Reactive Control Meets Runtime Verification: A Case Study of Navigation

Dogan Ulus(✉) and Calin Belta

Boston University, Boston, MA, USA
doganulus@gmail.com

Abstract. This paper presents an application of specification based runtime verification techniques to control mobile robots in a reactive manner. In our case study, we develop a layered control architecture where runtime monitors constructed from formal specifications are embedded into the navigation stack. We use temporal logic and regular expressions to describe safety requirements and mission specifications, respectively. An immediate benefit of our approach is that it leverages simple requirements and objectives of traditional control applications to more complex specifications in a non-intrusive and compositional way. Finally, we demonstrate a simulation of robots controlled by the proposed architecture and we discuss further extensions of our approach.

1 Introduction

Mobile robots are designed to work either in static and fully predictable environments such as automated warehouses or in open, partially unknown, and constantly changing environments. Classical deliberative control often works well for the former case while being inadequate or very inefficient for the latter. Alternatively, in reactive control approaches, robots continuously observe the environment at every level and thus are able to react and adapt to previously unknown circumstances. A common point between reactive control and runtime verification is that they both trade the completeness guarantees of deliberate control and model checking for online computation, practicality, and scalability. Following this synergy and growing interest in robotics using formal specifications, we think runtime verification techniques can raise the level of abstraction and assurance of reactive controllers in robotic applications.

In this paper, we explore the combination of reactive control and runtime verification techniques to construct controllers for mobile robots that satisfy given safety requirements and high-level mission specifications. To this end, we use a multi-layered architecture that can be seen in many reactive controllers and enhance each layer with runtime monitors[1] to search for desired behaviors on-the-fly. We depict our navigation architecture that contains several components from reactive control and runtime verification domains in Fig. 1. At the

[1] https://github.com/doganulus/python-monitors.

© Springer Nature Switzerland AG 2019
B. Finkbeiner and L. Mariani (Eds.): RV 2019, LNCS 11757, pp. 368–374, 2019.
https://doi.org/10.1007/978-3-030-32079-9_21

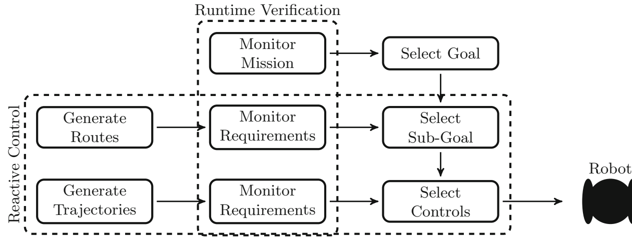

Fig. 1. The navigation stack used in the case study

bottom layer of the architecture, we employ limited trajectory search to devise the short-time motion of the robot. Runtime monitors are embedded to find trajectories that satisfy low-level safety properties such as collision avoidance and one-way regulations. The middle layer addresses the shortcomings of short-horizon trajectories by searching for a route over a connectivity graph of the environment. Mid-level safety properties for the graph traversal (e.g. avoiding specific areas) are similarly checked using runtime monitors in this layer. Once undesired trajectories and routes are filtered out, we use a number of features and heuristics to select the best one among the remaining. Repeating these procedures in real-time produces a safe motion for the robot to reach a specific (goal) location relative to trajectory/route generation specifics. Finally, the top layer is designated for high-level mission control that enforces the correct order of locations to be visited and we similarly employ runtime monitors constructed from mission specifications for the mission control.

