RTAEval: A Framework for Evaluating Runtime Assurance Logic

Kristina Miller[1(✉)], Christopher K. Zeitler[2], William Shen[1], Mahesh Viswanathan[1], and Sayan Mitra[1]

[1] University of Illinois Urbana Champaign, Champaign, IL 61820, USA
kmmille2@illinois.edu
[2] Rational CyPhy, Inc., Urbana, IL 62802, USA

Abstract. Runtime assurance (RTA) addresses the problem of keeping an autonomous system safe while using an untrusted (or experimental) controller. This can be done via logic that explicitly switches between the untrusted controller and a safety controller, or logic that filters the input provided by the untrusted controller. While several tools implement specific instances of RTAs, there is currently no framework for evaluating different approaches. Given the importance of the RTA problem in building safe autonomous systems, an evalutation tool is needed. In this paper, we present the RTAEval framework as a low code framework that can be used to quickly evaluate different RTA logics for different types of agents in a variety of scenarios. RTAEval is designed to quickly create scenarios, run different RTA logics, and collect data that can be used to evaluate and visualize performance. In this paper, we describe different components of RTAEval and show how it can be used to create and evaluate scenarios involving multiple aircraft models.

Keywords: Runtime assurance · Autonomous systems

1 Introduction

Safe operation of autonomous systems is critical as their real world deployment becomes more common place in domains such as aerospace, manufacturing and transportation. However, the need for safety is often at odds with the need to experiment with, and therefore deploy, new untrusted technologies in the public sphere. For example, experimental controllers created using reinforcement learning can provide better performance in simulations and controlled environments, but assuring safety in real world circumstances is currently beyond our capabilities for such controllers. *Runtime assurance (RTA)* [3,15–17] addresses this tension. The idea is to introduce a *decision module* that somehow chooses between a well-tested *Safety controller* and the experimental, *Untrusted controller*, assuring safety of the overall system while also allowing experimentation with the new untrusted technology where and when possible. Specific RTA technologies are being researched and tested for aircraft engine control [1], air-traffic management [4], and satellite rendezvous and proximity operations [9].

É. André and J. Sun (Eds.): ATVA 2023, LNCS 14216, pp. 302–313, 2023.
https://doi.org/10.1007/978-3-031-45332-8_17

The Simplex architecture [16,17] first proposed this idea in a form that is recognizable as RTA. Since then, the central problem of designing a *decision module* that chooses between the different controllers has been addressed in a number of works such as SimplexGen [3], Black-Box Simplex [13], and SOTER [5]. The two main approaches for building the decision module are based on (a) an RTASwitch which chooses one of the controllers using the current state or (b) an RTAFilter which blends the outputs from the two controllers to create the final output. In creating an RTASwitch, the decision can be based on forward-simulation of the current state [19], model-based [3] and model-free forward reachability [13], or model-based backward reachability [3]. The most common filtering method is Active Set Invariance Filtering (ASIF) [2], wherein a control barrier function is used to blend the control inputs from the safety and untrusted controllers such that the system remains safe with respect to the control barrier functions [7,12,14].

While these design methods for the decision module have evolved quickly, a software framework for evaluating the different techniques has been missing. In this paper, we propose such a flexible, low-code framework called RTAEval (Fig. 1). A *low-code* framework is one which simplifies the development of applications by providing a library of tools which reduces the amount of code required to be written by the user. Low-code frameworks are becoming more commonplace as the need to quickly experiment, deploy, and test new technologies becomes more urgent. One example of a low-code framework is the Scenic Library [11] which can be used to quickly and easily spin up new test environments for testing perception and control algorithms. In this work, we introduce a framework in a similar vein which can be used to test new runtime assurance technologies. This framework consists of a module for defining scenarios, possibly involving multiple agents; a module for executing the defined scenario with suitable RTASwitches and RTAFilters; and a module for collecting and visualizing execution data. RTAEval allows different agent dynamics, decision modules, and metrics to be plugged-in with a few lines of code. In creating RTAEval, we have defined standardized interfaces between the agent simulator, the decision module (RTA), and data collection.

