose FIACRE event port will be connected to the Event and Interrupt environment processes (see Section 5.5).
  - Basic skills with an interrupt port, for each interruptible skill.
  - Composite skills
  - Composite skill temporal watchdog to monitor possible (reachable) overshoot or undershoot.
  - Composite skill parallel branches, if any.
  - Event and interrupt environment process for PROSKILL event.

5.7 Offline and online formal verification of PROSKILL programs

Figure 6 presents the workflow from the PROSKILL program to the executable version (top part), and the verifiable version and its analyzed properties report (bottom part). One should keep in mind, that the robotic programmers just write the PROSKILL program, the C/C++ code which glue basic skill actions and PROSKILL events to the real robot and the additional LTL properties to check (all in light blue on Figure 6), the rest is fully synthesized and compiled automatically.

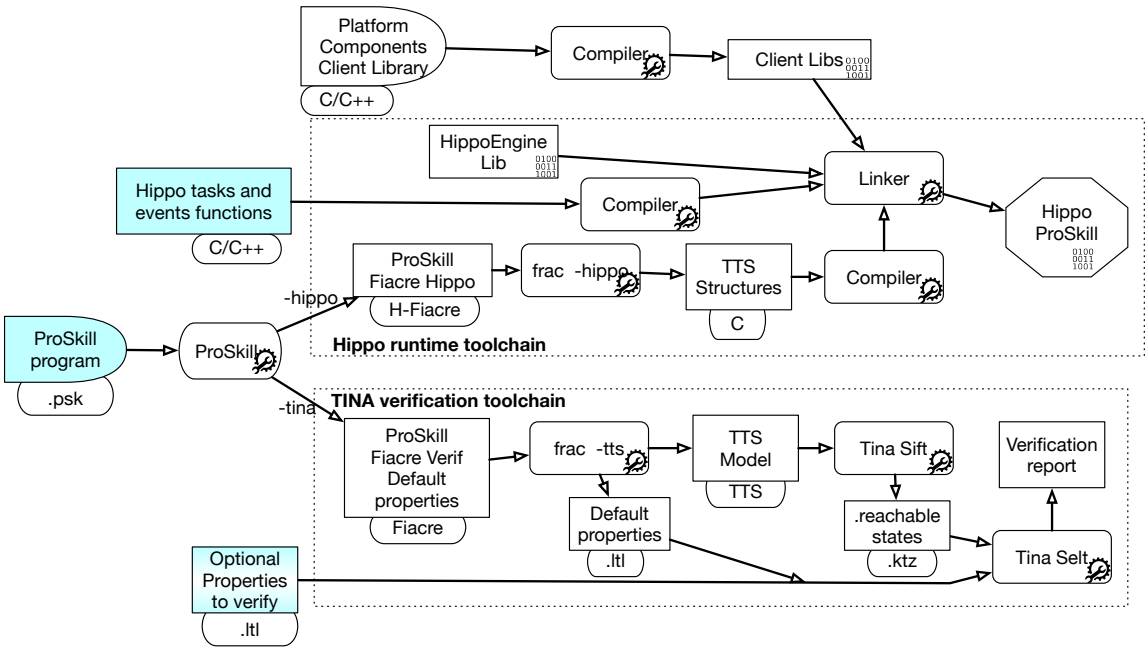

Figure 6: The FIACRE/HIPPO-TINA toolchain (only the data in the blue box need to be given, the rest is fully synthesized).

5.7.1 Offline formal verification

As shown on Figure 6, the offline verification is often done in two steps, one to synthesize the set of reachable states of the model (with sift), and one to check some properties in the mode (with selt).

When the FIACRE model for PROSKILL program is synthesized, a number of default properties are also synthesized and checked with selt.

- Check for each variable that there is no forbidden transition.
- For each basic skill that: it can run; it can succeed, for all successes; it can fail, for all failures; its invariant may fail; and it can be interrupted.
- For composite skill, we also add if it can undershoot or overshoot its time interval specification.

When these properties are checking that a particular state cannot be reached, if they are false, a counter example is given, to help the programmer to identify the problem and fix it.

5.7.2 Online (or runtime) verification

Unlike the offline verification which "model checks" the PROSKILL program, the runtime verification literally runs the same model. Thanks to the HIPPO engine, the TTS model is run, and commands are called, events are received, and the robot executes the mission it has been programmed to do in PROSKILL.

The H-FIACRE model already contains several tests which may lead to warnings and error messages at runtime:

- overshoot and undershoot skill execution,
- state variable illegal value transitions,
- command illegal returned value.

In the PROSKILL experiment, the HIPPO engine runs at 100 Hz, which is sufficient for the type of program and temporal constraints we handle.

Note that similarly to the **cheat_detector** monitor presented in Section 4.5, we can augment the H-FIACRE model with processes to monitor specific situations requiring actions.

6 An example: an UAV controller

We demonstrate our approach with an UAV for which we program in PROSKILL a survey mission. The functional layer of this experiment (Figure 7) has already been presented in [Dal Zilio et al., 2023], but suffice to say that it provides robust localization, navigation, flight control and allows us to command the drone. It is deployed using the GenoM specification language (which also maps in a formal framework to validate and verify the functional components) [Dal Zilio et al., 2023].

