Recommending Values for Parameter Sets in Simulation Applications

Diplomarbeit von Dirk Bächle

Technische Universität Hamburg-Harburg
Softwaresysteme
Prof. Dr. Ralf Möller

14. Januar 2005

Contents

1 Introduction

Over the last 20 years, simulation applications of various kinds have gained their respected places in the areas of science and economy. In physics—and many other fields like genetics or electronics—, engineers design and model experiments on the computer before the real tests are carried out. Operation parameters can be optimized at low cost and arising problems may be detected in an early stage. The ability to model and test software prototypes has cost-effectively enabled manufacturers to design and build everything from sophisticated aircraft and automobiles to electronic products. Simulating product performance reduces development expenses, time-to-market and warranty costs, positively impacting customer profitability.

Following the advice of DESCARTES, the part or system being investigated is decomposed to smaller pieces most of the time. For example, in the approach widely known as *Finite Element Method* (FEM) the component under duty gets replaced by a set of interconnected elements. Forces, or strains in general, attack at these elements and automatically lead to a mathematical model that can be evaluated by an appropriate equation solver in each incremental step. Optimized for speed and generality instead of comfort, these solvers offer a large number of parameters that have to be adjusted properly. Otherwise, the simulation might stop after a few increments—which may well equal a computation time of several hours. Apart from the inputs for a single step in time, there exists a vast count of variables for controlling the analysis in general. Both together can drive the inexperienced to despair easily and result in a very steep learning curve for the beginner.

Up to now, the existing simulation applications try to remedy this situation by offering a nice *Graphical User Interface* (GUI). This makes entering the parameters a lot easier, but the important relations between certain input fields remain veiled. A lot of time is lost by trying to start a simulation again and again, in order to find a set of input parameters that allow a 'successful' analysis. Here, 'successful' only means that all increments can be processed without interruption and the simulation is carried out to the end. It does not imply that the results are reasonable in any sense, so additional tweaking of variables might be required.

Picking a single parameter like the *element size*, a lot of circumstances have to be regarded. This input variable is a gauge for the average length of node connections and it is quite tempting to try and deduce a value for it from the dimension of the object. A simple formula that delivers a set of approximately 500 polygons—or polyeders for a three-dimensional analysis—per object, the so-called *mesh*, can suffice in not too complex cases. But if the process leads to a fold in the material, the input mesh might be too

coarse and will probably fail. This plain example illustrates that an answer to the question "How small is small enough?" can not easily be derived by a single linear function. It is still the task of the responsible engineer to meet the fine line between creating a model as realistic as possible and spending too much time—or resources in general—on a minute granularity that does not yield better results.
Based on his experiences with former simulation jobs of a similar kind, he will take into account several specific criteria and come up with a reasonable value quite easily. This applied methodology is commonly known under the term 'rule of thumb'. It allows a quick knowledge exchange between two persons by trading *rules* like: "**If** your process has the features A and B, **then** 200 is a good starting value for input parameter C." The bold-faced words outline the resemblance of this *rule* to a well-known computer language construct. Immediately the thought about implementing the expertise of specialists in a piece of software arises and delivers the starting point for the rest of this thesis.
The aim of this work is to develop a special *rule engine*. It can be thought of as a monolithic object that is controlled by the user interface of the application. Upon new user entries, the current data input window employs the *rule engine* in order to validate data ranges and to detect relations between the given parameters. For this, it operates on an internal set of parameters complemented by the so called *rules*, a set of arithmetic expressions.
Once a range violation or the possible application of a 'rule of thumb' has been detected, it can be fetched by the GUI. The input window then may inform the user about the conflict and its causes and ask how he wants to proceed.
For this, a reusable object is developed that consists of several parts. Two special lists manage the parameter set, as well as the rules for describing the relations among them. A simple language—the so called *Rule Expression Language* (REL)—is defined for all evaluation tasks. The stack processor, as another integral part of the rule engine, is able to interpret and compute arithmetic expressions.
Main features of the *rule engine* are:

- The set of rules and the set of parameters can be changed at the runtime of the supervising application.
- The language defining the rules is easy to understand and learn, such that the addition of rules to the system is simple and can be accomplished in a straightforward manner.

- The rule engine does not require an external expert system shell or library but comprises an independent and complete recommender system on its own.
- Documentation for the single parameters and the rules may be stored in the rule engine and can be requested by the application.

The structure of this thesis is as follows: section 2 specifies the problem that is about to be solved. Different common methodologies are discussed and the final approach is selected. Then, section 3 outlines the development and implementation of the single subparts and the rule expression language. A few simple examples for GUI dialogs and their interaction with the rule engine are shown in section 4. Finally, section 5 revises the done work and provides some basic ideas for further improvements.

