Towards the Verification of Safety-critical Autonomous Systems in Dynamic Environments

Adina Aniculaesei
TU Clausthal
38678 Clausthal-Zellerfeld, Germany
adina.aniculaesei@tu-clausthal.de

Daniel Arnsberger
TU Clausthal
38678 Clausthal-Zellerfeld, Germany
daniel.arnsberger@tu-clausthal.de

Falk Howar
TU Clausthal
38678 Clausthal-Zellerfeld, Germany
falk.howar@tu-clausthal.de

Andreas Rausch
TU Clausthal
38678 Clausthal-Zellerfeld, Germany
andreas.rausch@tu-clausthal.de

There is an increasing necessity to deploy autonomous systems in highly heterogeneous, dynamic environments, e.g. service robots in hospitals or autonomous cars on highways. Since unforeseen situations may occur in these environments, the verification results obtained with respect to the system and environment models at design-time might not be transferable to the system behaviour at runtime. For autonomous systems operating in dynamic environments, motion safety is a critical requirement. In this work, we present a two phase process in order to address the passive safety property. At the design phase, we exploit UPPAAL to formalize the autonomous system and its environment as timed automata and the safety property as TCTL formula. After verifying the correctness of these models with respect to the system's safety requirement, we build a monitor to check whether the assumptions made at design time are also correct at run time. If the current system observations of the environment do not correspond to the initial system assumptions, the monitor sends feedback to the system and the system enters a passive safe state.

1 Introduction

These days, autonomous systems are deployed mostly in known environments, e.g. industrial robot systems in production plants [13]. At design time, formal methods can offer strong guarantees with respect to specific properties of the system behaviour, if accurate system and environment models can be obtained. At runtime, if the system and environment behaviour fit their corresponding models, then it is guaranteed that the system behaviour satisfies the correctness properties formulated with respect to the system model at design time. However, there is an increasing necessity to deploy autonomous systems in highly heterogeneous, dynamic environments, e.g. service robots in hospitals or autonomous cars on the highways. Since unforeseen situations may occur in these environments, the verification results obtained with respect to the design-time models might not be transferable to the system behaviour at runtime.

Autonomous systems, such as mobile robots or autonomous vehicles belong to the spectrum of safety-critical applications. For this kind of systems, motion safety is a vital safety requirement. In this paper we present a novel concept for the verification of safety-critical autonomous systems in dynamic environments, with focus on the passive safety property. In [6], a system state is defined as safe under the passive safety property if there exists at least one braking maneuver starting in this state, which is collision-free for the duration of the system's braking time. In other words, when in such a state, the system does not actively collide with obstacles in its environment.

We establish the following premises for our verification problem. Firstly, the system's environment is dynamic and heterogeneous. Due to this heterogeneity, there is an infinite number of unforeseen situations, which cannot be modeled and verified at design time. Secondly, the autonomous system under verification is equipped with sensors, through which the system can detect in real time the changes occurred in the environment. Furthermore, the system starts to run with predefined assumptions about its environment. The system safety property is verified at design time against the modeled system's behaviour and the system's assumptions. In this paper we present a concept for the verification of autonomous systems in dynamic environments. Using this concept, we can ensure that the system's safety property is satisfied for a larger subset of the runtime environment than the one covered by the system's assumptions about its environment.

Our concept presents a two-phase process for our verification problem. In the design phase, we make use of the UPPAAL model checker [2] to formalize the system and environment models as timed automata and the system's safety property as TCTL formula. After we verify the system model against its safety property, we use the system and the environment model to build a monitor which checks whether the system assumptions made at design time are also correct at run time. Our case study and concept evaluation are built on the scenario of a mobile service robot driving towards a given goal in a simulation environment.

1.1 Paper Structure

The structure of this paper is as follows: Section 2 contains an overview on previous work; Section 3 presents our concept for the verification of autonomous systems in dynamic environments. Section 4 introduces the example scenario and Section 5 discusses the case study. The results of our evaluation can be found in Section 6. We draw conclusions and discuss future work in Section 7.

2 Related Work

Prior approaches [1, 3, 7, 9] focus on modeling and verification techniques to ensure collision safety for autonomous systems dynamic or unknown environments. Other works [8] present a more including approach, which reflects on the whole spectrum of cyber-physical systems. We consider these papers in turn and discuss the difference to our concept. The paper in [3] addresses the problem of passive motion safety of a mobile robot with limited field-of-view deployed in an unknown environment. The presented approach is a safety navigation scheme, which avoids braking inevitable collision states, i.e. states which regardless of the robot's trajectory lead to collision, to achieve collision safety in environments with moving obstacles.

