Martin Zimmermann, BSc

RBL: A Rule-based Language for Fault-Tolerant Systems

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Institute of Software Technology
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Graz, October 2019

Affidavit

Abstract

In modern society, fault-tolerant systems will become more and more important. Fault-tolerant systems will play a large part not only for safety-critical systems, to prevent injuries or even deaths, but also for consumer electronics, because no matter what, we want our electronic devices to work correctly. Unfortunately, physical faults are inevitable. Therefore, fault-tolerant systems are needed to keep electronic devices running after a physical fault.

This thesis introduces the domain-specific language RBL that makes it easy to model such fault-tolerant systems with the help of rules. These rules contain preconditions that need to be fulfilled to make them executable by the system. In return, every rule has postconditions that hold after the rule was executed successfully. With this, programmers can model complicated systems in a very abstract way. Our model is based on the Ph.D. thesis of Willibald Krenn with a few additions.

To test RBL, a fully functional compiler was implemented that is interoperable with Java. This makes it very easy to use RBL in a large variety of different projects. As an evaluation of RBL, a case study in the domain of autonomous driving was created that shows that RBL performs reasonably well for the intended use case.

Contents

List of Figures

Acknowledgement

First, I would like to thank my supervisor, Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa, from the Institute of Software Technology of Graz University of Technology. He always took time for my questions and offered help when I needed it. Throughout this thesis, he guided me but still allowed me to explore on my own.

I would also like to thank all my colleagues at the Institute of Software Technology for giving me technical as well as writing advice. Their tips helped me to avoid a lot of mistakes while writing this thesis.

Also, I would like to thank the projects "Lehrverbund Informatik Süd" and "Christian Doppler Lab for Quality Assurance Methodologies for Cyber-Physical Systems" in the course of which I could write this thesis.

Finally, I would like to express my gratitude to my parents and friends who supported and encouraged me throughout my studies.

1. Introduction

In the first part of this master's thesis, the motivation for this thesis is presented. After that, our objectives for the thesis and what we will cover in it is described. Lastly, a short outline of the structure is provided.

1.1. Motivation

More and more machines these days are powered by computers. However, often these machines have a single program flow and when this program flow is interrupted the machine stops completely. To prevent a total fault, fault-tolerant systems need to be deployed. A fault-tolerant system is a system where if, for example, a component of a machine fails during operation, the program is smart enough to detect this fault and find another, usually less optimal, path to achieve the same or equivalent goal. As soon as the component is repaired, the program gradually starts to use the old, more optimal, path again. This strategy prevents a total fault of the machine and ensures that the machine can continue with its work, even if some of its components fail.

Because writing such a fault-tolerant program in traditional programming languages like C or Java is hard and cumbersome, not many fault-tolerant systems exist. In this thesis, we will introduce a domain-specific language (DSL) that interoperates with Java. This language will enable the user to develop fault-tolerant systems quickly and reliably.

1.2. Objectives and Scope

One of the biggest objectives of this thesis is the development of the domain-specific language RBL (for **r**ule-**b**ased **l**angauge). RBL should be easy to use and convey the intention of the programmer clearly. To use and test the language, a domain-language-to-Java compiler should be developed. The compiler will enable the user to have just plain Java code in the end that can be used on millions of devices. Besides, an interpreter will be part of the compiler to test developed programs quickly. Because RBL will be quite different from traditional procedural languages, good documentation and a few sample programs will also be provided.

1.3. Structure

The second chapter contains a brief history and a quick introduction to fault-tolerant systems, artificial intelligence, and domain-specific languages. This should help the reader to understand the following chapters better. The next chapter describes the design of RBL and analyze the language concerning several aspects. This had been done before the actual implementation but was revised throughout the implementation phase to correct the mistakes we had made while planning. Implementation details like the architectural design and different algorithms used by the system can be found in the fourth chapter. Also, our encountered problems are described in this chapter. The fifth chapter showcases how the compiler can be used and gives two examples that could be implemented with RBL. In the sixth chapter, a case study that was conducted with RBL and gives the first empirical results is detailed. Finally, the last chapter summarizes our findings and gives an outlook for future work.

2. Preliminary

This thesis should be self-contained. Therefore, we will give an introduction to fault-tolerant systems, neuronal networks, and domain-specific languages. We will not cover the topics in all of their depths, but the interested reader can find additional information in the referenced papers.

2.1. Fault-Tolerant Systems

Algirdas gives an excellent definition of a fault-tolerant system in his paper [1] : "it is a system which has the built-in capability (without external assistance) to preserve the continued correct execution of its programs and input/output (I/O) functions in the presence of a certain set of operational faults."

In the early years of computing, this behavior was only crucial for highly critical systems like nuclear power plants and space missions. Because more and more people use computers these days and the applications get more and more complex, this also becomes a highly desirable property for regular programs.

2.1.1. History

As Carter and Bouricius write in [2] at the beginning of computers, most of the errors that happened were hardware faults, caused by a short life span of the relays and tubes used. The staff of Engineering Research Associates, Inc. even estimated that the ENIAC only were fully operational 50% of the time. At this time error detection was far more important than automatic error correction.

With the shift to transistors, the life span of the components increased dramatically, and error detection was focused on Input / Output devices. Around this time, the first computers implemented parity checks and error correction codes to make communication over wire or inside the computer more reliable.

The importance of this field is also shown by the formation of the IEEE Computer Society Technical Committee on Fault-Tolerant Computing in 1969. Since then, conferences on this topic were held regularly.

2.2. Artificial Intelligence

Replications of the human brains intrigued humans for centuries [3]. Still, the research in General Artificial Intelligence, which tries to have an Artificial Intelligence that can learn everything like a brain, is quite active [4], however, this seems to be a very challenging task.

The field of Artificial Intelligence is dotted with high expectations and significant breakthroughs, but also with many disappointments. In recent years computing power increased, and we have more and more data available, this led to a new wave of Artificial Intelligence usage. However, this time, the goal is to try to create Artificial Intelligence that is capable of a particular task, like playing chess.

Mapping inputs to outputs, through multiple layers of neurons, achieves this. Each neuron has one or more input signals and one or more output signals with different weights. Depending on the neuron, the input is mapped through different functions to an output. A variety of complex functionalities can be achieved when combining different weights and different functions. To get the correct weights and functions, usually, a feedback loop, mostly called learning, is implemented [5]. In Figure 2.1, we can see an example of a neural network.

2.3. Domain-Specific Languages

General speaking there are two types of programming languages used. First, general-purpose languages that can solve almost all problems and second,

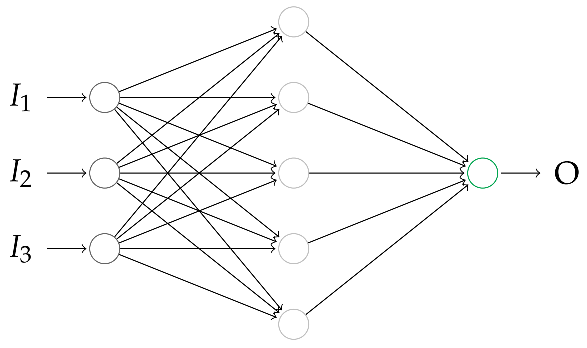

Figure 2.1.: Example of a Neural Network

domain-specific languages that are used for a specific domain. Most of the times, these domain-specific languages are more precise and more comfortable to read for humans in their respective domain. For example, writing the user interface of a website is much easier in HTML than it is in Python. Hinsen also points out in his paper [6] that some languages, although they are intended for a specific domain, like Tex for typesetting documents, can be misused as general-purpose languages.

Because of their ease of use for end-user, that not necessarily need to have a background in programming, this field is still very active. A search on ieeexplore.ieee.org for "Domain-Specific Language" gives around 4000 results and one can find a domain-specific language for almost every domain.

To no surprise, there was also a paper from Ahmad [7] that focuses on a very similar topic as we do in this theses. His approach is to encode the uncertainty of failure directly in the language by operations that should be executed but are not critical to the system.

3. Design

The design and implementation of a compiler for RBL was the main part of this thesis. First, we will describe the fault-tolerant system from [8], which is the basis for our language. Then we will continue with the design of RBL and the Java interface. In the end, we will analyze RBL based on nine criteria for modeling languages introduced in [9].

3.1. Fault-Tolerant System

Over the years, many fault-tolerant systems were proposed. The fault-tolerant system from [8] is one of the simpler ones. It is easy to understand but can deal with many situations. For example, a part of a machine is broken and is manually replaced after some time. In this scenario, the system would switch to another part of the machine that is still working without human interaction. After the part is replaced, the system will gradually start to use the replaced part again.

In the first subsection, we will describe the weight model, which models the world, and the abstract rules, which enables the fault-tolerant system to react to unforeseen events. In the second subsection, we will briefly discuss the plan search, followed by the execution and the update routines of the system.

3.1.1. Building Blocks of the Fault-Tolerant System

As stated before, the fault-tolerant system of this thesis is based on the weight model proposed in [8]. As an extension, we added the aging of the damping factor to counter some flaws in the model. Contrary to other models, it is not assumed that everything true in the model is also true in the world. Because the model is an imperfect representation of the world, actions that, in theory, should work can also fail, without a failure of the whole system. For example, a window can be stuck, without a visible indication, and although normally it is not a problem to open a window, in this case, the window can not be opened. Encoding all these unexpected situations in the model would require much tedious and error-prone work and probably would never reflect the real world completely. Instead, the system becomes tuned to such failures and tries to avoid actions that fail often. To continue the example above, this would mean trying a different window, instead of doing nothing or trying to open the same window multiple times.

To do that there are three basic building blocks:

Propositions	$\mathcal{P}$
Actions	$\mathcal{A}$
Rules	$\mathcal{R}$

- **Proposition:** Represent the world state. If a proposition is contained in the current model about the world, it is assumed to be true. (e.g., "bottle is open")
- **Action:** The actual actions or programs that the system should execute. (e.g., "drink")
- **Rule:** Is a combination of the pre-conditions that have to be met in order to execute a specific action and also specifies which post-conditions will hold if the rule was executed successfully. Furthermore, the rule has several properties attached to it, which, in combination, enable the fault-tolerant behavior. (e.g., "if the bottle is open you can drink from it if so the bottle will be empty")

3.1.1.1. System

The system is responsible for collecting and managing all the important information about the current state of the model. Furthermore, it executes a specific set of actions to achieve a certain goal. To achieve that, the system contains a set of rules that can be executed and a set of propositions. This set of propositions reflects what the system thinks is true about the world (e.g., "the window is open") we will refer to this set of propositions as memory from now on. We say something is contained in the memory if this proposition is part of the memory.

For every execution the system has three distinct phases:

- **Planning Phase:** In this phase, the system tries to find an optimal set of rules to achieve a certain goal, also considering the pre-conditions and post-conditions of the different rules. This phase will be further explained in Section 3.1.2.
- **Execution Phase:** In this phase, the system executes the actions that were calculated by the planning phase. This phase will be further explained in Section 3.1.4
- **Update Phase:** In this phase, the system updates the parameters of the rules depending on different criteria. This phase will be further explained in Section 3.1.4.

3.1.1.2. Rule

The main concept of the weight model is the rule. The system calculates a sequence of rules, which are unlikely to fail and lead to a given goal. After the execution, the fault-tolerant system updates all rules accordingly.

