A Rule-Based Smart Control for Fail-Operational Systems

Georg Engel, Gerald Schweiger, Franz Wotawa, and Martin Zimmermann(✉)

Institute for Software Technology,
CD Laboratory for Quality Assurance Methodologies for Autonomous Cyber-Physical Systems, Technische Universität Graz, Inffeldgasse 16b/2, 8010 Graz, Austria
{engel,gerald.schweiger,wotawa,martin.zimmermann}@ist.tugraz.at

Abstract. When systems become smarter they have to cope with faults occurring during operation in an intelligent way. For example, an autonomous vehicle has to react appropriately in case of a fault occurring during driving on a highway in order to assure safety for passengers and other humans in its surrounding. Hence, there is a need for fail-operational systems that extend the concept of fail-safety. In this paper, we introduce a method that relies on rules for controlling a system. The rules specify the behavior of the system including behavioral redundancies. In addition, the method provides a runtime execution engine that selects the rules accordingly to reach a certain goal. In addition, we present a language and an implementation of the method and discuss its capabilities using a case study from the mobile robotics domain. In particular, we show how the rule-based fail-operational system can adapt to a fault occurring at runtime.

Keywords: Fail-operational systems · Adaptive behavior · Self-healing systems · Autonomous and mobile systems

1 Introduction

Autonomous systems must have the capabilities to make decisions during operation in an independent fashion without considering external control. Ideally such systems must not only be able to react to external stimuli coming from the system's environment in a smart way, but also in case of internal faults or misinterpretations of sensor inputs causing deviations from ordinary behavior. Reacting on internal faults is especially relevant for safety critical systems like autonomous vehicles, where the vehicle itself has not only to detect the fault, but also to react in a smart way in order to reach a safe state autonomously. Such behavior requires that the system is capable of compensating faults at least for a certain amount of time. For example, if the autonomous vehicle detects a fault in its powertrain, stopping operation in a blind bend might not be safe for the passengers. It might be a wiser choice to go to the closest parking space and

F. Wotawa et al. (Eds.): IEA/AIE 2019, LNAI 11606, pp. 137–145, 2019.
https://doi.org/10.1007/978-3-030-22999-3_13

keep operation even under degraded conditions. Hence, such autonomous vehicles have to have build in capabilities for implementing fail-operational behavior. Obviously, fail-operational behavior can only be provided in case of redundancy. Redundancy can be achieved either via using spare parts that are used to replace broken parts autonomously, or re-configuring the system during operation to achieve the desired functionality. The latter deals with either using components in an undesired way to replace a broken function, or to go to degraded mode that can still be achieved with the available functionality. Regardless of the underlying redundancy, a fail-operational system requires a control program that enables the use of redundancy in case of a fault. In this paper, we contribute to the implementation of fail-operational systems, and introduce a rule-based programming language that is capable of deciding which redundant behavior to use during operation. In addition, we discuss a case study from the autonomous mobile robot's domain showing that the control system relying on the rule-based program is capable of dealing with faults in the powertrain by going to a degraded mode.

The idea behind the programming language RBL is to provide means for specifying control rules that select redundant behavior during runtime and to have an interface between the selected rules and the rest of the system. In particular, our implementation allows to call external Java methods used for communicating with the rest of the system. RBL also offers means for computing weights for rules that should be selected. These weights are not fixed, but vary during runtime based on the success of actions executed previously. Hence, actions that are not successful when carried out are less often executed. For example, if we have two rules implementing the same functionality but using redundant components that are triggered via their Java interface, the rule that most often lead to a successful execution will also be more likely used in the future. Whenever there is a fault in the corresponding component, the action will fail, the rule weight will decrease, and the other alternative rule will be more likely selected for execution. Besides the underlying foundations behind the RBL language, which is based on previous work [9,20], we introduce an extension that supports fine tuning of the weight calculation of the rules.

2 The RBL Programming Language

The RBL language is a programming language that uses rules to model fault-tolerant systems. RBL is an extension to a previously proposed rule-based language [20]. Every rule in RBL has a set of preconditions and post conditions. If all preconditions are satisfied the rule can be executed. If the rule was successfully executed, the post conditions will be true and can enable new rules to be executed afterwards. A simple example for a rule is: *"If the water bottle is full and open, you can pour out the water, after that the bottle is empty"*. In this section, we give a brief overview about the syntax and semantic of RBL and describe the interface between RBL, Java and a Modelica Model. For more details about RBL and a more formal definition of the underlying semantics, we refer the interested reader to a previous paper [20].