In the work in [7] the authors use theorem proving techniques to verify the passive safety and passive friendly safety properties for the collision avoidance algorithm dynamic window, for autonomous robotic ground vehicles. Both properties are verified with respect to an environment which contains stationary as well as moving obstacles. The paper makes use the differential dynamic logic [10] as a a modeling formalism for the hybrid models which describe the continuous physical motion of the robot as well as its discrete control choices.

The paper in [8] presents the ModelPlex approach, which combines offline verification of CPS models with runtime validation of system executions in order to provide correctness guarantees for CPS executions at runtime. The method uses theorem proving with sound proof rules to synthesize three runtime monitors, i.e. model monitor, controller monitor and prediction monitor, from hybrid system models.

The model monitor checks the system execution for deviations from the system model. The controller monitor tests the current controller decisions of the system implementation for compliance with the system model, while the prediction monitor evaluates the worst-case safety impact of the current controller decisions with respect to the predictions of a bounded deviation plant model.

Phan et al. [9] present a runtime approach based on the Simplex architecture [11], to ensure collision-freedom for robots with limited field-of-view and limited sensing range in unknown environments, i.e. environments where the detailed shapes and the locations of the obstacles are not known in advance. The switching condition between the advanced controller and the baseline controller is computed using extensive geometry reasoning. The approach guarantees collision-freedom if the obstacles are stationary. However, the authors claim that the approach can be extended for environments containing moving objects to ensure passive safety, if a bound on the obstacle velocity is known.

Alami et al. [1] present an approach which computes the maximum velocity profile of a mobile robot moving over a planned trajectory in an environment with an arbitrary number of obstacles. Both robot dynamics and environment, as well as the constraints of the robot sensors are considered in the computation of the velocity profile. The planned profile is indicative of maximum speeds that the robot may have along its path without colliding with any object which could intercept its future trajectory. The maximum possible velocity with which the mobile objects move in the environment is known in advance.

Kane et al. [5] address the runtime verification of an ARV with black-box commercial-off-the-shelf components, which are not amenable to instrumentation. Instead, the authors propose an approach to passive monitoring of the target system, by generating high-level property constructs from the observed network state. The runtime monitoring algorithm developed in this paper incrementally takes as input a system state and a MTL formula and checks the state trace for violations. In order to give the system enough time to react to environment changes, the authors propose to reduce the formula as soon as possible using history summarizing structures and simplifications based on formula-rewriting.

In contrast to these approaches, we work on the premise that the real environment is highly heterogeneous and infinitely many situations may appear which cannot be verified at design time. The autonomous system under verification has assumptions about the input it can receive from the environment, e.g. the maximum velocity of moving obstacles. However, the changes which may occur in the environment can invalidate these assumptions, and thus render the system behaviour as non-conform with regard to its motion safety specification.

3 Concept

In this section we present our concept for verification of safety-critical systems in dynamic environments. We developed our concept starting from the following premises:

- The environment is heterogeneous and the infinite number of possible situations cannot be modeled and verified at design time.
- The system starts to run with predefined assumptions about its environment.
- The system together with its assumptions has been verified at design time against its safety specification.
- The system observes the changes in the environment in real time.

We divide our concept in two parts: design time and runtime, as illustrated in Figure 1.

At design time, M_S describes the system behaviour, while M_E models the environment. These models are joined together through the system assumptions SA. Thus, SA is the explicit interface between M_S and