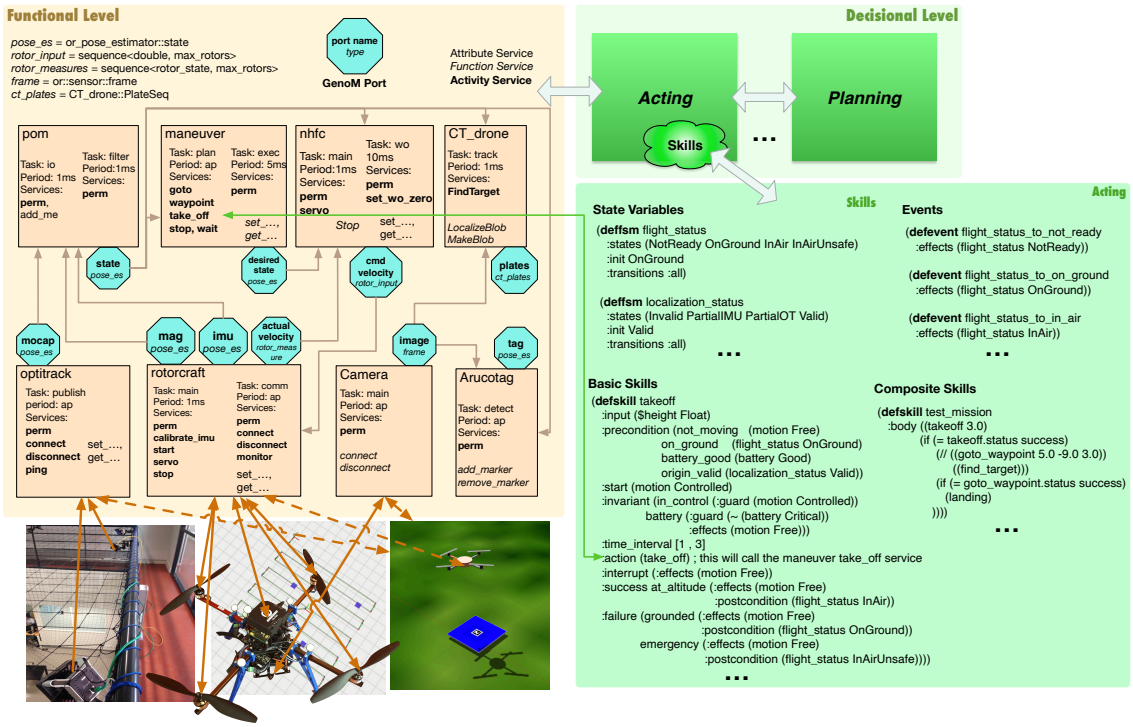

Figure 7: Architecture of the drone experiment.

Although eight functional components are required for this experiment, the set of primitive commands available for the *acting* component are **takeoff**, **landing**, **goto_waypoint**,

start_drone, **shutdown_drone**, and **camera_survey**. Each of them has a corresponding basic skill (like the one presented on Listing 3 for **takeoff**), to which we add a composite skill **uav_mission** (presented on Listing 4), and one skill to monitor for the *battery* level (Listing 5).

The state variables handled by the model are *flight_status* with values: NotReady, OnGround, InAir, InAirUnsafe and Lost; *target*: NotFound and Found; *mission_status*: Unknown, Ongoing, Failed and Succeeded; *localization_status*: Invalid, Coarse and Valid; *motion*: Free and Controlled; *battery*: Good, Low and Critical; and *camera*: On and Off.

Similarly, the available ProSkill external events are handling *localization_status* and *battery_status* updates.

The resulting Fiacre model has seven processes for the state variables, ten for external ProSkill events, six for basic skills, one for the monitoring skill, one for a composite skill, along two for its parallel branches (Listings 15 and 16) and one for a watchdog for its time interval of the composite skill (Listings 17), and one for the environment process. This amount to 1400 lines of Fiacre code, which compiles in a TTS with 97 places and 153 transitions.[8]

6.1 Offline verification results

Building the set of reachable states of the complete model of the whole ProSkill program takes 112 h 17 min on an AMD EPYC 7352 24-Core Processor (using one processor) and results in a set of reachable states which has 1 545 614 784 classes, 372 325 248 markings, 12 761 domains, 23 154 942 608 transitions. We are able to check all the default properties defined in section 5.7.1.

Some results are puzzling, but correct. For example, model checking shows that the invariant on the **landing** cannot fail. Indeed, checking the code, its invariant cannot be falsified in this particular setup. Similarly, the main composite skill can overshoot and undershoot. This can easily be explained as the wait for the *localization_status* to be valid is unbounded, hence it can take a priori any time.

All these results may indicate an error in the model, question some programming choices or confirm something expected. In any case, it offers to the programmer a complete exploration of the execution possibility to study and analyze.

Of course, state explosion is the limiting factor here, and we will explore some possible improvements to address it in Section 7.

6.2 Runtime verification results

We build a runtime version of the ProSkill specification and link it with the client libraries of the components which provide access to the sensors (i.e., events) and commands of the drone: pom for the *localization_status*; rotorcraft for *battery_status* but also the **start_drone** and **shutdown_drone** commands; camera for **camera_survey** and maneuver for **takeoff**, **landing** and **goto_waypoint**. The Hippo engine executes the TTS model at 100 Hz, it evaluates preconditions, invariants; tests conditions on state variables, or sets them when specified. It handles skill success and failure and detects runtime overshoot or undershoot skills execution.

[8] The resulting code can be found in the `examples` subdirectory of `git://redmine.laas.fr/laas/users/felix/proskill.git`.

An important aspect of running the model itself is to confirm that the execution result is consistent with what the programmer has in mind when he writes the ProSkill program. The ProSkill program becomes a formal model, and we run this model. We did several runs and showed that the drone acts as expected, surveys the area, and lands gracefully when its *battery* becomes Critical.

7 Limits, discussion, future work, and conclusion

7.1 Features and limits

ProSkill offers many of the desired features for an *acting* skill language. Skills can call each other with, if needed, a hierarchical organization. Resources management (not described here) is easily handled with resource variables (a test for availability and reservation is trivial to write in Fiacre, as all TTS transitions are atomically executed). Time management is provided, at the basic skill level, but also in the composite ones. Commands, events, and interruptions are properly handled too. This is another strong point of the ProSkill language: modeling nondeterminism (using `select`), raising from external events and from commands outcomes. Yet, the model remains predictable as all the execution paths resulting from this nondeterminism will be explored by model checking. At the end, the whole language can be mapped in a formal framework which opens a new realm for V&V of robotic systems.

Yet there are some limits to the expressiveness and power of the ProSkill language. Some of them are due to our approach that the language must map in Fiacre. For example, some basic types (e.g., float, string, etc.) are not currently handled by the Fiacre language. Therefore, this aspect of the ProSkill language cannot be model checked, nevertheless, these variables values can be made available at runtime and can even be involved in runtime verification.

Another manageable limitation is that the skills are not reentrant, and you cannot have skill with recursive calls (direct or indirect). In short there can only be one instance of each skill active at one time. But nothing prevents the programmer from creating as many instances of a particular skill as he may need.

If one considers most modern languages, those limits may seem extreme, but you have also to consider that often, plans given by automatic planner are already fully instantiated, and most variables are bound to a value. Overall, this is a classical tradeoff between the expressiveness of a language and its verifiability.

Uncertainty management is another area where ProSkill has little to offer. If we consider the *localization* in our drone example, we consider three discretized values Valid, Coarse, Invalid. Currently the covariance of the position of the drone (computed by the Kalman filter in pom) is checked and the proper event is synthesized accordingly. Similarly, Fiacre does not provide uncertainties on time intervals to model approximate time execution evaluation.