2 Environment, Robots, and Specifications

For our case study, we will work on a relatively complex 2D environment designed to give a representative view of real challenges without introducing too much detail. Depicted on the left of Fig. 2, our environment represents an office space with rooms (R1–R6), narrow passages (such as doors D1–D6), named locations (A–D), and some regulations at certain regions (one-way regions) including other

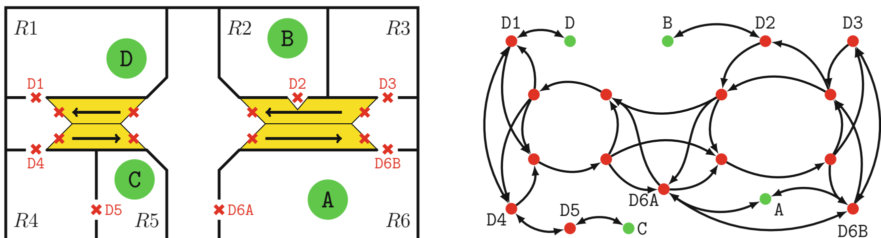

Fig. 2. Environment maps: Geometric on the left and topological on the right

(possibly uncontrolled) agents. We use a unicycle velocity-controlled model for the robot dynamics where the state space is defined by robot's position (x, y) and orientation θ, and controlled by forward and angular velocity commands $u = (v, \omega)$. It is of critical importance that the complexity of the environment determines the complexity of specifications and monitoring. For a static environment (that is to say, nothing changes outside of our control), we do not need any runtime monitoring at all. This is obviously a very strong assumption for many cases. On the other hand, if dynamic obstacles (such as other agents) exist in the environment, we have to at least add a basic monitoring mechanism that checks simple propositions—will the robot collide with anything soon or did the robot reach its goal? Moreover, if we have more complex regulations and tasks to complete in the environment, runtime monitors automatically constructed from rich specification languages seem a preferable option. Therefore, our robots in this study are assigned to perform complex navigation missions, specified by regular expressions, while avoiding static and dynamic obstacles as well as satisfying desired properties and regulations, specified by temporal logic formulas.

3 Search for Safe Motion

We here demonstrate an application of runtime monitors in searching for the desired safe behavior enhancing existing trajectory and route search algorithms. The general procedure can be summarized in three steps: (1) Generating a number of alternative behaviors (trajectories or routes), (2) discarding unsafe/undesired behaviors using runtime monitors, and (3) selecting the best remaining (thus safe) behaviors according to a predefined set of heuristics. Importantly, the extent of these search processes is limited due to available computational resources as well as that long-term complete plans may become invalid very quickly in dynamic and uncertain environments. In the following, we give more details about search procedures and actual properties used in the case study.

Trajectory Search. Dynamic Window Approach (DWA) [3] is a well-known collision avoidance and local motion planning algorithm that uses search procedures to find control actions (velocity commands) while considering robot's dynamics. The search space of DWA is limited by maximum acceleration available to the robot as depicted on the left of Fig. 3 and the algorithm samples a set of control actions. Then it calculates the future trajectories of each alternative action over a limited time horizon as illustrated on the right of the figure.

Originally being a collision avoidance algorithm, the only safety requirement over these trajectories considered in DWA is never getting dangerously close to obstacles, which is usually hard-coded into the algorithm. On the other hand, we are interested in checking such requirements using runtime monitors so that we can extend the approach for any temporal logic formula. We start our case study by expressing the collision avoidance requirement in temporal logic as follows.

$$\texttt{never(dangerously_close(obstacles))} \qquad \text{(CA)}$$

where dangerously_close is a predicate that computes whether any intersection occurs between obstacles and robot's footprint.