2 Design considerations

In this section the basic design for the rule engine is laid out. First, a more abstract look at the problem to be solved is taken. From the vast field of *Artificial Intelligence* some of its common practical approaches are picked and get discussed. A simple rule example helps in clarifying to which extent each of these methods can be applied and where difficulties might arise. Then, the final choice is made and a sketch of the recommender system is derived.

2.1 Basic problem

In order to derive a basic structure of the *rule engine* it seems advisable to take a close look at the intended usage scenario and try to get a more abstract view on it.
Some application requires the user to enter various parameters. These input values can be reduced to basic data types like strings, integers or floating-point numbers. They are used to control another object, but not in the sense of real-time supervision. Here, control refers to the basic setup of a simulation that can be compared to a physical experiment. Once everything is setup and the experiment is started, only two possibilities exist: either the experiment finishes up to the end or it stops somewhere in the middle, due to a setup failure.
All parameters have a direct impact on the behaviour of the carried out experiment. Unfortunately, the inexperienced user does not know how to set them in order to get it right in his first try. He is not aware of certain relations that exist between those parameters. Relations that enable an experienced person—someone who has carried out this, or a very similar, experiment/simulation before—to see difficulties arise from the selected input values. Thus, these experts are able to correct mistakes before the actual experiment is started and can help to save a lot of time. This is especially true for the area of FEM simulations, where a single analysis can take days or weeks to finish.
What basically happens is that the expert looks at the current set of parameters. By using some internal knowledge about the values and its relations he devises a new set of parameter values that is probably more successful.
At a very high level of abstraction this setup can be described by the mathematical model of *Constraint Satisfaction Problems* (CSP):

> A *constraint satisfaction problem* is a triple $< V, D, P >$ consisting of
>
> - a set $V = \{v_1, \ldots, v_n\}$ of variables;

- a set of sets $D = \{D_1, \ldots, D_n\}$, such that each D_i is the domain of possible values for the corresponding variable v_i; and
- a set $P = \{P_1, \ldots, P_m\}$ of constraint relations, where each P_j refers to some subset of variables in V, and every v_i in V appears in some P_j in P.

A simple algorithm for solving CSPs named `CSP-Solver` creates all possible combinations $< x_1, \ldots, x_n >$ of values of the variables in V, tests these against each constraint and returns all assignments that passed the check (see [17, p. 82]). Since the search space is very large in general, this procedure has a high complexity in time. Numerous other CSP algorithms try to be more intelligent in picking the assignments that should be tested by using advanced methods like *heuristics*, *logic programming*, *neural networks* or *constraint propagation*.
The rule engine in this work does not have to solve a complete CSP. Its task can be compared to performing the single step of testing an assignment—entered by the user—against the given restrictions. The 'algorithm' for finding the parameter sets to test, is carried out by the user himself, so to speak. If the parameter set contains errors and the check fails, the system does not stop with an error but recommends better values if possible.
So the general layout of the system that is to be built is as follows:

> Some kind of black box is able to store parameters. Upon request, the box checks the made parameter settings and corrects values where needed. Then, the corrected values may be read out again and can be processed further. Apart from recommending values, the system is also able to store default values and warn about range violations.

Section 2.4 gives a short introduction to some common approaches for deriving values in systems that show this or a similar kind of "intelligent behaviour". For a better comparison of the single applicable methods and systems a simple example rule from the area of FEM simulation will be used.

2.2 Example rule

A number of approaches exist to simulate the friction stresses between two bodies that are in contact and move relative to each other. From the discussion by Schafstall in [25, pp. 50], the model of Chen and Kobayashi in [7] is selected now to describe the friction between a workpiece and a die as

tool. Their approach via an *arctan* function, takes into account the influence and relations of the three parameters *tool velocity*, *relative sliding velocity* and *friction coefficient*. A general friction model is implemented in the current version of the simulation program MSC SuperForm, but the user has to set all three values by himself. Since he normally does not know about this formula and its theoretical background, he often enters disadvantageous values.
The example rule tries to offer some advice to the user by recommending a value for the relative sliding velocity r. To make it as simple as possible, the *arctan* function is linearized for smaller tool velocities and a certain range of the friction coefficient. It delivers an approximation for r as follows:

> "**If** the tool velocity v is less than $20\frac{mm}{s}$ and the friction coefficient f greater than 0.3, **then** a reasonable value for the relative sliding velocity r can be approximated by the linear function $r = 5 \cdot 10^{-4} \cdot v$."

which could be translated to a more computer language like syntax as:

```
if ((v < 20) AND (f > 0.3))
then recommend(5e-4*v)
```

This single rule has two input parameters and a recommended output value for the relative sliding velocity r. What makes it somewhat tricky is the fact that only one of the inputs—the tool velocity v—is used to compute the actual output. The friction coefficient f merely decides whether the rule gets applied or not.