Every RBL program comprises a set of rules, where the basic syntax of a rule is: `<preconditions> -> <postconditions> <action> [<weight modifiers>].` where `<preconditions>` and `<postconditions>` are sets of propositions that can also be empty. It is worth noting that in the postcondition we are able to add proposition and to remove proposition using `+` and `-` respectively before writing the proposition. A rule might be selected if all the preconditions are satisfied. The `<action>` is a Java class that implements a specific Java interface. A method of this class is called whenever the rule is executed. Before we discuss the `<weight modifiers>`, which are a new extension to the language, we will first briefly explain the fault-tolerant behavior of RBL.

For each execution RBL generates a list of rules, which when executed in succession, satisfies all precondition of the rules and the last rule in the list is a goal rule (denoted by "#" as postcondition). If there is more than one of such lists, meaning there are redundancies in the rules, RBL takes the list with the smallest number of rules and highest weight. The original rule-based language calculates the weight as follows:

$$w = (1 - current\_act) * (1 - damp). \tag{1}$$

After each run *current_act* and *damp* are updated. Where *current_act* is increased if the rule was selected in the path and decreased if it was not selected, meaning *current_act* is a measure of how often the rule was selected in the past. *damp* on the other hand is decreased if the rule was executed and was successful and increased when it failed, meaning it is a measure of how successful the rule was in the past. Formula (Eq. 1) together with the update enables the fault-tolerant behavior.

To fine-tune the fault-tolerant behavior to different situations, we introduce two new extensions to the language: (i) **Aging** is another update step that occurs after each run. *damp* will be decreased or increased by the aging value based on whether the aging target is currently smaller or greater than *damp*. (ii) **Activity and damping scaling** can be used to disable, or lessen, the effect of *damp* and *current_act* on the weight calculation. This leads to following new weight calculation:

$$w = (1 - (current\_act * act\_scaling)) * (1 - (damp * damp\_scaling)). \tag{2}$$

The `<weight modifiers>`, with which the two new concepts can be configured, are 5 values separated by a comma, if a value is left blank a default value is used. For example `[0.2,,,0,1]` is a valid `<weight modifiers>`. The different values have following meaning accordingly to their order:

1. **Damping value** is the value by which *damp* is increased or decreased
2. **Aging value** is the value by which *damp* ages
3. **Aging target** is the value towards *damp* ages
4. **Activity scaling** is the value of *act_scaling* in Equation (Eq. 2)
5. **Damping scaling** is the value of *damp_scaling* in Equation (Eq. 2)

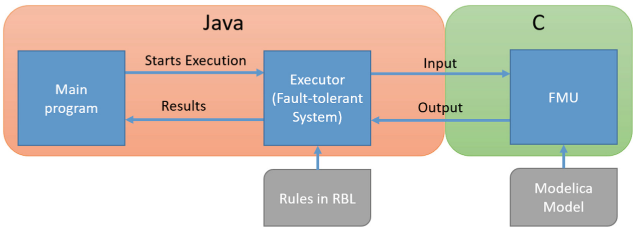

Fig. 1. Architecture of the RBL and FMU interface.

Table 1. Faults over time of the different fault modes.

	0 to 10 s	10 to 20 s	20 to 30 s
Normal	No fault	No fault	No fault
Fault A	Left wheel faulty	No fault	No fault
Fault B	No fault	Left wheel faulty	Right wheel faulty
Fault C	No fault	Left and right wheel faulty	No fault

For example, the rule *"If the water bottle is full and open, you can pour out the water, after that the bottle is empty."* can be expressed in RBL as follows:

```
b_open, b_full -> +b_empty -b_full action.emptyBottle [0.1,,,0.5,1].
```

3 The Mobile Robot Case Study

In this section, we discuss a case study where we outline the use of RBL to implement fail operational behavior for a mobile robot. The mobile robot includes a differential drive, where each wheel is connected with a motor, and a wheel encoder attached for measuring the rotational speed during operation. For details of the mathematical modelling of the powertrain we refer to [3]. In the case study we assume a controller that is able to set the voltage for the motors such that they start rotating. The speed of rotation is proportional to the voltage provided. The task of the controller is to set the voltage such that the robot moves straight. The controller itself only receives the rotational speed measured using the wheel encoders as inputs. We further assume that in case of a fault of the motor the rotational speed does not follow the given voltage anymore. For simplicity, we assume that in the fault case the rotational speed is half the expected speed. For the experiment based on the case study, we implemented the kinetics of the robot drive using Modelica [5].

