A Model-based Approach to the Development and Verification of Robotic Systems for Competitions

Marcus Santos, Madiel Conserva Filho and Augusto Sampaio
Centro de Informática
Universidade Federal de Pernambuco
Recife, Pernambuco, Brazil
{mvfs, mscf, acas}@cin.ufpe.br

2023 Latin American Robotics Symposium (LARS), 2023 Brazilian Symposium on Robotics (SBR), and 2023 Workshop on Robotics in Education (WRE) | 979-8-3503-1538-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/LARS/SBR/WRE59448.2023.10333030

***Abstract*—This paper proposes the adoption of modern Software Engineering techniques in the context of the development of robotic systems. Particularly, we use a design language, RoboChart, and a tool set, RoboTool, which support modelling, verification (via model checking) and simulation of such systems, among other facilities. Our application domain is an Unmanned Aerial Vehicle (UAV) and its control system, targetting at competition challenges. This system needs to be capable to perform navigation tasks in indoor environments and achieve some goals like detecting and finding objects, landing spots and maintaining a stable flight. We emphasise the important role of a design model both concerning formal verification of classical and domain-specific properties and as a basis for a systematic strategy to develop trustworthy implementations.**

I. Introduction

In recent years, the use of Unmanned Aerial Vehicles (UAV) has been significantly increasing; this is driven by technological advances such as the miniaturization of electronic components, increased battery life, and growth in computational power, among others. These vehicles have been used in the most diverse areas, from entertainment and commercial environments to military applications. These vehicles have also become increasingly complex and capable of performing more specific and complicated tasks, and it is expected that, in the near future, they will be adopted in the context of even more critical tasks, such as organ transport.

However, for them to be used in such critical applications, it is necessary to have guarantees about the correctness and robustness of the control code, since small flaws or thoughtless design decisions can cause irreparable damage. Therefore, it is necessary and recommended to use modern and rigorous software engineering techniques. Apart from functional correctness, such techniques also potentialise an increase in productivity during the development process, since desirable properties can be verified early in the design stage, and seamlessly preserved in the, systematically developed, implementation code.

The application domain we focus on is competition challenges. The more specific context is that of the Flying Robots Trial League (FRTL), a competition organized by RoboCup Brazil that aims to stimulate the development of autonomous robots in the execution of applications in the industrial and logistics areas. The main challenges involved in the competition are autonomous navigation, object detection, and stable flight, among others that are detailed in the next section. Teams compete in distinct stages aiming at the highest scores. As in every competition, some strict rules must be obeyed. For example, the use of GPS systems, sensors outside the arena, and human intervention are entirely prohibited.

The robotic system built is a modified quadcopter to carry out the phases with flight expectations between 20 and 30 minutes, with an onboard computer for processing. It also has a set of sensors, some for inferring the position of the drone and others for detecting targets. Furthermore, the high-level software used includes an implementation of visual-inertial odometry, a neural network for object detection, and our control software for drone movement.

Although a competition challenge is not as safe-critical as, for instance, the medical and the military domains, they have an extremely relevant educational impact, and are a suitable context to explore the advantages and limitations of state-of-the-art and state-of-the-practice software engineering techniques. We claim that such techniques have a direct impact on the competitive advantages of the teams that adopt them.

Following good software engineering practices, we adopt a model-based approach to the development of our robotic system for the FRTL competition. For modelling, we use RoboChart [1], a graphical, domain-specific, language for designing robotic systems. It is a UML [2] profile with a formal semantics defined using the CSP [3] (Communication Sequential Processes) algebra. Editing facilities for creating RoboChart models and the automatic generation of the corresponding CSP semantics are provided by a tool called RoboTool. For verification (via model-checking of CSP processes) we use FDR [4].