Following properties are part of each rule:

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	$\alpha' : [0,1] \rightarrow \mathbb{R}$
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	$damping \in [0,1]$

- **Precondition:** The pre-conditions are propositions that need to be contained in the memory for the rule to be executable. We combined the *Test* and the *Reach* properties from [8] to make execution simpler. Krenn himself sometimes does this in his dissertation to simplify things.
- **Action:** As described above, this is the action or program that the rule will execute when it is executed.
- **EffectAdd:** If the rule gets executed successfully, this proposition is added to the memory.
- **EffectDel:** If the rule gets executed successfully, these propositions are deleted from the memory.
- **TargetActivity:** Describes how often a rule should be selected.
- **ActivitySlope:** Is a function that helps to modify the weight of the rule in different ranges. See Figure 3.1 for an example.
- **DampingValue:** Is the value by which the damping should be increased or decreased if the rule was successfully executed or failed.
- **AgingValue:** Is the value by which the damping should be increased or decreased after every execution.
- **AgingTarget:** Is the value towards which the damping should age.
- **AgingZone:** Describes the range in which the damping should age.
- **Activity:** Is an indicator of how often the rule was already chosen.
- **Damping:** Describes how often the execution already failed or succeeded for this rule.

Precondition, *Action*, *EffectAdd*, *EffectDel*, *TargetActivity*, *ActivitySlope*, *DampingValue*, *AgingValue*, *AgingTarget* and *AgingZone* have to be designed by the programmer to achieve the desired behavior of the rules. *Activity* and *Damping* are dynamic values that are updated after every execution.

3.1.1.3. Plan

The fault-tolerant system always interprets *Paths*, which are 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, which the fault-tolerant system tries to reach, and is interpretable if the execution of all *Actions* in the *Path* would be successful.

Furthermore, a *Plan* is 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.

An example of a *Plan* would be:

The door is locked, and the goal is to get out of a room

> (get the key,
> put the key in the door lock,
> unlock the door,
> open the door,
> go out through the door)

However, if in the above example, the door would not be locked, this would only be a *Reasonable Path*.

3.1.2. Plan Search

One of the most important components of the system is the plan search. It searches through the rules and finds the list of rules that if executed in the given order, would most probably lead to a goal rule. Furthermore, all the pre-condition of the rules on this path must be met at the time the rule would be executed.

With a set of rules that enable the fault-tolerant system to react to events in different ways, there is also a need to decide which rules are more likely to succeed if multiple rules would satisfy the criteria mentioned above. For this purpose, the weight formula, shown in Equation 3.1, is used.

$$\begin{split}\gamma(activity, damping, target, \alpha') = \\ \alpha'(activity) * (target - activity) * (1 - damping)\end{split} \tag{3.1}$$

Several search algorithms can be used to perform the plan search. In Section 4.2, we will discuss the different plan search algorithms we implemented. It is important to note that although it would be good always to use a *Plan*, most of our algorithms actually produce *Reasonable Paths*. Therefore, we will relax the strict definition of a plan in the following chapters as just meaning a *Reasonable Path* that is minimal according to any plan search algorithm.

3.1.3. Execution

After the plan was calculated, the fault-tolerant system begins to execute the given plan, starting with the first rule. Either the execution is successful by reaching the desired goal rule, or a fault occurs while executing.

The important thing to note is that a fault does not mean the program will terminate. Instead, it is just assumed that the model of the world, which the fault-tolerant system had, was incorrect. Every action also has a repair function that can repair the memory and should do so if an error was encountered. For example, this gives the developer a chance to request some additional sensor data to check why the fault occurred and correct the memory accordingly.

After the memory is repaired, the fault-tolerant system updates the rules and starts to search for a plan again. Because of the updated memory and the updated rules, a plan without fault is likely to be taken.

3.1.4. Rule Update

The rules are updated after each execution. This ensures that the fault-tolerant system can react to the execution. The update also prevents the fault-tolerant system from using only just one execution path, but rather test all paths from time to time to find the optimal path, even if the system changes unnoticeably. We distinguish between three different updates, activity update, damping update, and aging.

3.1.4.1. Activity Update

Activity is recalculated for every rule, not just for the ones that were chosen. However, depending on whether the rule was part of the plan or not, it is calculated differently. Equation 3.2 gives the *Activity* value for the next iteration. Where $chosen_n$ is 0 if the rule was not chosen and 1 if the rule was chosen.

$$activity_{n+1} = \frac{1}{2}\left(chosen_n + activity_n\right) \tag{3.2}$$

3.1.4.2. Damping Update

If a rule was actually executed, depending on how the execution went, *Damping* is updated. When the rule was executed successfully *Damping* is decreased according to Equation 3.3 and if the rule was executed unsuccessfully *Damping* gets increased according to Equation 3.4.

(1) Damping factor computation in case of a successful execution:

$$damping_n = \begin{cases} 0.1, & (damping_{n-1} - dampingVal) \\ & < 0.1 \\ damping_{n-1} - dampingVal, & \text{otherwise} \end{cases} \tag{3.3}$$

(2) Damping factor computation in case of a unsuccessful execution:

$$damping_n = \begin{cases} 0.9, & (damping_{n-1} + dampingVal) \\ & > 0.9 \\ damping_{n-1} + dampingVal, & \text{otherwise} \end{cases} \tag{3.4}$$

3.1.4.3. Aging

After every execution *Damping* of every rule is aged. The aging is only done if the current *Damping* value evaluates to 1 if evaluated with the *AgingZone* function. Also *Damping* ages always to a specific value, this gives the programmer the possibility to age *Damping* in any direction he wants. In Equation 3.5 this process is formally specified.

$$damping_n = \begin{cases} damping_{n-1} + agingVal, & agingZone\left(damping_{n-1}\right) = 1 \\ & \land\, damping_{n-1} < agingTarget \\ damping_{n-1} - agingVal, & agingZone\left(damping_{n-1}\right) = 1 \\ & \land\, damping_{n-1} > agingTarget \\ damping_{n-1}, & \text{otherwise} \end{cases} \tag{3.5}$$

3.2. Language Design

Designing an easy to use and very expressive language was the main goal for us. Because rule-based languages are not as well known as procedural languages, we tried to make RBL as easy to understand and use as possible. We also decided to have many opt-in features and only make the most important parts mandatory. This not only allows a quick prototyping phase but also makes it easier to learn the language. First, we will describe the nature of our DSL. Then we will describe the language and its features in textual form, and afterward, we will give a formal definition in Backus-Naur form. The last part of this section will include the Java interface, which enables RBL to interact with Java.

3.2.1. Form of DSL Implementations

In [10] Spinellis identified eight design patterns on how to implement a DSL.

1. **Piggyback:** The piggyback pattern enables the DSL to use capabilities of the host language into the DSL. For example, this includes expressions or subroutines.
2. **Pipeline:** The pipeline pattern is used when the source DSL is translated into another DSL instead in the host language directly. This enables a pipeline of multiple DSL interpreters.
3. **Lexical processing:** The lexical processing pattern is used when there is no need for a tree-based translation. This can include simple text replacements, for example, the C preprocessor.

4. **Language extension:** The language extension pattern takes the host language and adds additional features to the language. An example would be the early versions of the C++ compiler.
5. **Language specialization:** The other direction is used in the language specialization pattern. This pattern removes specific instructions from the host language to make it safer or easier to use. For example, removing dynamic memory allocation.
6. **Source-to-source transformation:** The source-to-source transformation is used when the DSL is transformed into the host language. This enables the developer to leverage the power of the tools of the existing host language.
7. **Data structure representation:** This patterns enables a more user-friendly representation of complex domain data structures. For example, a visual graph is easier to understand than a list of node connections.
8. **System front-end:** The system front-end pattern is used when a program has more configuration options than is typically feasible for program parameters or a graphical user interface. By using this pattern, the program becomes easier to configure and is extensible. An example of this is the extensibility of Emacs.

RBL uses a combination of the source-to-source transformation, piggyback, and data structure representation patterns.

The source-to-source pattern is used because RBL will be translated to Java. This enables us to use a wide variety of tools already available for Java. Also, Java is a platform-independent language and, therefore, our end code can run on millions of devices.

For the implementation of the actual actions, which will be performed, we will use plain Java. Therefore, the piggyback pattern is also used by our language.

Finally, for the representation of the rules, we do not use the internal Java representation, but an easier to understand rule-based-language-style representation. This is possible through the data structure representation pattern.

3.2.2. Language Description

RBL borrows heavily from the syntax of the ATMS solver [11]. To illustrate the syntax of the language we will use the listing given in Listing 3.1.

```
active.
active -> +detected useSensor1.
active -> +detected useSensor2 (0 <= a < 0.25: a,
                     0.25 <= a < 0.5: a * 2, 1) 2.
active -> +detected useSensor3 [0.1, 0.01, 0.75|].
detected -> #processed -detected processObject.
```

Listing 3.1: Example RBL program

First, we have the propositions that are just denoted as "<proposition>.". *e.g.: active.*
These propositions represent the world state at the beginning of the execution.
Second, we have the rules. *e.g.: active -> +detected useSensor1.*
A rule has several parts. First on the left-hand side of the '->' the propositions are defined that need to be contained in the memory in order for the rule to be executed (Pre-conditions). Also, no propositions are possible.
On the right-hand side, there are multiple parts. The first part is either a single addition of a proposition to the memory (denoted by "+<proposition>") or a single goal that should be reached (denoted by "#<proposition>"). This part is optional. (EffectAdd) *e.g.: +detected or #processed*
The second part consists of zero or more removals of propositions from the memory (denoted by "-<proposition>"). (EffectDel) *e.g.: -detected*
The next part is the (Java) action that should be executed. This name should be given as a fully qualified name to a corresponding Java class as described below. (Action) *e.g.: useSensor1*
Next, we have the α' function. This is a list of ranges and their corresponding functions, all of the basic arithmetic operators are available, and the variable 'a' denotes the input value of the function and can be used in the calculation. The last value of the list is the "everything else" value and can be omitted if the whole range from 0 to 1 is covered. If no list is given the standard function $\alpha'(a) = 1$ is taken. *e.g.: (0 <= a < 0.25: a, 0.25 <= a < 0.5: a * 2, 1)* would lead to the slope shown in Figure 3.1. (ActivitySlope)

After that, we have the rule's rulegoal that determines how frequent the rule will be executed. If no rulegoal is given, the rulegoal is assumed to be 1. (TargetActivity) *e.g.: 2*
Lastly, we have the aging block. In this block enclosed by "[]", we have three numbers. The first number represents the value that damping will be decreased or increased if the rule was executed successfully or failed. The next value is the

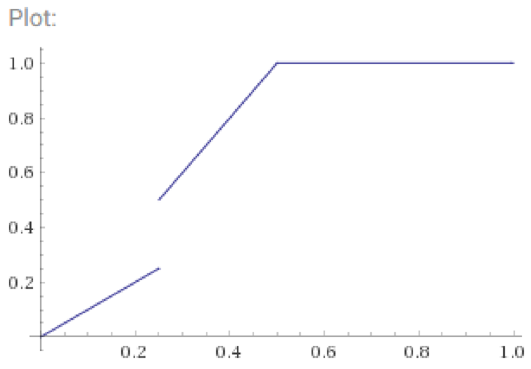

Figure 3.1.: Alpha slope for: (0 <= a < 0.25: a, 0.25 <= a < 0.5: a * 2, 1)

value by which damping will be increased or decreased at the aging step. The last value is the aging target which can have three forms either "x", "|x", or "x|", where x is always a number between 0 and 1. If the first form is chosen this means, the aging happens for all values of damping. The second form means that aging only happens for damping values greater than x. Moreover, the last form means that aging only happens for values smaller than x. *e.g.: [0.1, 0.01, 0.75|]*
Table 3.1 shows this by an example. If the whole aging block or single values of the aging block have been omitted the standard of "[0.1,0,0]" is taken, which means aging is not enabled by default.

current *Damping* /aging target	0.5	|0.5	0.5|
0.2	increase	nothing	increase
0.8	decrease	decrease	nothing

Table 3.1.: Example of aging with different *Damping* values and aging targets.

3.2.3. Backus–Naur Form

The Backus-Naur Form is a notation that is often used for describing syntaxes. We use the Backus-Naur form with the common extension of Regular Expression repetition symbols (*, +).