The ProSkill language does not offer fancy algorithmic constructs which are sometimes provided to ease the programming task. We want to keep the language as feature-limited as possible but complete enough to provide a powerful programming language. With the `if-then-else`, `while`, `do-until`, `goto`, `wait`, parallel branches execution and interrupt, we can write any of the complex programs we encountered. But the language remains open to any new constructs, assuming they cannot be programmed with the available ones and they can be mapped in Fiacre.

7.2 Discussion

The presented ProSkill framework has strong similarities with the SkiNet one [Albore et al., 2023], which also relies on the TTS formalism and the Tina toolbox. Yet their modeling choices and their runtime verification approach differ:

- Unlike ProSkill, they directly synthesize TTS (i.e., they skip the Fiacre language and the `frac` compilation steps).
- Even if TTS (an extension of Time Petri net) supports time, they do not currently model timing information in their basic and composite skills.
- They do not synthesize a controller which executes the ProSkill program as we do, instead they synthesize a controller which will monitor the regular program.

These choices can be discussed and there are pros and cons to each approach, yet the differences are sufficient to justify separate developments. Using Fiacre as an intermediate language has some strong advantages (legibility, expressivity, etc.), without any performance impact. Moreover, executing the program with the Hippo engine greatly improves the confidence that the initial ProSkill program does what the user wants, and providing we have a TTS player (Hippo), why not use it instead of writing another controller which needs to be monitored using the TTS model.

Note that we designed the ProSkill language, having in mind the automatic transformation to Fiacre. But if we consider again the systems presented in the state of the art in Section 2, other languages/systems can probably be automatically and unequivocally translated to Fiacre. Behavior trees and RMPL are probably suitable candidates for such mapping and this could lead to a valuable "formally verified" implementation.

Our previous work presented in [Dal Zilio et al., 2023] also produces a Fiacre model on which one can also perform offline and runtime verification. Although we could consider merging the two models and then make some verification on the joint model, it is unlikely to be effective considering we are already struggling to avoid state explosion. But for the runtime verification, this makes more sense, and we have run experiments where all the functional components are run with one Hippo engine running at 10 kHz along another engine executing the *acting* specifications at 100 Hz.

7.3 Future work

Although the current version of the language is fully operational, there are several improvements to consider:

- Extend the data type handled directly by Fiacre.[9]
- To avoid the reachable states explosion while model checking, we could consider a more abstract version of the offline verification model, yet we need to keep a faithful and complete version of the model for the runtime to avoid a semantics gap between what the programmer intended, and what is executed.
- We could consider performing some *planning* using the basic skills as an action model, and have a planner synthesize "composite skills",

[9] The Fiacre developper are working on adding rational numbers and strings types.

- Statistical Model Checking [Foughali et al., 2019] could be used to better explore the possible execution branches resulting from the FIACRE `select` instructions, based on probability/distribution obtained from regular or simulated runs.
- As we now have a formal model of the *acting* component, and of the functional components of the robot, we could consider making one of the middleware connecting them.
- Many autonomous robots are now deployed with humans present in the environment. To properly model their uncontrollable behaviors while they interact with the robots is quite challenging.

Nevertheless, all these improvements are extending the current implementation, while none requires a deep redesign of the approach with respect to V&V of autonomous robots.

7.4 Conclusion

Acting is a critical decisional functionality of autonomous systems such as robots. We propose PROSKILL, a new skill language to program the acting component of robots. The basic building blocks of the PROSKILL language are presented and so is the FIACRE formal framework. We then show how the PROSKILL primitives are mapped automatically and unambiguously in FIACRE. The obtained formal model can be used offline with model checking to verify logical and temporal properties, but also online for runtime verification while executing the program/model. The offline verification explores the model (for desirable or undesirable states, sequences, state variables values, etc.), while the runtime verification runs it and enforces it. We illustrate our integrated approach on a real platform: a drone executing a survey mission. The fact that the formal model is the one executing at run time ensures both its operational "accuracy" and enforce the use of formal tools from specifications to verification and runtime execution. This is a major step toward deploying V&V in the autonomous robot *acting* component and making it available to roboticists, even without any V&V background.

Acknowledgement

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

References

R. Alami, R. Chatilla, S. Fleury, M. Ghallab, and F. Ingrand. An Architecture for Autonomy. *International Journal of Robotics Research*, 17(4):315–337, 1998. doi:10.1177/027836499801700402.

A. Albore, D. Doose, C. Grand, J. Guiochet, C. Lesire, and A. Manecy. Skill-based design of dependable robotic architectures. *Robotics and Autonomous Systems*, 160:104318, 2023. ISSN 0921-8890. doi:10.1016/j.robot.2022.104318.

M. Barbier, J.-F. Gabard, D. Vizcaino, and O. Bonnet-Torrès. Procosa: a software package for autonomous system supervision. In *National Workshop on Control Architectures of Robots*, pages 37–47, 2006. URL https://www.lirmm.fr/gtcar/webcar/CAR2006/papers/ONERA-CERT.pdf.

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.

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.

B. Berthomieu, S. dal Zilio, and F. Vernadat. A FIACRE v3.0 primer, 2020. URL https://projects.laas.fr/fiacre/doc/primer.pdf.

D. Bjørner and K. Havelund. 40 Years of Formal Methods - Some Obstacles and Some Possibilities? *Formal Methods*, 2014. doi:10.1007/978-3-319-06410-9_4.

J. Bohren and S. Cousins. The SMACH High-Level Executive. *IEEE Robotics and Automation Magazine*, 17(4):18–20, Dec. 2010. doi:10.1109/MRA.2010.938836.

F. Boussinot and R. de Simone. The ESTEREL Language. *Proceeding of the IEEE*, 79(9): 1293–1304, Sept. 1991. doi:10.1109/5.97299.

R. I. Brafman, M. Bar-Sinai, and M. Ashkenazi. Performance level profiles - A formal language for describing the expected performance of functional modules. *Proceedings of the Conference on Intelligent Robots and Systems*, 2016. doi:10.1109/IROS.2016.7759280.

S. G. Brunner, F. Steinmetz, R. Belder, and A. Dömel. RAFCON: A graphical tool for engineering complex, robotic tasks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3283–3290, 2016. ISBN 978-1-5090-3762-9. doi:10.1109/IROS.2016.7759506.

A. Cavalcanti. Formal Methods for Robotics: RoboChart, RoboSim, and More. In *Formal Methods: Foundations and Applications*, pages 3–6, Cham, Nov. 2017. Springer International Publishing. ISBN 978-3-319-70848-5. doi:10.1007/978-3-319-70848-5_1.

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.

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

E. Coste-Maniere, B. Espiau, and E. Rutten. A task-level robot programming language and its reactive execution. In *IEEE International Conference on Robotics and Automation*, 1992. doi:10.1109/ROBOT.1992.219990.