Unfortunately, most approaches to the development of robotic systems adopt outdated software engineering techniques, focusing essentially on simulation and coding, but typically neglecting modelling and verification. There are, however, some initiatives concerned with a more systematic

www.robocup.org.br
robostar.cs.york.ac.uk/robotool/

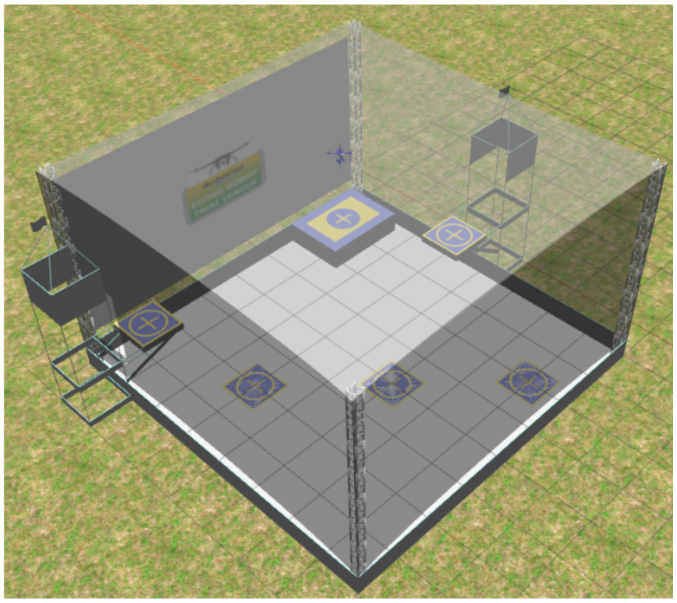


Fig. 1. Overview of the Arena

development approach. Particularly related to competition challenges, one can find some examples in [5]–[7]. Although these efforts embody some modelling activities, the system architecture and state machine models are presented in a rather informal way, without a precise syntax, let alone a formal semantics. Therefore, they cannot be subjected to formal analysis, nor as a precise reference to an implementation, as we advocate and unveil in this paper.

Section 2 provides additional details on the FRTL competition. Section III is dedicated to the modeling and verification of the proposed drone system. In Section 4, we briefly illustrate how the RoboChart modelling structure is reflected in the implementation. Finally, in Section 5 we summarise our contributions and discuss related and future work.

II. FRTL SYSTEM

Before we present (part of) the design, verification and implementation of an autonomous flying robot for the RobôCup Brazil Flying Robots Trial League competition, we first give an overview of the competition. It takes place face-to-face and consists of four phases, with each of them having different goals and scores. The team whose vehicle accumulates the highest score wins. During the actual competition, each phase is carried out by all the drones, before the next phase is started. A drone may fail or simply skip one phase and still proceed to attempt to perform the following phases of the competition.

In the first phase, the UAV must leave the launch base, look for another 5 bases that are placed around the arena, two of which are elevated bases, land on each one of them, and, in the end, return to the launch base. An overview of the arena is depicted in Figure 1. The launch base, in this arena, is the one located at the top rightmost corner. Three of the arena side walls and the rooftop are actually protection nets. The wall with the competition symbol (just behind the launch base) is a banner that serves as a fixed point reference for the drones that navigate based on images.

In the second phase, each base is assigned an identification character, say A, B, and so on. These are not physical marks, but rather a logical mapping between base coordinates and identifiers, provided as input to the teams so that the teams can relate the position of the mapped bases with the character identifiers provided. Also, in each base, a cube is placed with a QR code on its top face. This QR code has information about an identifier, which may or may not coincide with the base identifier. The vehicle must visit each of the bases, read the QR code on the cube, and check whether the identifier associated with this QR code is the base identifier. If so, no additional action is necessary and the vehicle proceeds its journey. Otherwise, the vehicle must carry the cube to the base whose identifier is the one associated with the QR code. After visiting all the bases, it must return to the launch base.

For the third phase, the robot must be able to move to the side of the arena that contains a shelf. There, he should be able to read the bar codes that are distributed in small boxes scattered throughout the structure.

For the last of the phases, the system must be able to take off from the launch base, look for a base that moves in the arena, and land on it.

To solve the challenges proposed by the competition, a drone was assembled. It is mounted on an F450 frame, with a Pixhawk 6X as controller. In addition, the vehicle has sensors such as a barometer, 3 accelerometers, 3 gyroscopes and others, all with redundancy for greater precision.

Furthermore, it has two cameras, one frontal and the other inferior, which are used to feed the VINS-Fusion, the visual-inertial odometry system implemented for localization, and the SSDMobilenetV2 with Detecnet, for the recognition of the items of interest in the phase, such as bases and QR codes.

In this system, all processing is carried out onboard the companion computer Jetson Nano, which is responsible for running all the algorithms and only informing the flight controller of the destination.

