Software Quality Journal
https://doi.org/10.1007/s11219-020-09519-w

An adaptive system for autonomous driving

Martin Zimmermann[1] · **Franz Wotawa**[1]

Published online: 04 July 2020

Abstract
Having systems that can adapt themselves in case of faults or changing environmental conditions is of growing interest for industry and especially for the automotive industry considering autonomous driving. In autonomous driving, it is vital to have a system that is able to cope with faults in order to enable the system to reach a safe state. In this paper, we present an adaptive control method that can be used for this purpose. The method selects alternative actions so that given goal states can be reached, providing the availability of a certain degree of redundancy. The action selection is based on weight models that are adapted over time, capturing the success rate of certain actions. Besides the method, we present a Java implementation and its validation based on two case studies motivated by the requirements of the autonomous driving domain. We show that the presented approach is applicable both in case of environmental changes but also in case of faults occurring during operation. In the latter case, the methods provide an adaptive behavior very much close to the optimal selection.

Keywords Self-adaptive systems · Validation using simulation · Adaptive control

PACS 68T05

1 Introduction

Our modern society relies more and more on various kinds of systems ranging from basic infrastructure like communication or power transmission networks to entertainment. Some of these systems are safety-critical like vehicles or airplanes where faults occurring at runtime may harm people, which has to be prevented by taking appropriate measures during development. In addition, there seems to be a trend for increased digitalization and automation for enabling new opportunities that capture almost all parts of our daily lives. For example, tight integration of product development and manufacturing considering user

✉ Franz Wotawa
wotawa@ist.tugraz.at

Martin Zimmermann
martin.zimmermann@tugraz.at

[1] Technische Universität Graz, Institute for Software Technology, Christian Doppler Lab for Quality Assurance Methodologies for Cyber-Physical Systems, Inffeldgasse 16b, 8010 Graz, Austria

requirements potentially lead to mass customization where people can order one specific product especially tailored to fit their needs. A challenge of increased pervasion of systems into our society is that we more and more depend on the correct functioning of these systems. Hence, it is of uttermost importance that the systems work as expected even in case of unexpected environmental changes or faults occurring during operation.

In the automotive industry, there is the additional motivation of avoiding traffic accidents and, therefore, saving lives behind further information. Bringing truly autonomous driving into practice, there are at least two challenges to solve. First, we have to provide guarantees that the autonomous vehicle is trustworthy providing the expected behavior. This requires extensive testing and other verification measures (see, e.g., Kalra N. and Paddock S.M. (2016)). Second, the autonomous vehicle has to compensate internal faults occurring during operation aiming at bringing passengers to a safe place under all conditions. Note that in ordinary cars with some degree of automation, e.g., cruise control, whenever there is a failure observed, control is passed to the driver who is responsible for taking action. In autonomous driving, this is not possible anymore.

In this paper, we contribute to the second challenge of autonomous driving and provide a methodology and tool that allow for implementing adaptive behavior. In particular, we provide means for representing redundancies and redundant control computations a system can take. The adaptive method relies on monitoring the system's behavior, and an adaptive control part that allows selecting the sequences of control actions by the system autonomously aiming at reaching a pre-defined goal. The selection itself considers previous executions of actions aiming at avoiding failing actions to be executed. Besides the methodology, we come up with a programming language for representing redundant actions and control sequences, and a tool for integrating the methodology into an ordinary software development process.

In our previous paper (Wotawa F. and Zimmermann M. 2018), we presented a mathematical model for a system, which enables such an adaptive behavior. This model was a subset of the model used in Krenn W.K. (2008/2009), which in turn is based on the work of Nilsson (1994). For this subset, we introduced a programming language that enables easy use of the mathematical model. Furthermore, we evaluated this language with two case studies. In Engel G. et al. (2019), we used the rule-based language, as it is presented in this paper with some small extensions, in combination with a Modelica model to simulate a robot, which has the task to drive straight and experiences some faults in its servo motors. Therefore, in Engel G. et al. (2019), we briefly touched on concepts that we will explain in full detail in this paper.

In this paper, we present the full mathematical model of Krenn W.K. (2008/2009) for adaptive systems. We focus on how to bring redundancies into control without any heavy underlying apparatus. Also, we extended the previously introduced programming methodology to cover the whole mathematical model of Krenn W.K. (2008/2009). This model is based on planning using behavioral rules. Each rule specifies how a certain precondition together with an action leads to a post-condition. For example, if we need the car's location, we use the GPS to gain it. This can be expressed by stating a rule formalizing that there is a need for a location and an action that calls the GPS in order to obtain the location. By stating such rules, and a final goal, e.g., presenting such a location to another part of the system or a human, we are able to formalize a control system that also easily allows specifying redundancies. For this purpose, it is only necessary to introduce alternative rules leading to the same post-conditions. The underlying methodology itself will select the best rule depending on the satisfied preconditions and also the reliability of the different actions in the past.

Furthermore, we present improvements to Krenn's work that address some shortcomings we noticed while creating the case studies for Wotawa F. and Zimmermann M. (2018). For validation purposes and to show that the proposed method can be used in practice, we furthermore introduce a case study from the autonomous driving domain focusing on selecting the right sensor for a computer vision system. Such sensors are especially error-prone, and they work best in different environments (see Steinbaeck 2017). E.g., cameras require certain lighting conditions for optimal performance. In the case study, we propose a vision system control program, which is able to automatically adapt to changes in the environment. Note that although we focus on vision systems, the same principles can be easily applied to write control programs for other sub-systems in autonomous cars as well. This is due to the fact that the control system considers redundancies of the system and makes choices during operation considering the previous succeeding and failing executions of system functions.

Therefore, the main contributions of this paper are:

- Extending the programming language introduced in Wotawa F. and Zimmermann M. (2018) to support the full mathematical model of Krenn W.K. (2008/2009)
- Improving the mathematical model as well as the programming language to deal with a specific kind of error described in Section 3.5
- Introducing a novel path planning algorithm, in the context of the mathematical model
- Presenting new case studies to validate the mathematical model and the programming language

This paper is organized as follows: First, we discuss the application domain of our case study focusing on three vision sensors, i.e., cameras, radar, and LiDAR. Afterward, we discuss the underlying adaptive control framework, including rule selection and weight adaptation. In Section 4, we introduce the rule-based language and its Java interface. Afterward, we introduce the case study we used for validation purposes and present the first empirical results obtained. Finally, we discuss related research, and conclude the paper.

2 Application domain of the case study

Although our rule-based language can be used for a wide variety of different systems (Wotawa F. and Zimmermann M. 2018; Engel G. et al. 2019), we specifically picked the autonomous driving domain as an example for our case study. The reason behind this is that we believe that autonomous driving is a perfect application domain for demonstrating the capabilities of our adaptive system framework due to the involvement of highly complex systems and available redundancies. Especially for autonomous driving, the capabilities to compensate for faults occurring at runtime and reacting to environmental changes is of uttermost importance. A truly autonomous car has to compensate faults in a way that assures reaching a safe state in any situation. In addition, such a car has to be independent of weather conditions and other environmental changes and react appropriately. With the help of the proposed framework and the underlying programming language, we can quickly prototype an adaptive system, and implement changes in the requirements quickly if the need arises.

For our case study, we focus on object detection of an autonomous vehicle, i.e., specifically the camera, LiDAR, and radar. All three sensors use different methods to detect objects, and, therefore, they have different strengths and weaknesses:

Camera: The camera is one of the most versatile sensors in an autonomous car. It can provide a human driver with additional images (Stamenkovic Z. et al. 2012) and can be used by an autonomous car to make better decisions. For example, traffic sign detection (Huang S.C. et al. 2017) and vehicle detection (Caraffi C. et al. 2012) are necessary to drive safely on the road. One of the biggest benefits of a camera is that it can detect color, which makes recognition of objects easier.