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

program       ::= [memory] r_rule
memory        ::= (predicate ".")
predicate     ::= ID
r_rules       ::= (r_rule ".")+
r_rule        ::= [predicates] "->" [("+" | "#") prediacte]
                  ("-" predicate)* action [alist]
                  [rulegoal] [aging]
predicates    ::= predicate ("," predicate)*
action        ::= ID("."ID)*
alist         ::= "(" ((alistentry ("," alistentry)* ["," expr])
                | expr) ")"
alistentry    ::= NUMBER ("<" | "<=") "a"  ("<" | "<=") NUMBER
                 ":" expr
rulegoal      ::= 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 ")"
```

3.2.4. Java Interface

The main advantage of RBL is that it nicely integrates with Java. As mentioned previously, RBL compiles to pure Java, hence the integration is straightforward. First, we will show the interface that is used to specify the rule actions. Then we will discuss how Java programs can leverage the fault-tolerant system as a part of their execution. Also exiting Java code can be reused when migrating to RBL.

3.2.4.1. Action

The first interface is the interface for the action that is executed if the action was chosen in a plan, and no previous action failed in the plan. Every class that will contain an action has to implement the following interface.

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

Listing 3.2: *RuleAction* Java interface

The *execute()* method is called when the action gets executed. It should perform the action immediately and return true if the execution was successful and false if it was not. The method can use the memory to change its behavior based on the current world state. However, this is not encouraged.

If the action fails, the framework will call the *repair()* method. This method should repair the memory to have a valid world state again. This function is vital that at the next iteration, a different rule is chosen.

3.2.4.2. Executor

The *Executor* is the main class for another Java program to use the framework. It provides all the necessary functionality to execute plans. Also, the *Executor* gets auto-generated according to the RBL source file.