In Sect. 2, we give an overview of RTAEval. In Sect. 2.1, we discuss how scenarios are defined, and, in Sect. 2.2, we discuss how the user should provide decision modules (also called the RTALogic). In Sect. 2.3, we discuss data collection, evaluation, and visualization. Finally, in Sect. 3, we show a variety of examples implemented in RTAEval. A tool suite for this framework can be found at https://github.com/RationalCyPhy/RTAEval.

2 Overview of the RTAEval Framework

The three main components of RTAEval are (a) the scenario definition, (b) the scenario execution, and (c) the data collection, evaluation, and visualization module (See Fig. 1). A scenario is defined by the agent and its low-level controller, the unsafe sets, the untrusted and safety controllers, the time horizon for analysis, and the initial conditions. Given this scenario definition, the scenario is executed iteratively over the specified time horizon.

During each iteration of the closed-loop execution of the RTA-enabled autonomous system, the current state of the agent and the sets of unsafe states are collected. This observed state information is given to both the untrusted and safety controller, which each compute control commands. Both of these commands are evaluated by the user-provided decision module (i.e., RTA logic), which computes and returns the actual command to be used by the agent. The agent then updates its state, and the computation moves to the next iteration. While the execution proceeds, data – such as the RTA computational performance, controller commands, agent states, and observed state information of the unsafe sets – is collected via the data collection module. At the end of an execution, this data is evaluated to summarize the overall performance of the RTA. This summary includes computation time of the RTA logic, untrusted versus safety controller usage, and the agent's distance from the unsafe set. We also provide a visualization of this data.

A low-code tool suite of the RTAEval framework is written in Python, which was chosen for its ease of implementation and interpretibility. The tool is flexible in that it allows for a wide variety of simulators and coding languages and can be generalized to scenarios where multiple agents are running a variety of different RTA modules. Simple Python implementations of vehicle models (some of which we provide in simpleSim) can be incorporated directly. However, users can incorporate new agent models within simpleSim as long as the agent has a function `step` that defines the dynamics and low-level controller of the agent and returns the state of the agent at the next time step. An example of this is provided in Example 1 and Fig. 4. The safety and untrusted controllers should also be encoded in `step`, which simply takes in the command (or mode) to be used over the next time step. Higher fidelity simulators such as CARLA [6] and AirSim [18] can also be used in place of simpleSim for the execution block. The observed state information would need to be provided to our data collection, evaluation, and visualization tool in the format seen in Fig. 2.

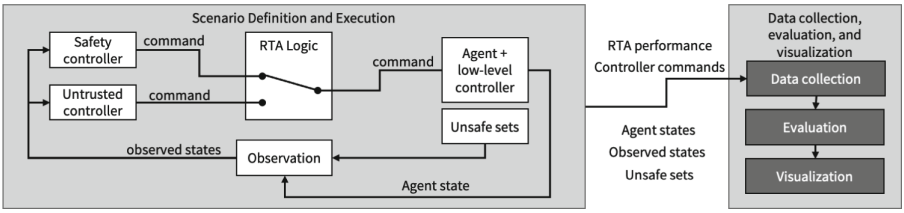

Fig. 1. RTAEval framework. (Scenario Definition and Execution) Some user-defined scenario is executed. The scenario is defined by the safety controller, untrusted controller, plant and low level controllers, unsafe sets, and sensor. (Data collection, evaluation, and visualization) The execution data is collected, evaluated, and visualized using our provided suite of tools.