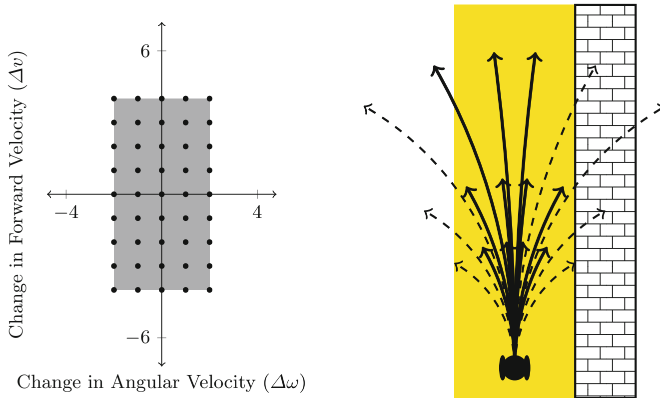

Fig. 3. (Left) A finite set of admissible velocity commands for the next time step relative to the current velocity. The search space, depicted in gray, is constrained by maximum allowed accelerations of the robot. (Right) Future trajectories of the robot simulated for each admissible velocity command. Dashed trajectories contain a violation in specification so commands that lead to these trajectories are discarded.

In this case study, besides collision avoidance, we also want our robot to obey one-way regulations of the environment, which state that robots have to move in a single direction inside certain regions. The direction of one-way regions is either west or east in our environment. We call these regions westways and eastways accordingly and predicates inside_westway and inside_eastway check whether the robot is in these regions. Moreover, we define some auxiliary formulas to detect whether the robot just entered a one-way region such that

```
entered_eastway : inside_eastway and not previously inside_eastway
entered_westway : inside_westway and not previously inside_westway
```

The desired direction in a one-way region is checked by predicates going_east and going_west and we write our safety properties for each type of one-way regions as follows:

```
inside_eastway implies (going_east since entered_eastway)     (OW-E)
inside_westway implies (going_west since entered_westway)     (OW-W)
```

Finally, we construct our runtime monitor to check the conjunction of (CA), (OW-E), and (OW-W) requirements over generated trajectories. Control actions that produce violating trajectories are discarded before the selection phase. This ensures the safety of selected control action if there exists one in alternatives

otherwise we apply a full brake. The last piece of trajectory search is to select the best one among safe trajectories according to a weighted sum of some predefined heuristics, namely final speed of the trajectory (higher is better), final-distance-to-goal (lower is better), minimum-distance-to-obstacles (higher is better). In the case study, the actual values of weights are found empirically.

Route Search. Given a connectivity graph of these locations, we can search for a route from the current location to the actual goal location and each node on the route is passed to the lower layer as a (sub) goal. In the search of a suitable route, we need to take into account some extra requirements. On the other hand, external runtime monitors are desirable to enforce application-specific properties as in trajectory search rather than generating a new graph search algorithm for each and every one of them. For example, consider a property such that the robot never uses the door `D6A` when going from the location `D` to `A`, which can be expressed as follows.

```
(visit(A) && once visit(D)) -> (!visit(D6A) since visit(D))    (ND)
```

We then construct a runtime monitor from the property (ND) to check routes generated over the graph. In particular, we use an off-the-shelf implementation of the shortest path algorithm [8] that generates simple paths starting from the shortest one. Sequentially checking these paths using runtime monitors constructed from temporal logic formulas [4,6] ensures that the we select the shortest route that satisfies specified properties and then we can update the route of the robot accordingly.

4 Navigate by Regular Expressions

In this section, we use regular expressions to specify complex navigation missions and guide the mission execution via runtime monitors constructed from the specification. Navigation missions describe the desired behavior of the robot over a set of observations and regular operations of sequential composition (`;`), alternative choice (`|`), and repetition (`*`) are used to express the ordering between these observations. For example, a robot is said to reach a region `A` when it was outside for a while and then entered the region `A`. We can specify such a behavior using regular expressions as follows:

```
reach(A) = (outside(A))*; inside(A)
```

where atomic propositions `inside(A)` and `outside(A)` check whether the robot is in the region `A` or not. Similarly more complex missions are obtained by composing simple missions as below.

```
mission1 : (reach(C); reach(B)|reach(D); reach(A))*    (M1)
```

which specifies a (robot) behavior to repeatedly visit the regions `A`, and `C` while visiting `B` or `D` in-between. From this expression, we construct a runtime monitor [7] that associates a Boolean state variable for each proposition and updates them according to previous states and robot's position at each time step. The next sub-goal of the robot is determined according to the state vector of the monitor.