For human operators to communicate with this machine, just for interventions and to start the challenges, there is the radio control and a wifi and Bluetooth network communication with an external computer. On the occasion of a significant behaviour deviation (for instance, if the drone's autonomous navigation is leading it to collide with the arena protection nets) the operator may also take manual control of the drone.

III. MODEL AND VERIFICATION IN ROBOCHART

This section presents the design and the verification of some desired properties of the autonomous flying robotic system in RoboChart. Our focus here is on the overall control flow of the drone followed by the detailed design of the software for the first phase of the competition.

A robotic application in RoboChart is defined by a module that specifies a robotic platform and one or more (parallel) software controllers that run on this platform. The physical robot is abstracted into the robotic platform through events,

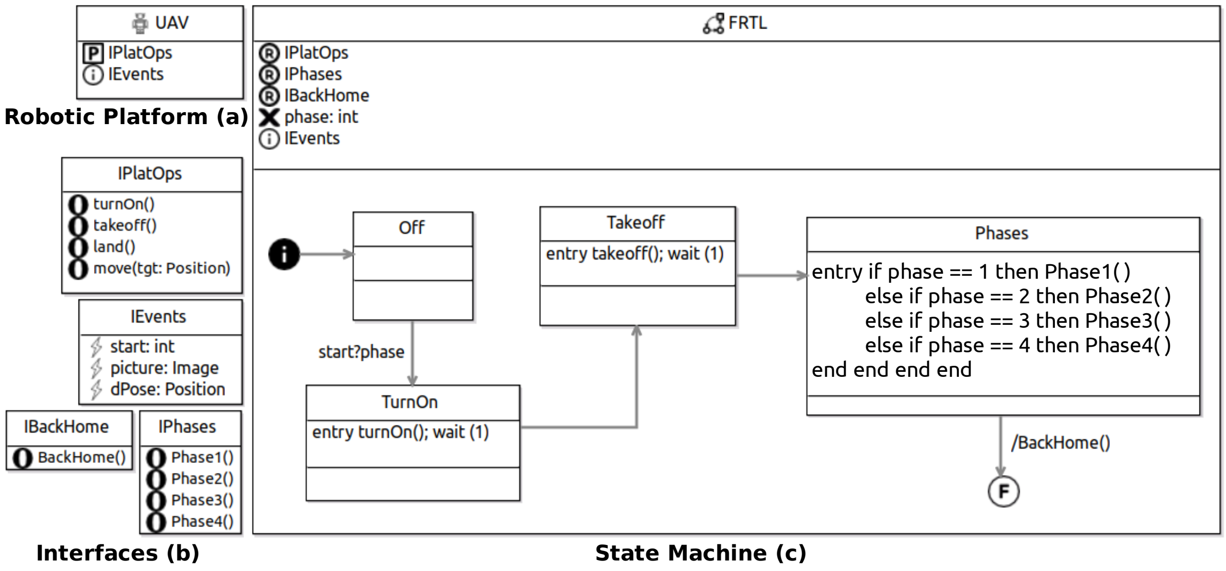


Fig. 2. RoboChart Model

operations, and variables that represent sensors and actuators of the robot that are available to the software controllers. RoboChart uses interfaces to group these elements for reuse and to describe dependencies. The robotic platform UAV for our system is in Figure 2(a). It provides (Ⓟ) the interface IPlatOps that contains operations to turn on, takeoff, move, and land the drone. The platform UAV also defines (ⓘ) the interface IEvents with some events (⚡) explained as needed. These interfaces are illustrated in Figure 2(b).

A software controller in RoboChart encapsulates one or more (parallel) state machines. We omit the diagram of the controller, as we have just a single state machine, FRTL, shown in Figure 2(c). The machine FRTL requires (Ⓡ) three interfaces: IPlatOps, provided by the platform UAV, IPhases, which contains the software operations that model the phases of the competition, and IBackHome that contains a software operation, BackHome(), to return the drone to its initial position: the launch base. The machine FRTL also defines the interface IEvents and a local variable (✖), phase, that identifies the phase of the competition to be accomplished.

In a RoboChart state machine, the initial junction (❶) indicates its starting point of execution. So, in FRTL (Figure 2(c)), the initial transition leads to the state Off. Transitions in RoboChart have the following optional features: a trigger (input) event, a guard condition, and an action. A transition can be fired when its trigger event happens and the guard condition holds, in which case the associated action is executed. For the initial transition, however, only an action is included and it is immediately fired since there are no triggers nor guards (which is the same as an implicit true guard).