Unfortunately, the camera is not a perfect sensor. Non-stereo cameras lack depth information and thus cannot provide a 3D view of the environment. Furthermore, cameras do not produce their own light, so they are heavily dependent on environmental light conditions. The most common light source for cameras is sunlight. However, a few methods to use other light sources have already been proposed, e.g., see Kawai S. et al. (2012).

Radar: Radar has been around for a long time already. The first uses of radar were over 40 years ago. Since then, radar has been used for multiple applications like blind spot detection, automatic brake systems, and collision avoidance systems. Radar has very high reliability because of that, and it is considered one of the key sensors for autonomous driving (Meinel H.H. 2014). Nevertheless, there are also weaknesses of radar systems, e.g., they are less angularly accurate than LiDAR, and they cannot distinguish multiple objects in cases where these objects are close to each other.

LiDAR: In the past, LiDAR was mostly used as a sensor for aerial vehicles and other special vehicles because it was too heavy and expensive to use in mass production. In the past, there were some recent advancements both regarding weight and costs so that LiDAR made its way into the car industry (Jeong N. et al. 2018). Because LiDAR uses mostly self-produced infrared light to detect objects, it is not as dependent on different light conditions as a camera, but still is not as reliable as a radar (Rasshofer R.H. and Gresser K. 2005). In addition, LiDAR accuracy is heavily dependent on snow, fog, rain, and dusty weather conditions.

In the case study, we make use of the three sensors and combine them in order to obtain a reliable object detection. The idea is to come up with a control program that compensates for weather condition changes and for sensor faults that arise during operation. We discuss the control program and the two different instances of the case study in Section 5. In the next section, we introduce the underlying framework for adaptive control.

3 Adaptive control framework

The adaptive framework we are going to discuss in this section is based on Krenn's Ph.D. thesis (Krenn W.K. 2008/2009), where the mathematical foundations of the rule-based language are outlined in detail. However, to be self-containing, we briefly explain the approach behind self-adaptivity comprising rules, a weight model, and the path planning algorithm. For a deeper understanding of the basic ideas behind the underlying framework, the weight model, and the rules, we refer the interested reader to Krenn's Ph.D. thesis. The underlying idea behind the adaptive control framework is to provide rules that capture redundant ways of reaching a certain control specific goal and let the underlying control framework select the most appropriate set of rules during operation considering successful executions of these rules over time.

To accomplish the fault-tolerant behavior, the adaptive framework has three phases that are executed in succession to complete a full run. All phases will be discussed in more detail further on.

1. **Path Planning:** The control model searches for a list of rules that would lead to a goal if their actions are executed in succession and have the highest chance of success.
2. **Rule Execution:** The control model executes the prior found list of rules. This is either stopped by a rule which is failing or when a goal was reached.
3. **Rule Update:** After the execution, all rules' parameters are updated to reflect the new knowledge about the successfulness of the rules. Meaning that unsuccessful rules will get chosen rarer in the future, but success rules will be chosen more frequently in the future.

In this section, we first discuss the control model, comprising of rules, propositions, and actions. After that, we take a closer look at path planning, where we introduce two new path planning algorithms that are not included in Krenn's Ph.D. thesis as well as a summary of the algorithm proposed by Krenn. Further on, we compare the algorithms and describe *Rule Execution* and *Rule Update* that provides the new rule update function *Aging*, which extends Krenn's original work.

3.1 The control model

The control model, described in Krenn's Ph.D. thesis (Krenn W.K. 2008/2009), captures the fundamental behavior of a system in interaction with its environment. Basically, the control model represents the current state of the environment and actions that can be chosen accordingly to the specified rules. Moreover, we specify a specific goal that has to be reached. The execution part of the self-adaptive framework takes the model and chooses the rules that, when executed, lead to the goal considering the state of the environment. For selecting the rules, we are relying on a weight model that ideally should indicate to us the optimal rule and, therefore, action to choose. In case the internal representation of the environment deviates from the real environment causing an action to fail, the system is able to choose a different rule and to adapt the weight model accordingly to capture this deviation in further executions. Hence, the whole self-adaptive framework can be seen as a high-level control program that tries to find rules enabling the system to reach its goal. It is also worth noting that self-adaptivity, in this case, is limited by the stated rules. If there are no redundant rules available or at least sequences of rules that can be executed in order to compensate for a failing rule, the system is not able to adapt itself so that it can reach its goal. Furthermore, if, from the current state, there are no rules such that a goal can be reached, the system cannot operate and will terminate with an error. If this happens, there are two options. First, the model can be adapted to enable the desired behavior. Second, the system has to be put in a state from which it can reach a goal. Unfortunately, detecting such situations is only possible during runtime, as the states can change dynamically.

Let us now introduce the control model that comprises the following three basic building blocks:

– **Propositions ($\mathcal{P}$):** A proposition is a single piece of information that can be true or false about the environment. $\mathcal{P}$ is the set of all propositions. Hence, the powerset of $\mathcal{P}$ reflects all different states of the environment To represent the current state of the environment, propositions can be saved in the "memory." If a proposition is included in the memory, it is assumed, but not expected, to also be true in the real world. An example would be *"the sensor is active."*
– **Actions ($\mathcal{A}$):** $\mathcal{A}$ is the set of all actions. Each action, when executed, brings the environment into a new state. The system should take actions in order to achieve a specific goal. An example would be *"use the sensor to detect objects."* Usually, actions are not

high-level as in the previous example but rather program code (in our experiments Java code). If an action, for whatever reason, is not able to be executed successfully, we say that the action failed. For example, a robot assumes that a door is open. Because of this assumption, he wants to go through the door. However, in the meantime, the door was closed. When the robot now tries to go through the door, it will collide with the door, and the action of moving through the door fails.

– **Rules ($\mathcal{R}$):** A rule encapsulates a possible action a system can take in regards to some preconditions and post-conditions. In addition, the rule contains many properties that make the fault-tolerant behavior of the system possible. $\mathcal{R}$ is the set of all these rules. The rules are further explained in the next subsection. An example would be *"if the sensor is active you can use the sensor to detect objects."*

3.2 Rules

The rules are the most essential part of the control model. As previously stated, the rules give the system the possibility to take actions under certain precondition. Rules change the internal view on the environment, i.e., the propositions. After a rule is executed, propositions might be added to the memory or deleted from the memory, depending on the rule. The goal of the system is to find a set of rules that, when executed in sequence, lead to a pre-defined goal. Furthermore, the system tries to find the best set of rules considering previous failures or successes of rule executions. If there are two possible rules that the system can choose to achieve a goal, the system will choose the rule that is more likely to succeed. The idea behind this is that rules, which often fail, should be considered less for execution in order to optimize the behavior of the overall system. When the system actually executes the rules, further called run, it actually executes the action of each rule. If the execution of an action fails, the rule also fails. A run is completed if either a rule failed, then the whole run failed, or a goal rule is reached.

Each rule comprises several parts as follows, which we summarize in Table 1:

– **Precondition:** If all preconditions are in the memory, this rule is able to be executed.
– **Action:** This is the action the system will take when the rule is executed. Note that in our implementation, this is the fully qualified name of a Java class.
– **EffectAdd:** The proposition *EffectAdd* will be added to the memory after the successful execution of a rule.
– **EffectDel:** Each element of the set *EffectDel* will be removed from the memory after the successful execution of a rule.
– **TargetActivity:** Is the value that should be reached as the *activity*. With this value, it is easy to change how often a rule should be executed compared to other rules.
– **ActivitySlope:** This function modifies the weight in specific zones, this is mostly used to model more complex scenarios.
– **DampingValue:** Is the value by which *damping* should be increased or decreased in the Rule Update.
– **AgingValue:** Is the value by which *damping* should be increased or decreased in the Rule Update.
– **AgingTarget:** Is the value to *damping* should age toward in the Rule Update.
– **AgingZone:** Is the range for which *damping* should be updated in the Rule Update.
– **Activity:** Indicates how often the rule was already chosen. It is updated in the Rule Update.
– **Damping:** Indicates how successful a rule was. It is updated in the Rule Update.