```
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();
}
```

Listing 3.3: *Executor* Java class

For all *executesX* methods hold that for the first execution of any of these functions the predicates given in the RBL source file will be loaded into the memory. After the execution, the memory is not erased. Each further execution of such a function will also not load the predicates of the RBL source file again.

Furthermore, if it is not possible to find a plan, a NoPlanFoundException is thrown.

3.3. Analysis of the Language

In their paper [9], Paige et al. define nine criteria by which a modeling language can be measured. These nine criteria are based on the five criteria given in [12] by Hoare. Because of the semi-graphical representation of RBL, we think that these nine criteria should also hold for our language.

3.3.1. Simplicity

Paige et al. argue that simplicity is the most important criterion when designing a language. A simple language means that it is small and can be learned by its entirety by the programmer.

We think RBL scores very well at this criterion. The semi-graphical syntax (*e.g.:* ->) gives the programmer an immediate context. Because there are no keywords, only a few key-symbols, the syntax of the language can be learned very quickly. Furthermore, the many optional parameters make it easy to get started with RBL and learn more advanced concepts when they are necessary.

3.3.2. Uniqueness

Uniqueness refers to the features of a language. Every language should try to provide one and only one way of modeling a concept. Redundant or only slightly different features should be avoided. This helps the programmer to understand the language quicker and take off some decision burden from the developer.

Seeing that there are only two language features, memory and rules, which map directly to the concepts of the fault-tolerant system, RBL is also very good in terms of this criterion. Furthermore, even all the parameter of the rules have a direct mapping to concepts of the fault-tolerant system and could not be expressed any other way in RBL.

3.3.3. Consistency

For a language to be consistent means that all features of this language aid to further the purpose of this language. In contrast, all features that do not aid the purpose of the language should be discarded.

Because RBL was developed to efficiently designing fault-tolerant systems and is based on the idea of the weight model mentioned above, we only included these features. This means that also regarding this criterion RBL scores well.

3.3.4. Seamlessness

A language that can be translated into executable code achieves the criterion of seamlessness. Furthermore, the notations and concepts for the modeling language and the target language should not differ. For example, this means, when a class is modeled in an object-oriented modeling language also the compiler should generate these classes.

Regarding this criterion, RBL scores not so well. The language can indeed be translated to executable code, as we will show in the next chapter, but the concepts between RBL and the target language differ. In RBL, we have rules, but in Java, we have object instances which represent these rules. In addition, the complete inner working of the fault-tolerant system is hidden from the user.

3.3.5. Reversibility

Reversibility refers to reversing the process from a modeling language to a target language. This means that if the code in the target language gets updated the model in the modeling language will also reflect that change. Paige et al.

argue that if this is not possible, only the code will be maintained and the model will be discarded over time because it is too much effort to keep it up to date. This usually needs also some tool support.

In theory, RBL could support this too. Generating a model from the Java code should not be difficult, and all necessary parts of the model are present in the Java code. However, because this is just an academic research prototype, no efforts to provide such tooling support have been made.

3.3.6. Scalability

For a modeling language to be useful for more than "toy" systems, scalability is essential. Scalability refers to the ability to model small systems as well as larger systems. A technique that supports this is called grouping, which allows the programmer to abstract bigger concepts while modeling.

For this criterion, our language scores even worse than in reversibility. RBL provides no mechanism for grouping a set of rules together to abstract a more complicated action. For example, it would be desirable to have a subpart of the rules only concerning moving the arm of a robot, so the developer does not have to be concerned how the movement works in detail and can just use the subpart.

3.3.7. Supportability

Supportability refers to the ability that a human can use the language. Which means that it should be easy to use the modeling language on a computer. The tools used for that should also help the programmer to spot error easily and early. For example, syntax checking should be performed.

Because RBL is text-based, this is one of the principals which is not very applicable. Creating a program in RBL on a computer is very easy. In addition, a compiler that compiles the language will perform lexical, syntax, and semantic checks while compiling to guaranty that the input adheres to the language specification.

3.3.8. Reliability

A significant criterion for languages is also that the resulting target language code is reliable and coherent to the specification and the compiler should throw an error if there is an unexpected input. A good way to test for this criterion is formal analysis and testing.

No formal analysis was done for our compiler. However, we have tested the compiler thoroughly, and we are quite confident that there are no significant errors in the compiler. However, as Dijkstra famously said in [13]: "Testing shows the presence, not the absence of bugs."

3.3.9. Space Economy

The Space Economy refers mostly to the space used by the modeling language on paper. In smaller models, there is less to see and therefore also less to understand.

Because RBL uses descriptive key-symbols, instead of keywords, and a minimal syntax, the space that our language requires is minimal. This is further enhanced by the fact that many features are optional and do not have to be specified if the standard values are used.

3.3.10. Summary

Because of the small, yet powerful, domain that RBL can be used in, we were able to achieve a high score in *Simplicity*, *Uniqueness*, *Consistency* and *Space Economy*. The small keyword count, as well as the many optional parameters, make it a good language to prototype fault-tolerant systems quickly.

Because RBL is based on the ATMS language and therefore on rule-based languages rather than procedural languages, we have a poor score in *Seamlessness*. On the other hand, this makes it possible to develop fault-tolerant systems much quicker. Therefore, we think this is a necessary trade-off.

Reversibility and *Scalability* are two areas where our language could need some improvements. However, as stated previously, because this is just an academic research prototype, we did not see the need to implement these features.

4. Implementation

After we designed RBL to fit our needs, we began to implement the compiler. A working prototype of the compiler was an essential goal of this thesis. Therefore, we put much work into developing the compiler. In this chapter, we will first explain our architectural design and explain the different components. In the second section, we will explain the different Path Search Algorithms that were implemented and compare them. The last section will deal with the discussion of problems we encountered while implementing and testing the compiler.

4.1. Architectural Design

As for most compilers, we build a pipeline. This enables us to reuse or replace existing components very easily. The main components of our compiler are shown in Figure 4.1. The interfaces between the components were kept generic to facilitate simple reuse of components further. A good example of this is that the interpreter for the interpreter mode and the interpreter for the compiler mode is nearly identical. In the following sections, we will describe the individual components in more detail.

4.1.1. Precompile

An essential step is performed before the actual compilation if the interpreter mode is chosen.

Because the only way to load classes written in Java dynamically at runtime of a Java program is through the Java compiler service, which is only included in

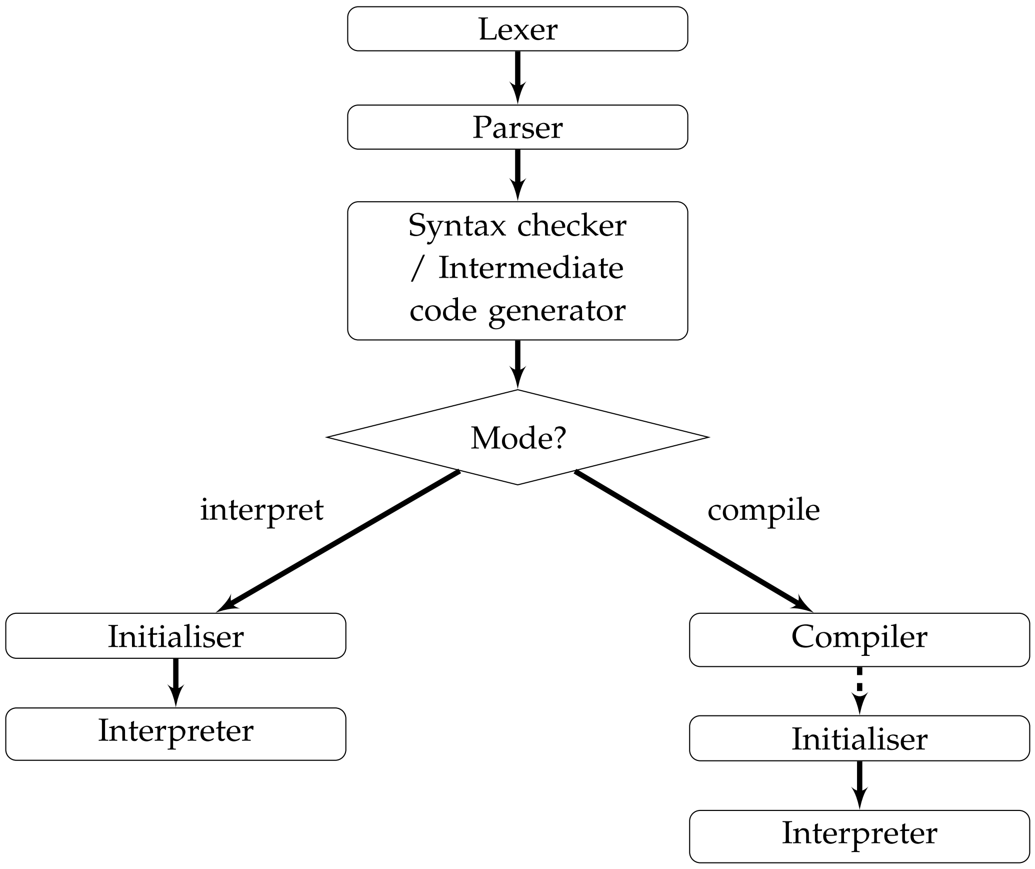

Figure 4.1.: Compiler phases

the Java Development Kit (JDK), the JDK is required for the interpreter mode. It is, therefore, checked if the JDK is installed.

Furthermore, every class in the folder provided is compiled and loaded into the Java classpath, if the interpreter mode is selected. This is done independently from whether the classes are used or not. Because of that, we can instantiate these classes if needed.

4.1.2. Lexer / Parser

We refrained from implementing our own lexer and parser. Instead, we used the excellent library ANTLR 4, which is described in great detail in [14].

ANTLR generates the lexer and parser automatically from a .g4 source file. In this file, all the lexer and parser rules are defined, and it is very human readable. Also, it is straightforward to change the grammar of the language, which made the prototyping phase much quicker than if we had developed our own lexer and parser.

First, the lexer creates a token stream out of the RBL source file. This token stream is then further processed by the parser. The parser then segments the token and creates an abstract syntax tree (AST). The AST is then the final output of the parser and is further processed by the syntax checker and intermediate code generator.

4.1.3. Syntax Checker / Intermediate Code Generator

Compared to other languages our syntax checker performs only minimal checks. The main reason for this is that we do not have a type system at all. Still, a few syntax checks are performed like, if the whole range for the α'-function is covered as described in Section 3.2.2.

Checks if all predicates from the preconditions have at least the chance to be added are left out on purpose because the actions of the rules can manipulate the memory freely.

The intermediate code generator then produces the memory and rules as a Java data structure according to the specification of RBL. Important to note is that in this phase the compiler does not check whether for every action given in the RBL source file a Java class exists. This was done to have more similarity between the compiler and the interpreter because they both can create these classes differently, as we will show in the next subsections.

4.1.4. Initialiser

The initializer has the important role of creating the memory and rules as described in Section 3.1.1. Depending on what mode is chosen, different steps are performed.

- **Interpreter / Dynamic compilation:** In this mode also the classes get initialized that were added to the classpath in the precompile step(Section 4.1.1).
- **Normal compilation:** Because the classes were hardcoded into the interpreter, the classes could be compiled as part of the program. Therefore, no extra initialization of classes that represent the actions is necessary.

4.1.5. Interpreter

The interpreter is the part of the program where pathfinding, execution of the actions, and updating of the weight model is happening. For the interpreter, it does not make a difference if he was invoked through the interpreter mode or the compiler mode and functions in both cases the same. Only the interface to the interpreter is a bit different.

4.1.6. Compiler

Our compiler is a to-source compiler. Which means that the output of the compiler is Java source code. This enables us to be flexible with the output program.

At the compile step the Java data structure that represents the memory and rules are written in a Java file. Depending on the chosen mode this Java file can then be used in another project either together with the library or standalone.

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

Listing 4.1: Simple example that is used to demonstrate the different algorithms

In the other Java program, the memory and rules are again transformed to a Java data structure and can then be used with the interpreter (Section 4.1.4).

4.2. Path Search Algorithms

One of the main problems with RBL is the search for rules that can be taken. In [8] this is also stated, and one algorithm (in our thesis called Bottom Up Path Search) is proposed to solve this problem. The problem with this algorithm is that it is not searching for the optimal path and has some limitations. Therefore, we propose two new algorithms that do not have these limitations. In the case of the Top Down Path Search, which also searches not for the optimum, it has a similar performance as the Bottom Up Path Search. The Optimal Path Search searches for the optimal path, which takes much longer to find a solution than the other two, but always gives back the optimal path. Benchmark of the different algorithms can be found at the end of this section.

Throughout this section, we will use Listing 4.1 to demonstrate the different algorithms.

4.2.1. Bottom Up Path Search

The Bottom Up Path Search is the Path Search Algorithm proposed in [8]. It starts with the goals of the rule set, and then it searches all the rules, which would enable this rule to be satisfied. If the next rules in turn also depend on other rules to be satisfied, it also searches the rules that would satisfy them. Either this goes on until all rules 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,

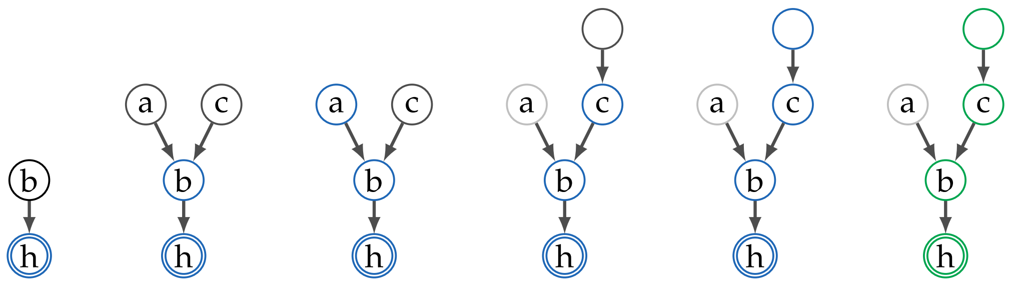

Figure 4.2.: Bottom Up Path Search example

no valid path can be found. Figure 4.2 shows the algorithm with the example given above and Figure 4.3 illustrates the algorithm as a flow diagram.

4.2.2. Top Down Path Search

The Top Down Path Search takes the approach from the other side. It starts with the memory as current propositions and searches for every rule that is possible with the current memory. If one of the rules is a goal rule, it takes the rule and terminates the path search with a valid path. If none of the rules is a goal rule it takes the rule with the greatest weight. Now the memory is recalculated, and the process begins again. If no rule can be added to the path anymore, the algorithm will return with no path found. Figure 4.4 illustrates this algorithm with the example from above and Figure 4.5 illustrates the algorithm as a flow diagram.

4.2.3. Optimal Path Search

The Optimal Path Search is similar to the Top Down Path Search. It not only searches until it finds one valid path; it searches through all paths and saves every valid path. After that, all paths where a rule could be removed, and the path stays still interpretable are removed. After that, the plan with the greatest weight, according to [8] is selected.

This generates a significant overhead but ensures that the globally optimal path is selected.

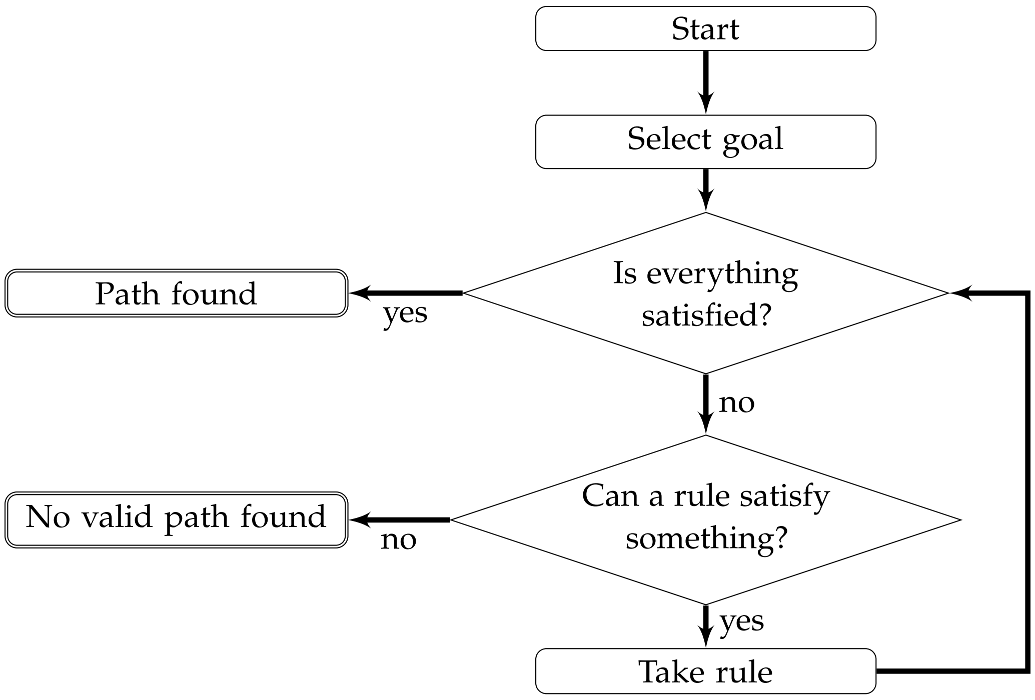

Figure 4.3.: Bottom Up Path Search flow chart

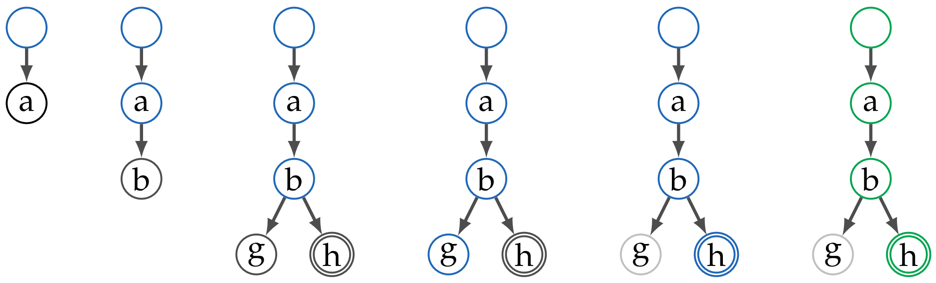

Figure 4.4.: Top Down Path Search example

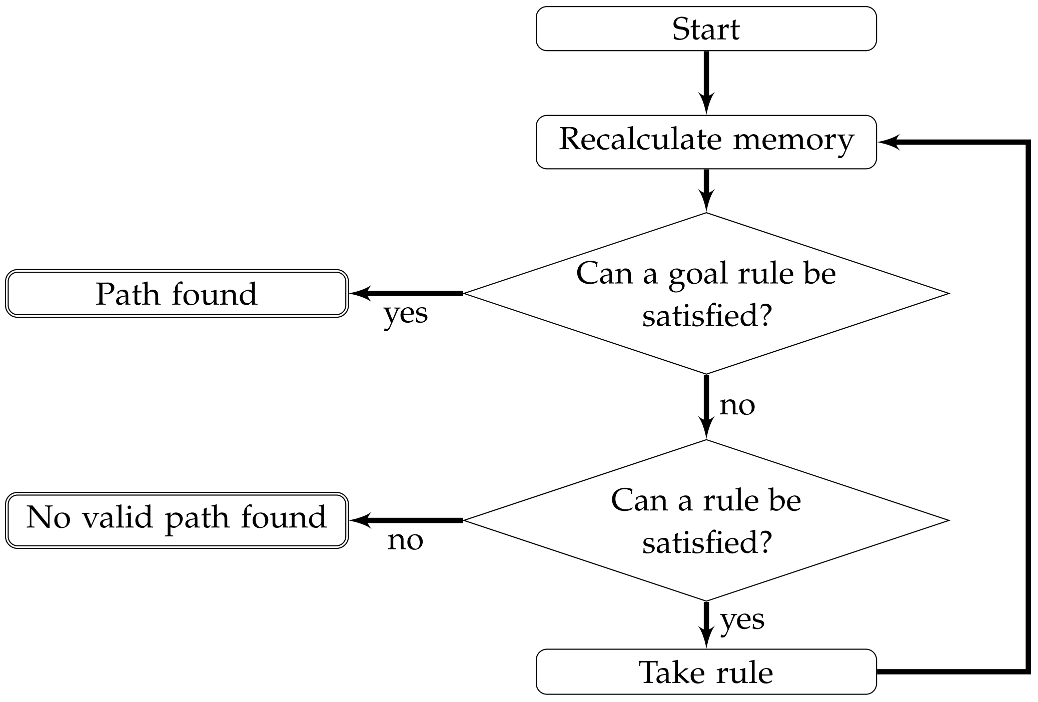

Figure 4.5.: Top Down Path Search flow chart

4.2.4. Algorithm Comparison

To compare the three algorithms, we executed random tests with different configurations. Because of the randomness, there are also a few rules that do not have any valid plan at all. All tests were run on a laptop with an Intel Core i5-7200 with a clock rate of 2.5GHz and 8GB of RAM.

No surprise is that the Optimal Path Search Algorithm takes the longest, as shown in Figure 4.6. Also, an interesting observation is that the Top Down Path Search performs roughly the same on Average (Figure 4.6) and in successful cases (Figure 4.8) as the Bottom Up Path Search Algorithm, even though he has a higher success rate as can be seen in Figure 4.7.

The success rate for the Top Down approach is even as good as the Optimal Plan Search for small rule sets. The drop of successes for the Optimal Plan search can be explained because the Optimal Plan Search uses much memory, and with 11 and 15 rules, this is already too much for our laptop.

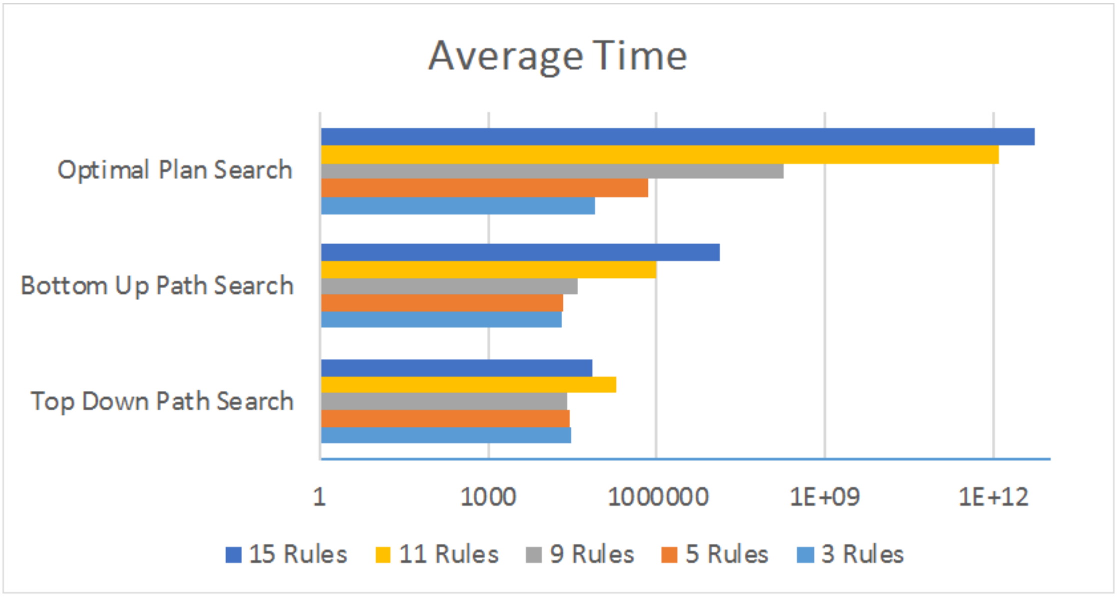

Figure 4.6.: Comparison average time

Another interesting finding is that all three algorithms perform roughly the same except for 11 and 15 rules, as shown in Figure 4.9.

For the findings given above, we conclude that the Top Down approach is not only faster but also gives more consistent and better results. The only downside to the Top Down approach is, that it performs the worst in terms of the weight, as can be seen in Figure 4.10.

Readers should note, that as previously stated, we did our test on randomly generated rules. Which algorithm is the best depends heavily on the rules and the application the user has in mind. If the user needs the optimal and correct solution and has not many rules, but can deal with a longer computing time, we would recommend the Optimal Plan Search Algorithm. For a faster approach that can also deal with deletions, but has a not so optimal plan, we would recommend the Top Down Path Search. When the user wants to have a balance between optimization and computing time and does not use deletions and the rule pattern described in Section 4.3.2, we would recommend the Top Down Path Search.

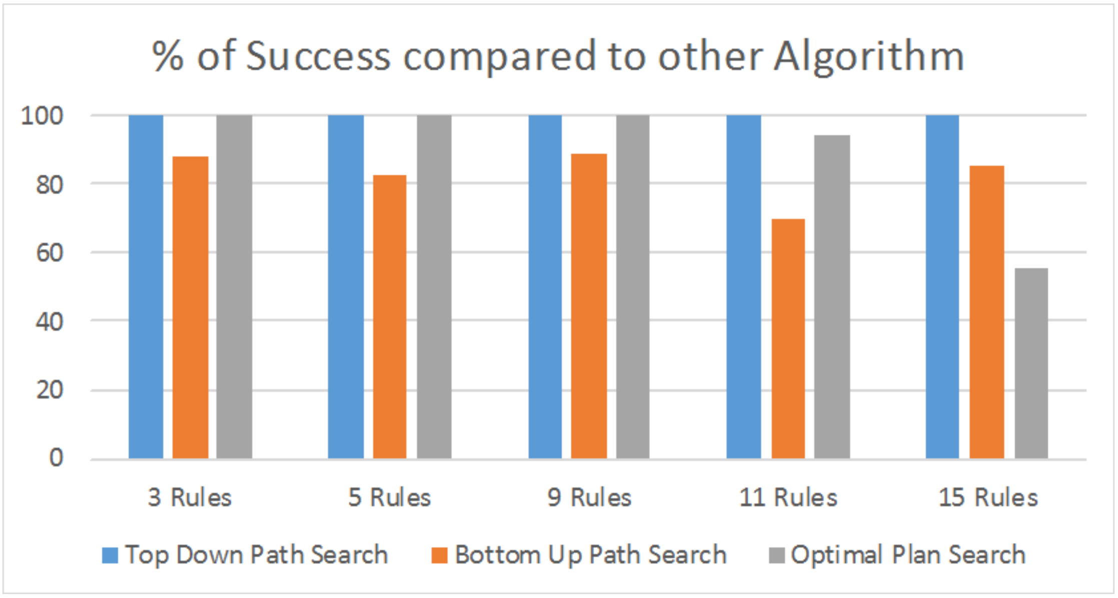

Figure 4.7.: Comparison successes

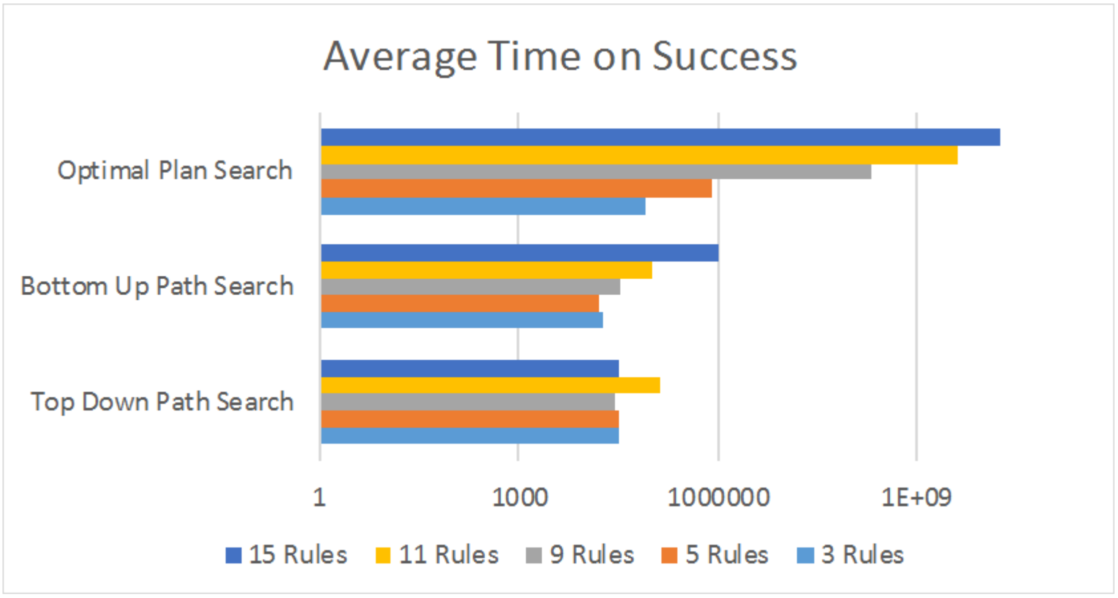

Figure 4.8.: Comparison average time for successes

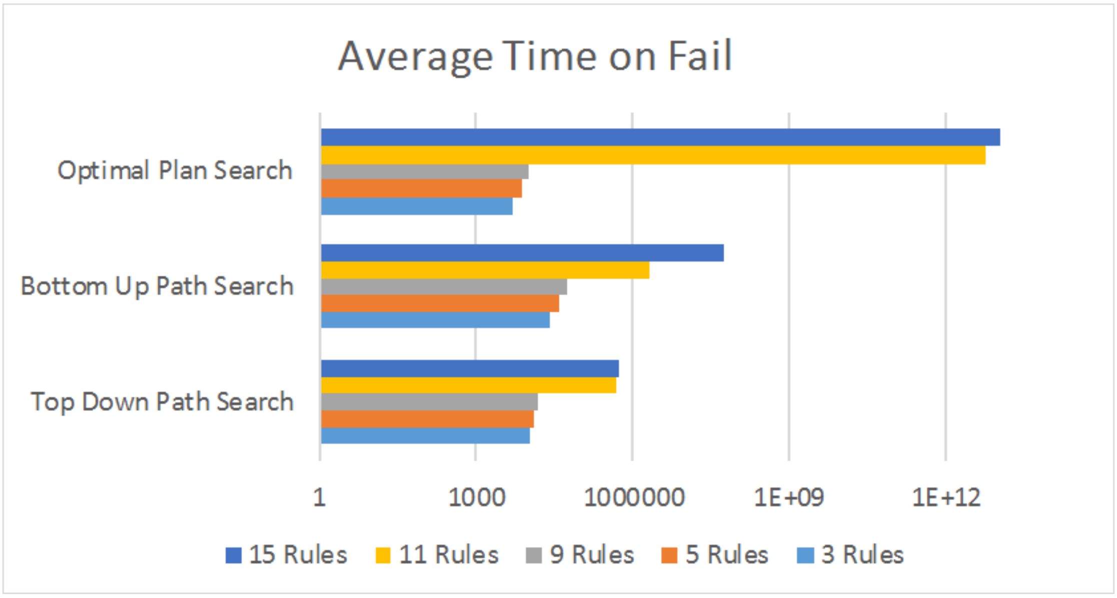

Figure 4.9.: Comparison average time for fails

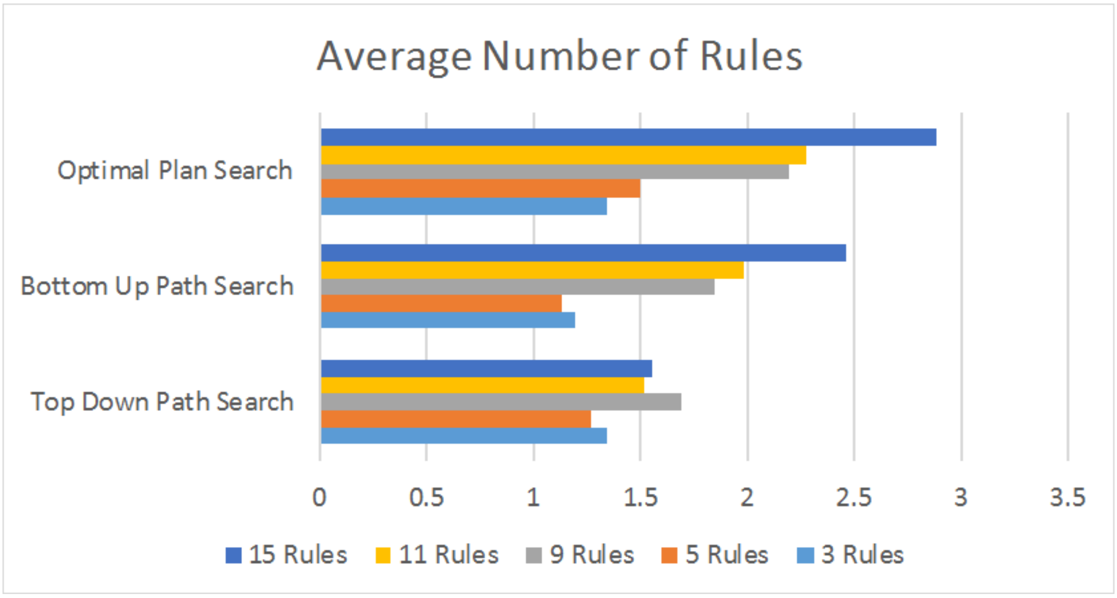

Figure 4.10.: Comparison average numbers of rules (higher is better)

$$activity_n = \frac{1}{2} \cdot chosen_n + \frac{activity_{n-1}}{2} = \sum_{k=0}^{n-1} chosen_{n-k} \cdot \frac{1}{2^{k+1}}$$

Figure 4.11.: Activity Factor from [8]

4.3. Encountered Problems

As with every implementation, we also had some problems while implementing the compiler. Some things looked very promising in theory, but through testing, we found some errors in our assumptions. First, we will discuss the floating-point precision problem, and propose our solution to that. Then we will show that the algorithm proposed in [8] leads to wrong results in special circumstances. Finally, we will discuss the starvation of rules that can lead to a significant performance decrease.

4.3.1. Floating Point Precision

From the definition of the activity factor given in Figure 4.11 we can see that after some iteration we already have very small numbers. For that reason the `float` and `double` Java data type were not sufficient to hold the numbers. Java has the `BigDecimal` class, which provides an arbitrary-precision signed decimal number.

In all instances where using a `double` could lead to unintended behavior in our program, a `BigDecimal` is used instead.

The downside to using `BigDecimal` is a runtime overhead compared to `double`. Also, the code is a bit more messy, because Java has no operator overloading and therefore no infix notation is supported for `BigDecimal` in Java.

4.3.2. Violation of Plan Definition of the Bottom Up Algorithm

In [8] Krenn defines a plan as follows:

```
a -> +b action.
b -> +a action.
a, b -> #goal action.
```

Listing 4.2: Example that gives a wrong path with the Bottom Up Algorithm

$$n \in \mathcal{R}, \mathcal{M} \in \Sigma \cdot \frac{\begin{array}{l} n.\mathit{Test} \subseteq \mathcal{M}, \\ n.\mathit{EffectPos} \in \mathit{ToReach}\left(\langle r_1, \ldots, r_n \rangle\right) \setminus \mathcal{M}, \\ \left(\forall r_i \cdot n.\mathit{EffectNeg} \cap \left(r_i.\mathit{Test} \cup r_i.\mathit{Reach}\right) = \emptyset\right) \end{array}}{\left(\mathcal{M}, \langle r_1, \ldots, r_n \rangle\right) \rightarrow \left(\mathcal{M}, \langle n, r_1, \ldots, r_n \rangle\right)} \text{RCR}$$

Figure 4.12.: Rule Choose Rule from [8]

Definition 1 *A plan is a reasonable path RPath that does not contain any rule (not goals) that may be removed and the path is still reasonable given $\mathcal{M}$.*

Further, a reasonable path is given as:

Definition 2 *Hence, a reasonable path is a sequence of rules that all have a guard that is true as soon as the rule gets interpreted and that leads to the fulfillment of at least one goal.*

That means that each reasonable path should be interpretable from the first rule to the goal if all actions also evaluate to true.

Also important to note is that Krenn combined the two preconditions Test and Reach to one called Guard.

Now let us look at Listing 4.2 (where x -> is the precondition "x" given as *Reach*):

Because we have no predefined memory, we neither can execute the first, second, or third rule in this example, because all of them require a precondition.

Now the conditions to add a rule to the plan, which we can see in Figure 4.12, are as follows:

$$ToReach(t) =_{def} \bigcup_{r_i \in t} r_i.Reach \setminus \bigcup_{r_i \in t} \{r_i.EffectPos\}$$

Figure 4.13.: ToReach from [8]

Definition 3 *Rule RCR (Rule Choose Rule) lists some of the conditions that apply when looking to extend a plan: (1) the test propositions of the new rule must evaluate to true, (2) the new rule must not remove a proposition that a rule within the partial plan checks for, (3) the new rule must provide a proposition (EffectPos) that is needed by some other rule within the partial plan.*

Where *ToReach(t)* is given as shown in Figure 4.13. In addition, the condition for a finished plan is as stated:

Definition 4 *"Plan is complete when there is no dangling reach-proposition, as indicated by the empty reach-set."*

From Definition 3 of the RCR, we can first add rule 3 and then add rule 1 and rule 2 to our plan, because both satisfy the criterion given for the RCR. Furthermore, with those two rules in our plan, rule 3 is satisfied, and those the plan is deemed complete as given by Definition 4.

Therefore from the algorithm, as given in [8], both plans 1, 2, 3 and 2, 1, 3 would be a valid plan according to the algorithm. However, these plans clearly violet the definition of a reasonable path from Definition 2 and in succession, the definition of a plan given in Definition 1.

If this problem is given to the SEPIAS Runtime that is also presented in [8], the SEPIAS runtime runs into an infinite loop, which can be tested with the Prolog code in Listing 4.3:

4.3.3. Starvation of Rules

With the algorithm proposed in Krenn's Ph.D. thesis [8] it is very easy to get in a state where there are three rules and one rule is never chosen again, although