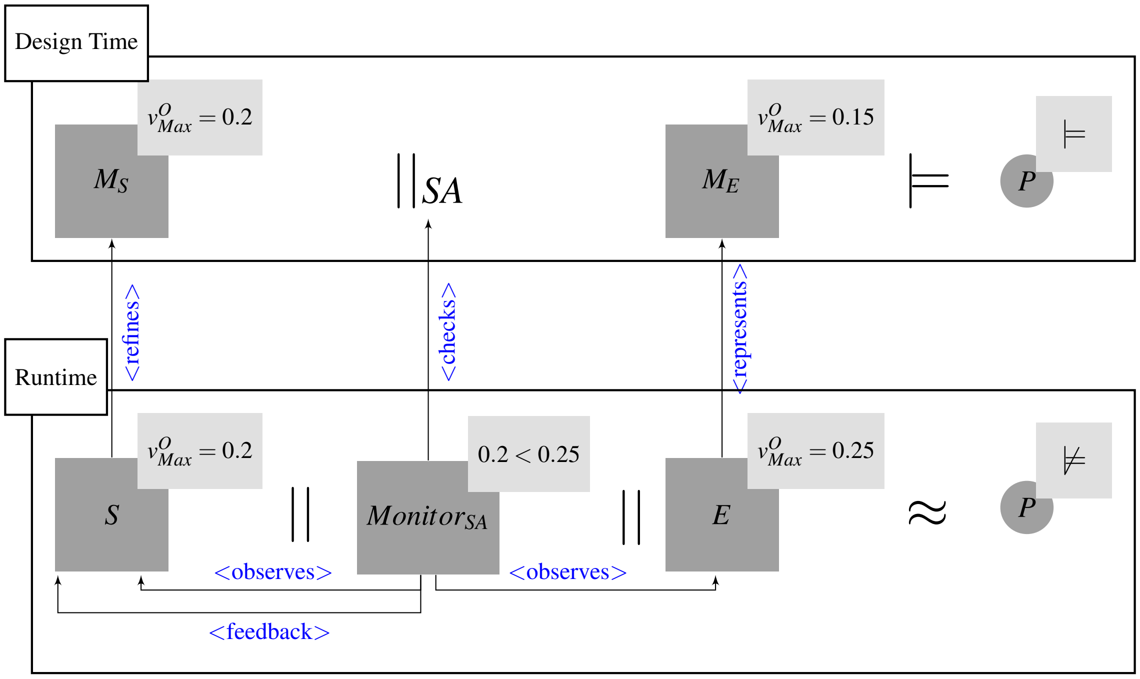


Figure 1: Verification concept with runtime monitoring

M_E. The passive safety property is formulated with regard to the system model M_S in its environment described in M_E and is expressed as the property specification P. Model checking is used to verify whether the system model's behaviour satisfies P or not. The environment model M_E is constructed so that M_S always satisfies P.

We complement formal verification methods used at design time with runtime monitoring of the environment. In contrast to design time when all system executions can be inspected, only the current system run can be verified against the safety property. The system S refines the behaviour described in its model M_S. During its runtime, the system S operates in its environment E and a monitor $Monitor_{SA}$ observes both in order to check whether the system assumptions made during design time are still correct at runtime. If the system assumptions are correct, the property specification P is satisfied. Otherwise, the monitor gives feedback to the system and the system enters a passive safe state.

In figure 1 example values are attached to each component involved in the verification problem. At design time, the system assumption about the environment is *0.2* and the environment has a real upper bound of *0.15*. Therefore, the property P is indeed satisfied. At runtime, the environment has an upper bound of *0.25* instead of *0.15* while the system assumption remains unchanged. The monitor detects that the assumption is invalid and hence that P is violated.

4 Scenario

Our scenario takes place in a simulation environment in which a mobile service robot is commissioned to perform deliveries reach a given destination. The environment features both stationary and moving objects.

Figure 2 depicts an abstract view of the environment, in which the robot drives to its goal, while an obstacle moves on the same lane from the opposite direction. Static obstacles occupy the neighboring left and right lanes. The robot has partial knowledge of its surroundings due to its sensory limitations, i.e. its field of view spans up to range R.

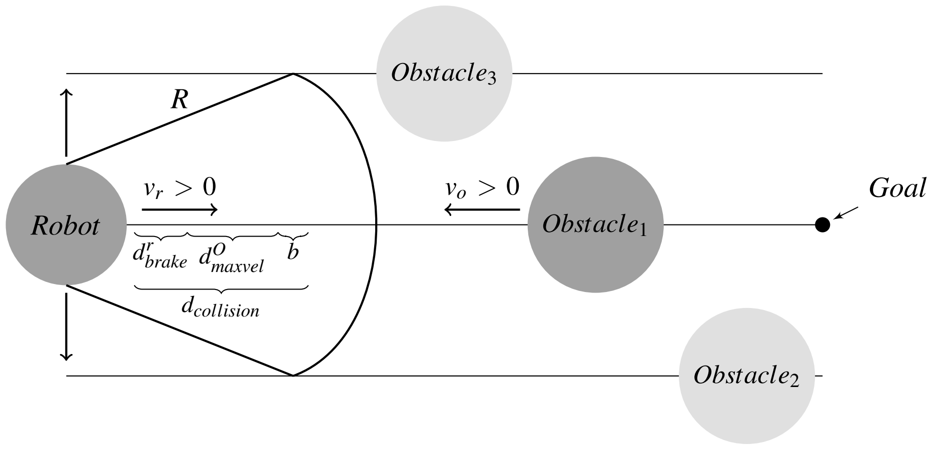


Figure 2: Scenario and Physical Model.

There are a few constraints we have imposed on the robot and on its environment, without affecting the generality of our concept. The robot and the dynamic obstacle move towards each other in a two-dimensional space. The robot has the ability to observe its environment and react to its changes. However, the robot reacts only to environment changes which occur in front of it. Even though it cannot rotate, the robot can change lanes by moving sideways.