Once in Off, the behaviour of the state machine remains blocked in such a state until it receives the event start from the robotic platform. This input event indicates the beginning of the autonomous navigation of the drone. It also communicates the phase to be executed that is recorded in the variable phase.

After receiving the event start, the control flows, from the state Off, to the state TurnOn. In RoboChart, a state may have an entry action that is immediately executed when the state is entered. After its execution, the transitions out of such a state become available if their trigger events, if any, can occur and if their guards, if any, hold. In TurnOn, the entry action calls the platform operation turnOn() to turn on the drone and, in sequence (;), the machine waits for one time unit, as indicated by the wait statement wait(1); this is (an abstraction of) a time budget for the platform to react to the operation turnOn().

The transition from the state TurnOn to the state Takeoff fires immediately, since it has no trigger nor guard. Similarly to TurnOn, Takeoff also has an entry action that calls the platform operation takeoff(), to takeoff the drone, followed again by wait(1). Afterwards, the control flows to the state Phases whose entry action is a (nested) conditional statement to call a software operation that implements a phase of the competition based on the value of the variable phase. For instance, if the value of phase is 1, then the software operation Phase1() is called; the behaviour of this operation, which implements the first phase of the competition, is captured by a separate component presented in Figure 3. This allows for a modular design presentation. After the execution of Phase1(), the control returns back to the machine, still in the state Phases, where the transition to the final state (Ⓕ) is taken. The action of this final transition calls the operation BackHome() to return the drone to its initial position: the launch base. We recall that, as presented in Section II, the phases are performed independently, one at a time, so we do not model a loop imposing any sequence of phases.

We now consider the design of Phase1() (see Figure 3). A software operation in RoboChart is just like a state machine, except that it has (optional) parameters.