2.1 Scenario Definition and Execution

A *scenario* is defined by the *agent*, *unsafe sets*, *safety* and *untrusted controllers*, *initial conditions*, and *time horizon* $T > 0$. The *simulation state at time* $t \in [0, T]$ consists of the *agent state*, the *unsafe set definition*, and the *control command* at time t. The agent has an identifier, a state, and some function `step` that takes in some control command at time t and outputs the system state at time $t + 1$. The *unsafe sets* are the set of states that the system must avoid over the execution of the scenario. We say that the agent is *safe* if it is outside the unsafe set. The safety and untrusted controllers compute control commands for the system, which are then filtered through the RTA logic, as discussed further in Sect. 2.2. The initial conditions define the simulation state at time 0. Then, given a scenario with some time horizon T and an RTA logic, an *execution* of the scenario is a sequence of time-stamped simulation states over $[0, T]$. Note that, while we define an execution as a discrete time sequence of simulation states, the actual or real-world execution of the scenario may be in continuous time; thus, we simply sample the simulation states at a predefined interval. We call the part of the execution that contains only the sequence of agent states the *agent state trace*. Similarly, we call the part of the execution that only contains the sequence control commands the *mode trace* and the part that only contains the sequence of unsafe set states the *unsafe set state trace*.

In order for our evaluation and visualization to work, the execution must be given to the data collection as a dictionary, the structure of which is shown in Fig. 2. Here, there are three levels of dictionaries. The highest level dictionary has the keys `'agents'` and `'unsafe'`, which point to dictionaries containing the state and mode traces of the agents and state traces of the unsafe sets respectively. The second level of dictionaries has keys that correspond to different agents and unsafe sets. We call these keys the agent and unsafe set IDs. Each agent ID points to a dictionary containing the state and mode traces of that agent. The state trace is a list of time-stamped agent states, and the mode trace is a sequential list of control commands. Each unsafe set ID points to a dictionary containing the set type and state trace of that unsafe set. The set type is a string that tells RTAEval what type of set that particular unsafe set is. Currently, RTAEval supports the following set types: *point*, *ball*, *hyperrectangle*, and *polytope*. Each set has a *definition* that, together with the type, defines the set of states contained within the unsafe set. Then, the state trace for an unsafe set is a sequence of time stamped definitions of the set.

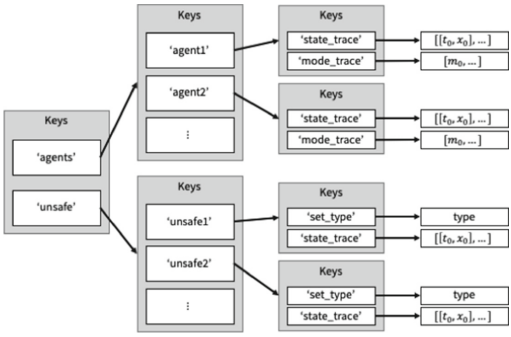

Fig. 2. Execution structure required by the RTAEval evaluation and visualization

Example 1. Consider the following adaptive cruise control (ACC) scenario shown in Fig. 3 as a running example. An agent with state $x = [p, v]^\top$ has dynamics given by

$$f(x, m) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} a(x, m),$$

where $a(x, m) = g_{\mathtt{S}}(x)$ if $m = \mathtt{S}$ and $a(x, m) = g_{\mathtt{U}}(x)$ if $m = \mathtt{U}$. The agent tries to follow at distance $d > 0$ behind a leader moving at constant speed $\bar{v}$. The position of the leader at time t is given by $p_L(t)$. Then, the untrusted controller $g_{\mathtt{U}}$ and safety controller $g_{\mathtt{S}}$ are given by

$$g_{\mathtt{S}}(x) = k_1((p_L(t) - d) - p) + k_2(\bar{v} - v) \text{ and } g_{\mathtt{U}}(x) = \begin{cases} a_{\max} & (p_L(t) - p) > d \\ -a_{\max} & \text{else} \end{cases},$$

where $k_1 > 0$ and $k_2 > 0$. The function `step` is a composition of the untrusted controller, the safety controller, and the dynamics function of the system.

A collision between the agent and leader occurs if $\|p_L(t) - p(t)\| \leq c$, $c < d$. There is then an unsafe set centered on the leader agent, and it is defined by $\mathcal{O} = \{[p, v, t]^\top \in \mathcal{X} \times \mathbb{R}_{\geq 0} \mid \|p_L(t) - p\| \leq c\}$. The function `updateDef` then takes in the current state of the simulator and creates the unsafe set centered on the leader. The initial conditions for this scenario are then the initial agent state x_0, the initial leader state x_{L0}, and the time horizon $T > 0$.

