Drawing on SFL for Making Intelligent Decisions in RBL

Martin Zimmermann and **Ingo Pill** and **Franz Wotawa**
Christian Doppler Laboratory for Quality Assurance Methodologies for Cyber-Physical Systems
Institute for Software Technology, Graz University of Technology
Inffeldgasse 16b/II, 8010 Graz, Austria, email: {mzimmerm, ipill, wotawa}@ist.tugraz.at

Abstract

Even when facing issues, we expect today's systems to evaluate the situation and make intelligent choices in order to fulfill their tasks. With RBL, we recently proposed a language that we can use for describing and enabling smart system behavior such that a system can autonomously react to issues occurring during operation. The proposed framework needed a designer to fine-tune internal parameters though and suffered from performance problems for specific scenarios. We now show how to exploit diagnostic reasoning for enabling a self-configuring plug and play setup of an intelligent system that can make the most promising choices to fulfill its tasks. We also show that this concept significantly improves the performance for previously problematic application scenarios without sacrificing overall performance.

1 Introduction

Intelligent behavior is a feature we have grown to expect today's technical systems to exhibit. Be it our smartphones having to last through the day by efficiently regulating power consumption, automated industry plants that have to intelligently react to problems like local power loss or conveyor congestions, or autonomous vehicles that have to cope with an abundance of possible faults but have to maintain safety under all circumstances. We expect all these systems to make intelligent decisions in order to effectively and efficiently carry out their tasks—regardless of encountered issues and slight changes in the environment. In the context of faults, we call systems that implement such smart behavior to be *fail-operational*, such that they can still discharge their duties, albeit possibly with degraded performance.

There is a variety of techniques available that a designer can employ as technological basis for implementing such behavior. This includes calculi like the situation calculus [1; 2] or dynamic planning concepts that allow us to react to changes in the environment [3]. In our work, we focus with the Rule-based Language (RBL) framework [4] on a language and the corresponding framework that allow us to describe a system's possible actions with pre- and postconditions. Most importantly, we can constantly reason about viable plans (described as action sequences) for achieving our goals. Essential aspects are that we can reason about (necessary) plan updates considering live data about the individual actions' success and that we will in turn take the actions' success rates for future smart decisions into account. Furthermore, a corresponding framework for automatically generating JAVA code from a RBL file to implement intelligent behavior for already existing systems is available on GitHub[1]. In [5], we showed, for example, that we can use an earlier version of this framework to design fail-operational behavior for a robot—so that it can carry out its jobs even when experiencing some internal or externally triggered faults. Originally, RBL targeted fail-operational behavior only, but it is apparent that with RBL's concept, we can describe any setting that fits this syntactic setup of actions and their pre- and postconditions.

When employing RBL, a designer has to define rules with pre- and postconditions for a system's actions. These can be *automatically* translated into JAVA code to be integrated into the system code. This code indeed contains all parts and functionality necessary to enable the desired behavior, including the control part improved by our current work.

From our experiments in [5] we saw that there are two potential disadvantages to deploying RBL. The first is related to the issue that we have to define and fine-tune several parameters concerning the decision-making process. Second, we saw in our experiments that with configuration problems there is a problem class where using RBL resulted in inferior performance. With our current work, we replace the manually tuned parameters with a diagnostic reasoning concept. There, we take the individual actions' success (and thus in turn the confidence in the individual rules' effects) towards achieving our goals into account. Via integrating a *lightweight diagnostic reasoning* concept for assessing the successfulness of the individual actions, we can avoid having to fine-tune internal parameters on one hand, and on the other hand, we can achieve the desired performance also for the class of configuration problems—without sacrificing performance for other classes.

For the underlying diagnostic reasoning concept, we translate spectrum-based fault localization (SFL) [6] to a *live* and rule-based setting. That is, we recently showed in [7] how to adopt SFL (which is traditionally used to evaluate the executions of a test suite) for static diagnostics of knowledge-bases used in reasoning processes. The RBL setting where we reason with our RBL-rule set during live operation of a system is similar, but it requires us to extend the original static scope. In this manuscript, we show how to adopt the concept for RBL and how to extend it to accommodate a *live* setting. In a first step, this allows us to

[1] https://github.com/martinzimmermann/RBL-Framework/releases/DX2020

continuously evaluate the success of the individual actions during their execution. Then, based on these data, we can constantly update our internal predictions concerning how successful the individual rules are in achieving the desired effects. These success rates then serve as a basis for an educated and smart decision concerning the use of the most successful rules (and thus the most effective action sequences) for achieving the desired goal(s).