H. Costelha and P. U. Lima. Robot task plan representation by Petri nets: modelling, identification, analysis and execution. *Autonomous Robots*, 33(4):337–360, Mar. 2012. doi:10.1007/s10514-012-9288-x.

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. ISSN 0921-8890. doi:10.1016/j.robot.2022.104301. URL https://hal.laas.fr/hal-03683044.

A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. UPPAAL SMC tutorial. *International Journal on Software Tools for Technology Transfer*, pages 1–19, Apr. 2015. doi:10.1007/s10009-014-0361-y. URL http://dx.doi.org/10.1007/s10009-014-0361-y.

R. W. De Araújo and A. A. D. de Medeiros. Verification of Procedural Reasoning System (PRS) Programs Using Coloured Petri Nets (CPN). In *Artificial Intelligence Applications and Innovations: IFIP 18th World Computer Congress*, pages 421–433. Springer, 2004. doi:10.1007/1-4020-8151-0_36.

L. de Silva, F. Meneguzzi, and B. Logan. An Operational Semantics for a Fragment of PRS. In *International Joint Conference on Artificial Intelligence*, pages 1–8, July 2018. doi:10.24963/ijcai.2018/27.

O. Despouys and F. Ingrand. Propice-Plan: Toward a Unified Framework for Planning and Execution. In *European Workshop on Planning*, 1999. doi:10.1007/10720246_22.

G. Dowek, C. Muñoz, and C. S. Pasareanu. A Formal Analysis Framework for PLEXIL. In *Workshop on Planning and Plan Execution for Real-World Systems*, pages 1–7, Sept. 2007. URL https://shemesh.larc.nasa.gov/people/cam/PLEXIL/.

T. Eckstein and G. Steinbauer. Action-based programming with YAGI - an update on usability and performance. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 557–569. Springer, 2020. doi:10.1007/978-3-030-55789-8_48.

B. Espiau, K. Kapellos, and M. Jourdan. Formal verification in robotics: Why and how? In G. Giralt and G. Hirzinger, editors, *International Symposium on Robotics Research*, 1996. doi:10.1007/978-1-4471-1021-7_26.

M. Figat and C. Zieliński. Parameterised robotic system meta-model expressed by Hierarchical Petri nets. *Robotics and Autonomous Systems*, 150:103987, 2022. doi:10.1016/j.robot.2021.103987.

A. Finzi, F. Ingrand, and N. Muscettola. Model-based Executive Control through Reactive Planning for Autonomous Rovers. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004. doi:10.1109/IROS.2004.1389463.

R. J. Firby. An investigation into reactive planning in complex domains. In *AAAI Conference*. Seattle, WA, 1987. URL http://www.aaai.org/Papers/AAAI/1987/AAAI87-036.pdf.

M. Foughali, F. Ingrand, and C. Seceleanu. Statistical Model Checking of Complex Robotic Systems. In *International SPIN Symposium on Model Checking of Software*, 2019. doi:10.1007/978-3-030-30923-7_7.

J. Frank and A. K. Jónsson. Constraint-Based Attribute and Interval Planning. *Constraints*, 8(4), 2003. doi:10.1023/A:1025842019552.

M. Georgeff and F. Ingrand. Monitoring and Control of Spacecraft Systems Using Procedural Reasoning. In *Proceedings of the Space Operations-Automation and Robotics Workshop*, 1989. URL https://hal.laas.fr/hal-01981584.

M. Ghallab. On Chronicles: Representation, On-line Recognition and Learning. In *Knowledge Representation and Reasoning*, pages 597–606, 1996. doi:10.5555/3087368.3087439.

M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun, and D. Weld. Pddl - the planning domain definition language. Technical report, AIPS, 08 1998.

M. Ghallab, D. S. Nau, and P. Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016. doi:10.1017/CBO9781139583923. URL https://projects.laas.fr/planning/.

D. Hähnel, W. Burgard, and G. Lakemeyer. GOLEX — bridging the gap between logic (GOLOG) and a real robot. In *KI Advances in Artificial Intelligence*, pages 165–176. Springer, 1998. doi:10.1007/BFb0095437.

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.

D. Höller, G. Behnke, P. Bercher, S. Biundo, H. Fiorino, D. Pellier, and R. Alford. Hddl: An extension to pddl for expressing hierarchical planning problems. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 9883–9891, 2020. doi:10.1609/aaai.v34i06.6542.

M. D. Ingham, R. J. Ragno, and B. C. Williams. A Reactive Model-based Programming Language for Robotic Space Explorers. In *International Symposium on Artificial Intelligence, Robotics and Automation for Space*, 2001. URL http://robotics.estec.esa.int/i-SAIRAS/isairas2001/papers/Paper_AM053.pdf.

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.

F. Ingrand, R. Chatilla, R. Alami, and F. Robert. PRS: a high level supervision and control language for autonomous mobile robots. In *IEEE International Conference on Robotics and Automation*, pages 43–49, 1996. doi:10.1109/ROBOT.1996.503571.

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

L. P. Kaelbling. Goals as parallel program specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Minneapolis-St. Paul, Minnesota, 1988. URL https://www.aaai.org/Papers/AAAI/1988/AAAI88-011.pdf.

L. P. Kaelbling and N. J. Wilson. Rex programmer's manual. Technical Report 381, Artificial Intelligence Center, SRI International, Menlo Park, California, 1988. URL https://apps.dtic.mil/sti/pdfs/ADA461661.pdf.

M. Kim and K. C. Kang. Formal Construction and Verification of Home Service Robots: A Case Study. In *Automated Technology for Verification and Analysis*, pages 429–443, Berlin, Heidelberg, 2005. Springer. ISBN 978-3-540-31969-6. doi:10.1007/11562948_32.

M. Klotzbücher and H. Bruyninckx. Coordinating Robotic Tasks and Systems with rFSM Statecharts. *Journal of Software Engineering for Robotics*, 2012. URL https://lirias.kuleuven.be/handle/123456789/369166.

A. Kovalchuk, S. Shekhar, and R. I. Brafman. Verifying Plans and Scripts for Robotics Tasks Using Performance Level Profiles. In *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021*, pages 673–681. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/ICAPS/article/view/16016.

C. Lesire and F. Pommereau. ASPiC: an Acting system based on Skill Petri net Composition. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1–7, Sept. 2018. doi:10.1109/IROS.2018.8594328.

V. Mataré, T. Viehmann, T. Hofmann, G. Lakemeyer, A. Ferrein, and S. Schiffer. Portable high-level agent programming with golog++. In *International Conference on Agents and Artificial Intelligence*, 2021. doi:10.5220/0010253902180227. URL https://www.scitepress.org/Papers/2021/102539/102539.pdf.