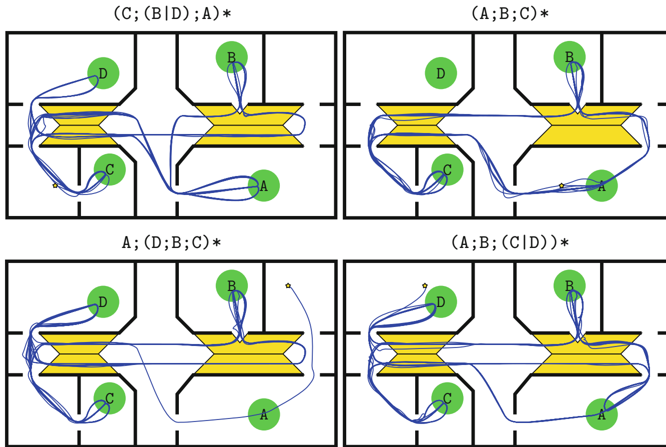

Fig. 4. Trajectories of robots G1–G4 assigned with missions M1–M4, respectively. (Color figure online)

Finally we present our simulation results of four robots G1–G4 operated in the same environment and controlled by the proposed architecture. We assign the first robot G1 with the mission M1 and the rest G2–G4 with missions M2–M4 below, respectively.

```
mission2 : (reach(A); reach(B); reach(C))*                (M2)
mission3 : (reach(A); (reach(D); reach(B); reach(C))*     (M3)
mission4 : (reach(A); reach(B); (reach(C)|reach(D))*      (M4)
```

In Fig. 4, we separately show the simulated trajectories of the robot for a certain duration that covers several loops as specified in the mission. The initial position of the robot is marked by a yellow star. Robots get close to each other quite frequently and evading maneuvers cause small variations among loops seen in the figure. Overall we see that the robots successfully avoid each other and static obstacles and obey regulations of the environment while performing their formally-specified missions over achieving reasonable trajectories.

5 Conclusion

We presented an example and novel use of provably correct runtime monitors to control a mobile robot subject to complex safety requirements and mission

specifications in a dynamic environment. We embedded runtime monitors into a layered reactive control architecture together with other simple and scalable components to achieve a navigation solution that does not require strong assumptions. Our approach amounts to a more active use of runtime monitors beyond checking assumptions of an offline motion planner at runtime [1,2,5]. We believe the simplicity and breadth of runtime monitors would make them ideal to cover many use cases and increase the level of assurance in robotic applications.

References

1. Medina Ayala, A.I., Andersson, S.B., Belta, C.: Temporal logic motion planning in unknown environments. In: Intelligent Robots and Systems (IROS), pp. 5279–5284. IEEE (2013)
2. Desai, A., Dreossi, T., Seshia, S.A.: Combining model checking and runtime verification for safe robotics. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 172–189. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_11
3. Fox, D., Burgard, W., Thrun, S.: The dynamic window approach to collision avoidance. IEEE Robot. Autom. Mag. **4**(1), 23–33 (1997)
4. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. Int. J. Softw. Tools Technol. Transfer **6**(2), 158–173 (2004)
5. Lahijanian, M., Maly, M.R., Fried, D., Kavraki, L.E., Kress-Gazit, H., Vardi, M.Y.: Iterative temporal planning in uncertain environments with partial satisfaction guarantees. IEEE Trans. Rob. **32**(3), 583–599 (2016)
6. Ulus, D.: Online monitoring of metric temporal logic using sequential networks. arXiv preprint arXiv:1901.00175 (2019)
7. Ulus, D.: Sequential circuits from regular expressions revisited. arXiv preprint arXiv:1801.08979 (2018)
8. Yen, J.Y.: Finding the k shortest loopless paths in a network. Manage. Sci. **17**(11), 712–716 (1971)