As it starts to drive, the robot accelerates until it has reached its maximum velocity. Then, it continues to drive with this velocity until it reaches its destination or until it brakes due to collision danger. In order to detect a possible collision, the robot computes the distance $d_{collision}$ and compares it with the current distance between the robot and the moving obstacle. The distance $d_{collision}$ is calculated as follows:

$$d_{collision} = d^r_{brake} + d^o_{maxvel} + b \tag{1}$$

where:

- $d^r_{brake} = v^r * t^r_{brake}$ is the braking distance of the robot based on its current velocity v^r,
- $d^o_{maxvel} = v^o_{max} * t^r_{brake}$ is the distance covered by the obstacle moving with maximum velocity v^o_{max} during the robot's braking time t^r_{brake}, and
- b is a look-ahead buffer distance based on the assumptions. This is necessary in order for the robot to stop before causing a collision with the obstacle.

When detecting a possible collision, the robot also checks for the possibility to change the lane rather than braking immediately. Without reducing the generality of our concept, we assume that there is only one moving obstacle in the robot's environment. The obstacle's velocity is bounded by a maximum value, $v_o \in (0, v^o_{max}]$ and can change randomly in the interval boundaries given by v^o_{max}.

5 Case Study

In this section, we present our case study, on the basis of which we evaluate our concept. In the scenario introduced in Section 4, we presented an informal specification for a mobile service robot, i.e. the robot must reach a specific point, while avoiding obstacles as well as possible by braking timely before a collision takes place.

In order to check if the robot complies with its specification, we use model checking to formally verify the behaviour of the robot against its specification. We use the UPPAAL model checker [**?**] to describe the robot's behaviour and that of its environment, and to formalize the robot's specification. UPPAAL uses timed automata as its modeling language and Timed Computation Tree Logic (TCTL), a subset of Computation Tree Logic (CTL), as its query language.

5.1 System and Environment Models

In Figure 3 and Figure 4 two abstract automata are illustrated, which model the behavior of the robot and that of the obstacles in the environment. In order to simplify the models, we abstract from their characteristics as physical objects and consider the robot and the obstacles as discrete points in a two-dimensional space. The robot's behaviour is verified against the previously given specification. Therefore, the robot cannot drive through the obstacles, since this would correspond to a collision actively caused by the robot. However, an obstacle can move through the robot, thus emulating its movement past the robot.

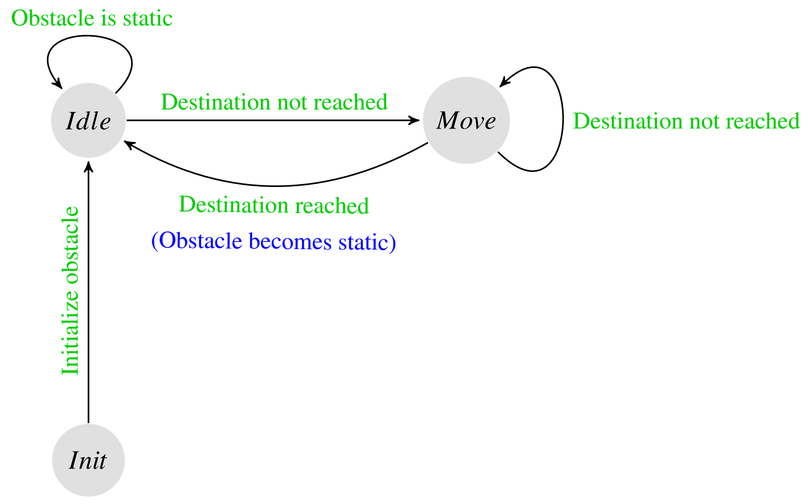


Figure 3: Abstract obstacle automaton

Since the environment contains multiple obstacles, the template of the obstacle automaton is instantiated with various parameters in order to depict the two types of obstacles in the environment. The automaton has three locations, the initial location *Init* and the locations *Idle* and *Move*. On the first transition from *Init* to *Idle*, the obstacle's parameters, i.e. identification number, initial position, whether it is static or not, are stored in a global obstacle array. In the *Idle* state an obstacle can be either static or it can begin to move towards the robot in a continuous motion. The obstacle velocity has an upper bound and the obstacle automaton chooses arbitrarily its velocity from the interval $(0, v^o_{max}]$, each time the obstacle executes a move. Based on the current velocity, the new obstacle position is computed and updated. The obstacle automaton is modeled so that a moving obstacle has a destination, and upon reaching it, the obstacle becomes static, i.e. the automaton enters the location *Idle*. This behavior ensures that the obstacle eventually stops moving.