B. McClelland, D. Tellier, M. Millman, K. Go, A. Balayan, M. Munje, K. Dewey, N. Ho, K. Havelund, and M. D. Ingham. Towards a systems programming language designed for hierarchical state machines. In *IEEE International Conference on Space Mission Challenges for Information Technology,(SMC-IT 2021)*, pages 23–30, 2021. doi:10.1109/SMC-IT51442.2021.00010.

C. McGann, F. Py, K. Rajan, R. Henthorn, and R. McEwen. A deliberative architecture for AUV control. In *IEEE International Conference on Robotics and Automation*, pages 1049–1054, 2008. doi:10.1109/ROBOT.2008.4543343.

A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede. A Survey on Domain-Specific Modeling and Languages in Robotics. *Journal of Software Engineering for Robotics*, 7(1): 1–25, July 2016. doi:10.6092/JOSER_2016_07_01_p75.

B. Pelletier, C. Lesire, C. Grand, D. Doose, and M. Rognant. Predictive runtime verification of skill-based robotic systems using petri nets. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10580–10586. IEEE, 2023. doi:10.1109/ICRA48891.2023.10160434.

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.

R. G. Simmons and D. Apfelbaum. A task description language for robot control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1931–1937 vol.3, 1998. doi:10.1109/IROS.1998.724883.

R. G. Simmons and C. Pecheur. Automating Model Checking for Autonomous Systems. In *AAAI Spring Symposium on Real-Time Autonomous Systems*, Mar. 2000. URL https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.7359.

D. E. Smith, J. Frank, and W. Cushing. The ANML Language. In *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 2008. URL http://ti.arc.nasa.gov/m/profile/de2smith/publications/ICAPS08-ANML.pdf.

J. Turi. *Planning from operational models for deliberate acting in robotics*. PhD thesis, University of Toulouse, February 2024.

J. Turi, A. Bit-Monnot, and F. Ingrand. Enhancing Operational Deliberation in a Refinement Acting Engine with Continuous Planning. In *ICAPS, WS Integrated Acting, Planning and Execution (IntEx)*, Prague, Czech Republic, July 2023. URL https://hal.science/hal-04107355.

V. Verma, A. K. Jónsson, C. Pasareanu, and M. Iatauro. Universal executive and PLEXIL: engine and language for robust spacecraft control and operations. In *American Institute of Aeronautics and Astronautics Space*. AIAA Space Conference, 2006. doi:10.2514/6.2006-7449.

D. E. Wilkins and K. L. Myers. A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation*, 5(6):731–761, December 1995. doi:10.1093/logcom/5.6.731.

B. C. Williams and M. D. Ingham. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. *Proc. of the IEEE: Special Issue on Modeling and Design of Embedded Software*, 91(1):212–237, 2003. doi:10.1109/JPROC.2002.805828.

A Complete H-Fiacre process for a basic skill

Note: the H-Fiacre code presented in these appendices is automatically synthesized from ProSkill specifications. As a result, some code may seem cumbersome (e.g., `ignoreb` variables not used) or unnecessary (e.g., `if (true) then ... else ... end`, or `from x wait[0,0]; to y`). The frac compiler keeps or simplifies these constructs as needed.

Listing 13: The H-Fiacre process specification of the **takeoff** basic skill.

```
process skill_takeoff
    [interrupt_takeoff: sync]
    (&skill: skill_array, &flight_status: sv_flight_status, &target: sv_target,
     &mission_status: sv_mission_status, &localization_status: sv_localization_status,
     &motion: sv_motion, &battery: sv_battery, &camera: sv_camera) is

states start_, check_precondition, precondition_satisfied, precondition_unsatisfied,
    action, action_sync, action_dispatch, error, action_sync_overshoot,
    action_sync_not_undershoot, interrupted, failed_invariant, failure_grounded,
    check_failure_postcondition_grounded, failed_failure_postcondition_grounded,
    failure_emergency, check_failure_postcondition_emergency,
    failed_failure_postcondition_emergency, success_at_altitude,
    check_success_postcondition_at_altitude, failed_success_postcondition_at_altitude, done,
    ether

var ignoreb: bool, ret_val: takeoff_ret_val_type, overshoot, undershoot : bool

from start_
    wait [0,0];
    on (not (skill[takeoff].caller = None));
    skill[takeoff].status := no_status;
    // Skill 'takeoff' has been called
to check_precondition

from check_precondition
    wait [0,0];
    on (not (invariant_active(skill, flight_status, target, mission_status, localization_status, motion,
         battery, camera)));
    if ((motion = Free) and (flight_status = OnGround) and (battery = Good) and (localization_status =
         Valid) and true) then
        to precondition_satisfied
    else
        to precondition_unsatisfied
    end

from precondition_unsatisfied
    wait [0,0];
    // Skill 'takeoff' precondition ((motion = Free) and (flight_status = OnGround) and
    // (battery = Good) and (localization_status = Valid) and true) UNsatisfied
to done

from precondition_satisfied
    wait [0,0];
    // Skill 'takeoff' precondition ((motion = Free) and (flight_status = OnGround) and
    // (battery = Good) and (localization_status = Valid) and true) is satisfied
    motion := Controlled;
    skill[takeoff].inv_active := true;
to action

from action // Action: (takeoff)
    // Skill 'takeoff' calling its action (start task)
    overshoot := false;
    undershoot := true;
    start Fiacre_takeoff_action_task (skill[takeoff]);
to action_sync