```
interpretRuleSetOnce([
    rule(ruleOne, 1, 0.5, 1, [1,0,1], 1, 0.5, [],
        [a], action, (b), [], 0 ),
    rule(ruleTwo, 1, 0.5, 1, [1,0,1], 1, 0.5, [],
        [b], action, (a), [], 0 ),
    rule(ruleThree, 1, 0.5, 1, [1,0,1], 1, 0.5, [],
        [a, b], action, (0,goal), [], 0 )],
    [], Plan, FinalMem, FinalRules).
```

Listing 4.3: Example that leads to an infinite loop in the SEPIAS runtime

it would be the best rule. In this section, we will first explain the problem in more detail and then present our solution to the problem.

4.3.3.1. Problem Description

The problem arises if the first rule generally performs better than the other two, but, because of external factors, the first rule has a temporary fault. After the first rule recovers from the fault, the first rule is never chosen again.

Two factors contribute to this problem.

1. As stated in Chapter 3.1.4.2 damping only is updated when the rule actually gets executed.
2. We can see from the Equation 3.1 that even though the activity is very small (e.g., 0) the weight can never be higher than $(1 - Damping)$.

In Figure 4.14 we can see how the weight will develop for a rule with damping of 0.9 over time if it is not selected. On the other hand, in Figure 4.14 we can see that a rule that has damping of 0.1 and is chosen every second time will oscillate between $1/3$ and $2/3$.

From this, we can see that no matter how small the activity of the first rule gets it will never reach a point where it will be considered the best rule again because the maximum weight it can get is 0.1.

This problem is further increased by the linear addition and subtraction of the damping factor. Meaning even though the second and third rule would only be successfully 50% of the time there is a big chance that the second or third

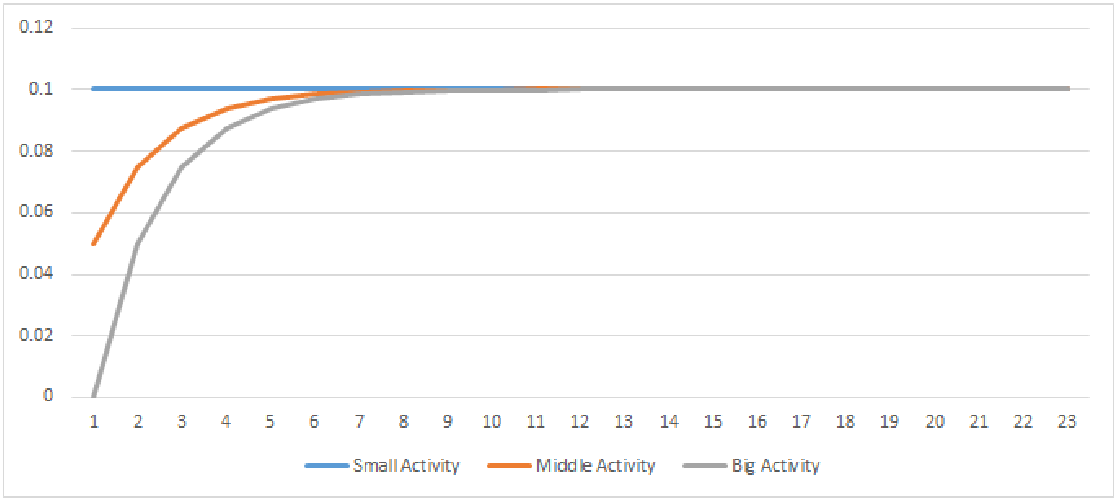

Figure 4.14.: Convergence of the weight of a rule with a damping factor of 0.9

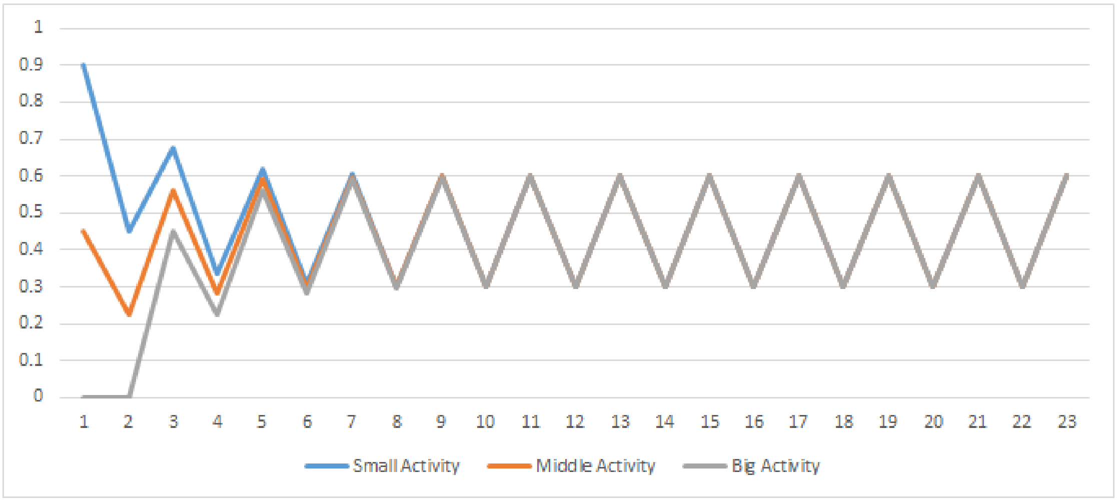

Figure 4.15.: Convergence of the weight of a rule with a damping factor of 0.1 and selected every second time

rule will never hit a damping factor small enough that the first rule will be considered better again.

Because of this problem, it could be that a rule that usually performs better (e.g., 100%), but has a total fault for a short time, will, after recovery, not be chosen over rules that perform worse (50%). This behavior can lead to a significant performance decrease of the system over time.

4.3.3.2. Proposed Solution

To mitigate this behavior, we added the aging of the damping factor to the model as described in Section 3.1.1. We designed the aging to be very flexible because, in our tests, we found out that there is no perfect approach that can fit all use cases. Overall, the aging parameters are just a tool for the developer to counter this flaw; however, in certain scenarios, we could increase the performance of the system by almost 10% by including aging.

5. Usage and Examples

In this chapter, we will describe how the interpreter, compiler, and RBL can be used. We will discuss the different modes of operation and all the options of the compiler in the first section. In the second section, we will have a look at some examples that showcase the power of RBL and the compiler.

5.1. Program

For using the compiler, there are two main modes. The interpreter and the compiler. Following man-page-style usage describes the usage of the program. In the first subsection, we will further describe the interpreter mode, and in the second subsection, we will describe the compiler mode.

```
rule interpret [-times n] PATHTOJAVAFILES PATHTORULEFILE
rule compile [-o OUTPUTPATH] [-p PACKAGENAME] [-d]
                             [-lib] PATHTORULEFILE
rule (-h | --help)
```

5.1.1. Interpreter

The interpret option interprets the rules given in the PATHTORULEFILE using the Java classes located in PATHTOJAVAFILES n times. If no –times parameter is given, the rule set will be interpreted only once. For the interpretation, the JDK must be used because the Java classes are compiled on the fly.

5.1.2. Compiler

The compile option compiles the RBL source file into Java code. The –d enables dynamic class creation, which means that the classes that are specified in the rule file are created at runtime of the resulting Java files. In this mode, the developer has to make sure that the classes are available in the classpath of the class loader before the *Executor* can be called. Without the –d option the classes will be hard-coded into the resulting Java files, this enables better compiler support but can lead to false positives. The –lib option only generates the Executor.java, which contains all the definitions for the rules. If this mode is selected the rule.jar file has to be included in the build path. With this option, there is less clutter in the project, but an additional dependency on the .jar file is needed.

5.2. Recommended Workflow

Seamless integration into a build workflow is a major concern these days. Especially with the rise of build systems and continues integration, integration into a workflow is necessary. Therefore, we also made sure that our compiler is easy to integrate into a build system. We would recommend Gradle as a build system, but the proposed parameters also work with other build systems. Our recommended *build.gradle* is given in Listing 5.1. Where the *rule.jar* has to be in the root path of the project.