The abstract robot automaton is displayed in Figure 4 and it has the following locations:

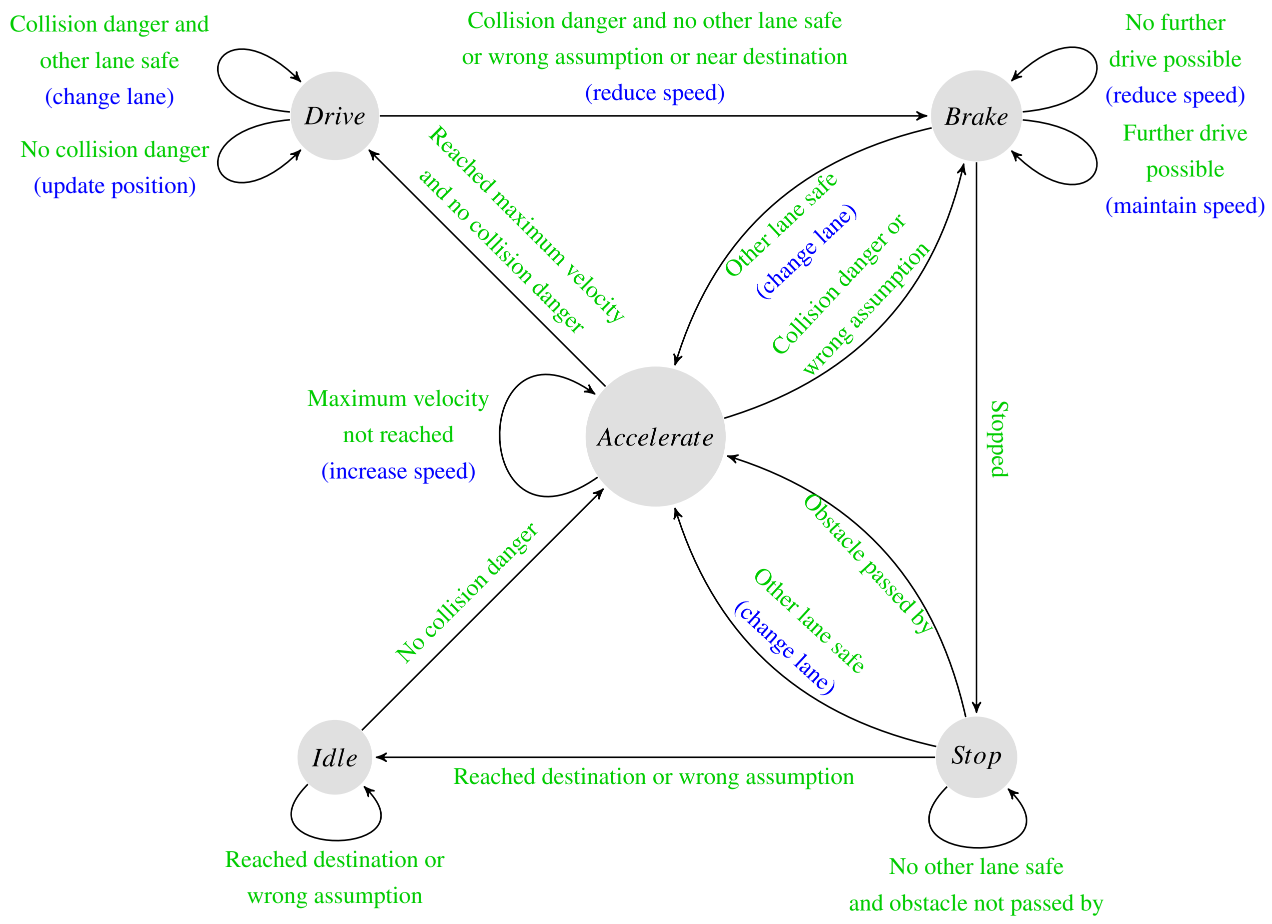

Drive
Brake
Accelerate
Idle
Stop
Collision danger and other lane safe (change lane)
No collision danger (update position)
Collision danger and no other lane safe or wrong assumption or near destination (reduce speed)
No further drive possible (reduce speed)
Further drive possible (maintain speed)
Reached maximum velocity and no collision danger
Other lane safe (change lane)
Collision danger or wrong assumption
Maximum velocity not reached (increase speed)
Stopped
No collision danger
Obstacle passed by
Other lane safe (change lane)
Reached destination or wrong assumption
Reached destination or wrong assumption
No other lane safe and obstacle not passed by

Figure 4: Abstract robot automaton

- **Idle** is the initial location of the robot automaton, because the robot is stationary at the beginning. When it reaches its destination, the robot stops, so this is also the final state.
- **Accelerate** is the location in which the automaton remains as long as the robot has not reached its maximum speed.
- **Drive** is the location in which the automaton remains as long as the robot can drive without any collision danger.
- **Brake** is the location which the automaton enters if collision danger or wrong assumption is detected.
- **Stop** is the location which the automaton enters when the robot stands still after braking.