```

```
from action_sync
    select
        sync Fiacre_takeoff_action_task ret_val;
        // Skill 'takeoff' action returned (sync task)
        if undershoot then
            // WARNING: Action 'takeoff' undershoot 1 s (100 ticks)
            null
        end;
        to action_dispatch
    []// undershoot
        on undershoot;
        wait [100, 100];
        to action_sync_not_undershoot
    []// overshoot
        on not undershoot and not overshoot;
        wait [200, 200]; // 200 because we already waited 100 for not_undershoot
        // WARNING: Action 'takeoff' overshoot 3 s (300 ticks)
        to action_sync_overshoot
    []//invariant
        wait [0,0];
        on (not (motion = Controlled));
        // Skill 'takeoff' failed invariant in_control: (motion = Controlled)
        skill[takeoff].val := takeoff_ret_val(takeoff_failed_inv_in_control);
        to failed_invariant
    []//invariant
        wait [0,0];
        on (not (not (battery = Critical)));
        // Skill 'takeoff' failed invariant battery: (not (battery = Critical))
        motion := Free;
        // Skill 'takeoff' failed invariant battery concluding effects: motion := Free;
        skill[takeoff].val := takeoff_ret_val(takeoff_failed_inv_battery);
        to failed_invariant
    []// we got an interrupt from the event port...
        interrupt_takeoff;
        motion := Free;
        // Skill 'takeoff' interrupted from event port
        // Skill 'takeoff' is interrupting its own action takeoff
        ignoreb := Fiacre_takeoff_interrupt_action();
        to interrupted
    end

from action_sync_overshoot
    overshoot := true;
to action_sync

from action_sync_not_undershoot
    undershoot := false;
to action_sync

from action_dispatch
    wait [0,0];
    if (ret_val = takeoff_success_at_altitude) then
        to success_at_altitude
    end;
    if (ret_val = takeoff_failure_grounded) then
        to failure_grounded
    end;
    if (ret_val = takeoff_failure_emergency) then
        to failure_emergency
    end;
    if (ret_val = takeoff_interrupted) then
        skill[takeoff].val := takeoff_ret_val(takeoff_interrupted);
        // Skill 'takeoff' has been interrupted while executing its action.
        to interrupted
    end;
to error // a priori unreachable
```

```
121
122 from interrupted
123     wait [0,0];
124     skill[takeoff].inv_active := false;
125     skill[takeoff].status := interrupted;
126 to done
127
128 from failed_invariant
129     wait [0,0];
130     skill[takeoff].inv_active := false;
131     skill[takeoff].status := failed_inv;
132 to done
133
134 from success_at_altitude
135     wait [0,0];
136     skill[takeoff].val := takeoff_ret_val(takeoff_success_at_altitude);
137     // Skill 'takeoff' success 'at_altitude'
138     skill[takeoff].inv_active := false;
139     motion := Free;
140 to check_success_postcondition_at_altitude
141
142 from check_success_postcondition_at_altitude
143     wait [0,0];
144     if (flight_status = InAir) then
145         skill[takeoff].status := success;
146         to done
147     else
148         // Skill 'takeoff' failed success_postcondition 'at_altitude': (flight_status = InAir)
149         to failed_success_postcondition_at_altitude
150     end
151
152 from failed_success_postcondition_at_altitude
153     wait [0,0];
154     skill[takeoff].status := success;
155 to done
156
157 from failure_grounded
158     wait [0,0];
159     skill[takeoff].val := takeoff_ret_val(takeoff_failure_grounded);
160     // Skill 'takeoff' failure 'grounded'
161     skill[takeoff].inv_active := false;
162     motion := Free;
163 to check_failure_postcondition_grounded
164
165 from check_failure_postcondition_grounded
166     wait [0,0];
167     if (flight_status = OnGround) then
168         skill[takeoff].status := failure;
169         to done
170     else
171         // Skill 'takeoff' failed failure_postcondition 'grounded': (flight_status = OnGround)
172         to failed_failure_postcondition_grounded
173     end
174
175 from failed_failure_postcondition_grounded
176     wait [0,0];
177     skill[takeoff].status := failure;
178 to done
179
180 from failure_emergency
181     wait [0,0];
182     skill[takeoff].val := takeoff_ret_val(takeoff_failure_emergency);
183     // Skill 'takeoff' failure 'emergency'
184     skill[takeoff].inv_active := false;
185     motion := Free;
186 to check_failure_postcondition_emergency
```

```
187
188 from check_failure_postcondition_emergency
189     wait [0,0];
190     if (flight_status = InAirUnsafe) then
191         skill[takeoff].status := failure;
192         to done
193     else
194         // Skill 'takeoff' failed failure_postcondition 'emergency': (flight_status = InAirUnsafe)
195         to failed_failure_postcondition_emergency
196     end
197
198 from failed_failure_postcondition_emergency
199     wait [0,0];
200     skill[takeoff].status := failure;
201 to done
202
203 from done
204     wait [0,0];
205     skill[takeoff].caller := None;
206     // Skill 'takeoff' returning control to caller and back to 'ether'
207 to ether
208
209 from error
210     wait [0,0];
211     // Skill 'takeoff' has an error in its model, check the returned values from real actions
212 to ether
213
214 from ether
215     wait [0,0];
216 to start_
```