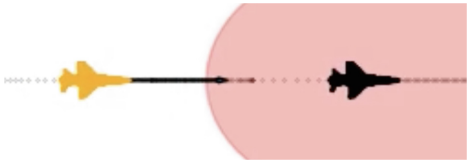

Fig. 3. Example visualization of the scenario defined in Example 1. The leader is shown in black, the follower is shown in orange, and the unsafe region is shown in red.

This scenario is shown in our low code framework in Figs. 4 and 5. The dynamics of the agent are defined in `step` in lines 11–26 of Fig. 4. The proportional controller is defined in lines 1–4 and the bang-bang controller is defined in lines 6–9. This is all contained within a class `AccAgent`. In Fig. 5, we set up the scenario. In lines 2–5, we define the goal point for the agent. In lines 7–15, we create the agent, the leader, and the unsafe set. Finally, in line 18, we initialize the scenario to be executed; in lines 21–26, we add the agents and unsafe sets to the scenario; and in lines 29–30, we set up the scenario parameters.

2.2 RTA Logics

We provide an RTA base class that can be used in RTAEval. The user must provide the RTA logic to be evaluated. This logic takes in an observed state and outputs the control command to be used by the plant. This observed state information has to be provided in the format shown in Fig. 2 for data collection, evaluation, and visualization to work. The RTA base class is shown in Fig. 6. We provide the functions `RTASwitch` and `setupEval`. Users must provide the switching logic as `RTALogic`. When creating RTA, the user can decide to use

```
1  def P(self, time_step): #Proportional controller
2      xrel = self.goal_state[0] - self.state_hist[-1][0]
3      vrel = self.goal_state[1] - self.state_hist[-1][1]
4      return self.Kp[0]*xrel + self.Kp[1]*vrel
5
6  def BangBang1D(self, time_step): # Bang-bang controller
7      x_err_curr = self.goal_state[0] - self.state_hist[-1][0]
8      v_err_curr = self.goal_state[1] - self.state_hist[-1][1]
9      return np.sign(x_err_curr)*self.a_max
10
11 def step(self, mode, initialCondition, time_step, simulatorState):
12         self.goal_state = self.desired_traj(simulatorState)
13         if mode == 'SAFETY':
14             self.control = self.P
15         elif mode == 'UNTRUSTED':
16             self.control = self.BangBang1D
17         else:
18             self.control = self.no_control()
19         a_curr = self.control(time_step)
20         if abs(a_curr) > self.a_max:
21             a_curr = np.sign(a_curr)*self.a_max
22         x_next = initialCondition[0] + initialCondition[1]*time_step
23         v_next = initialCondition[1] + a_curr*time_step
24         if abs(v_next) >= self.v_max:
25             v_next = np.sign(v_next)*self.v_max
26         return [x_next, v_next]
```

Fig. 4. Controller and step functions for the agent in Example 1. The first function defined is the proportional controller (Safety) and the second function is the Bang-bang controller (Untrusted). The step function (lines 11–26) takes in the current mode and state of the agent, as well as the time stpe and current suimulator state. In lines 13–18 it decides which controller to use, and in lines 19–26, it updates the staet of the agent.

our data collection by running `setupEval` in `__init__`. This will create a data collection object called `eval`, which saves the data used for our evaluation (see Sect. 2.3). The switch is performed in `RTASwitch`, which also stores the current perceived state of the simulator from the point of view of the agent, as well as the time to compute the switch. The user provided switching logic `RTALogic` takes in the current state of the simulator and returns the mode that the agent should operate in. To create different logics, the user must create an RTA class derived from the RTA base class, which implements the function `RTALogic`. An example of this is given in Example 2.