2 Preliminaries

The RBL framework aims to provide developers with a light-weight method to model and deploy systems that exhibit intelligent behavior in a changing environment. The resulting system does not rely on a fixed execution path but instead intelligently and continuously (re-)designs a viable action sequence during operation. In order to achieve this, the RBL framework offers two components; a specific modeling language and a runtime engine for the necessary reasoning. The underlying mathematical model for RBL and this reasoning is based on the Ph.D. thesis of Krenn [8] and draws on the use of beliefs and rules.

The RBL runtime engine models the environment by keeping a list of corresponding beliefs. For modeling the system, we use *rules*. In terms of syntax, we have that a rule comprises a set of preconditions, a set of postconditions, an action (Def. 2), a repair routine (Def. 3), and a weight that catches our confidence in this rule's success and whose computation is described in more detail in Sec. 3. The RBL runtime engine uses this model when trying to find a rule sequence (a plan) that (a) fulfills the preconditions for the individual rules, and (b) leads to the desired and specified goal with (c) optimal weights.

In order to enable reactive intelligent behavior such that we can choose between plans, the model should allow multiple viable (in respect of aspects (a) and (b)) plans for achieving some goal. Based on the rules' weights, the runtime engine then calculates the credibility in succeeding for each plan. Finally, in respect of aspect (c), the runtime engine selects the plan with the highest credibility. If the selected plan, i.e., one of its actions/rules, fails during execution, the runtime engine calls the failed rule's repair routine in order to bring the runtime engine's beliefs in a coherent state.

As we will see also in Sec. 4, the concept for computing the weights has a significant impact on the system's performance. Improving the weight calculation thus has been the major focus for our work, where we will discuss in detail in Sec. 3 how to exploit light-weight diagnostic reasoning.

Definition 1 *A* rule $R = (pre, post, a, r, w)$ *consists of a finite set of preconditions pre, a finite set of postconditions post, a single action a, a repair routine r, and a weight w. Rule R can only be executed if all $p \in$ pre are known as beliefs. Iff the runtime engine executes rule R, its action a is executed. If this is successful, all $p \in$ post are individually added to (or removed from) the runtime engine's beliefs, depending on the individual p. If a is not successfully executed, the repair routine r is executed.*

Definition 2 *An* action *a is a function that interacts with the environment and returns $\top$ (true) iff the interaction was successful—otherwise it returns $\perp$ (false).*

Definition 3 *A* repair routine *r is a function designed by the user that adds and/or removes beliefs from the runtime engine's beliefs such as reach a coherent set of beliefs.*

Definition 4 *A* finite plan *is a finite rule sequence $R_1, \ldots, R_n$ such that all R_i's preconditions are met and the goal is achieved.*

The modeling language for RBL as introduced in [4] implements the following syntax: Beliefs as used to describe the environment are written as "`<belief>.`". These beliefs are added to the runtime's beliefs with the first execution. Rules to describe the system and its possible actions are written as "`<pre> -> <goal> <post> <action>.`". Each rule consists of the following elements:

- **pre** is a finite set of beliefs that the runtime engine must know in order for that rule to become executable. Please note that this set can be empty for describing a rule that we can always execute. The syntax is "`<belief1>, ..., <beliefn>`". In the literature, sometimes the term *guard* is used to refer to this set.
- **goal** indicates that executing this rule achieves a specific goal. The syntax is "`#<goal>`". A designer can ask the runtime engine to either reach a certain goal, or any of the goals present in the model.
- **post** is a finite set of beliefs that are either to be individually added to or individually removed from the runtime engine's belief, iff the rule executes successfully. $p \in$ *post* to be added are preceded by "+" and those that are to be removed are preceded by "-", resulting in the syntax "`+<belief1> -<belief2> ... +<beliefn>`". The beliefs get added or removed from left to right.
- **action** is an identifier for a JAVA class that aggregates the function associated with the system action and a repair routine to be called when the action function fails during execution. Please note that these classes use fully qualified JAVA class names.

For more details about RBL, we refer the interested reader to [4; 5] for more extensive discussions of RBL's syntax.

RBL's runtime engine controls planning *and* executing the plan. Planning from rules is not a new idea and has been proposed before [9; 10; 11]. Different from most planning systems though, the RBL runtime engine focuses not only on planning but also on executing the plan and uses the execution's feedback to improve the continuous (re-)planning. The runtime engine loops over three distinct tasks.