Table 1 Elements of a rule

Name	Domain
Precondition	$\langle precondition_i \in \mathcal{P} \rangle$
Action	$action \in \mathcal{A}$
EffectAdd	$posEffect \in \mathcal{P}$
EffectDel	$\langle negEffect_i \in \mathcal{P} \rangle$
TargetActivity	$activityGoal \in [0, 1]$
ActivitySlope	$activitySlope : [0, 1] \rightarrow \mathbb{R}$
DampingValue	$dampingVal : \mathbb{R}$
AgingValue	$agingVal : \mathbb{R}$
AgingTarget	$agingTarget : \mathbb{R}$
AgingZone	$agingZone : [0, 1] \rightarrow \mathbb{Z}_2$
Activity	$activity \in [0, 1]$
Damping	$damping \in [-1, 1]$

The calculation of the total *weight* of a rule is given in Eq. 1. From this equation, we see that we prefer rules that have lower *activity* and lower *damping* values. The underlying semantics is that rules that have been chosen less frequently and are more successful, are more likely to be chosen again during system operation. In Fig. 1, we show the relationship between *damping*, *activity*, and *weight*, with the assumption that $activitySlope(x) = 1$ and $activityGoal = 1$.

$$\begin{aligned} &weight(activity, damping, activityGoal, activitySlope) = \\ &activitySlope(activity) \times (activityGoal - activity) \times (1 - damping) \end{aligned} \tag{1}$$

3.3 Path planning

For our system to work, we also need a path planning algorithm, which searches for a sequence of rules that can be executed after each other so that a goal will be reached. It is

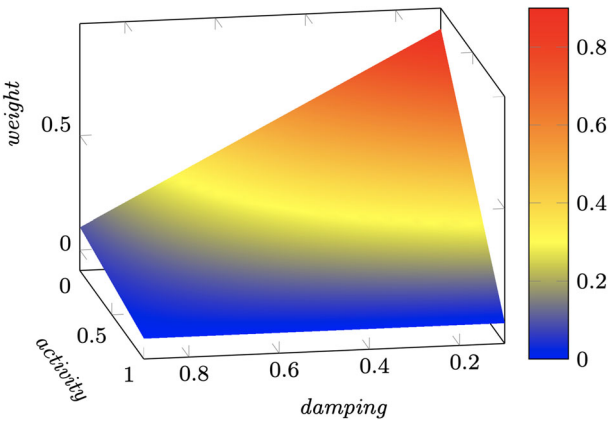

Fig. 1 Relation between *damping*, *activity*, and *weight*

important to have a good path planning algorithm both to get a good sequence as well as do not have too much overhead for the path planning with large rule sets.

In this section, we first explain what a Path is in the context of our system. Then, we present three path planning algorithms. The first algorithm is a novel algorithm introduced in this paper. It is fast but does not give us the optimal path. The second is Krenn's original algorithm, which also does not compute the optimal path and has some problems. However, we use it as a baseline for our comparison. The last one was introduced in Wotawa F. and Zimmermann M. (2018) and is rather slow but always computes the optimal path. At the end of the section, we compare the three algorithms.

3.3.1 Path

The fault-tolerant system always interprets *Paths*, which is simply an ordered list of rules. To actually do something, it is necessary to restrict this list further. A *Reasonable Path*, therefore, is a *Path* that contains a goal rule, and all preconditions are met when the rule would be executed.

Furthermore, we define a *Plan* as a *Reasonable Path* that contains no rule that could be removed and the *Reasonable Path* still stays a *Reasonable Path*. This ensures that only *Reasonable Paths* with minimum lengths are selected. It is highly desired to find actual *Plans* for the execution; however, this is a very computationally intensive task, as we will see with the *Optimal Path Planning Algorithm*, which always finds a *Plan*. However, both the *Top Down Path Planning Algorithm* as well as *Krenn's Algorithm* do not guarantee to find a *Plan*, but at least a *Reasonable Path*.

To avoid infinity loops, we further restrict a *Path* to only contain each rule once. It is to note, that this is only a simple restriction to avoid infinity loops, more comprehensive methods could be developed, i.e., cycle detection, to prevent infinity loops.

As every algorithm at least produces a *Reasonable Paths*, we will, for simplicity, further on refer to *Reasonable Paths* as just *Paths*.

3.3.2 Top down path planning algorithm

To find a path, this algorithm starts with the current memory and calculates all the rules that could currently be reached. If there is a rule with a goal in this set, the goal rule will be taken, and the algorithm finishes. Otherwise, the rule with the highest weight will be taken. After a new rule is taken, the memory will be updated with the proposition additions or deletions of that rule, and again all rules that could be reached are calculated. If there is no more rule to add, the algorithm backtracks to the last time where it made a decision to take a rule and takes the rule with the next highest weight. If there is no more possibility to add any rule, which means all paths have been explored, the algorithm returns with an error.

To illustrate the algorithm, we make use of the following program, where each line represents a rule. For each rule, we have the precondition on the left side, and the propositions to be added (using "+") or removed (using "−") on the right side together with the action written in capitalized letters. The goal state is also indicated on the right side as a proposition having a preceding "#." For more information about the syntax of the language, we refer to Section 4.

```
-> +a A.
 a -> +b B.
```

```
c -> +b C.
b -> +g -b E.
b -> #h -b D.
```

In Fig. 2, we illustrate the execution of the first algorithm the *Top Down Path Planning Algorithm*. When starting with an empty memory, we are only able to execute the rule `-> +a A.`, which adds `a` to the memory. In the second step, the algorithm can only execute the rule `a -> +b B.` because only for this rule its precondition is fulfilled. Adding `b` to the memory enables us to reach two new rules `b -> +g -b E.` and `b -> #h -b D.`. First, the algorithm chooses `b -> +g -b E.` because we assume that this rule has a higher weight, but unfortunately there is no rule any more to add and we cannot reach the goal. Hence, the algorithm backtracks to the previous rule and chooses the alternative rule `b -> #h -b D.` and we reach the goal.

3.3.3 Krenn's algorithm (bottom up path planning)

This algorithm was proposed in Krenn W.K. (2008/2009). First, it starts with all the goal rules and picks the one with the highest weight. Then it searches for all the rules that would add preconditions that are needed by the goal rule to be executable. From this set, it picks the rule with the highest weight. If more preconditions of rules, including the preconditions for the newly added rule, are not satisfied, this continues until either all preconditions of all rules in the list are satisfied, or there are no more rules that could satisfy this path. In the first case, a valid path is found. In the second case, no valid path can be found for the current path. If this happens, the algorithm backtracks to the last location at which it made a decision and takes the rule with the next highest weight. If there are also no more goal rules to choose from, no valid path can be found.

In Fig. 3, we illustrate the execution of Krenn's algorithm. First, we start with the goal rule `b -> #h -b D.`. From this, we see that the precondition `b` is not satisfied. Both rules `a -> +b B.` and `c -> +b C.` can add `b` to the memory. We assume that `c -> +b C.` has the higher weight so we choose this rule. Now, we need to add `c` to the memory as `c` is the precondition for `c -> +b C.`. However, there is no rule that can add `c`; therefore, the algorithm backtracks until the last decision and chooses `a -> +b B.` instead of `c -> +b C.`. The only rule that can add `a` to the memory is `-> +a A.`. and therefore, we pick this rule. Now, all preconditions are satisfied and we found a valid path.