Example 2. An example of a simple RTA switching logic can be seen in Fig. 7. This is a simulation-based switching logic that was designed for the adaptive cruise control introduced in Example 1. Here, the future states of the simulator are predicted over some time horizon T and saved as `predictedTraj` in line 2. We then check over this predicted trajectory to see if the agent ever enters the unsafe set in lines 3–11. If it does, then the safety controller is used, and if it does not, then the untrusted controller is used. Once `RTALogic` is created, we add it to a new class called `accSimRTA` and use it to create an RTA object called `egoRTA` for `egoAgent1`. We can then change line 22 in Fig. 5 to `RTAs = [egoRTA, None]`. This will associate `egoRTA` with `egoAgent1` and run the RTA switching logic every time the state of `egoAgent1` is updated.

```
# Define desired goal point for the follower agent (ego):
def agent1_desiredTraj(simulationTrace):
    lead_state = simulationTrace['agents']['leader']['state_trace'][-1][1:]
    return [lead_state[0] - 10, lead_state[1]]

# Create the ego agent:
agent1 = AccAgent("follower",file_name=controllerFile)
agent1.follower = True
agent1.desired_traj = agent1_desiredTraj

# Create the leader agent:
leader = AccAgent("leader",file_name=controllerFile)

# Create the unsafe set centered on leader agent:
unsafe1 = relativeUnsafeBall("unsafe1", [5], 7, "leader")

# Initialize the ACC scenario:
accSim = simpleSim()

# Initialize agents and unsafe sets in the scenario:
agents = [agent1, leader]
RTAs = [None, None]
modes = [ccMode.UNTRUSTED, ccMode.NORMAL]
inits = [[0,1], [5,1]]
accSim.addAgents(agents=agents, modes=modes, RTAs=RTAs, initStates=inits)
accSim.addUnsafeSets(unsafe_sets = [unsafe1])

# Set simulation parameters
accSim.setSimType(vis=False, plotType="2D", simType="1D")
accSim.setTimeParams(dt=0.1, T=5)
```

Fig. 5. Python code snippet defining the scenario in our low-code RTAEval framework. The untrusted and safety controllers are contained within the dynamics of the `AccAgent`, which is defined in a separate file. The scenario is executed in a simply python simulator, which is initiated on line 18. Initially, the agents are not assigned RTAs, but this will be done in Sect. 2.2. The agents and unsafe sets are added to the scenario, and the simulation parameter are set in lines 29 and 30.

2.3 Data Collection, Evaluation, and Visualization

We now discuss the data collection, evaluation, and visualization tool which is provided as a part of RTAEval. This tool is a class that has some collection functions and post-processing functions. To use the data collection and evaluation functionalities provided, the user must add the line `self.setupEval()` when creating the RTA object. Data collection occurs via the functions `collect_trace` and `collect_computation_times`. Here, `collect_trace` collects the simulation traces, and `collect_computation_times` collects the time it takes for the RTA module to compute a switch. An example of how the data collection can be incorporated in the RTA module is shown in Fig. 7. The traces are collected and stored as a dictionary of the form shown in Fig. 2. Once the data has been collected over a scenario, we can use them to evaluate the performance of the RTA over a scenario. Examples of the data evaluation, as well as screenshots from our simulator are shown in Sect. 3. A summary of the RTA's performance in the scenario can be quickly given by running `eval.summary()`. The main metrics that we study are the following: **Computation time** gives the running time of `RTASwitch` each time it is invoked. We provide the average, minimum, and maximum times to compute the switch. **Distance from unsafe set** is the distance

```python
class baseRTA(abc.ABC):
    def __init__(self):
        self.do_eval = False # Don't automatically set up RTAEval
        pass

    @abc.abstractmethod
    def RTALogic(self, simulationTrace: dict) -> Enum:
        # User provided logic for switching RTA
        pass

    def RTASwitch(self, simulationTrace: dict) -> Enum:
        start_time = time.time()
        rtaMode = self.RTALogic(simulationTrace)
        running_time = time.time() - start_time

        if self.do_eval:
            self.eval.collect_computation_time(running_time)
            self.eval.collect_trace(simulationTrace)
        return rtaMode

    def setupEval(self):
        # Add this to init when inheriting baseRTA to inclde evaluation
        self.do_eval = True
        self.eval = RTAEval()
```