In order to couple the Java-based control program with the Modelica model capturing the kinematics of the robot, we made use of co-simulation. Co-simulation is a simulation method involving a collaboration of various solvers and often tools [6]. Usually, co-simulation is chosen when different specialized tools are employed to model different subsystems of a heterogeneous complex system.

Various co-simulation interfaces are available [4], a prominent one is the Functional Mockup Interface (FMI) [1]. It has been developed as tool-independent standard in the ITEA2 European Advancement project MODELISAR, and is by now widely supported by many tools. FMI supports both model exchange and co-simulation of dynamic models, providing a zip of an executable and xml-files describing metadata for the subsystem, for a detailed discussion see e.g. [16]. For our experiments, we used the open-source library javaFMI [7] as an interface between a controller implemented in Java and a dynamical simulation model implemented in Modelica.

We briefly discuss the underlying co-simulation framework used for the experimental evaluation. The framework comprises three main components: the main program, the executor, and the Functional Mock-up Unit (FMU). In Fig. 1 we depict the general underlying architecture. The main program starts the execution and evaluates the results. The executor is generated out of the RBL rules and is responsible for the fault-tolerant behavior. The executor communicates through javaFMI with the FMU and evaluates the results from the FMU. The FMU is generated directly from the Modelica model of the robot.

4 Experimental Evaluation

In this section we will describe our experimental evaluation. First, we will give an overview of our experimental setup, then we will present and discuss our results.

Experimental Setup: For our experimental evaluation we coupled the former mentioned Modelica Model with the RBL language, as we described in Sect. 3. The goal of the robot was to drive in a straight line and if faults occur going in a degraded mode.

The only inputs the fault-tolerant system can give to the robot was that either both motors of the wheels get the same voltage, or the left or right motor gets half as much voltage as the other motor. In case of a fault this would mean that the healthy wheel should match the speed of the faulty wheel, by reducing the voltage of the healthy wheel to half. As a feedback the fault-tolerant system only knows if both wheels turn at the same speed, or not.

The ruleset of RBL for our experiment was very simple as we can see in the following listing. To note is that we disabled that *current_act* has any influence on our system, this means the only relevant measure for choosing a rule for our experiment was how often the rules succeed.

```
-> #drive actions.leftHalfDrive [0.2,,,0,1].
-> #drive actions.rightHalfDrive [0.2,,,0,1].
-> #drive actions.normalDrive [0.2,,,0,1].
```

To see the fail-tolerant behavior in action we tested 4 different configurations of the model, where each was executed for 30 s. The concrete faults can be seen in Table 1.

Table 2. Results of the different fault modes for runs with and without RBL rules.

	Without RBL		With RBL	
	% of straight	Distance from Y	% of straight	Distance from Y
Normal	100.0%	0.0	99.8%	3
Fault A	66.7%	15,264.6	99.7%	695.2
Fault B	33.4%	1,762.3	99%	1,473.8
Fault C	100.0%	0.0	99.8%	3
Average	75.0%	4,256.7	99.6	543.7

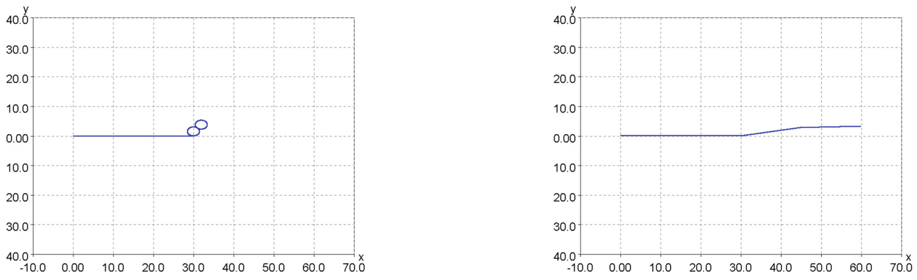

Fig. 2. Visual comparison of the test without RBL (left) and with RBL (right) with fault B. We can clearly see that the robot with RBL drives a more straight line towards infinity than the robot without RBL.