1. **Planning:** The runtime engine searches for the plan with the highest likelihood of success given the individual rules' previous execution results.
2. **Execution:** The runtime engine executes each rule in the plan, tracking the successfulness of the rules' execution. If a rule was not executed successfully, i.e., the associated action failed, the execution of the plan is stopped and the failed rule's repair routine is invoked in order to bring the system's beliefs in a coherent state.
3. **Update:** The runtime engine updates the rules' execution history and recalculates the rules' weight based on these data.

Before describing in Sec. 3 our approach to exploit a diagnostic reasoning concept for computing the weights and in turn enabling automated smart choices in RBL, let us describe some basic diagnostic concepts first.

When aiming to reason about the sources of some encountered program or system failure, model-based diagnosis (MBD) [12; 13] is, without doubt, a powerful technique.

It comes at the disadvantage though, that we need a special model for the reasoning. Furthermore, while MBD is complete with respect to the used model, the entailed computations can become quite complex. In particular, we suffer from the fact that the search space for diagnoses is exponential in the number of the "component" health state variables that we introduced to the model in order to be able to reason about faulty components. While larger numbers of these variables thus result in a computational disadvantage, the set of health state variables indeed defines which diagnoses we can find and which not—so that we often intentionally suffer from computational issues.

With SFL, we take a different approach and consider the involvement of components in individual failing *and* passing behavior, following the idea that some component that is always involved in faulty behavior but never in correct one is very suspicious of being the source of the problem (and vice versa). Since we can seldomly enjoy the luxury of having such an extreme case where a component is only involved in either faulty or correct behavior, over the years, many *similarity coefficients* [14; 15] for computing each component's individual suspiciousness have been proposed. When computing these coefficients, we take each component's involvement or non-involvement in faulty and correct behavior into account, and compute a corresponding suspiciousness value for each component that we can then use to rank the components. In order to implement this light-weight approach, we thus only need to collect the corresponding execution data about which component was involved in which behavior (stored in a matrix also referred to as spectrum - see Def. 5), as well as data about which behavior considered in the spectrum is violating or complying with our expectations (the so-called error vector - see Def. 6). Then we evaluate the data according to a chosen metric like Ochiai and establish a suspiciousness ranking for the components via simple computations (no model being involved).

Definition 5 *An activity matrix or spectrum A is an $n \times m$ matrix, where for each of the n system components we have m rows for m considered behaviors b_j. Cell a_{ij} is labeled with 1 iff component c_i is involved in b_j, otherwise with 0.*

Definition 6 *An error vector e for some spectrum A (Def. 5) is a vector of length m (an $1 \times m$ matrix) s.t. $e_j = 1$ iff b_j in A violates the expectations, and $e_j = 0$ otherwise.*

From A and e, we derive for each c_i four *frequencies* $n_{CN}(c_i)$, $n_{CE}(c_i)$, $n_{VN}(c_i)$ and $n_{VE}(c_i)$ that capture in how many **C**orrect and **V**iolating behaviors (the rows) in A some c_i was **E**xecuted or **N**ot. From these numbers we can compute several similarity coefficients that shall describe a component c_i's suspiciousness $D(c_i)$ of being faulty. Let us introduce with Ochiai, Tarantula, and Jaccard a selection of well-known metrics:

$$\mathit{Ochiai}: D(c_i) = \frac{n_{VE}(c_i)}{\sqrt{(n_{VE}(c_i) + n_{VN}(c_i)) \cdot (n_{VE}(c_i) + n_{CE}(j))}}$$

$$\mathit{Tarantula}: D(c_i) = \frac{\frac{n_{VE}(c_i)}{n_{VE}(c_i)+n_{VN}(c_i)}}{\frac{n_{VE}(c_i)}{n_{VE}(c_i)+n_{VN}(c_i)} + \frac{n_{CE}(c_i)}{n_{CE}(c_i)+n_{CN}(j)}}$$

$$\mathit{Jaccard}: D(c_i) = \frac{n_{VE}(c_i)}{n_{VE}(c_i) + n_{VN}(c_i) + n_{CE}(c_i)}$$

It is apparent that SFL is an excellent technique for evaluating the executions of a test suite for some software so that we can easily collect the required data. While it is not a straightforward adoption, we showed in [7] that we can exploit SFL also in the context of logic reasoning with knowledge-bases. That is, while a knowledge base is not a program that we execute in the traditional sense, one can *record the rules that we use when reasoning about an individual problem* and use these data *to define a spectrum*. From an abstract point of view, the reasoning process for some individual problems with the same knowledge-base (which plays the role of program or system then) replaces the individual behaviors for some program as mentioned above. In order to define an *error vector*, we suggested to *inspect whether we would derive a contradiction and whether we would fail to derive the expected conclusions*.