Fig. 6. Base RTA class used in the low-code RTAEval framework. Users need only provide the decision logic, which we call `RTALogic`.

```python
def RTALogic(simulationTrace):
    predictedTraj = simulate_forward(simulationTrace)
    egoTrace = predictedTraj['agents'][egoAgent.id]['state_trace']
    for unsafeSet in self.unsafeSets:
        unsafeSetTrace = predictedTraj['unsafe'][unsafeSet.id]['state_trace']
        for i in range(len(unsafeSetTrace)):
            egoPos = egoTrace[i][1]
            unsafeSetDef = unsafeSetTrace[i][1]
            pos_max = unsafeSetDef[0][0] - unsafeSetDef[1]
            if egoPos > pos_max:
                return egoModes.SAFETY
    return egoModes.UNTRUSTED
```

Fig. 7. Example RTA switching logic for Example 2. Here, the trajectory of the follower agent is simulated forward, and if it ever comes within collision distance of the leader, then the safety controller is used.

between the ego agent and the unsafe sets. We also allow the user to find the distance from other agents in the scenario. **Time to collision (TTC)** is the time until collision between the ego agent and the other agents if none of them change their current trajectories. Finally, we also provide information on the **percent controller usage**, which is the proportion of time each controller is used over the course of the scenario. We also provide information on the number of times a switch occurs in a scenario. Example results are shown in Sect. 3.

3 RTAEval Examples

In this section, we present some examples using our provided suite of tools for RTAEval. We evaluate two different decision module logics: SimRTA and

ReachRTA. SimRTA is the simulation based switching logic introduced in Example 2. ReachRTA is similar to SimRTA but uses reachable sets that contain all possible trajectories of the agent as the basis of the switching logic. We evaluate these RTAs in 1-, 2-, and 3-dimensional scenarios with varying numbers of agents. These scenarios are described in more detail in Table 1. Here, the workspace denotes the dimensions of the physical space that the systems live in. Note that, while all the examples presented have some physical representation, this is not a necessary requirement of the tool. We also provide pointers to where the dynamics of the agents can be found, as well as the untrusted and safety controllers used. Visualizations of the scenarios can be seen in Figs. 3 and 8.

Table 1. Brief description of evaluated scenarios.

	ACC	Dubins	GCAS
Workspace	1	2	3
Dynamics Untrusted	Example 1 Bang-bang controller (Example 1)	Dubin's car [8] PID with accleration [10]	Dubin's plane [8] PID with acceleration [10]
Safety	PID (Example 1)	PID with deceleration [10]	PID with deceleration and pitching up [10]
Unsafe	Leader (ball)	Leader (ball) and building (rectangle)	Leader (ball) and ground (polytope)
Visualization	Fig. 3	Fig. 8	Fig. 8
Scenario length	10 s	20 s	40 s

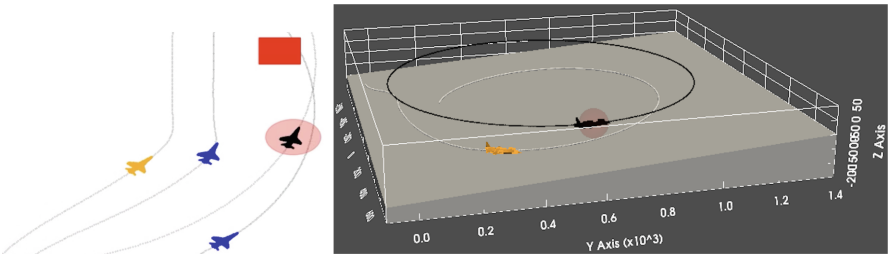

Fig. 8. Example scenarios in Table 1. Left: 2-dimensional dubins aircraft with building collision avoidance. The leader is shown in black, and the followers are shown in orange and blue. The desired trajectories are shown in gray. Right: Ground collision avoidance. The leader is shown in black and the follower is shown in orange. The desired trajectory is shown in white.