The transitions from one location to another are guarded with different conditions (depicted in green), which express if the respective transition is enabled or not. Update statements (illustrated in blue) are used to change accordingly the values of various variables, e.g. robot speed or position.

We elaborate only on one of the functions used in the transitions guards, the function *collisionDanger* shown in Listing 1. It calculates whether a possible collision is ahead or not. In order to perform its computation, the function considers the maximum obstacle velocity assumed by the robot.

```
bool collisionDanger() {
    int i = 0;
    while (i < N) {
        if (robotLane == obstacles[i].lane) {
            if (robotPosition <= obstacles[i].position && obstacles[i].
                position - robotPosition <= visualRadius) {
                if ((robotPosition + brakingDistance(vMax) +
                    obstacleDrivingDistance(vMax)) >= obstacles[i].position
                    - BUFFER && !obstacles[i].static) {
                    return true;
                }
                else if (robotPosition + brakingDistance(vMax+1) >=
                    obstacles[i].position && obstacles[i].static) {
                    return true;
                }
            }
        }
        i++;
    }
    return false;
}
```

Listing 1: Guard function which checks for collision danger

The function checks for all obstacles in the environment, if the obstacle is on the same lane as the robot and if a collision is possible. The computations for the detection of collision danger are performed if and only if the obstacle has entered the robot's visual range. Two other functions are used to in this computation: *brakingDistance* and *obstacleDrivingDistance*. The first function computes the robot's braking distance for its maximum velocity, while the latter calculates the distance which the robot assumes the obstacle can still drive during its braking time. We further add a look-ahead buffer distance, in order to account for the fact that the robot gets the current informations about the obstacle one time step after the update is sent. Therefore, the collision distance between the robot and the obstacle is an upper bound. We also distinguish between dynamic and static obstacles, hence the two different computations. If there is a static obstacle in front of the robot, the computation is performed using only the robot's own speed.

5.2 Safety Property

The robot specification states that it must at all times comply with its safety property, namely never actively collide with an obstacle. This is expressed in Equation 2, which translates to the robot having no speed at the moment when an obstacle and the robot occupy consecutive positions on the same lane.

$$\begin{aligned} A[]\ forall\ (i : int[0, N-1])\ R.y\ &==\ obstacles[i].y \\ &and\ obstacles[i].x\ >\ R.x \\ &and\ (obstacles[i].x\ -\ R.x\ <=\ 1) \\ &imply\ R.v\ ==\ 0 \end{aligned} \tag{2}$$

The satisfaction of this specification depends on the robot's assumption about the maximum obstacle speed. If the assumption is correct or greater than the real value, then the specification is satisfied and the robot stops in time. Otherwise, the robot brakes too late and a collision cannot be avoided.

5.3 From UPPAAL Models to Implementation

The implementation of this models was done with the help of automata-based programming [**?**]. Without reducing the generality of our concept, we implemented the robot without the functionality of changing lanes. Thus, we focused only on the timely braking of the robot, namely on the conditions in which this takes place, i.e. robot parameters or environment inputs.

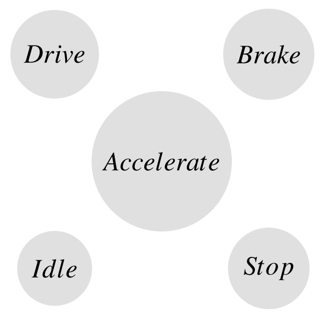
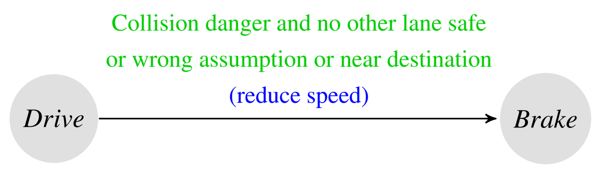
UPPAAL	**Implementation**
	`class RobotStates(Enum):` `Idle = 0` `Accelerate = 1` `Drive = 2` `Brake = 3` `Stop = 4`
 Collision danger and no other lane safe or wrong assumption or near destination (reduce speed) Drive Brake	`if (self.state == RobotStates.Drive and` `(self.collisionDanger() or self.` `wrongAssumption() or self.` `nearDestination())):` `self.current_velocity_x -=` `self.velocity_increment` `self.state = RobotStates.Brake`

Table 1: Transformation from UPPAAL models to Python source code.