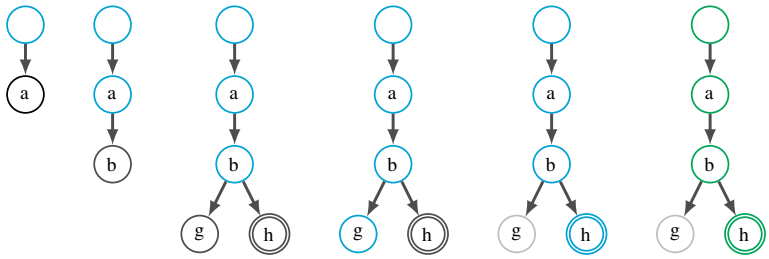

Fig. 2 Top Down Path Planning Algorithm example. Blue, current path; black, reachable rules; gray, dead ends; green, accepted path

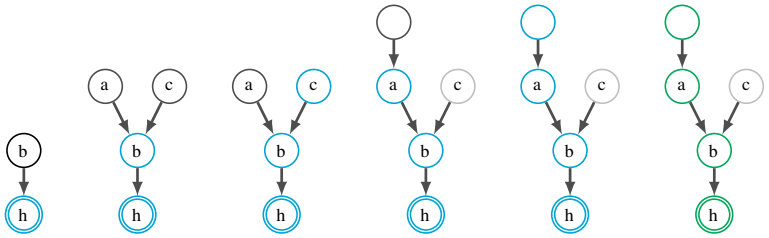

Fig. 3 Bottom Up Path Planning Algorithm example. Blue, current path; black, candidate rules; gray, dead ends; green, accepted path

3.3.4 Optimal path planning algorithm

This algorithm finds the optimal path, as defined in Krenn's Ph.D. thesis. In his thesis, the optimal path is a path that is a *Plan* as defined in Section 3.3.1, has the smallest number of rules of any *Plan*, and has the maximum weight of these *Plans*. To achieve this, we first generate all possible paths. This is very computationally complex because all combinations of rules have to be explored. After this, all the paths, where a rule can be removed and the path would still be executable (meaning all precondition of all rules in the path can still be fulfilled), are removed. This is done by removing each rule from the path and checking if it is still valid. If we find a path where we can remove a rule, we can remove the whole path without checking all other rules, as the new path has to be included in all possible paths. This leaves us with only *Plans* as defined in Section 3.3.1. From the remaining set of paths, the path with the lowest number of rules but the highest weight is selected. The *Optimal Path Planning Algorithm* guarantees to find the optimal solution because it considers all possible paths. This is, of course, at the cost of higher computational complexity and is not feasible for larger rule sets, as we will see in the comparison of the algorithms.

3.3.5 Comparing the path planning algorithms

In order to compare the newly introduced path planning algorithm, the path planning algorithm introduced in Wotawa F. and Zimmermann M. (2018) and the one introduced in Krenn's Ph.D. thesis (Krenn W.K. 2008/2009), we make use of a set of randomly generated rules. Obviously, because of random generation, there might be sets of rules that have no valid path, i.e., no path leading to the goal state. We generated the rules according to the parameters given in Table 2. For each row of the table, we created 50 random test cases. This gives us 750 test cases in total.

From Fig. 4, we see that the *Optimal Path Planning Algorithm* takes the longest time to complete. We are further able to conclude that the *Top Down Path Planning Algorithm* proposed in Wotawa F. and Zimmermann M. (2018) is a bit faster than the *Bottom Up Path Planning Algorithm* proposed by Krenn W.K. (2008/2009), in cases where there are more rules. In Fig. 5, we illustrate the success rate of the algorithms in terms of finding the best solution. Again, from this figure, we can conclude that the top-down approach provides better results than the other algorithms. The reasons behind this are as follows: (i) The *Bottom Up Path Planning Algorithm* cannot deal with proposition deletions well, that is why it is not as good as the *Top Down Path Planning Algorithm*. (ii) The *Optimal Path Planning Algorithm* should, in theory, always find the best solution. However, just with 11 rules, the set of rules was too large to calculate an optimal solution using a Laptop with

Table 2 Parameters for the random creation of rules

# of rules	# of goal rules	Max. # of propositions	Max. # of preconditions
3	1	1	1
3	1	1	2
3	1	2	3
5	1	2	1
5	1	2	2
5	2	4	3
9	1	2	2
9	2	4	4
9	2	6	4
11	1	4	2
11	2	6	4
11	4	8	6
15	1	4	2
15	2	6	4
15	4	8	6

an Intel Core i5-7200U 2.50GHz CPU and 8GB RAM running on Windows 10 Enterprise (Build 17134). During the path planning, we run into out of memory errors and, therefore, could not find a solution. Hence, from this evaluation, we see that the new *Top Down Path Planning Algorithm* for our self-adaptive control framework is superior compared to the other algorithms considering randomly generated rule sets.

3.4 Execution

After the system determined the best list of rules to execute, the system moves to the execution phase. In this phase, the system executes each of the rules actions in the list. After a successful execution of a rules action, the memory is updated according to the rule. If the execution of the action fails, the system stops the current run and calls a repair function that can bring the memory in a valid state again. Bringing the memory in a valid state again can

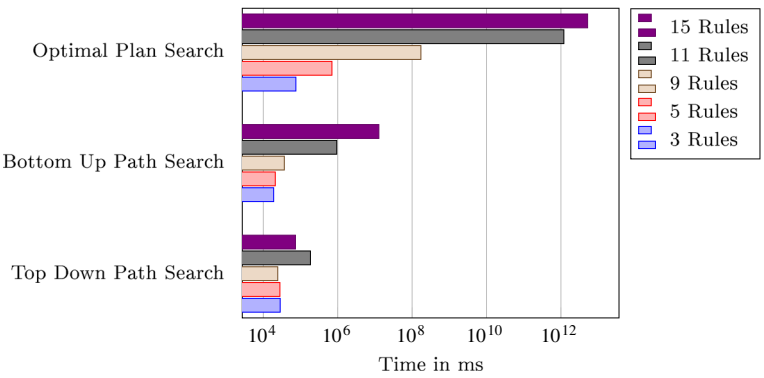

Fig. 4 Comparison between the algorithms, average time per execution in ms

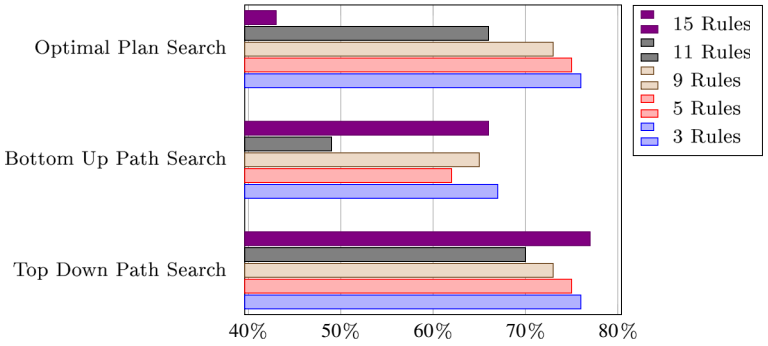

Fig. 5 Comparison between the algorithms percentage of success

be achieved by adding propositions to the memory or removing propositions from the memory. A simple approach would be to add or remove the proposition that had been added or removed during the current run. Another approach could be to gather additional data (e.g., by querying sensors) to find out why the execution of the action failed exactly and update the memory with the new information.

3.5 Rule update

To enable the fault-tolerant behavior, the rules' *activity* and *damping* are updated, before the path planning is started again. In total, we have three different update phases.