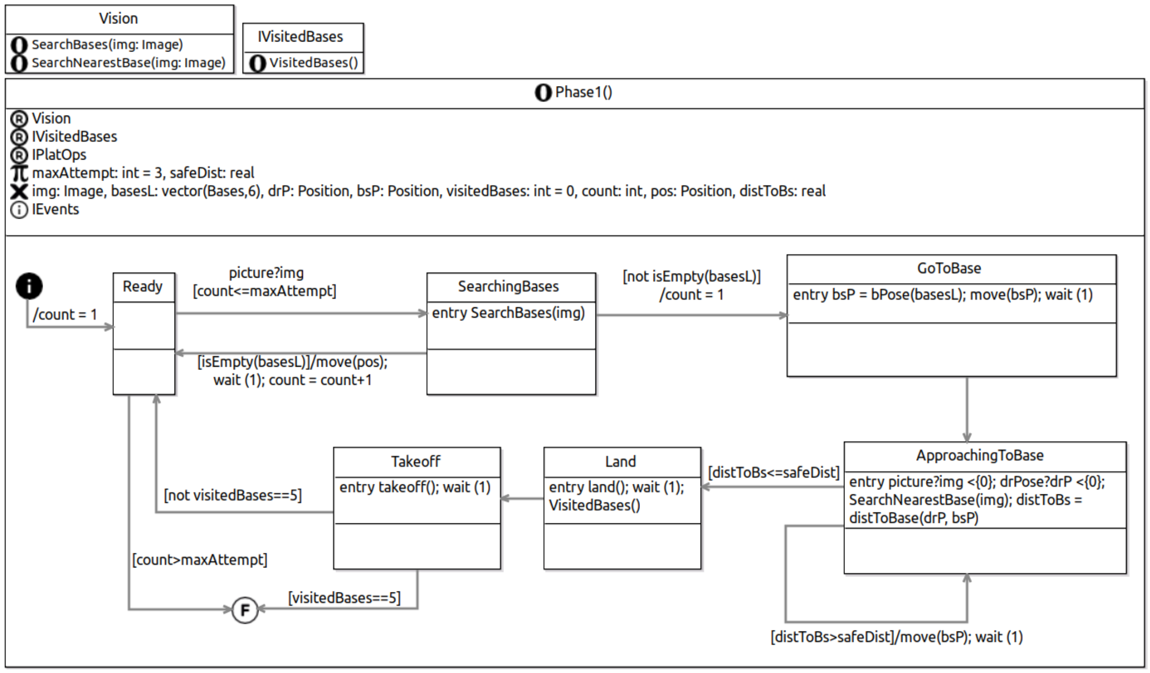

Vision
SearchBases(img: Image)
SearchNearestBase(img: Image)
IVisitedBases
VisitedBases()
Phase1()
Vision
IVisitedBases
IPlatOps
maxAttempt: int = 3, safeDist: real
img: Image, basesL: vector(Bases,6), drP: Position, bsP: Position, visitedBases: int = 0, count: int, pos: Position, distToBs: real
IEvents
/count = 1
Ready
picture?img
[count<=maxAttempt]
SearchingBases
entry SearchBases(img)
[isEmpty(basesL)]/move(pos);
wait (1); count = count+1
[not isEmpty(basesL)]
/count = 1
GoToBase
entry bsP = bPose(basesL); move(bsP); wait (1)
ApproachingToBase
entry picture?img <{0}; drPose?drP <{0};
SearchNearestBase(img); distToBs =
distToBase(drP, bsP)
[distToBs<=safeDist]
Land
entry land(); wait (1);
VisitedBases()
Takeoff
entry takeoff(); wait (1)
[not visitedBases==5]
[count>maxAttempt]
[visitedBases==5]
F
[distToBs>safeDist]/move(bsP); wait (1)

Fig. 3. RoboChart Software Operation Phase1

The operation Phase1() requires three software operations: (i) SearchBase(), which implements an algorithm, based on the neural network SSDMobileNetV2, that processes an image to identify the target bases (two overhead bases and three mobile bases), (ii) SearchNearestBase yields the nearest base, both of them are declared in the interface Vision, and (iii) VisitedBases(), declared in the interface IVisitedBases, that updates the status of a base after the drone lands on it. For conciseness, these operations are omitted here. Phase1() also requires the interface IPlatOps and defines the interface IEvents, already presented. It also declares some local variables and constants (π) explained on demand.

The behaviour of Phase1() is as follows. The initial transition leads to the state Ready. Its only action is an assignment count = 1. The local variable count is incremented at each failed attempt to find the target bases and must not exceed the value of the local constant maxAttempt.

In Ready, the operation remains blocked until it receives the event picture. We recall that our robotic system includes two cameras. We use the event picture to receive an image from such cameras that is recorded in the local variable img.

Once the event picture occurs and if count is less than or equal to maxAttempt, then the control flows from the state Ready to the state SearchingBases, where its entry action calls the software operation SearchBases(). If target bases have been found, this operation updates the variable basesL, which records the list of the bases found (such a variable is shared between the operations Phase1() and SearchBases()). There are two transitions from SearchingBases. If basesL is empty, then the transition back to Ready is fired, where its action first calls the platform operation move(), to move the drone to a new position (defined by the local variable pos). Next, we have a time budget (wait(1)). We recall that this represents a delay that gives time for the drone to move. Finally, the counter is incremented (count = count + 1). This cyclic flow goes on until the value of count has exceeded the value of maxAttempt, when the transition out of Ready to the final state of Phase1() takes place.

From SearchingBases, there is also a transition to the state GoToBase. It is reached if target bases have been found (not isEmpty(basesL)), which leads to an assignment (count = 1). In GoToBase, its entry action first assigns to the variable bsP the position of the next target base in basesL, that has not yet been visited, calculated using the function bPose (whose definition is omitted). After that, it calls the platform operation move() to move the drone to the base, followed by wait(1).

Afterwards, the software operation Phase1() transitions to the state ApproachingToBase, which aims at checking whether the drone is close enough (to the base) to land safely. For that, its entry action initially reads values from the events picture and drPose, with a deadline of zero ($< \{0\}$) to indicate these values should always be available, and, then, it assigns them to local variables img and drP. Next, it calls the software operation SearchNearestBase() that updates the variable bsP, with the position of the nearest base, which is ensured to be the same target base previously found. Finally, the distance from the drone to the base is calculated using the function distToBase (omitted) and assigned to local variable distToBs. If this distance is greater than the constant safeDist, then the self-transition is fired and the flow returns back to ApproachingToBase, moving the drone again, as it is not

https://github.com/dusty-nv/jetson-inference at a safe distance to land on the target base. Otherwise, the control flows to the state Land. In our design, we are assuming that this safe distance is feasible.