Table 1 shows two examples of how the UPPAAL models are transformed into Python code. The first row shows how the five different robot states are declared. The left side shows an abstract automaton of the UPPAAL model. In Python, we use an enumeration class to set up the corresponding states. The second row exemplifies the transition step of the UPPAAL automaton through the transition between the states *Drive* and *Brake*. The functions *collisionDanger* and *wrongAssumption* implement the functionality specified in the UPPAAL model. We consider only one lane at runtime, as we chose not to implement the functionality of changing lanes in the robot. The same transformation was done for all the other transitions of the robot automaton. The obstacle automaton undergoes the same kind of transformation in Python source code, in order to move the dynamic obstacle in the simulation environment.

6 Evaluation and Discussion

In order to evaluate the approach presented in Section 3, we built the scenario in a simulation environment for robots and implemented the behavior which we have modeled in the case study.

We built a test suite, in which we chose the robot's assumption about the maximum obstacle velocity to be always 0.2 m/s. We experimented with various maximum obstacle velocities as well as different radii of the robot's reaction area. When it observes a wrong assumption inside its reaction area, the robot begins to brake in order to enter a passive safe state. This area can be smaller than the visual range covered by the robot's sensors. As evaluation criterion for our concept we use the number of collisions which take place.

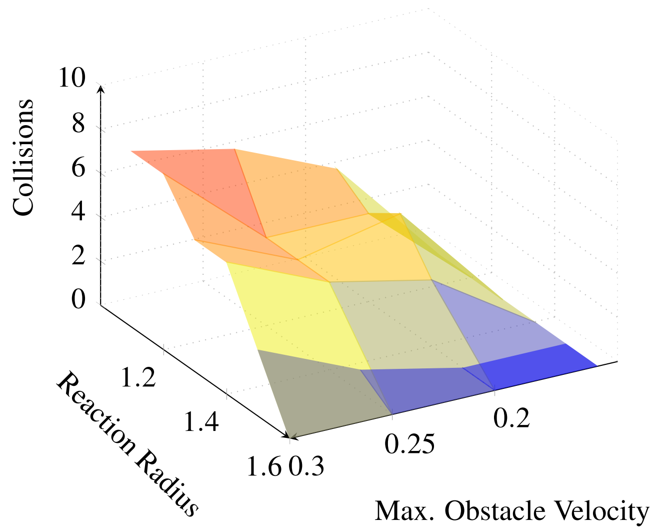


Figure 5: Evaluation Results

Figure 5 shows the tests results. The blue region illustrates the tests which were already covered through model checking at design time. The maximum obstacle velocity did not exceed the robot's assumption and the reaction area was large enough for the robot to brake in time. The test cases in which the collisions took place are ordered in ascending order after the number of collisions and distributed in color-coded regions, from yellow to red, accordingly. When the reaction radius was chosen too small, the robot could not avoid a collision, even if its assumption was not exceeded by the obstacle. Furthermore, collisions took also place when the robot's reaction radius was chosen large enough, but the maximum obstacle speed was too high. Nevertheless, we were able to reduce the number of collisions by choosing an appropriate reaction radius for the different maximum obstacle velocities.

7 Conclusions and Future Work

This paper proposed the use of runtime monitoring to complement formal methods in the verification of safety-critical autonomous systems in heterogeneous environments. We developed a two-phase process to verify the behaviour of a mobile service robot in a simulation environment with respect to the passive safety property. In the design phase, we formalized the system and the environment models as timed automata in UPPAAL. We verified these models along with the system's assumptions about the environment against the system's safety property expressed as a TCTL formula. At runtime, we built a runtime monitor to check whether the system assumptions made at design time are still correct at runtime.

Our evaluation results show that we were able to ensure the satisfaction of the system's safety property for a larger subset of the runtime environment than the one covered by the assumptions which the system has about this environment. The size of this subset depends not only on the maximum obstacle velocity, but also on the reaction area of the robot.

In future work we want to explore possible dependencies between system's assumptions and the effects one or a subset of incorrect assumptions may have on the runtime system's behavior. In this work, the implementation of the functionality described in the system model was done manually. Automated model transformation methods can be applied to extract the system implementation more easily from the system model. However, this transformation must in turn be verified to ensure it is performed correctly. In this respect, we want to identify a suitable verification method for model transformation (see [4] for an extensive survey) and apply it to our case study. Furthermore, we intend to expand to several dynamic obstacles in the runtime environment and include sideway motion in an arbitrary manner for the dynamic obstacles, e.g. obstacles crossing the path of the autonomous system. Another aspect we want to consider is refined kinematic capabilities for the autonomous system, e.g. rotating, driving backwards or being outrun by moving obstacles.

References

[1] R. Alami, K. M. Krishna & T. Siméon (2007): *Provably Safe Motions Strategies for Mobile Robots in Dynamic Domains*. *Autonomous Navigation in Dynamic Environments*, pp. 85–106, doi:10.1007/978-3-540-73422-2_4. Available at `http://dx.doi.org/10.1007/978-3-540-73422-2_4`.