1. **Activity Update:** updates *activity* of every rule
2. **Damping Update:** updates *damping* of every rule that was executed
3. **Aging:** updates *damping* of every rule

Activity Update The activity update is performed for every rule, regardless if the execution of the rule was successful or not, or a rule was chosen or not. The update is calculated as stated in Eq. 2, where $chosen$ is $activityGoal$ if the rule was chosen or 0 if the rule was not chosen. The primed version of a value in our equation always refers to the updated value.

$$activity' = \frac{1}{2}(chosen + activity) \tag{2}$$

Damping Update *damping* is only updated by this mechanism if the rule actually was executed. If the rule was not chosen or the execution failed at a previous rule, meaning the run stopped before the rule was executed, the *damping* will not be updated. If the rule was executed and the execution was successful, *damping* will be decreased according to Eq. 3. If the execution failed *damping* will be increased according to Eq. 4.

$$damping' = \begin{cases} 0.1, & (damping - dampingVal) \\ & < 0.1 \\ damping - dampingVal, & \text{otherwise} \end{cases} \tag{3}$$

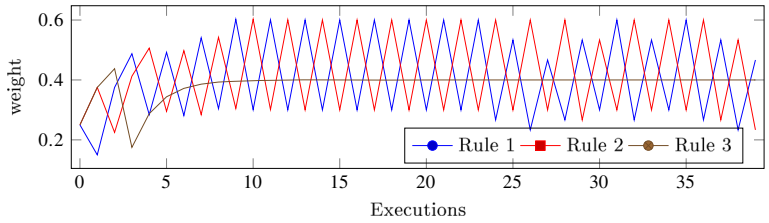

Fig. 6 Weight of three rules without aging over time. We see that the weight of Rule 3 stays the same after a few executions and never reaches a point where the rule would be considered again, because of the low *damping*

$$damping' = \begin{cases} 0.9, & (damping + dampingVal) \\ & > 0.9 \\ damping + dampingVal, & \text{otherwise} \end{cases} \tag{4}$$

Aging The aging is like the activity update done for every rule and updates the value of *damping*. Equation 5 describes how aging is defined.

$$damping' = \begin{cases} damping + agingVal, & agingZone\,(damping) = 1 \\ & \wedge\, damping < agingTarget \\ damping - agingVal, & agingZone\,(damping) = 1 \\ & \wedge\, damping > agingTarget \\ damping, & \text{otherwise} \end{cases} \tag{5}$$

As we mentioned previously, Krenn developed most of the weight model described in this paper. When we implemented the weight model as a programming language and ran some tests described in Wotawa F. and Zimmermann M. (2018), we found out that the weight model did not perform well in specific scenarios. Further investigation led us to a problem where, under certain circumstances, a rule, although it is the best rule, is never chosen again.

This was caused by a rule having a low *damping*, for example, caused by a previous execution that failed. A low damping keeps the rule also at a low *weight*, no matter how small *activity* gets (see Eq. 1). If the *weight* is low, a rule is unlikely to be chosen again. However, *damping* cannot be updated without an execution. This leads to the situation where a rule never gets chosen again because its *weight* stays very small because of the small *damping*. We see this behavior in Fig. 6 (the brown line is the rule with a small *damping*. Subsequently, it never reaches a *weight* that is higher than one of the other rules and is therefore never chosen again).

To counter this behavior, we added aging to the weight model. This enables programmers to change *damping* over time even though the rule is not executed. In Fig. 7, we see that Rule 3 steadily increases its weight due to the increase of the damping value. After some time, the rule has a high enough weight again to be chosen for execution again.

4 Rule-based language

In this section, we introduce our extended rule-based language that is based on the weight model, which we described in Section 3, enabling developers to quickly build fault-tolerant

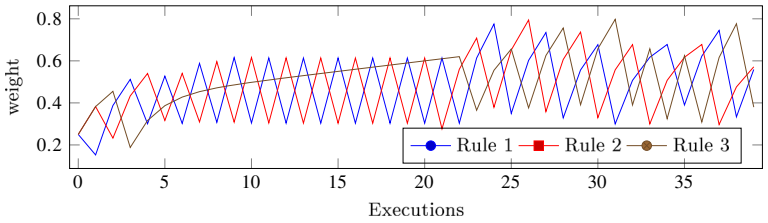

Fig. 7 Weight of the three rules with aging of 0.01 over time. We see that the weight of Rule 3 is gradually increased until the rule has a high enough weight to be considered for executions again

systems. In addition to the language, we developed a compiler that translates the rule-based language into Java code. Furthermore, parts of the language, particularly the actions, are developed in a way such that they can be directly mapped to Java classes implementing the actions. Because of relying on Java, the language can be used on many different platforms without any changes. In this section, we describe the syntax of the language and the interface between the language and Java. With this extension, the rule-based language now supports all of Krenn's mathematical concepts. The compiler of the language as well as the runtime engine can be found on Github[1].

4.1 Syntax of the language

We wanted to make the language as intuitive, easy to use, and readable as possible. Therefore, we decided on a more visual representation of control models as rules. An example of a program can be seen in Fig. 8.

To keep our language concise, almost all the concepts mentioned below are optional. If nothing is specified, standard values are used. The only exception to this is that there needs to be at least one rule with an action. This enables us to write really short programs in RBL, for example "-¿ a." would be a valid program.

Each program starts with the initialization of the memory, followed by the definition of the rules. The initialization of the memory is just a proposition followed by a ".". For example, "active." is a proposition stored at initialization time into the memory. Multiple initializations can be at the top of the file.

After the initialization, we have the set of rules where every rule again comprises multiple parts. Each rule consists of a left-hand part and a right-hand part divided by an arrow and concluded with a dot "left hand -¿ right hand.". The left-hand part only contains all the preconditions as propositions that must hold so that the rule can be executed. "active -¿" e.g., means that the rule can only be executed if "active" is in the memory.

The right-hand side of a rule contains multiple parts. The first part is either a goal, which the system will try to reach, donated by "#," e.g., "#objectProcessed," or a proposition addition denoted by "+," e.g., "+objectDetected." After that, multiple proposition deletions denoted by "−" can be written, e.g., "−objectDetected." After the deletion of propositions, the action has to be provided. In our language, this is the full name of a Java class, that inherits from the interface we describe in Section 4.2, e.g., "sensors.useSensor1."

After the action the function for *activitySlope* can be defined in the form "$(x_1 <= a <= x_2 : f_1, ..., x_n <= a <= x_{n+1} : f_m, f_s)$," where x_1 to x_{n+1} are values that the programmer

[1] https://github.com/martinzimmermann/RBL-Framework/releases/SQJQ2020

```
active.
active -> +objectDetected sensors.useSensor1.
active -> +objectDetected sensors.useSensor2 (0 <= a <= 0.5: a/2, 1) 2.
active -> +objectDetected sensors.useSensor3 [0.1, 0.01, 0.75|].
objectDetected -> #objectProcessed -objectDetected processObject.
```

Fig. 8 Example for a program in RBL

can define as well. These values should always be a range. Either the whole range from 0 to 1 should be covered, or the f_s function must be given, which is used for all values that are not covered by the range. For the function, the programmer can use all standard arithmetic expressions and can use the variable "a," representing the current weight, in the function as well. The provided function could look like "(0 ¡= a ¡= 0.5: a/2, 1)." The next part is the activity goal, denoted as a number, followed by the last part, which is the aging. Aging is a triple of numbers in square brackets, e.g., "[0.1, 0.01, 0.75—]." The first value is the damping value. The second one is the aging value and the last one the aging target. Where the aging target can have the form "x," "—x" or "x—," where just "x" means the range for aging is from 0 to 1, "—x" means the aging range is from x to 1 and "x—" means that the aging range is from 0 to x. To note is that although in the mathematical weight model, it is possible to have an aging range between two arbitrary numbers, this is currently not possible in the language. The reason was that we think the most common use cases are to age a value from 0 or from 1 toward a value, as this feature was implemented to mitigate the problem described in Section 3.5.

In Fig. 9, we depict the Backus-Naur form of our self-adaptive control language.

4.2 Java interface

To be interoperable with many programs and also operating systems, we decided to use Java as a base for our language. This enables us to integrate the self-adaptive control language in many different projects. In addition, it makes it possible to use the language only in small sub-systems of a larger system. To implement the Java interface, we have to define a bridge between our language and the Java program. This bridge comprises two parts. In the first part, we developed a compiler that compiles programs written in the self-adaptive control language into Java. In this step, we also include an execution mechanism into the Java program that takes care of the specified rules. Second, we provide an interface to Java classes implementing the actions.