B Complete H-FIACRE processes for a composite skill

Listing 14: The H-FIACRE main process of the **uav_mission** composite skill.

```
1  process skill_uav_mission
2      (&skill: skill_array, &flight_status: sv_flight_status, &target: sv_target,
3       &mission_status: sv_mission_status, &localization_status: sv_localization_status,
4       &motion: sv_motion, &battery: sv_battery, &camera: sv_camera) is
5
6  states start_, check_precondition, precondition_satisfied, precondition_unsatisfied,
7      body_branch, NS1, NS1_NS3_sync, NS3, NS5, NS5_NS4_sync, NS4, N3_T, N3_T_NS6_sync,
8      NS8, NS6, NS7, NS9, NS10, N2_T, N2_T_NS11_sync, NS12, NS11, N1_T, N1_T_NS13_sync,
9      NS14, NS13, NS15, failure_mission_failed, success_mission_accomplished, done, ether
10
11 var ignoreb: bool
12
13 from start_
14     wait [0,0];
15     on (not (skill[uav_mission].caller = None));
16     skill[uav_mission].status := no_status;
17     // Skill 'uav_mission' has been called
18 to check_precondition
19
20 from check_precondition
21     wait [0,0];
22     on (not (invariant_active(skill, flight_status, target, mission_status, localization_status, motion,
           battery, camera)));
23     if (true) then
24         to precondition_satisfied
25     else
26         to precondition_unsatisfied
27     end
28
```

```
from precondition_unsatisfied
    wait [0,0];
    // Skill 'uav_mission' precondition (true) UNsatisfied
to done

from precondition_satisfied
    wait [0,0];
    // Skill 'uav_mission' precondition (true) is satisfied
    mission_status := Ongoing;
to body_branch

from body_branch
    wait [0,0];
to NS1 //

// from: NS1 to: NS3 type: ET_GOAL expr: (start_drone)
from NS1
    wait [0,0];
    // Pass the control to (start_drone)
    // Skill 'uav_mission' calling (start_drone)
    skill[start_drone].caller := uav_mission;
    // synthesized action arg index 0
    skill[start_drone].ArgIndex := 0;
to NS1_NS3_sync

from NS1_NS3_sync
    wait [0,0];
    // Wait the control back from (start_drone)
    on (skill[start_drone].caller = None);
    // Skill 'uav_mission' call to (start_drone) returned
to NS3

// from: NS3 to: NS5 type: ET_GOAL expr: (^ (localization_status Valid))
from NS3
    // Wait on state variable value (localization_status Valid)
    wait [0,0];
    // Skill 'uav_mission' waited on condition (localization_status = Valid)
    on (localization_status = Valid);
to NS5

// from: NS5 to: NS4 type: ET_GOAL expr: (takeoff height 3.0 duration 0)
from NS5
    wait [0,0];
    // Pass the control to (takeoff height 3.0 duration 0)
    // Skill 'uav_mission' calling (takeoff height 3.0 duration 0)
    skill[takeoff].caller := uav_mission;
    // synthesized action arg index 1
    skill[takeoff].ArgIndex := 1;
to NS5_NS4_sync

from NS5_NS4_sync
    wait [0,0];
    // Wait the control back from (takeoff height 3.0 duration 0)
    on (skill[takeoff].caller = None);
    // Skill 'uav_mission' call to (takeoff height 3.0 duration 0) returned
to NS4

// from: NS4 to: N3 type: ET_IF expr: (= takeoff.status success)
from NS4
    wait [0,0];
    // Test on (= takeoff.status success)
    // Skill 'uav_mission' testing expression (= takeoff.status success)
    if (skill[takeoff].status = success) then
        // Skill 'uav_mission' expression (= takeoff.status success) is TRUE
        to N3_T
    else
```

```
95          // Skill 'uav_mission' expression (= takeoff.status success) is FALSE
96          to NS8
97      end
98
99  // splitting on N3_T, level 1, index 0
100 from N3_T
101     wait [0,0];
102     // Pass the parallel control
103     skill[uav_mission_branch_1_0].caller := uav_mission;
104     skill[uav_mission_branch_1_1].caller := uav_mission;
105 to N3_T_NS6_sync
106
107 from N3_T_NS6_sync
108     wait [0,0];
109     // wait the control back from all // branches
110     on (
111         (skill[uav_mission_branch_1_0].caller = None) and
112         (skill[uav_mission_branch_1_1].caller = None) and
113         true);
114 to NS6
115
116 // from: NS8 to: NS7 type: ET_GOAL expr:
117 from NS8
118     wait [0,0];
119 to NS7
120
121 // from: NS6 to: NS9 type: ET_GOAL expr:
122 from NS6
123     wait [0,0];
124 to NS9
125
126 // from: NS7 to: NS10 type: ET_GOAL expr: (printf "Mission failed")
127 from NS7
128     wait [0,0];
129     // will print user trace: "Mission failed"
130     // Skill 'uav_mission' prints a user message
131 to NS10
132
133 // from: NS9 to: N2 type: ET_IF expr: (= goto_waypoint.status success)
134 from NS9
135     wait [0,0];
136     // Test on (= goto_waypoint.status success)
137     // Skill 'uav_mission' testing expression (= goto_waypoint.status success)
138     if (skill[goto_waypoint].status = success) then
139         // Skill 'uav_mission' expression (= goto_waypoint.status success) is TRUE
140         to N2_T
141     else
142         // Skill 'uav_mission' expression (= goto_waypoint.status success) is FALSE
143         to NS12
144     end
145
146 // from: NS10 to: NS38 type: ET_GOAL expr: (failure mission_failed)
147 from NS10
148     wait [0,0];
149     // Skill 'uav_mission' is done executing and returns with failure mission_failed
150 to failure_mission_failed
151
152 // from: N2_T to: NS11 type: ET_GOAL expr: (landing)
153 from N2_T
154     wait [0,0];
155     // Pass the control to (landing)
156     // Skill 'uav_mission' calling (landing)
157     skill[landing].caller := uav_mission;
158     // synthesized action arg index 0
159     skill[landing].ArgIndex := 0;
160 to N2_T_NS11_sync
```

```
161
162 from N2_T_NS11_sync
163     wait [0,0];
164     // Wait the control back from (landing)
165     on (skill[landing].caller = None);
166     // Skill 'uav_mission' call to (landing) returned
167 to NS11
168
169 // from: NS12 to: NS8 type: ET_GOAL expr:
170 from NS12
171     wait [0,0];
172 to NS8
173
174 // from: NS11 to: N1 type: ET_IF expr: (= landing.status success)
175 from NS11
176     wait [0,0];
177     // Test on (= landing.status success)
178     // Skill 'uav_mission' testing expression (= landing.status success)
179     if (skill[landing].status = success) then
180         // Skill 'uav_mission' expression (= landing.status success) is TRUE
181         to N1_T
182     else
183         // Skill 'uav_mission' expression (= landing.status success) is FALSE
184         to NS14
185     end
186
187 // from: N1_T to: NS13 type: ET_GOAL expr: (shutdown_drone)
188 from N1_T
189     wait [0,0];
190     // Pass the control to (shutdown_drone)
191     // Skill 'uav_mission' calling (shutdown_drone)
192     skill[shutdown_drone].caller := uav_mission;
193     // synthesized action arg index 0
194     skill[shutdown_drone].ArgIndex := 0;
195 to N1_T_NS13_sync
196
197 from N1_T_NS13_sync
198     wait [0,0];
199     // Wait the control back from (shutdown_drone)
200     on (skill[shutdown_drone].caller = None);
201     // Skill 'uav_mission' call to (shutdown_drone) returned
202 to NS13
203
204 // from: NS14 to: NS12 type: ET_GOAL expr:
205 from NS14
206     wait [0,0];
207 to NS12
208
209 // from: NS13 to: NS15 type: ET_GOAL expr: (printf "Mission Accomplished")
210 from NS13
211     wait [0,0];
212     // will print user trace: "Mission Accomplished"
213     // Skill 'uav_mission' prints a user message
214 to NS15
215
216 // from: NS15 to: NS14 type: ET_GOAL expr: (success mission_accomplished)
217 from NS26
218     wait [0,0];
219     // Skill 'uav_mission' is done executing and returns with success mission_accomplished
220 to success_mission_accomplished
221
222 from success_mission_accomplished
223     wait [0,0];
224     skill[uav_mission].val := uav_mission_ret_val(uav_mission_success_mission_accomplished);
225     // Skill 'uav_mission' success 'mission_accomplished'
226     mission_status := Succeeded;
```

```
227     skill[uav_mission].status := success;
228 to done
229
230 from failure_mission_failed
231     wait [0,0];
232     skill[uav_mission].val := uav_mission_ret_val(uav_mission_failure_mission_failed);
233     // Skill 'uav_mission' failure 'mission_failed'
234     mission_status := Failed;
235     skill[uav_mission].status := failure;
236 to done
237
238 from done
239     wait [0,0];
240     skill[uav_mission].caller := None;
241     // Skill 'uav_mission' returning control to caller and back to 'ether'
242 to ether
```