[2] G. Behrmann, A. David & K. G. Larsen (2004): *A Tutorial on Uppaal*. In M. Bernardo & F. Corradini, editors: *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, *LNCS* 3185, Springer, Bertinoro, Italy, pp. 200–236, doi:10.1109/ITSC.2008.4732685. Available at `http://dx.doi.org/10.1007/978-3-642-22306-8_11`.

[3] S. Bouraine, T. Fraichard & H. Salhi (2012): *Provably Safe Navigation for Mobile Robots with Limited Field-of-Views in Unknown Dynamic Environments*. In: *IEEE International Conference on Robotics and Automation (ICRA 2012)*, pp. 174–179, doi:10.1109/ICRA.2012.6224932. Available at `https://hal.inria.fr-hal-00768527`.

[4] D. Calegari & N. Szasz (2013): *Verification of Model Transformations: A Survey of the State-of-the-Art*. In Y. Donoso & R. Santos, editors: *Proceedings of the {XXXVIII} Latin American Conference in Informatics (CLEI)*, *ENTCS* 292, pp. 5–25, doi:10.1016/j.entcs.2013.02.002. Available at `http://dx.doi.org/10.1016/j.entcs.2013.02.002`.

[5] A. Kane, O. Chowdhury, A. Datta & P. Koopman (2015): *A Case Study on Runtime Monitoring of an Autonomous Research Vehicle (ARV) System*. In E. Bartocci & R. Majumdar, editors: *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015, Proceedings*, *LNCS* 9333, Springer International Publishing, Vienna, Autria, pp. 102–117, doi:10.1007/978-3-319-23820-3_7. Available at `http://dx.doi.org/10.1007/978-3-319-23820-3_7`.

[6] K. Maček, D. Vasquez, T. Fraichard & R. Siegwart (2008): *Safe Vehicle Navigation in Dynamic Urban Scenarios*. In: *11th International IEEE Conference on Intelligent Transportation Systems 2008 (ITSC 2008)*, *LNCS* 8734, Springer Berlin Heidelberg, Beijing, China, pp. 482–489, doi:10.1109/ITSC.2008.4732685.

[7] S. Mitsch, K. Ghorbal & A. Platzer (2013): *On Provably Safe Obstacle Avoidance for Autonomous Robotic Ground Vehicles*. In P. Newman, D. Fox & D. Hsu, editors: *Proceedings of Robotics: Science and Systems*, IX, Berlin, Germany, doi:10.15607/RSS.2013.IX.014.

[8] S. Mitsch & A. Platzer (2014): *ModelPlex: Verified Runtime Validation of Verified Cyber-Physical System Models*. In B. Bonakdarpour & S. A. Smolka, editors: *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, *LNCS* 8734, Springer, pp. 199–214,

doi:10.1007/978-3-319-11164-3_17. Available at http://dx.doi.org/10.1007/978-3-319-11164-3_17.

[9] D. Phan, J. Yang, D. Ratasich, R. Grosu, S. A. Smolka & S. D. Stoller (2015): *Collision Avoidance for Mobile Robots with Limited Sensing and Limited Information About the Environment*. In E. Bartocci & R. Majumdar, editors: *Proceedings of the 6th International Conference on Runtime Verification (RV 2015)*, *LNCS* 9333, Springer International Publishing, Swizerland, pp. 201–215, doi:10.1007/978-3-319-23820-3_13. Available at http://dx.doi.org/10.1007/978-3-319-23820-3_13.

[10] A. Platzer (2001): *Differential Dynamic Logic for Hybrid Systems*. *Journal of Automated Reasoning* 41(2), pp. 143–189, doi:10.1007/s10817-008-9103-8. Available at http://dx.doi.org/10.1007/s10817-008-9103-8.

[11] J. Rivera, A. Danylyszyn, C. Weinstock, L. Sha & M. Gagliardi (1996): *An Architectural Description of the Simplex Architecture*. Technical Report CMU/SEI-96-TR-006, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. Available at http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12521.

[12] A. A. Shalyto (2001): *Logic Control and "Reactive" Systems: Algorithmization and Programming*. *Automation and Remote Control* 62(1), pp. 1–29, doi:10.1023/A:1002837232103. Available at http://dx.doi.org/10.1023/A:1002837232103.

[13] M. Weißmann, S. Bedenk, C. Buckl & A. Knoll (2011): *Model Checking Industrial Robot Systems*. In A. Groce & M. Musuvathi, editors: *Model Checking Software: 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*, *LNCS* 8734, Springer Berlin Heidelberg, pp. 161–176, doi:10.1007/978-3-642-22306-8_11. Available at http://dx.doi.org/10.1007/978-3-642-22306-8_11.