RuleAction

```
public interface RuleAction {
boolean execute (Memory model);
void repair (Memory model);
}
```

The *RuleAction* interface is a representation of an *action*. Every action must implement this interface and provide two functions: first, the execute function which will be called when the associated rule will be executed and, second, the repair function, which will be called in case the execution of the execute function fails. This gives the programmer the possibility to either call some external code when an error occurs, e.g., a log function, or update the model directly if it is possible to infer the difference between the model and the real environment.

```
DIGIT        ::= "0".."9"
NUMBER       ::= DIGIT+ | DIGIT+ "." DIGIT+
LETTER       ::= ("A".."Z" | "a"..."z")
ID           ::= LETTER*

program      ::= [memory] r_rules
memory       ::= (predicate ".")
predicate    ::= ID
r_rules      ::= (r_rule ".")+
r_rule       ::= [predicates] "->" [("+" | "#") prediacte]
                 ("-" predicate)* action [alist]
                 [activitygoal] [aging]
predicates   ::= predicate ("," predicate)*
action       ::= ID("." ID)*
alist        ::= "(" ((alistentry ("," alistentry)* ["," expr])
                 | expr) ")"
alistentry   ::= NUMBER ("<" | "<=") "a"  ("<" | "<=") NUMBER
                 ":" expr
activitygoal ::= NUMBER
aging       ::= "[" NUMBER? ',' NUMBER? ',' agingTarget? "]"
agingTarget ::= NUMBER
                | "|" NUMBER
                |  NUMBER "|"
expr         ::= sign value
                | expr mulop expr
                | expr sign expr
                | value
sign         ::= "+" | "-"
mulop        ::= "*" | "/"
value        ::= "a" | NUMBER | "(" expr ")"
```

Fig. 9 The Backus-Naur form of the self-adaptive control language

Executor

```
public class Executor {
public boolean executesTillGoalReached(int limit = 10);
public void executesOnce();
public void executesNTimes(int n);
public void executesForever();
public void resetMemory();
public List<String> getMemory();
}
```

The *Executor* class is the point where the execution of the system can be started. From the functions of the class, we can already see that there are multiple ways to start an execution. *executeTillGoalReached* executes the system until there is a goal reached, or the limit of the executions is reached. *executeOnce* executes the system just once, regardless if a goal was reached or not. *executeNTimes* executes the system N times. *executesForever* puts the program in an endless loop that continuously executes the system. *resetMemory* resets the memory to the starting condition. Moreover, *getMemory* gives back a list of the current propositions in the memory. It is to note that all execute functions do not reset the memory, meaning the memory will be preserved after the goal was reached, or the execution failed.

5 Case studies

For the case studies, we extended the case studies that we proposed in Wotawa F. and Zimmermann M. (2018). All codes we used for our experiments and the results are available on Github[2]. We developed a small simulation that runs different scenarios. In the simulation, we make use of three sensors: a camera, a radar, and a LiDAR. In addition, there are different environmental states, for example, a clear sky when considering weather conditions. For each state and each sensor, we defined a probability with which a successful detection of an object would happen. In Table 4, we depict such a distribution for the weather simulation. The simulation then executes a sequence of these states with the rules provided. The goal is to get as many successful object detection as possible. To benchmark our approach, we use two different measurements, i.e., (i) random probability, where we would theoretically pick one of the sensors randomly, and (ii) max success, where we theoretically always pick the sensor with the highest success rate for each state. It is to note that both random probability and max success are computed theoretical values.

The rule-based program we used for the case study comprises three initialization, and three rules and one goal rule:

```
lidar_ok.
radar_ok.
camera_ok.

camera_ok -> +objectDetected
             actions.detectObjectWithCamera.
lidar_ok -> +objectDetected
            actions.detectObjectWithLiDAR.
radar_ok -> +objectDetected
            actions.detectObjectWithRADAR.
objectDetected -> #objectProcessed
                  -objectDetected
                  actions.processObject.
```

This program, further called *Unmodified*, represents the unmodified model from Krenn, as no new additions are used. From this program, we also created two program variants. (i) *Aging to 0.5*, where we added an aging value of 0.01 and the aging target of 0.5 to each detectObjectWith*X* rule where *X* takes one of the following values: Camera, LiDAR, RADAR, and (ii) *Generated*, where we used a genetic algorithm to find the best damping value, aging value, and aging target for the detectObjectWithX for each scenario. Table 3 shows the values that we obtained for program variant *Generated*. To have the fairest comparison, we used the *Optimal Path Planning Algorithm* (Section 3.3.4) for finding paths.

To get meaningful results, we ran each program 3000 times on each scenario and computed the average, minimum, 1. quartile, median, 3. quartile, and maximum.

5.1 Weather scenario

Setup The weather scenario represents a typical weather situation that can occur in practice. First, there are 1000 times a clear sky, then 1000 times rain, 1000 times storm, and after that 1000 times clear sky again. We depict this weather scenario in Fig. 10.

[2]https://github.com/martinzimmermann/SQJQ2020-Experiments/releases/Submission

Table 3 Result of optimization. (W.: Weather scenario, T.: Temporary fault scenario)

	Camera	LiDAR	RADAR
W. damping	0.3267278328062617	1.0000000000000000	0.6696486256642918
W. aging	0.8291299100196624	0.6995987368000960	0.5913889693574242
W. aging target	0.8374017642841604	0.1269329708101536	0.6696486256642918
T. damping	1.0000000000000000	1.0000000000000000	0.8895599998603136
T. aging	0.3118268094218060	0.8872067356020341	0.7706575091074656
T. aging target	1.0000000000000000	1.0000000000000000	0.0639508625534764

For the probabilities of detecting an object, have a look at Table 4. The camera performs well in good weather conditions. However, as the rain gets heavier, it drastically loses accuracy. The LiDAR is not as good as the camera in good weather conditions, but also its accuracy is not reduced so drastically when the weather gets worse. Radar, on the other hand, is completely unaffected by the weather but is not as good as the other two sensors in good weather conditions.

Results The results for the weather scenario are not overwhelming, as the *Random Probability* already achieves a good result of a 73.3% success rate. With the *Unmodified* program, on average, we have a success rate of 74.4%; this is an increase of 1.1% compared to *Random Probability*. From Fig. 11, we can also see, that for some runs the *Unmodified* program is worse than *Random Probability*. The inclusion of the aging parameter was helpful. On average, we have a success rate of 76.3%; this is an increase of 3% compared to *Random Probability*. However, with *Aging to 0.5* now every run is better than *Random Probability*. Unfortunately, even with the *Generated* program, on average, we only get a success rate of 77.8%, which is an increase of 4.5% compared to *Random Probability*. However, we should keep in mind that the maximum success rate that could theoretically be achieved by predicting the future is also only 85%.

These results let us believe that for scenarios where there is much random fluctuation, the parameters have to be selected carefully to get good results and that using optimization, in particular, a genetic algorithm, for obtaining the parameters is a good idea.

5.2 Temporary fault scenario

Setup The temporary fault scenario simulates the faults of different sensors over time. For example, snow could block the camera completely, but after some time, the snow will melt and drop off the car, making the camera usable again. A fault is simulated by changing success from 100% probability to 0% probability. In Table 5, we state the probabilities for the temporary fault scenario.

Fig. 10 States of the weather scenario

Table 4 Success rate of the sensors for the weather scenario

Success rate	Clear	Rain	Storm	Clear	Total
Camera	100%	65%	45%	100%	77.5%
LiDAR	80%	70%	60%	80%	72.5%
Radar	70%	70%	70%	70%	70%
Random probability	83.33%	68.33%	58.33%	83.33%	73.33%
Max success	100%	70%	70%	100%	85%

The scenario consists of 4000 states in total. In the first 1000 states, no sensor is faulty. In the next 1000 states, the camera is faulty, followed by the next 1000 states where both the camera and the LiDAR are faulty, and finally, in the last 1000 steps, all sensors are working again. We visualize this scenario in Fig. 12.