Results: To understand our result, we used 3 metrics: (i) The percentage of the time the robot actually drove straight, meaning both wheels turned at the same speed. (ii) the sum of the distances from the robot to the y axis per time step. (iii) we plotted the path of the robot in absolute coordinates to compared them visually. We compared runs without the RBL rules , and runs with RBL, i.e., where the fault-tolerant system could change the supplied voltage to the wheels in case of a fault.

We conclude from Table 2 that in the case no fault occurs (Normal), or both wheels are faulty (Fault Mode C), the robot without the RBL rules performance a bit better than the robot with RBL rules. This is due to the nature of the fault-tolerant system that first has to find the correct voltage supply for the robot. However, the other tests show that the robot with RBL rules perform much better. It has always a success rate of over 99%. Interesting to note is that the % of driving straight does not correlate to the distance from the y axis. In Fig. 2, we can see why. When we look at the Figure of the path without the RBL rules (left), we can see that even though the robot drives wrong most of the time it always circles back to the y axis which leads to a value comparable to the distance to the Y axis from the robot with RBL rules. On the figure however we can clearly see that the robot did not fulfil its target to drive in a straight line. All tests run with the RBL rules however fulfil this target within a small margin.

5 Related Research and Conclusions

Early work in the domain of adaptive and self-healing systems include [15] and [10] where the authors focus on reconfiguration and self-repair of hardware. Musliner et al. [11] introduced an approach for self-adaptive software to be used in real-time environments. The use of self-healing in the context of autonomous systems and automotive is not novel. Seebach and colleagues [17] introduced an approach for implementing such a behavior.

Pell and colleagues [12] were one of the first making use of model-based reasoning [2,14] for allowing systems gaining additional autonomy, which finally led to a model-based control system for space probes [13]. In the latter papers, models are used to allow the system to react to internal faults occurring during operation in a smart way. Steinbauer and Wotawa [18] used this idea and implemented a system for mobile robots that was able to detect software issues and to restart software modules causing these issues at runtime to bring the robot back to operation. Hofbaur et al. [8] introduced a system that is able to change control models for mobile robots at runtime in case of faults in the robot's drive. For a summary, of model-based approaches for self-adaptive systems we refer to [19]. See [21] for more information and details regarding the formal background of model-based reasoning for self-adaptive systems.

In this paper we presented a case study that uses RBL to combine a control-system with a physical model of a robot that captures the kinematics of the robot. Hence, we are able to demonstrate that the proposed approach for self-adaptive systems also works in a simulated environment that is very close to the real world. We tested our approach by modeling a robot and introducing faults in the system. The experimental results indicate that RBL can be effectively used for controlling such a robot in case of faults. In future research, we will extend the case study capturing different fault scenarios and also different drives. Moreover, we will compare the outcome with other means for implementing fail-operational behavior like model-based reasoning.

Acknowledgments. The research was supported by ECSEL JU under the project H2020 737469 AutoDrive - Advancing fail-aware, fail-safe, and fail-operational electronic components, systems, and architectures for fully automated driving to make future mobility safer, affordable, and end-user acceptable. AutoDrive is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program "ICT of the Future" between May 2017 and April 2020. More information https://iktderzukunft.at/en/bmvit. The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