Listing 15: The H-Fiacre process (first parallel branch) of the **uav_mission**.

```
1  process skill_branch_uav_mission_1_0
2       (&skill: skill_array, &flight_status: sv_flight_status, &target: sv_target,
3        &mission_status: sv_mission_status, &localization_status: sv_localization_status,
4        &motion: sv_motion, &battery: sv_battery, &camera: sv_camera) is
5
6  states start_, body_branch, N3_T, N3_T_NS6_sync, NS6, done, ether
7
8  var ignoreb: bool
9
10 from start_
11     wait [0,0];
12     on (not (skill[uav_mission_branch_1_0].caller = None));
13 to body_branch
14
15 from body_branch
16     wait [0,0];
17 to N3_T //
18
19 // from: N3_T to: NS6 type: ET_GOAL expr: (camera_tracking)
20 from N3_T
21     wait [0,0];
22     // Pass the control to (camera_tracking)
23     // Skill 'uav_mission' calling (camera_tracking)
24     skill[camera_tracking].caller := uav_mission_branch_1_0;
25     // synthesized action arg index 0
26     skill[camera_tracking].ArgIndex := 0;
27 to N3_T_NS6_sync
28
29 from N3_T_NS6_sync
30     wait [0,0];
31     // Wait the control back from (camera_tracking)
32     on (skill[camera_tracking].caller = None);
33     // Skill 'uav_mission' call to (camera_tracking) returned
34 to NS6
35
36 // Joining on NS6, this branch is done
37 from NS6 // join
38     wait [0,0];
39 to done
40
41 from done
42     wait [0,0];
43     skill[uav_mission_branch_1_0].caller := None;
44 to ether
45
46 from ether
47     wait [0,0];
48 to start_
```

Listing 16: The H-Fiacre process (second parallel branch) of the **uav_mission**.

```
process skill_branch_uav_mission_1_1
     (&skill: skill_array, &flight_status: sv_flight_status, &target: sv_target,
      &mission_status: sv_mission_status, &localization_status: sv_localization_status,
      &motion: sv_motion, &battery: sv_battery, &camera: sv_camera) is

states start_, body_branch, N3_T, N3_T_NS1_sync, NS1, NS2, NS2_NS5_sync, NS5, NS6, done, ether

var ignoreb: bool

from start_
   wait [0,0];
   on (not (skill[uav_mission_branch_1_1].caller = None));
to body_branch

from body_branch
   wait [0,0];
to N3_T //

// from: N3_T to: NS1 type: ET_GOAL expr: (goto_waypoint x 1 y 2 z 3 yaw 0 duration 0)
from N3_T
   wait [0,0];
   // Pass the control to (goto_waypoint x 1 y 2 z 3 yaw 0 duration 0)
   // Skill 'uav_mission' calling (goto_waypoint x 1 y 2 z 3 yaw 0 duration 0)
   skill[goto_waypoint].caller := uav_mission_branch_1_1;
   // synthesized action arg index 2
   skill[goto_waypoint].ArgIndex := 2;
to N3_T_NS1_sync

from N3_T_NS1_sync
   wait [0,0];
   // Wait the control back from (goto_waypoint x 1 y 2 z 3 yaw 0 duration 0)
   on (skill[goto_waypoint].caller = None);
   // Skill 'uav_mission' call to (goto_waypoint x 1 y 2 z 3 yaw 0 duration 0) returned
to NS1

// from: NS1 to: NS2 type: ET_GOAL expr: (^ 2)
from NS1
   // Wait 2 seconds
   // Skill 'uav_mission' waited 2 seconds (200 ticks)
   wait [200,200];
to NS13

// from: NS13 to: NS15 type: ET_GOAL expr: (goto_waypoint x -3 y -2 z 4 yaw 1.4 duration 0)
from NS13
   wait [0,0];
   // Pass the control to (goto_waypoint x -3 y -2 z 4 yaw 1.4 duration 0)
   // Skill 'uav_mission' calling (goto_waypoint x -3 y -2 z 4 yaw 1.4 duration 0)
   skill[goto_waypoint].caller := uav_mission_branch_1_1;
   // synthesized action arg index 3
   skill[goto_waypoint].ArgIndex := 3;
to NS13_NS15_sync

from NS13_NS15_sync
   wait [0,0];
   // Wait the control back from (goto_waypoint x -3 y -2 z 4 yaw 1.4 duration 0)
   on (skill[goto_waypoint].caller = None);
   // Skill 'uav_mission' call to (goto_waypoint x -3 y -2 z 4 yaw 1.4 duration 0) returned
to NS15

// from: NS15 to: NS6 type: ET_GOAL expr: (camera_tracking.interrupt)
from NS15
   wait [0,0];
   // will interrupt camera_tracking
   // Skill 'uav_mission' is interrupting camera_tracking
   ignoreb := Fiacre_camera_tracking_interrupt_action();
```

```
to NS6

// Joining on NS6, this branch is done
from NS6 // join
    wait [0,0];
to done

from done
    wait [0,0];
    skill[uav_mission_branch_1_1].caller := None;
to ether

from ether
    wait [0,0];
to start_
```

Listing 17: The H-Fiacre process monitoring under and over shooting of the **uav_mission** skill.

```
process skill_uav_mission_watchdog(&skill: skill_array) is

states start_, monitor, skill_overshoot, skill_not_undershoot

var ignorep: nat, overshoot, undershoot : bool

from start_
    wait [0,0];
    on (not (skill[uav_mission].caller = None)); // Skill 'uav_mission' watchdog started
    overshoot := false;
    undershoot := true;
to monitor

from monitor
    select
        on (skill[uav_mission].caller = None);
        if undershoot then // execution terminated before the undershoot elapsed
            // WARNING: Skill 'uav_mission' undershoot 60 s (6000 ticks)
            null
        end;
        to start_
    []// undershoot
        on undershoot;
        wait [6000, 6000]; // wait the undershoot time value
        to skill_not_undershoot
    []// overshoot
        on not undershoot and not overshoot;
        wait [6000, 6000]; // we already waited 600 for undershoot + 600 = 1200
        // WARNING: Skill 'uav_mission' overshoot 120 s (12000 ticks)
        to skill_overshoot
    end

from skill_overshoot
    wait [0,0];
    overshoot := true;
to monitor

from skill_not_undershoot // we got here before the execution terminated
    wait [0,0];
    undershoot := false; // hence we did NOT undershoot
to monitor
```