Results For the temporary fault scenario, the results are much better compared to the weather scenario. This is no surprise as usually predicting events that either happen or not is easier than predicting events with a certain probability. Also, it is worth noting that because of the success probability of the sensors being either 0% or 100% the program is deterministic. From the results depicted in Fig. 13, we see that the approach has a very high success rate for all our programs. We see that just with the *Unmodified* program, we obtain a success rate of 93%, this is already an increase by 18% compared to *Random Probability*, which has a success rate of 75%. The parameter selection of *Aging to 0.5*, however, makes the success rate worse compared to the *Unmodified* program. *Aging to 0.5* has only a success rate of 91%. Our *Generated* program even achieves a success rate of 100%. However, it is worth noting that the generated parameters are highly specific for this scenario and probably will perform poorly in other scenarios. This was just to highlight the influence of the parameters on the result and is maybe not applicable in a real-world situation.

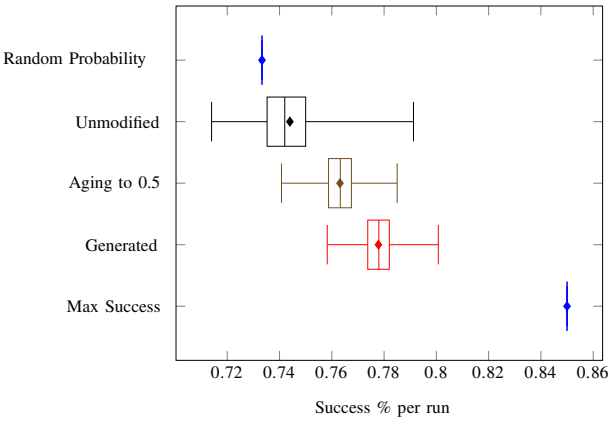

Fig. 11 Percentage of the success % of the experiments from the weather scenario. *Max Success* and *Random Probability* are theoretical values

Fig. 12 States of the temporary fault scenario

6 Related research

There has been a lot of research in various scientific areas aiming at providing adaptivity to systems to cope with internal or external events. In classical STRIPS planning (see Fikes and Nilsson 1971), knowledge in the form of actions with pre- and post-conditions are used to find a plan, i.e., a sequence of actions that enables the system to evolve from an initial state to a goal state. In this kind of planning, originally plan creation and plan execution are separated and further assumptions like that each action can be carried out without any failure applies. In order to overcome these limitations, Nilsson (1994) introduced teleo-reactive programs where planning and action execution are tightly interconnected.

Nilsson's work on teleo-reactive programs was one of the motivations behind work from Krenn W.K. (2008/2009) and Willibald K. et al. (2009). Although Krenn and Wotawa also relied on a rule-based representation of actions, they extended teleo-reactive programs via introducing a bio-inspired way of learning, which was based on adapting the selection of rules depending on previous success or failure. In other work, e.g., Willibald K. and Wotawa (2007a) and Willibald K. and Wotawa (2007b), the same authors showed the applicability of the technique for configuration and diagnosis. In the work presented in this paper, which is based on Krenn W.K. (2008/2009), we improved the computation of rule sequences and the activity value computation. In addition, we discussed other case studies in detail.

Besides planning, there has also been some research on making use of model-based reasoning principles for implementing adaptive systems. Pell et al. (1996) introduced the concepts behind a space probe that is based on reactive planning in combination with onboard diagnosis. Williams and Nayak (1997) discussed the foundations behind this system. Later, Brandstötter et al. (2007) came up with the basics behind a robotic system that is able to react to internal hardware faults occurring at runtime. Steinbauer and Wotawa (2013) summarized these early approaches. Most recently, Wotawa (2019) provided an algorithm for integrating various methods of diagnostics reasoning into a system that is able to detect and also correct faults during operation. Similar to these approaches, we are also making use of rules for providing the knowledge required to control adaptive systems. However, we make use of advanced rule selection that itself varies over time, depending on past interactions between the system and its environment. Moreover, the proposed approach does not

Table 5 Success rate of the sensors for the temporary fault

Success rate	C/L/R	L/R	R	C/L/R	Total
Camera	100%	0%	0%	100%	50%
LiDAR	100%	100%	0%	100%	75%
Radar	100%	100%	100%	100%	100%
Random probability	100%	66.66%	33.33%	100%	75%
Max success	100%	100%	100%	100%	100%

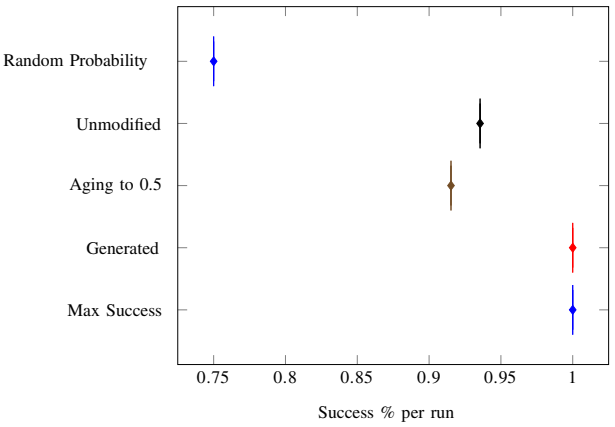

Fig. 13 Percentage of the success % of the experiments from the temporary fault scenario. *Max Success* and *Random Probability* are theoretical values

require sophisticated and computationally demanding algorithms in contrast to model-based reasoning.

The idea of having self-adaptive systems that can react to various internal and external threats autonomously is very much appealing both from a theoretical point of view but also from practice where it is relevant to come up with methods, techniques but also architectures and design principles for developing truly adaptive systems. For achieving autonomic computing, IBM suggested a reference model for autonomic control loops (see (IBM 2006)), i.e., the MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) loop. This very high-level architecture comes together with a discussion on its use but does not indicate a concrete methodology behind for implementing the required functionality. Making use of this and other similar architectures self-adaptive system's technology has many application areas, including cloud and elastic computing (see Barna et al. 2017) or the internet of things (IoT) (see Ifikhar et al. 2017). The method introduced in this paper nicely complements work on self-adaptive systems. We provide means for implementing self-adaptive behavior that considers rules representing different ways a system can interact with its environment.

In addition to architectures regarding self-adaptive systems, there is also work on providing guarantees that a system fulfills its requirements even in case of adaptation (see, e.g., Shevtsov et al. 2017. Providing guarantees is always important but even more in case of systems that can change their behavior. The approach we follow in this paper has the advantage of modeling the overall system's behavior as a collection of pre-defined rules that correspond to actions delivering a certain functionality. Hence, assuming that the implementation of the actions as well as the implementation of the whole approach is correct, the resulting sequence will provide a valid solution, which is exactly what we want to assure. Moreover, the underlying different actions allow for concretely specifying alternatives and redundancies.

There are other more general approaches to provide adaptivity often inspired by nature. In swarm computing, a behavior emerges from the interaction of more or less independent agents using communication for achieving a certain goal. Such systems may use different kinds of interactions and parameters to perform different behaviors. Changes in the environment and in a collection of agents do not necessarily impact the final outcome and may also trigger adaptation. The methods and techniques provided in this paper are substantially

different from such agent-based approaches. First, we are not relying on communicating agents for obtaining an emerging behavior. Second, given the rules, we fix more or less at design time the potential interactions with the environment.

7 Conclusion

In this paper, we introduced a framework for implementing adaptive fault-tolerant systems focusing on control logic. The underlying idea behind the framework is a rule-based language for representing actions, their preconditions, and consequences. The framework makes use of rule selection and execution based on the previous success of execution. It allows specifying alternative routes to reach a specific goal, e.g., sensing a value and sending it to a server. In case one potential route is not available, the system searches for a new one and, therefore, adapts itself over time. Besides the framework, we also introduce a programming language that is tightly coupled with Java and provide empirical results using case studies originating from the autonomous driving domain. In particular, we showed that the approach is applicable in cases where different sensors are used for obstacle detection. In one case study, we focused on the adaptivity in case of changing weather conditions, whereas in the other one, we handled the case of sensor faults occurring during operation. In both cases, the system based on the framework was able to adapt itself accordingly.