References

1. Blochwitz, T., et al.: The functional mockup interface for tool independent exchange of simulation models. In: 8th International Modelica Conference 2011, pp. 173–184 (2009)
2. Davis, R.: Diagnostic reasoning based on structure and behavior. Artif. Intell. **24**, 347–410 (1984)
3. Dudek, G., Jenkin, M.: Computational Principles of Mobile Robotics. Cambridge University Press, Cambridge (2010)
4. Engel, G., Chakkaravarthy, A.S., Schweiger, G.: A general method to compare different co-simulation interfaces: demonstration on a case study. In: Obaidat, M.S., Ören, T., Rango, F.D. (eds.) SIMULTECH 2017. AISC, vol. 873, pp. 351–365. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-01470-4_19
5. Fritzson, P.: Object-Oriented Modeling and Simulation with Modelica 3.3 - A Cyber-Physical Approach, 2nd edn. Wiley-IEEE Press, New York (2014)
6. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: state of the art. CoRR abs/1702.0, February 2017
7. Hernández-Cabrera, J.J., Évora-Gómez, J., Roncal-Andrés, O.: javaFMI. https://bitbucket.org/siani/javafmi/wiki/Home. Accessed 25 Jan 2019
8. Hofbaur, M.W., Köb, J., Steinbauer, G., Wotawa, F.: Improving robustness of mobile robots using model-based reasoning. J. Intell. Rob. Syst. **48**(1), 37–54 (2007)
9. Krenn, W., Wotawa, F.: Intelligent, fault adaptive control of autonomous systems. In: Madrid, N.M., Seepold, R.E.D. (eds.) Intelligent Technical Systems. Lecture Notes in Electrical Engineering, vol. 38, pp. 175–188. Springer, Dordrecht (2009). https://doi.org/10.1007/978-1-4020-9823-9_13
10. Moreno, J.M., Madrenas, J., Faura, J., Cantó, E., Cabestany, J., Insenser, J.M.: Feasible evolutionary and self-repairing hardware by means of the dynamic reconfiguration capabilities of the FIPSOC devices. In: Sipper, M., Mange, D., Pérez-Uribe, A. (eds.) ICES 1998. LNCS, vol. 1478, pp. 345–355. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0057636
11. Musliner, D., Goldman, R., Pelican, M., Krebsbach, K.: Self-adaptive software for hard real-time environments. Intell. Syst. Appl. **14**(4), 23–29 (1999)
12. Pell, B., et al.: A remote-agent prototype for spacecraft autonomy. In: Proceedings of the SPIE Conference on Optical Science, Engineering, and Instrumentation, Volume on Space Sciencecraft Control and Tracking in the New Millennium, Bellingham, Waschington, U.S.A., Society of Professional Image Engineers (1996)
13. Rajan, K., et al.: Remote agent: an autonomous control system for the new millennium. In: Proceedings of the 14th European Conference on Artificial Intelligence (ECAI), Berlin, Germany, August 2000
14. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987)
15. Rincon, F., Teres, L.: Reconfigurable hardware systems. In: Proceedings of the International Semiconductor Conference, New York, NY, USA, vol. 1, pp. 45–54 (1998)
16. Schweiger, G., Gomes, C., Engel, G., Hafner, I., Schoeggl, J.P., Nouidui, T.S.: Functional mockup-interface : an empirical survey identifies research challenges and current barriers. In: American Modelica Conference 2018 (2018)

17. Seebach, H., et al.: Designing self-healing in automotive systems. In: Xie, B., Branke, J., Sadjadi, S.M., Zhang, D., Zhou, X. (eds.) ATC 2010. LNCS, vol. 6407, pp. 47–61. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16576-4_4
18. Steinbauer, G., Mörth, M., Wotawa, F.: Real-time diagnosis and repair of faults of robot control software. In: Bredenfeld, A., Jacoff, A., Noda, I., Takahashi, Y. (eds.) RoboCup 2005. LNCS (LNAI), vol. 4020, pp. 13–23. Springer, Heidelberg (2006). https://doi.org/10.1007/11780519_2
19. Steinbauer, G., Wotawa, F.: Model-based reasoning for self-adaptive systems – theory and practice. In: Cámara, J., de Lemos, R., Ghezzi, C., Lopes, A. (eds.) Assurances for Self-Adaptive Systems. LNCS, vol. 7740, pp. 187–213. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36249-1_7
20. Wotawa, F., Zimmermann, M.: Adaptive system for autonomous driving. In: 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 519–525, July 2018
21. Wotawa, F.: Reasoning from first principles for self-adaptive and autonomous systems. In: Lughofer, E., Sayed-Mouchaweh, M. (eds.) Predictive Maintenance in Dynamic Systems, pp. 427–460. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05645-2_15