Each of these scenarios is executed using simpleSim, and the two RTA logics are created for them. Data is collected over the scenario lengths in Table 1. Note that the scenario length is the simulation time for the scenario and not

the real time needed to run the scenario. We run these scenarios with varying numbers of agents and present the running time of the scenario execution and evaluations in Table 2. The simulation time step is set to 0.05 for all scenarios. Here, exec time is the time it takes to run the scenario, RTA comp time is the average time it takes to run the user-provided RTA logic per iteration, and % RTA comp is the percentage of the exec time that is taken by the RTA decision module. That is, % RTA computation is roughly the number of time steps in a scenario multiplied by the the average RTA comp time and divided by the execution time. Finally, eval time is the time it takes to get a full summary of how the RTA performs for each agent. The evaluation summary includes the average decision module computation time, controller usage, distance from the unsafe sets and other agents, and time to collision with the unsafe sets and other agents. We note that a majority of the run time for the scenario execution is due to the RTA logic computation time and not our tool. Additionally, while the run time of the evaluation is affected by the number of agents in the scenario, it is mostly affected by the set type of the unsafe set, where the polytope in the GCAS scenario causes the biggest slow down in evaluation time.

Table 2. Running time for execution and evaluation of RTAs with the tool suite provided for RTAEval.

		SimRTA				ReachRTA			
Scenario	Num agents	Exec time (s)	RTA comp time (ms)	% RTA Comp	Eval time (s)	Exec time (s)	RTA comp time (ms)	% RTA Comp	Eval time (s)
ACC	1	18.49e-3	0.07	76.63	7.27e-3	0.35	1.71	97.89	8.22e-3
	2	50.08e-3	0.10	84.12	18.16e-3	1.12	2.76	98.66	17.96e-3
	5	0.18	0.16	90.47	87.96e-3	6.01	5.96	99.10	0.10
Dubins	1	2.32	4.99	86.06	32.88e-3	15.18	37.10	97.76	34.15e-3
	3	15.30	11.84	92.89	0.18	71.60	58.83	98.60	0.11
	10	203.87	49.77	97.65	0.70	461.71	114.08	98.83	0.76
GCAS	1	5.85	6.08	83.12	30.62	39.28	47.65	97.02	31.423
	1	45.27	17.84	94.60	83.61	174.00	71.11	98.07	98.10

The summary of an RTA performance is given out in a text file from which visualizations like the one in Fig. 9 can be easily created. In addition to the computation time, distance from the unsafe sets, distance from the other agents, and controller usage, the minimum times to collision (TTC) for the unsafe sets and other agents are also reported. The summary information is saved in such a way that users can pull up snapshots of the scenario at any point in time. This means that the user can examine the state of the scenario that caused an unwanted result. Such functionality aids in the rapid prototyping of RTA technologies and logics.

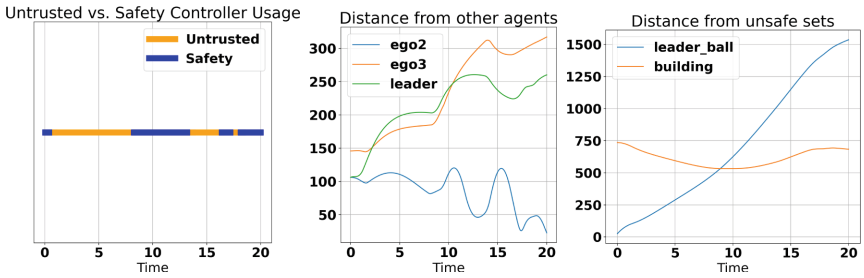

Fig. 9. Example visualization from three agent dubins scenario. From left to right: Controller usage plot, distance from other agents, and distance from unsafe sets. **ego2** and **ego3** denote the other aircraft.

4 Conclusion

We presented the RTAEval suite of Python-based tools for evaluating different runtime assurance (RTA) logics. Different RTA switching logics can be quickly coded in RTAEval, and we demonstrate its functionality in rapid prototyping of RTA logics on a variety of examples. RTAEval can be used in multi-agent scenarios and scenarios with perception models. Interesting next steps might include extending the functionality of RTAEval to filtering methods such as ASIF and scenarios that involve effects of proximity-based communication.