At this point, in Land, the drone lands on the target base, as initially defined by its entry action that calls the platform operation land(), followed by wait(1), as usual. Finally, the entry action calls the software operation VisitedBases(). It records the currently visited base as already visited and increments the (shared) variable visitedBases.

From Land, the flow reaches the state Takeoff whose entry action calls the platform operation takeoff(), to takeoff the drone from the previously landed base, followed by wait(1). There are two transitions from this state. If the drone has landed once in all the five bases, visitedBases == 5, the transition to the final state is taken. Otherwise, the control returns back to the state Ready.

The autonomous flying robotic system has been analysed using RoboTool, a modelling and verification tool for RoboChart. RoboTool automatically generates the CSP semantics for the RoboChart models and several standard assertions for checking classical properties like deadlock freedom, divergence freedom and determinism. These assertions are verified using FDR, the *d*e facto model checking tool for CSP. Besides these classical properties, we have encoded the following domain-specific properties directed in CSP.

- P1: After three failed attempts to find the bases, the drone must return to its initial position: the launch base.
- P2: The drone lands on a base if the distance to the base is greater than the allowed distance.
- P3: The drone only lands on a base if it is at a safe distance.
- P4: Every navigation action (such as move, land, takeoff) determined by the control software gives the drone one time unit to react.
- P5: The drone must be able to land and takeoff in all the five bases, and then returns to the launch base.

The verification results are shown in Table I. We used a machine with quad-core Intel i7 5500U, and 16 GB of RAM in an Ubuntu system. We ran each assertion 10 times, and the verification time given is the average.

From the results presented in Table I, we note that all properties have passed, except P2. This is an expected behaviour in our design, as the drone must not land on a base if it is not at a safe distance. This scenario is reflected by the self-transition in the state ApproachingToBase of the operation Phase1().

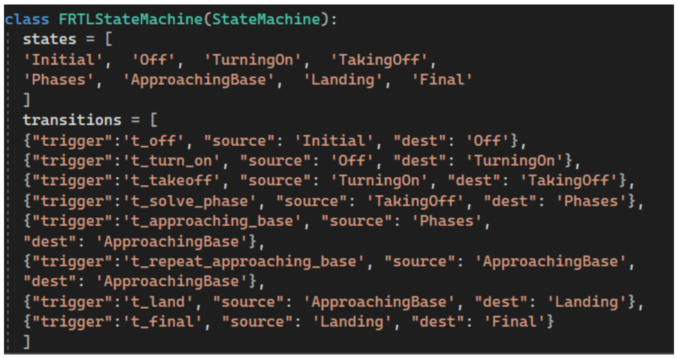
IV. Implementation

With the aim of maintaining the fidelity of what was modeled in the tools and facilitating the understanding of how the system works, we opted for structuring the implementation in such a way as to directly reflect a state machine. For this, we use the pytransitions Python library that offers support for the implementation of state machines and other extra features such as machine parallelism and hierarchical machines.

All CSP assertions are available at https://encurtador.com.br/dvEPV
https://github.com/pytransitions/

TABLE I
Verification results

	Property	FDR Result
Safety	deadlock freedom	Passed
	livelock freedom	Passed
	determinism	Passed
Domain	P1	Passed
	P2	Failed
	P3	Passed
	P4	Passed
	P5	Passed

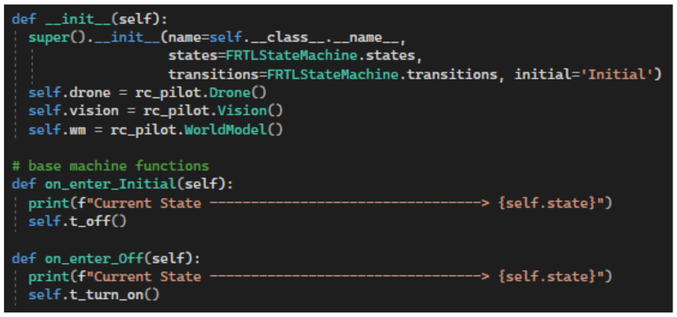


```
class FRTLStateMachine(StateMachine):
  states = [
  'Initial',  'Off',  'TurningOn',  'TakingOff',
  'Phases',  'ApproachingBase',  'Landing',  'Final'
  ]
  transitions = [
  {"trigger":'t_off', "source": 'Initial', "dest": 'Off'},
  {"trigger":'t_turn_on', "source": 'Off', "dest": 'TurningOn'},
  {"trigger":'t_takeoff', "source": 'TurningOn', "dest": 'TakingOff'},
  {"trigger":'t_solve_phase', "source": 'TakingOff', "dest": 'Phases'},
  {"trigger":'t_approaching_base', "source": 'Phases',
  "dest": 'ApproachingBase'},
  {"trigger":'t_repeat_approaching_base', "source": 'ApproachingBase',
  "dest": 'ApproachingBase'},
  {"trigger":'t_land', "source": 'ApproachingBase', "dest": 'Landing'},
  {"trigger":'t_final', "source": 'Landing', "dest": 'Final'}
  ]
```