In Sec. 3, we will show how to extend this concept to the live scope of RBL such as to derive a measure for our confidence in the individual rules working out as expected.

3 Making decisions based on diagnostic data

As we've been discussing in the previous sections, the calculation of the individual rules' weights and thus in turn that of a plan as defined by the sum of the invoked rules' individual weights is crucial for performance. With our previous work, we already made some improvements over Krenn's original reasoning, but this required us to introduce additional parameters. Selecting values for these parameters, while being obviously critical, has not been a straight forward task and required tuning as well as domain knowledge and some try and error experimentation. Furthermore, the performance we experienced for configuration scenarios turned out to be below our expectations.

Before describing our concept for replacing these manual parameters with light-weight diagnostic reasoning, let us first describe how RBL weights are currently computed. This computation is based on two values: *activity* for measuring how often a rule was already selected, and *damping* for measuring how often a rule failed so far. When discharging the update task (see Sec. 2), the runtime engine updates *activity* for every rule according to Eq. 1, where *chosen* is 1 if the rule was selected and 0 if not. The primed version of a value in our equation always refers to the updated value. The *damping* values get updated in two steps: (1) for those rules included in the plan, the value is increased by *dval* iff the rule execution failed (Eq. 2) and is decreased by *dval* if rule execution succeeded (Eq. 3) — with an upper bound of $1 - \mathit{dval}$ and a lower bound of *dval*. In a second step, the value is updated for *all* rules according to Eq. 4, regardless of whether they were part of the plan or not. The parameters *dval*, *val*, *target*, and *zone* have to be provided and tuned by the designer, where *dval*, *val*, and *target* are float values in the range $(0, 1)$ and *zone*() is a function that returns $\top$ or $\bot$.

The final weight is then calculated from *activity* and *damping* according to Eq. 5, where the designer has to choose float values for *activity_scaling* and *damping_scaling* in the range $(0, 1)$.

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

$$\mathit{damping}' = \begin{cases} 1 - \mathit{dval} & \text{if } (\mathit{damping} + \mathit{dval}) \\ & \quad > 1 - \mathit{dval} \\ \mathit{damping} + \mathit{dval} & \text{otherwise} \end{cases} \tag{2}$$

$$damping' = \begin{cases} dval & \text{if } (damping - dval) < dval \\ damping - dval & \text{otherwise} \end{cases} \tag{3}$$

$$damping' = \begin{cases} damping + val & \text{if } zone\,(damping) = \top \wedge damping < target \\ damping - val & \text{elif } zone\,(damping) = \top \wedge damping > target \\ damping & \text{otherwise} \end{cases} \tag{4}$$

$$\begin{aligned} weight =& (1 - (activity * activity\_scaling)) * \\ & (1 - (damping * damping\_scaling)) \end{aligned} \tag{5}$$

Important to note is that higher weights indicate better performance in this calculation, which is in contrast to our new SFL based computation described later on. Please also note that for Krenn's original approach and our previous version of RBL, only minimal plans where we cannot remove any rule (but which can be of varying cardinality) are considered when selecting the most promising one. Ensuring this minimality is important, since in principle adding actions would increase the weight/attractiveness of a plan.

As we discussed in Sec. 2, SFL has been used traditionally for identifying faulty components when given data about a test suite execution for some system. In our context, rules represent the considered components, and our aim is now the exact opposite. That is, we aim to select the most promising rules, or in other words, those rules that are the least likely to fail. Furthermore, we are operating in a live setting where we continuously collect new execution data.

Employing SFL for our purposes does not require massive changes in the runtime engine. In principle, we continuously execute a system (in detail, this means executing plans π_i to achieve the current goals), get feedback about the success of the execution of some π_i, and compute the *frequencies* $n_{CN}(c_i), n_{CE}(c_i), n_{VN}(c_i)$ and $n_{VE}(c_i)$ (see Sec. 2) for the individual rules c_i on-the-fly. Formally, whenever we know whether some π failed or succeeded, we would add another column to the spectrum A catching which rules were part of π and which were not. Furthermore, we enlarge the error vector e with one field for π and add the info whether π failed or succeeded. These data then define the desired frequencies and in turn the values for the chosen similarity coefficient.