Funding information Open access funding provided by Graz University of Technology. The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

References

Barna, C., Khazaei, H., Fokaefs, M., Litoiu, M. (2017). Delivering elastic containerized cloud applications to enable DevOps. In *2017 IEEE/ACM 12Th international symposium on software engineering for adaptive and self-managing systems (SEAMS), Buenos Aires, pp 65–75*.

Brandstötter, M., Hofbaur, M.W., Steinbauer, G., Wotawa, F. (2007). Model-based fault diagnosis and reconfiguration of robot drives. In *2007 IEEE/RSJ International Conference on Intelligent Robots and System. pages 1203–1209*.

Caraffi C., Vojíř T., Trefný J., Šochman J., Matas J. (2012). A system for real-time detection and tracking of vehicles from a single car-mounted camera. In *2012 15th International IEEE Conference on Intelligent Transportation Systems, pp 975–982 https://doi.org/10.1109/ITSC.2012.6338748*.

Engel G., Schweiger G., Wotawa F., Zimmermann M. (2019). A rule-based smart control for fail-operational systems. In *2019 Advances and Trends in Artificial Intelligence. From Theory to Practice, pp 137–145, https://doi.org/10.1007/978-3-030-22999-3_13*.

Fikes, R.E., & Nilsson, N.J. (1971). Strips: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*, 189–208. Elsevier.

Huang S.C., Lin H.Y., Chang C.C. (2017). An in-car camera system for traffic sign detection and recognition. In *2017 Joint 17th World Congress of International Fuzzy Systems Association and 9th International Conference on Soft Computing and Intelligent Systems (IFSA-SCIS), pp 1–6 https://doi.org/10.1109/IFSA-SCIS.2017.8023239*.

IBM (2006). An architectural blueprint for autonomic computing. Autonomic Computing – White Paper. IBM Technical Report.

Iftikhar, M.U., Ramachandran, G.S., Bollansée, P., Weyns, D., Hughes, D. (2017). Deltaiot: a self-adaptive Internet of Things exemplar. In *2017 IEEE/ACM 12Th international symposium on software engineering for adaptive and self-managing systems (SEAMS), Buenos Aires, pp 76–82*.

Jeong N., Hwang H., Matson E.T. (2018). Evaluation of low-cost lidar sensor for application in indoor uav navigation. In *2018 IEEE Sensors Applications Symposium (SAS), pp 1–5, https://doi.org/10.1109/SAS.2018.8336719*.

Kalra N., & Paddock S.M. (2016). Driving to safety: how many miles of driving would it take to demonstrate autonomous vehicle reliability? Https://www.rand.org/pubs/research_reports/RR1478.html.

Kawai S., Takeuchi K., Shibata K., Horita Y. (2012). A method to distinguish road surface conditions for car-mounted camera images at night-time. In *2012 12th International Conference on ITS Telecommunications, pp 668–672 https://doi.org/10.1109/ITST.2012.6425265*.

Krenn W.K. (2008/2009). Self reasoning in resource-constrained autonomous systems. dissertation, Graz University of Technology.

Meinel H.H. (2014). Evolving automotive radar - from the very beginnings into the future. In *The 8th European Conference on Antennas and Propagation (EuCAP 2014), pp 3107–3114, https://doi.org/10.1109/EuCAP.2014.6902486*.

Nilsson, N. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, *1*, 139–158.

Pell, B., Bernard, D., Chien, S., Gat, E., Muscettola, N., Nayak, P., Wagner, M., Williams, B. (1996). A remote-agent prototype for spacecraft autonomy. In *Proc. of the SPIE Conference on Optical Science, Engineering, and Instrumentation, Volume on Space Sciencecraft Control and Tracking in the New Millennium, Bellingham, Washington, U.S.A., Society of Professional Image Engineers*.

Rasshofer R.H., & Gresser K. (2005). Automotive radar and lidar systems for next generation driver assistance functions. *Advances in Radio Science*, *3*, 205–209. https://doi.org/10.5194/ars-3-205-2005. https://www.adv-radio-sci.net/3/205/2005/.

Shevtsov, S., Weyns, D., Maggio, M. (2017). Handling new and changing requirements with guarantees in self-adaptive systems using SimCA.

Stamenkovic Z., Tittelbach-Helmrich K., Domke J., Lörchner-Gerdaus C., Anders J., Sark V., Eric M., Šira N. (2012). Rear view camera system for car driving assistance. In *2012 28th International Conference on Microelectronics Proceedings, pp 383–386 https://doi.org/10.1109/MIEL.2012.6222882*.

Steinbaeck, J., Steger, C., Holweg, G., Druml N. (2017). Next generation radar sensors in automotive sensor fusion systems. In *2017 Sensor data fusion: Trends, Solutions, Applications (SDF), pp 1–6*, https://doi.org/10.1109/SDF.2017.8126389.

Steinbauer, G., & Wotawa, F. (2013). Model-based reasoning for self-adaptive systems – theory and practice. In *Assurances for Self-Adaptive Systems, Springer LNCS 7740*.

Williams, B.C., & Nayak, P.P. (1997). A reactive planner for a model-based executive. In *Proceedings 15th International Joint Conf. on Artificial Intelligence. pages 1178–1185*.

Willibald K., & Wotawa, F. (2007). Gradient-based diagnosis. In *Proceedings of the International Workshop on Principles of Diagnosis (DX), pages 314–321*.

Willibald K., & Wotawa, F. (2007). Configurating collaboration of software modules at runtime. In *Proceedings of the AAAI Workshop on Configuration, pages 19–24*.

Willibald K., Wotawa, F., Madrid Natividad, M. (2009). Intelligent, fault adaptive control of autonomous systems. In Seepold, R. (Ed.) *Intelligent Technical Systems, volume 38 of LNEE. Springer*.

Wotawa, F.ranz. (2019). Reasoning from first principles for self-adaptive and autonomous systems. In Lughofer, E., & Sayed-Mouchaweh, M. (Eds.) *Predictive maintenance in dynamic systems ? advanced methods, Decision Support Tools and Real-World Applications, Springer*, https://doi.org/10.1007/978-3-030-05645-2.

Wotawa F., & Zimmermann M. (2018). Adaptive system for autonomous driving. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp 519–525 https://doi.org/10.1109/QRS-C.2018.00093*.

Martin Zimmermann received a M.Sc. in Computer Science (2019) from the Graz University of Technology. He is currently a PhD student at the Graz University of Technology and a member of the Christian Doppler Laboratory for Quality Assurance Methodologies for Cyber-Physical Systems (QAMCAS) working on verifying adaptive systems and on its foundations. He wrote his Master's Thesis on implementing and evaluating a self-adaptive control framework also as part of the QAMCAS team.

Franz Wotawa received a M.Sc. in Computer Science (1994) and a PhD in 1996 both from the Vienna University of Technology. He is currently professor of software engineering at the Graz University of Technology. Since the founding of the Institute for Software Technology in 2003 to the year 2009 Franz Wotawa had been the head of the institute. His research interests include model-based and qualitative reasoning, theorem proving, mobile robots, verification and validation, and software testing and debugging. Starting from October 2017, Franz Wotawa is the head of the Christian Doppler Laboratory for Quality Assurance Methodologies for Cyber-Physical Systems. During his career Franz Wotawa has written more than 350 peer-reviewed papers for journals, books, conferences, and workshops. He supervised 86 master and 35 PhD students. For his work on diagnosis he received the Lifetime Achievement Award of the Intl. Diagnosis Community in 2016. Franz Wotawa has been member of a various number of program committees and organized several workshops and special issues of journals. He is a member of the Academia Europaea, the IEEE Computer Society, ACM, the Austrian Computer Society (OCG), and the Austrian Society for Artificial Intelligence and a Senior Member of the AAAI.