Fig. 4. States and Transitions

To explain the structure of our implementation, consider the FRTL RoboChart machine in Figure 2. We present in Figure 4 a fragment of code where the states and transitions for the FRTLStateMachine class are defined. In the states clause, we pass as parameters all the states that compose the machine. In the transitions clause, we first define a name for the trigger, then inform which states are related by this trigger, passing first the source and then the destination state as parameters. For example, the transition t_off, has as source state Initial and as destination Off.

In Figure 5, we have the constructor of the FRTLStateMachine class, where the states, transitions and are passed as parameters. In addition, we have the instantiation of other objects that are necessary for the resolution of the phases, as is the case of drone, vision and wm. The first represents the

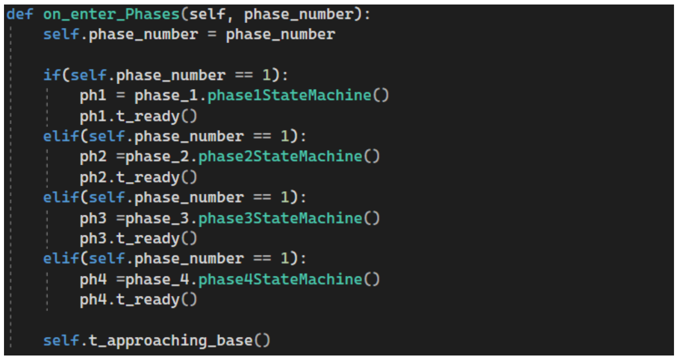


```
def __init__(self):
  super().__init__(name=self.__class__.__name__,
                   states=FRTLStateMachine.states,
                   transitions=FRTLStateMachine.transitions, initial='Initial')
  self.drone = rc_pilot.Drone()
  self.vision = rc_pilot.Vision()
  self.wm = rc_pilot.WorldModel()

# base machine functions
def on_enter_Initial(self):
  print(f"Current State ----------------------------------> {self.state}")
  self.t_off()

def on_enter_Off(self):
  print(f"Current State ----------------------------------> {self.state}")
  self.t_turn_on()
```

Fig. 5. Starting and changing



```
def on_enter_Phases(self, phase_number):
    self.phase_number = phase_number

    if(self.phase_number == 1):
        ph1 = phase_1.phase1StateMachine()
        ph1.t_ready()
    elif(self.phase_number == 1):
        ph2 =phase_2.phase2StateMachine()
        ph2.t_ready()
    elif(self.phase_number == 1):
        ph3 =phase_3.phase3StateMachine()
        ph3.t_ready()
    elif(self.phase_number == 1):
        ph4 =phase_4.phase4StateMachine()
        ph4.t_ready()

    self.t_approaching_base()
```

Fig. 6. The Phase selector

physical drone and is where the movement commands are sent that will be translated into the real world. The second abstracts the neural network and visualization resources needed for object detection. The third represents the *world model*, where the drone positioning resources captured by the odometry algorithm are implicit.

Finally, we have the definitions of the on_enter_ <State> functions that correspond to the RoboChart entry actions. For example, we have on_enter_Off, which when starting informs the current state and then transitions to the next state through self.t_turn_on(), which is the trigger we defined earlier.

A subtle detail about the machine transition can be seen in the ”on_enter” action. This event, as previously mentioned, is triggered when a transition enters the state. However, as the machine is already started within Initial, this action is not triggered when the machine is started.