In practice, we keep track of the current values for the frequencies, and whenever a plan π fails or succeeds, we increase the appropriate frequencies by one. Then we can compute the corresponding similarity coefficients for the individual rules via the simple formulae depicted in Sec. 2. In Algorithm 1, we can see a more detailed version of this concept for supervising the plan execution, including all the loops and decisions made. In this algorithm, we directly call the actions and repair routines associated with a rule's action class. The function `update_frequencies(E,res)` is used to update the frequencies and subsequently recompute the weights for all the individual rules. It has two arguments: E representing a list containing those rules that have been executed for plan π, and *res* that catches whether the plan failed or succeeded.

The updated weights can then be used in the planning part of the runtime engine. With the new weight model, we are

Algorithm 1 An algorithm that supervises a plan's execution and updates the frequencies and weights for all rules.

Input: a valid plan π and a function *update_frequencies* that updates the frequencies and rule weights
Output: $\top$ if the plan execution succeeded, $\bot$ otherwise

```
 1: procedure EXECUTE_PLAN($\Pi$, $update\_frequencies$)
 2:     $R \leftarrow \pi$ in reverse order
 3:     $E \leftarrow \emptyset$
 4:     $res \leftarrow \bot$
 5:     while $R \neq \emptyset$ do
 6:         $r \leftarrow R.pop()$
 7:         $res \leftarrow r.action.action()$
 8:         if $res = \bot$ then
 9:             $r.action.repair()$
10:             break
11:         else
12:             $E \leftarrow E \cup \{r\}$
13:         end if
14:     end while
15:     $update\_frequencies(E, res)$
16:     return $res$
17: end procedure
```

searching for those plans with the least weight since in our case a plan's weight is associated with the risk of the plan failing. That is, if a rule has a low weight this means the rule is less likely to fail, and again a plan π's weight is computed as the sum of its rules' weights. Thus, we also do not have to reason about minimal plans (as opposed to the earlier computation model described above). That is, since minimal weight is the optimization criterion for the selection, when considering all viable plans for the decision ensures that there is no subplan that can achieve the same goal. Please note that if there are multiple plans with the same optimal weight, we choose the one created first.

Our new concept does not result in higher run-times. That is, instead of updating *damping* and *activity*, we now update the *frequencies*. This is less complex since these are just additions, instead of divisions and additions. Also, for the weight calculation, the run-time costs stayed nearly the same. Instead of Eq. 5, we now have to calculate the similarity coefficients as suggested in Sec. 2—where the formulae for Ochiai, Jaccard, and Tarantula are also not really complex. Overall, the run-time differences are negligible.

A problem that we encountered was that for the similarity coefficients, no valid value exists when we have insufficient data such that we would have a division by zero. Since we're using the values directly as weights, we assigned a small value to a rule's weight in such cases, following the idea that after a cold start—which is when we suffer from insufficient data—a rule (and the action behind it) can be assumed to be healthy rather than faulty.

4 First Experiments

In order to evaluate our new approach, we ran all our previous examples as well as two new examples with different RBL versions and configurations. Since most of the examples depend on random factors, we ran each example with each configuration 50 times and report average values.

All our examples were executed on a Windows 10 PC with JAVA version 1.8.0_171. The PC has an Intel Core i5-7200 @ 2.5GHz CPU, 8 GB of RAM, and an SSD. Our most complex example, the *Mobile Robot*, took on average 3.4 seconds for *Krenn* and on average 1.9 seconds for *Jaccard* for the whole simulation (including a 3D simulation of the robot in its environment). This means less than 1.9 ms were used for each of the 1000 time steps and RBL took only a fraction of it. Please note that our code (including the experimental setup) is publicly available[2].

Now let us briefly introduce the examples. They have between 3 and 5 rules, which, although not large, still shows the viability of our approach.