```
group 'example'
version '1.0-SNAPSHOT'
apply plugin: 'java'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit',
                version: '4.12'
    compile files('rule.jar')
}

jar {
    manifest {
        attributes('Main-Class': 'example.Test')
    }
}

ext.rule = [
        packagename: "rule",
        ruleSource: './src/main/example.rule',
        destinationDir: "./src/main/java/"
]

task(runRuleCompiler){
    javaexec {
        main="-jar";
        args = [
                "rule.jar", "compile",
                "-o", rule.destinationDir,
                "-p", rule.packagename,
                "-lib", rule.ruleSource
        ]
    }
}

compileJava{
    dependsOn runRuleCompiler
}
```

Listing 5.1: Recommended *build.gradle* file

With this build script, the compiler is executed before every build, so all changes made to the RBL source file will be synchronized immediately with the actual Java code.

The *compile* flag tells the compiler that it should compile the given RBL source file, given as the variable *rule.ruleSource*. The *-o* parameter, together with *rule.destinationDir* specifies the destination. This makes it easy to integrate the generated files into a package hierarchy.

With the *-p* switch and the *rule.packagename* the package name of the generated file can be given and should adhere to the Java specification based on the folder hierarchy.

Lastly, the *-lib* options only generates the minimum amount needed and uses all dependent classes from *rule.jar* instead of generating them. This dramatically reduces the clutter in the project.

We would also recommend against using the *-d* option in this scenario because all the classes that are required for the rule set to run should be implemented in the project. Introducing the dynamic compile option would only cause the compiler to have a weak type-checking against the actions and brings no benefits here.

To note is that because this configuration builds the rules again at every compilation, it has an impact on the compile time. For small sets of rules, it should not be noticeable, but for larger rule sets, developers may choose only to recompile the rules if they changed.

5.3. Examples

To demonstrate the capabilities of RBL and provide a better understanding of its use cases, we will provide two examples in this section. The first example will be about a robot that has to leave a room. The second will be the model of a weather station that collects data and transmits it over WiFi or GSM.

5.3.1. Robot leaving a Room

```
window_open.
door_closed.

window_closed -> +window_open -window_closed openWindow
                (0 <= a < 0.5: 1, 0.5 <= a <= 1: 0.5) 0.2.
door_closed -> +door_open -door_closed openDoor 0.2.

door_open -> #exit goThroughDoor (0 <= a <= 1: a/2).
```

```
window_open -> #exit goThroughWindow 0.2.
```

Listing 5.2: Robot example RBL source file

This example demonstrates how a robot could be modeled to leave a room. The last two rules specify the desired goal *#exit*. To achieve that the robot can either go through the open window or through the door, which the robot must first open. The problem is that the *goThroughWindow* rule has a much lower frequency than the *goThroughDoor* method. Therefore, the robot will first try to open the door and then go through it. Only when this fails, the robot will consider going through the window.

5.3.2. Weather Station

```
-> +temperature_collected collectTemperature.
-> +humidity_collected collectHumidity.
temperature_collected, humidity_collected ->
    +data_prepared -temperature_collected
    -hummidity_collected prepareData.

-> +connected connectGSM 0.2.
-> +connected connectWifi.
connected, data_prepared -> #send -data_prepared sendData.
```

Listing 5.3: Weather station example RBL source file

In this example, we see the model of a weather station. First, the station needs to collect the data, then prepare it, and then tries to send it. The interesting part here is, that only if the sending fails, the *connect* predicate will be removed by a low-level repair routine. Therefore, the device does not connect to GSM or Wifi again as long as there is no error while sending.

6. Case Study

To see if our compiler works reliable and to assess the real-world usefulness of RBL, we developed a case study. The case study found in this thesis is a extension of the case study presented in [15]. The case study proposed in [15] allowed us to have a first glance at the usability of RBL and enabled us to find a few errors, which ultimately lead to the inclusion of aging in RBL.

6.1. Scenarios

For this case study, we developed a small simulation that simulates sensors for object detection in an autonomous car in different states. This is modeled by having a matrix, where the columns define the state and the rows define the different sensors. Each cell then contains the probability that the object detection is successful with a given sensor in a given state. An example of this would be the first three rows of Table 6.1.

Each simulation has a sequence of states, further called steps, which are run through when the simulation is executed. Such a sequence could look like 1000 steps clear sky, then 1000 steps rain, 1000 steps storm and 1000 steps clear sky again.

For benchmarking, we compare our system against two different metrics. The first metric is the success rate if we would choose a random sensor (out of the three) at each step, further called Random select. The second one is the success rate if we would choose the sensor with the highest probability at each step, meaning choosing the perfect sensor for the given situation, basically predicting the future, further called Max success. An example of this can be seen in Table 6.1.

Success rate	Clear sky	Rain	Storm
Camera	100%	65%	45%
LiDAR	80%	70%	60%
RADAR	70%	70%	70%
Random select	83.33%	68.33%	58.33%
Max success	100%	70%	70%

Table 6.1.: Example success rate of the sensors in different states

6.1.1. Weather Scenario

Our first scenario is a typical weather scenario where we first have 1000 steps of clear sky, then 1000 steps of light rain, followed by 1000 steps of heavy rain and then 1000 steps of clear sky again. This is also illustrated in Figure 6.1.

The success rate of the different states is given in Table 6.2 along with the success rate for using only a single sensor (last column, first three rows) and the Random Select as well as Max Success's success rate. Figure 6.2 shows this also graphically. We can see that the camera is pretty good when there is clear sky, but the rate with which it can detect objects gets worse quickly if there is rain or a storm. LiDAR, on the other hand, does not have such a good detection rate at clear sky but also does not get so much worse with more rain. RADAR, however, does not have a good base rate. However, it is not affected by rain or storm at all.

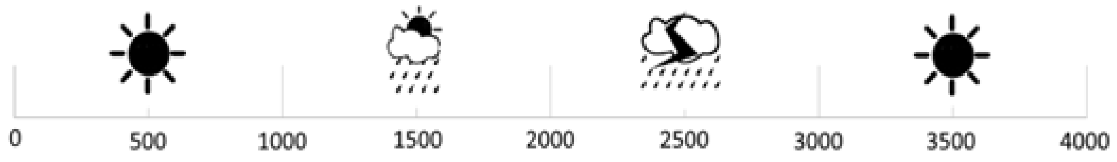

Figure 6.1.: States of the Weather Scenario

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

Table 6.2.: Success rate of the sensors for the Weather Scenario

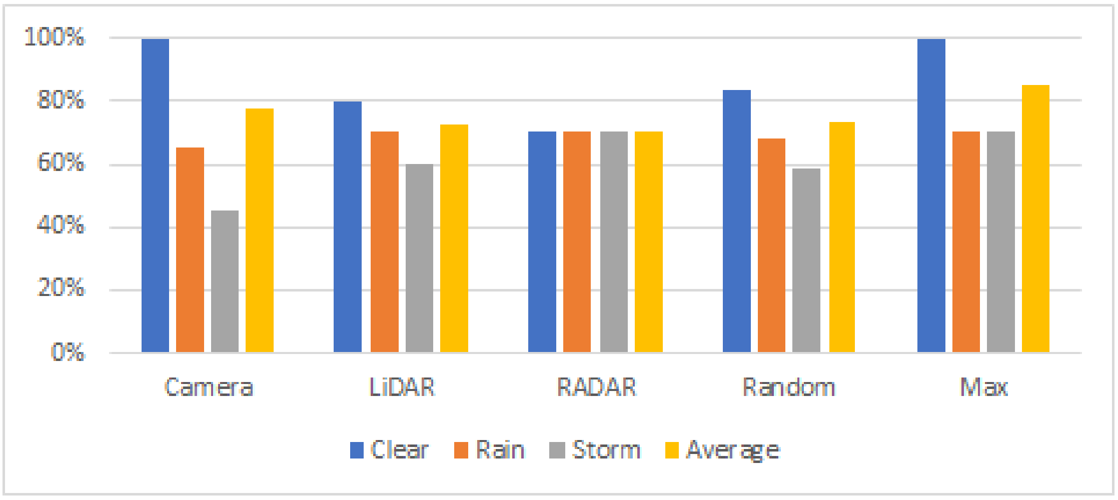

Figure 6.2.: Graphical representation of the success rate of the Weather Scenario

6.1.2. Temporary Fault Scenario

The second scenario we looked into was a temporary fault scenario, meaning that normally all sensors run with 100% success rate, but at some point, the sensor would stop working completely for a given amount of steps. In Figure 6.3 this scenario is illustrated. For the first 1000 steps, all sensors work as expected after this only the camera has a fault for 1000 steps, then the camera and LiDAR has a fault for 1000 steps and for the last 1000 steps all sensors work again. This is modeled by the success rates given in Table 6.3.

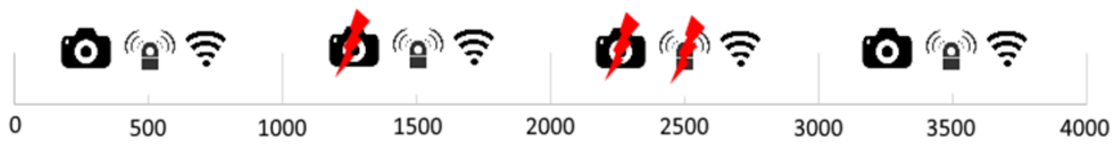

Figure 6.3.: States of the Temporary Fault Scenario

Success rate	C/L/R	L/R	R	C/L/R	Average
Camera	100%	0%	0%	100%	50%
LiDAR	100%	100%	0%	100%	75%
RADAR	100%	100%	100%	100%	100%
Random select	100%	66.66%	33.33%	100%	75%
Max success	100%	100%	100%	100%	100%

Table 6.3.: Success rate of the sensors for the Temporary Fault Scenario

6.2. Results

To get meaningful results, we compared the number of successful object detections for each scenario with five different methods. The first two are Random Select and Max Success, which were introduced in the previous section. Then, we have three different methods for which we run the program. Because of the random nature of our program, it was hard to compare single runs because often there was no significant difference or even complete opposite results than we expected. Therefore, we ran each method 3000 times and took the average of all these runs. The three methods are Normal, which is just the program without any parameters set for the aging, Aging to 0.5, which has for every detect rule the aging value set to 0.01 and the aging target is 0.5, and Generated, which has aging values that were calculated by a genetic algorithm, by training on the simulation. Because this was only trained on one simulation natural overfitting is the result, and the results given in this theses for Generated should be taken with cautiousness because on other simulation this values may perform worse than other methods. All programs for the different methods can be found in Appendix A.

6.2.1. Weather Scenario

The results of the weather scenario surprised us. As we can see in Table 6.4 the improvement from the Random selection to the Normal execution were only 1,6% this poor performance was the reason we double-checked the programs and ultimately led to the inclusion of the aging parameters introduced in Chapter 3.1.4.3. The reason for this poor performance is further explained in Chapter 4.3.3. Besides, Aging of 0.5 did not perform so well, but we can see that there is a little increase. This is mostly also due to wrong parameter selection. Finally, the Generated method is doing great. Here we have an increase of 6.3%. In Figure 6.4 we can see that this is more than 50% of the increase we could achieve from the Random select to the Max success. Here we would like to remind the reader that Max success is the maximum success rate we could achieve if we could predict the future. Therefore, it is unrealistic for any general system to achieve this value.

Method	Success rate
Random select	73.33%
Normal	74.93%
Aging to 0.5	75.01%
Generated	79.61%
Max success	85.00%

Table 6.4.: Success rate of the different methods for the Weather Scenario

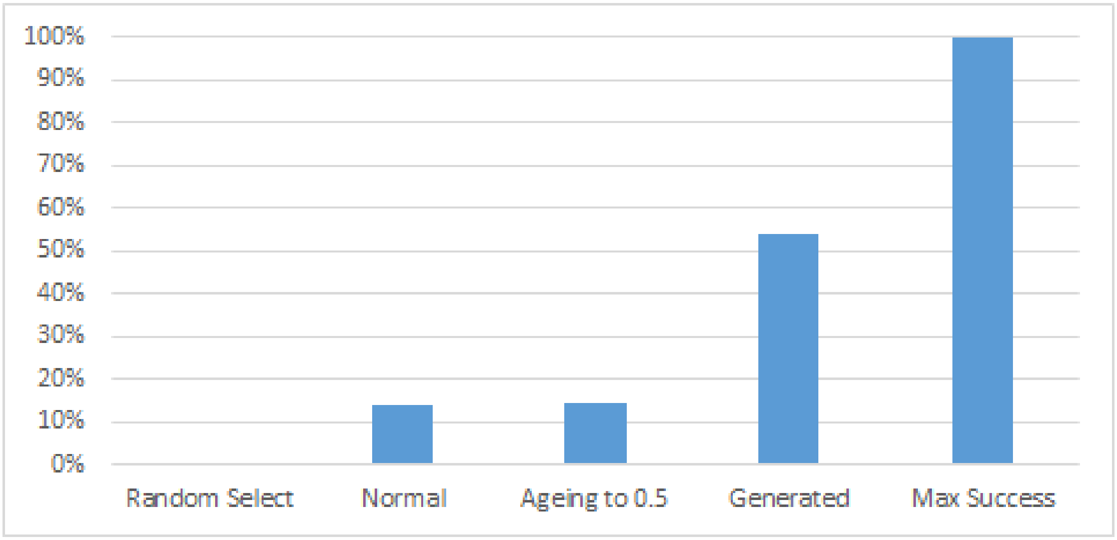

Figure 6.4.: Percentage of the success rate increase from Random select (0%) to Max success (100%) for the Weather Scenario

6.2.2. Temporary Fault Scenario

Not as surprising as the Weather Scenario was the Temporary Fault Scenario. Nevertheless, with this example, we can see the strength of the programming language. This scenario does not suffer from the rule starvation as described in Chapter 4.3.3, we can easily see this when we look at Table 6.5. Here we can see that the Normal and the Aging to 0.5 methods are roughly the same. In Figure 6.5 we can further see that even the Normal method is already excellent. Here we get an increase of 18.59% from 75.00% to 93.59% from the Random select to the Normal method. This is a bit over 70% of the maximum increase we could get.

On the other hand, some readers might surprise the result from the Generated method because it has a 100% success rate. If we remember that these values are specifically generated for this scenario and that there was a sensor that is working 100% of the time (RADAR) it is no surprise that the algorithm chooses values that always use RADAR, and, therefore, have a 100% success rate.

Method	Success rate
Random select	75.00%
Normal	93.59%
Aging to 0.5	93.59%
Generated	100.00%
Max success	100.00%

Table 6.5.: Success rate of the different methods for the Weather Scenario

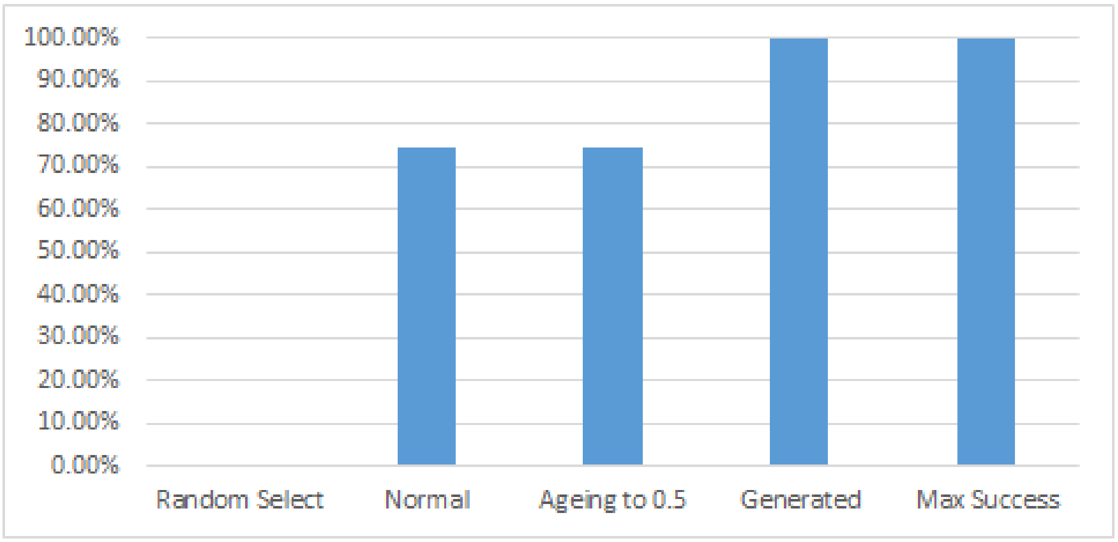

Figure 6.5.: Percentage of the success rate increase from Random select (0%) to Max success (100%) for the Temporary Fault Scenario

7. Summary

Combining artificial intelligence, logic-based languages, and compiler construction was a fascinating topic. In the end, we think we got a good prototype for how rule-based languages can look in the future. Even though finding the globally optimal path is NP-complete, we can make good approximations as the benchmarks showed.

We think we will see more fault-tolerant systems in the future. Our devices get more and more capable and more often than not, they have redundant execution paths. Having a dedicated language for implementing fault-tolerant behavior is helpful for sure.

7.1. Outcome

Our compiler and interpreter for RBL was completed while writing this thesis. They have all the capabilities described in this thesis. An extensive test suite and manual testing ensured that it should be free of the most severe errors. In addition, we think this is a good starting point for future work.

We also showed in chapter 6 that with RBL, we can increase the reliability of redundant systems.

7.2. Future Work

To use RBL in production environments, a mechanism for code reuse should be implemented; right now, no such mechanism exists. Also, debugging capability would greatly increase the ease of use of RBL.

Another extension point would be to implement other algorithms to find plans, or dynamically switching between algorithms.

Appendix

Appendix A.

Case Study Programs

A.1. Normal