References

1. Aiello, A., Berryman, J., Grohs, J., Schierman, J.: Run-time assurance for advanced flight-critical control systems. In: Proceedings of AIAA Guidance, Navigation, and Control Conference, AIAA 2010–8041, Toronto, Ontario Canada, Aug., 2010 (2010)
2. Ames, A.D., Coogan, S., Egerstedt, M., Notomista, G., Sreenath, K., Tabuada, P.: Control barrier functions: theory and applications. In: 2019 18th European control conference (ECC), pp. 3420–3431. IEEE (2019)
3. Bak, S., Manamcheri, K., Mitra, S., Caccamo, M.: Sandboxing controllers for cyber-physical systems. In: 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems, pp. 3–12. IEEE (2011)
4. Cofer, D., et al.: Flight test of a collision avoidance neural network with run-time assurance. In: Digital Avionics Systems Conference (2022)
5. Desai, A., Ghosh, S., Seshia, S.A., Shankar, N., Tiwari, A.: Soter: a run-time assurance framework for programming safe robotics systems. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 138–150. IEEE (2019)
6. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: Carla: an open urban driving simulator. In: Conference on robot learning, pp. 1–16. PMLR (2017)
7. Dunlap, K.: Run Time Assurance for Intelligent Aerospace Control Systems. Ph.D. thesis, University of Cincinnati (2022)
8. Dunlap, K., Hibbard, M., Mote, M., Hobbs, K.: Comparing run time assurance approaches for safe spacecraft docking. IEEE Control Syst. Lett. **6**, 1849–1854 (2021)

9. Dunlap, K., Mote, M., Delsing, K., Hobbs, K.L.: Run time assured reinforcement learning for safe satellite docking. J. Aerosp. Inf. Syst. **20**(1), 25–36 (2023)
10. Fan, C., Miller, K., Mitra, S.: Fast and guaranteed safe controller synthesis for nonlinear vehicle models. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 629–652. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_31
11. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: a language for scenario specification and scene generation. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 63–78 (2019)
12. Hibbard, M., Topcu, U., Hobbs, K.: Guaranteeing safety via active-set invariance filters for multi-agent space systems with coupled dynamics. In: 2022 American Control Conference (ACC), pp. 430–436. IEEE (2022)
13. Mehmood, U., Sheikhi, S., Bak, S., Smolka, S.A., Stoller, S.D.: The black-box simplex architecture for runtime assurance of autonomous cps. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NFM 2022. LNCS, vol. 13260, pp. 231–250. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-06773-0_12
14. Mote, M.L., Hays, C.W., Collins, A., Feron, E., Hobbs, K.L.: Natural motion-based trajectories for automatic spacecraft collision avoidance during proximity operations. In: 2021 IEEE Aerospace Conference (50100), pp. 1–12. IEEE (2021)
15. Schierman, J., Ward, D., Dutoi, B., et al.: Run-time verification and validation for safety-critical flight control systems. In: AIAA Paper 2008–6338, Proceedings of the AIAA Guidance, Navigation, and Control Conference, Honolulu, Hawaii, Aug., 2008 (2008)
16. Seto, D., Krogh, B., Sha, L., Chutinan, A.: The simplex architecture for safe online control system upgrades. In: American Control Conference (ACC) (1998)
17. Sha, L., et al.: Using simplicity to control complexity. IEEE Softw. **18**(4), 20–28 (2001)
18. Shah, S., Dey, D., Lovett, C., Kapoor, A.: AirSim: high-fidelity visual and physical simulation for autonomous vehicles. In: Hutter, M., Siegwart, R. (eds.) Field and Service Robotics. SPAR, vol. 5, pp. 621–635. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-67361-5_40
19. Wadley, J., et al.: Development of an automatic aircraft collision avoidance system for fighter aircraft. In: AIAA Infotech@ Aerospace (I@ A) Conference, p. 4727 (2013)