- **Total fault:** Object detection with different sensors. The system has 3 sensors to choose from, where it can only choose one sensor per time step, and over time, the different sensors experience intermittent faults. The goal is to choose one of the sensors that are currently working. The only feedback to the system is whether the currently chosen sensor is working or not [4].
- **Temporary fault:** The fault patterns of the sensors in this example are such that fewer sensors are available at the same time compared to the previous example. The poor performance of Krenn's approach for this example motivated our previous extensions.
- **Weather scenario:** Object detection during different weather conditions. Again, there are 3 sensors, and each has a different probability of detecting objects in specific weather conditions. Again, we have to choose one sensor per time step. It was not always possible to select a sensor with 100% accuracy, i.e., for some weather conditions, the best accuracy was 70%. Consequently, there are always some failures (see [4]).
- **Mobile robot:** Here, we developed a robot's Modelica model comprising two differential drives and used a physics model to simulate different faults in the power unit. The system controls the voltage supply of the two differential drives and has to go into a degraded mode if there is a fault in the power unit. The goal is to select the right voltage supply such that the robot always drives straight, no matter whether the robot encounters a fault or not. Different to [5], for our comparison, we use the distance between the worst run and the ideal point, i.e., the point where the robot would hold if the robot would drive without any faults, as a measure of success. Since the robot always encounters some faults during our example, the robot can not reach the point where he would end up driving without faults. Therefore, it is not possible to achieve 100% success.
- **Robot conf.:** This example is very similar to the previous one, but instead of selecting the voltage directly, the system must first choose the voltage for the differential drives and then initiate the drive action. Only at the drive action, the system will get feedback. This modification results in a configuration problem.
- **Light bulb:** In this example the system has to select a working battery and light bulb combination to make the bulb light up. Over time the batteries are drained and replaced, and also the bulbs break and get replaced. The goal is to always select a battery and light bulb combination that will work.

We used the following RBL configurations:

[2]https://github.com/martinzimmermann/RBL-test-programs/releases/DX2020

Table 1: Success rates for different RBL configurations.

	Krenn	Aging	D. only	Jaccard	Ochiai	Tarantula
Total Faul	99.6	99.7	**99.9**	**99.9**	2.8	**99.9**
Temporary Fault	93.6	91.6	**99.9**	**99.9**	75.2	**99.9**
Weather Szenario	74.8	77.0	78.2	**82.1**	77.8	78.0
Mobile Robot	41.1	64.3	**83.1**	81.4	0.0	75.2
Robot conf.	58.6	58.6	4.3	**81.4**	0.0	75.2
Light Bulb	54.8	54.8	14.0	**94.4**	6.8	61.6

- **Krenn** uses the original weight model from Krenn's Ph.D. thesis as used also in [4].
- **Aging** uses our previous extensions as described in Sec. 3. Setting val to 0.01, $zone$ to always return true, and $target$ to 0.5, our aim was to bring *damping* back to it's original value over time
- **Damping only** uses the same extensions as **Aging**. Here, we set *activity_scaling* to 0 and *dmping_scaling* to 1—as used to test the mobile robot in [5].
- **Jaccard** refers to our new approach using SFL for RBL (see Sec. 3) and the Jaccard similarity coefficient.
- **Ochiai** refers to our new approach using SFL for RBL when using the Ochiai similarity coefficient.
- **Tarantula** refers to our new approach using SFL for RBL when using the Tarantula similarity coefficient.

From Table 1, we can see that our previous extensions could increase the success rate over Krenn's weight calculation. The parameter-tuning makes a big difference. On one hand, for the *Mobile Robot* example *Damping only* performed much better than the other versions. On the other hand, for the *Robot conf.* example it performed badly.

Surprisingly, *Original* and *Aging* did not perform that bad on the configuration problems. Still, there is room for improvement as shown by our new SFL versions.

In our experiments not all similarity coefficients performed well. We noticed that using *Ochiai* resulted in terrible performance. However, *Jaccard* and *Tarantula* offered the best or close to the best performance. Overall, Jaccard seems to be the most suitable choice in our case. Only for the *Mobile Robot* example, the *Damping only* version of RBL performed better, as we can see also in Table 1. We argue that *Ochiai*'s bad performance might be rooted in the problem that it often calculates NaN with insufficient information, but more extensive experiments are required for isolating the real issue.

5 Related Work

Early on, different methods for representing planning problems have been introduced. For example, Nilsson introduced Teleo-reactive Programs in [16], Fikes and Nilsson later developed the famous STRIPS language in [9] and Blum and Furst refined STRIPS in [10]. Today the most prominent formal language for planning is probably the Planning Domain Definition Language (PDDL) [17].