Following the fired transitions, the machine flow progresses until it reaches the state Phases. Its on_enter action is shown in Figure 6, where one can observe a non-trivial state that receives phase_number as a parameter and uses it to instantiate a new state machine referring to the phase that will be executed in the competition. It is worth emphasising that this coding strategy supports a compositional structure, closely mimicking the mudularity facilities provided by RoboChart with operations that are also modelled using state machines. When the machine that implements the state Phases (see Figure 2) is called, the two machines actually run in parallel, with the main machine waiting for the newly instantiated one to finish so that it can resume its previous execution flow.

V. Conclusion and Future Work

Following a model-driven approach, we presented a high-level design model for the control software of a flying robot whose purpose is to perform autonomous navigation activities targeting the FRTL competition challenge. The effort of building precise system architecture and behavioural models brings important insights for thoughtful design decisions. Furthermore, these models with well-defined syntax and semantics allow the formal verification of desirable properties and serve as a reference for the actual implementation. These have been extensively addressed in Sections III and IV. Particularly, using a library for implementing state machines in Python, the structure of the code is very much the same as that of the design model.

As already discussed in the introduction, as far as we are aware, a rigorous approach to the development of robotic systems for competition challenges, as we propose here, is an original contribution. Considering the more general context of the development of robotic systems, there are some approaches in the literature. For example, RobotML [8], SafeRobots [9] and FlexBE [10] provide DSLs for modelling robotic systems and facilities for animation, simulation, or even code generation, but no support for formal verification of behavioural properties. Modelling of time aspects is also more restrictive than in RoboChart.

Despite the promising results so far, there are several opportunities for Future work. We have considered only part of the FRTL scope. We plan to further detail the models to consider, for example, error handling using a state machine running in parallel with the FRTL machine in Figure 2(c). We also plan to have tool support to automatically generate the Python state machine implementations from the models.

Acknowledgements

Our work is funded by Brazilian Research Agencies, via project INES, grants CNPq/465614/2014-0 and FACEPE/APQ/0388-1.03/14.

References

[1] A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, J. Timmis, and J. C. P. Woodcock, “RoboChart: modelling and verification of the functional behaviour of robotic applications,” *Software & Systems Modeling*, vol. 18, no. 5, pp. 3097–3149, 2019.

[2] OMG, *OMG Unified Modeling Language*, Object Management Group Std., 2015. [Online]. Available: www.omg.org/spec/UML/2.5/

[3] A. W. Roscoe, *The Theory and Practice of Concurrency*, ser. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[4] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe, “FDR3 - A Modern Refinement Checker for CSP,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2014.

[5] R. B. Grando, P. M. Pinheiro, N. P. Bortoluzzi, C. B. da Silva, O. F. Zauk, M. O. Piñeiro, V. M. Aoki, A. L. Kelbouscas, Y. B. Lima, P. L. Drews, and A. A. Neto, “Visual-based autonomous unmanned aerial vehicle for inspection in indoor environments,” in *2020 Latin American Robotics Symposium, 2020 Brazilian Symposium on Robotics (and 2020 Workshop on Robotics in Education*, 2020, pp. 1–6.

[6] A. Cruzde Araújo, A. M. Nogueira Lima, J. M. Mattila, and R. Muthusamy, “Model-based robotic hand tracking and gripper state determination,” in *2017 Latin American Robotics Symposium (LARS) and 2017 Brazilian Symposium on Robotics (SBR)*, 2017, pp. 1–6.

[7] J. G. Melo, F. Martins, L. Cavalcanti, R. Fernandes, V. Araújo, R. Joaquim, J. G. Monteiro, and E. Barros, “Towards an autonomous robocup small size league robot,” in *2022 Latin American Robotics Symposium (LARS), 2022 Brazilian Symposium on Robotics (SBR), and 2022 Workshop on Robotics in Education (WRE)*, 2022, pp. 1–6.

[8] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2012, ch. RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications, pp. 149–160.

[9] A. Ramaswamy, B. Monsuez, and A. Tapus, “Saferobots: A model-driven framework for developing robotic systems,” in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 1517–1524.

[10] P. Schillinger, S. Kohlbrecher, and O. von Stryk, “Human-Robot Collaborative High-Level Control with an Application to Rescue Robotics,” in *IEEE International Conference on Robotics and Automation*, 2016.