```
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.
```

A.2. Aging to 0.5

```
lidar_ok.
radar_ok.
camera_ok.

camera_ok -> +objectDetected
             actions.detectObjectWithCamera
             [,0.01,0.5].
lidar_ok  -> +objectDetected
             actions.detectObjectWithLiDAR
             [,0.01,0.5].
radar_ok  -> +objectDetected
             actions.detectObjectWithRADAR
             [,0.01,0.5].

objectDetected -> #objectProcessed
                  -objectDetected
                  actions.processObject.
```

A.3. Generated for Weather Scenario

```
lidar_ok.
radar_ok.
camera_ok.

camera_ok -> +objectDetected
             actions.detectObjectWithCamera
             [0.20596888723176832, 0,
              0.041475146318857502].
lidar_ok  -> +objectDetected
             actions.detectObjectWithLiDAR
             [0.6833358431503067, 1,
              0.881142409368942].
radar_ok  -> +objectDetected
             actions.detectObjectWithRADAR
             [0.1557141357473 5216,
              0.8708851154862152,
              0.4026986546719830 7].

objectDetected -> #objectProcessed
                  -objectDetected
                  actions.processObject.
```

A.4. Generated for Temporary Fault Scenario

```
lidar_ok.
radar_ok.
camera_ok.

camera_ok -> +objectDetected
             actions.detectObjectWithCamera
             [1,
              0.21929041003377692,
              0.9180425416496725].
lidar_ok  -> +objectDetected
             actions.detectObjectWithLiDAR
             [1,
              0.9375198282694639,
              0.9774431420796214].
radar_ok  -> +objectDetected
             actions.detectObjectWithRADAR
             [0.08879874804379395,
              0.68037852833377403,
              0.337074236021326 47].

objectDetected -> #objectProcessed
                  -objectDetected
                  actions.processObject.
```

Bibliography

[1] A. Avizienis, "Fault-tolerant systems," *IEEE Trans. Computers*, vol. 25, no. 12, pp. 1304–1312, 1976. DOI: `10.1109/TC.1976.1674598`. [Online]. Available: `https://doi.org/10.1109/TC.1976.1674598` (cit. on p. 4).

[2] W. C. Carter and W. G. Bouricius, "A survey of fault tolerant computer architecture and its evaluation," *Computer*, vol. 4, no. 1, pp. 9–16, Jan. 1971, ISSN: 0018-9162. DOI: `10.1109/C-M.1971.216739` (cit. on p. 4).

[3] P. McCorduck, *Machines Who Think: A Personal Inquiry into the History and Prospects of Artificial Intelligence*. AK Peters Ltd, 2004, ISBN: 1568812051 (cit. on p. 5).

[4] B. Goertzel and C. Pennachin, *Artificial General Intelligence (Cognitive Technologies)*. Berlin, Heidelberg: Springer-Verlag, 2007, ISBN: 354023733X (cit. on p. 5).

[5] R. Lippmann, "An introduction to computing with neural nets," *IEEE ASSP Magazine*, vol. 4, no. 2, pp. 4–22, Apr. 1987, ISSN: 0740-7467. DOI: `10.1109/MASSP.1987.1165576` (cit. on p. 5).

[6] K. Hinsen, "Domain-specific languages in scientific computing," *Computing in Science Engineering*, vol. 20, no. 1, pp. 88–92, Jan. 2018, ISSN: 1521-9615. DOI: `10.1109/MCSE.2018.011111130` (cit. on p. 6).

[7] M. Ahmad, "First step towards a domain specific language for self-adaptive systems," in *2010 10th Annual International Conference on New Technologies of Distributed Systems (NOTERE)*, May 2010, pp. 285–290. DOI: `10.1109/NOTERE.2010.5536629` (cit. on p. 6).

[8] W. K. Krenn, "Self reasoning in resource-constrained autonomous systems," dissertation, Graz University of Technology, 2009 (cit. on pp. 7, 10, 29, 30, 36–38).

[9] R. Paige, J. Ostroff, and P. Brooke, “Principles for modeling language design,” vol. 42, pp. 665–675, Jul. 2000 (cit. on pp. 7, 20).

[10] D. Spinellis, “Notable design patterns for domain-specific languages,” *J. Syst. Softw.*, vol. 56, no. 1, pp. 91–99, Feb. 2001, ISSN: 0164-1212. DOI: `10.1016/S0164-1212(00)00089-3`. [Online]. Available: `https://doi.org/10.1016/S0164-1212(00)00089-3` (cit. on p. 14).

[11] F. Wotawa, *The logic reasoning system – a brief introduction*, 2008 (cit. on p. 15).

[12] C. A. R. Hoare, “Hints on programming language design.,” Stanford, CA, USA, Tech. Rep., 1973 (cit. on p. 20).

[13] J. N. Buxton and B. Randell, Eds., *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970 (cit. on p. 23).

[14] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd. Pragmatic Bookshelf, 2013, ISBN: 978-1-93435-699-9 (cit. on p. 27).

[15] F. Wotawa and M. Zimmermann, “Adaptive system for autonomous driving,” in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Jul. 2018, pp. 519–525. DOI: `10.1109/QRS-C.2018.00093` (cit. on p. 47).