For self-healing systems, Rincon and Teres proposed a reconfiguration hardware system [18]. Wotawa used Model-based diagnostics in [19] and [20] to detect errors that occur at runtime and then restart the faulty components on-the-fly.

Wilkins advocates in [21] reactive planning for agents, to be better equipped for the real world. For him, reactive planning couples planning and execution, instead of having 2 separated processes. However, he achieves this by planning multi path plans and replanning when an error occurs. Georgeff follows in [22] a similar approach. Instead of replanning, he uses partial planning and delayed decisions to make decisions only when the maximum of information about the decisions is available.

6 Conclusions

We showed how to adopt SFL for a live setting in order to generate a metric catching our RBL rules' healthiness. We used the resulting similarity coefficients as weights for the individual rules, and in turn to select rule sequences that are most likely succeeding in achieving our goals. Combining SFL with RBL enabled us to improve the rule performance predictions. Although neither using feedback from a plan's execution to improve planning, nor using SFL for rule-bases are novel in general, combining both and adopting them for RBL is a novel contribution that leads to attractive results. Now there is no need to fine-tune parameters, but we can rely on a plug and play setup. We can avoid catastrophic results due to wrong parameters and we achieve premium performance also for previously problematic applications.

In our experiments, we did not only compare our approach with previous work, but experimenting with several similarity coefficient metrics we identified Jaccard to offer superior performance for almost all our examples. Only for one example, another variant was *slightly* better.

A side effect of using our concept is that we do not need to check a plan's minimality in the planning stage anymore—which enables us to consider more planning algorithms. In future research, we will thus focus also on improving RBL's planning algorithm in order to keep the runtime overhead as small as possible. We intend to investigate also different strategies to discount the experience from older executions, as might be suitable for highly dynamic applications.

Acknowledgements

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] C. Boutilier, R. Reiter, M.l Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *17th Nat. Conf. on AI and 12th Conf. on Innovative Applications of AI Intelligence*, pages 355–362, 2000.

[2] S. Gspandl, I. Pill, M. Reip, G. Steinbauer, and A. Ferrein. Belief management for high-level robot programs. In *Int. Joint Conf. on AI*, pages 900–905, 2011.

[3] D. Connell and H. M. La. Dynamic path planning and replanning for mobile robots using rrt. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1429–1434, 2017.

[4] F. Wotawa and M. Zimmermann. Adaptive system for autonomous driving. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 519–525, July 2018.

[5] G. Engel, G. Schweiger, F. Wotawa, and M. Zimmermann. A rule-based smart control for fail-operational systems. In *Advances and Trends in Artificial Intelligence. From Theory to Practice*, pages 137–145, 2019.

[6] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *ASE*, pages 88–99. IEEE Computer Society, 2009.

[7] I. Pill and F. Wotawa. Spectrum-based fault localization for logic-based reasoning. In *2018 IEEE Int. Symp. on Software Reliability Engineering Workshops, ISSRE Workshops*, pages 192–199, 2018.

[8] W. Krenn. *Self Reasoning In Resource-Constrained Autonomous Systems*. dissertation, Graz University of Technology, 2008/2009.

[9] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189 – 208, 1971.

[10] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Art. Intelligence*, 90(1):281 – 300, 1997.

[11] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *13th Nat. Conf. on AI - Vol. 2*, pages 1194–1201, 1996.

[12] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[13] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[14] J. A. Jones and M. Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, pages 273–282. ACM, 2005.

[15] R. Abreu, P.r Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. of Systems and Software*, 82(11):1780–1792, 2009.

[16] Nils J. Nilsson. Teleo-reactive programs for agent control. *J. Artif. Int. Res.*, 1(1):139–158, January 1994.

[17] M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL—The Planning Domain Definition Language, 1998.

[18] F. Rincon and L. Teres. Reconfigurable hardware systems. In *1998 Int. Semiconductor Conference*, volume 1, pages 45–54 vol.1, Oct 1998.

[19] G. Steinbauer, M. Mörth, and F. Wotawa. Real-time diagnosis and repair of faults of robot control software. In *RoboCup 2005: Robot Soccer World Cup IX*, volume 4020, pages 13–23. 2006.

[20] F. Wotawa. Reasoning from first principles for self-adaptive and autonomous systems. In *Predictive Maintenance in Dynamic Systems: Advanced Methods, Decision Support Tools and Real-World Applications*, pages 427–460. 2019.

[21] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[22] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *6th Nat. Conf. on AI - Vol. 2*, pages 677–682, 1987.