2.3 Additional requirement: editable rules

Of course, these kind of simple rules can be easily implemented in a GUI environment by hard-coding them as appropriate functions at the time of program development. After reading the necessary parameters, each check function is called and returns a recommendation value if needed. The GUI could then ask the user how to proceed or fix the error by setting the recommended value automatically.
However, in this work a system is to be built whose behaviour can be adjusted by the user or a developer without having to change source code and recompiling the GUI program. It is an essential requirement that the rules are kept in some form of external database.
This unusual approach offers two main advantages:

1. **Ease of development**: An external set of rules is easier to maintain, test and develop—assuming all the required interface functions between the database and the program exist. Rules can be changed or added at remote sites without the need to install a complete IDE and the source code. An exchange or update of a database is possible by exchanging or overwriting one (or several) files, no recompilation of the application is necessary.

2. **Customization**: A single simulation application is normally used for a broad range of process types by the users. This makes it difficult, if not impossible, to find a single set of recommendation values that can suffice the demands of all customers. Especially larger companies will sooner or later ask for a customized version that is tailored for their needs and sets the parameters to the default values they use in daily practice. The separation of the rule database from the rest of the program, means less effort while meeting the customer's requests. No complete software release has to be created, shipped and installed. A simple exchange of files does the trick.

2.4 Common approaches

2.4.1 Artificial Intelligence

The field of *Artificial Intelligence* (AI) emerged from the three branches of mathematical logic, algorithm theory and computer technic in the fifties. Back then, this new academic discipline was founded by the collaboration of different research groups in order to isolate the key features of intelligent behaviour. Its development until today can be roughly divided into three time sections.
The first of these eras from about 1950 to the middle of the sixties was mainly influenced by research in the two fields of game theory and logical deduction. It could be shown that both problems can be attacked by the same approaches, since both consist of a start state, possible transformations of this state and one or more dedicated end states. Several strategies for the search of the solution in the problem space emerged:

1. *Generate and test*, where solutions are created randomly—driven by heuristics most of the time—and then tested for correctness.

2. *Forward chaining*, where the system tries to find a list of transformations that transfers the start state to a valid end state.

3. *Backward chaining*, where the system tries to find a way from the end state through all possible transformations back to the start state.

While designing systems that can play chess or are able to deduce a given hypothesis from a set of axioms, numerous fundamentals for the later research in AI were developed. Among them the symbolic programming language LISP, introduced by McCarthy in 1958.
The next era up to the middle of 1970 focused on the understanding of natural language. Several systems like SHRDLU successfully interpreted commands from the user that were input as normal sentences. The research was mainly influenced by linguistics and the work of Chomsky and Minsky. Its results had a large impact on todays human-computer interfaces. A second major breakthrough occurred in the area of pattern and object classification as systems were able to identify objects and to categorize them automatically, according to their properties.
The last section is dominated by the development of the so-called expert systems. They represent a new way of intelligence by storing knowledge in a separate database. Instead of using a single conventional algorithm to solve a task, expert systems manage their database separate from the machine or system that operates on the data. This flexible approach makes even large problems track-able. The main methods for storing data in the knowledge base are: rules, logical expressions, frames and semantic nets.
Today, *Artificial Intelligence* is a vast field of interdisciplinary research. It has become clear that the implementation of truly 'intelligent behaviour' is still not at reach within the next 10 or 20 years. Thus, instead of trying to find a holistic approach, current work concentrates on single aspects of intelligence.
Referring to Lunze in [19, pp. 10] a number of different goals can be specified for nowadays AI:

Problem solving and theorem proving Finding the solution to a problem by searching the problem space or reducing the theorem to known basic theorems.

Natural language understanding Transferring naturally spoken language to the computer and understand its semantics.

Image processing and vision Recognizing objects and groups of objects within images.

Learning Automatically acquiring knowledge in order to improve the program/system.

Expert systems Storing and processing of expert knowledge.

Qualitative reasoning Representing and analyzing physical systems by using qualitative descriptions.

Robotics Creating and programming intelligent roboters that can plan and perform tasks on their own.

AI hard- and software Creating specialized hard- and software for the support of AI applications.

Different methods and organizational forms of information processing exist to achieve these goals. They can be split in two main complexes. The first is based on symbolic representations of data and is mainly used in fields like the *Qualitative Reasoning* and *Natural language understanding*.
The second division of methods from which a few are introduced and discussed in the following sections, try to apply numeric calculations. Since the problem of value recommendation does not require any special knowledge about the objects that they characterize, this work concentrates on these practical numeric approaches only.

2.4.2 Artificial neuronal nets

Prerequisites *Artificial neuronal nets* (ANN) try to simulate intelligent behaviour, by mimicking the atomic parts of our brain and their interactions. Single neurons and their information exchange via electrical signals (*stimuli*) are rebuilt by means of software and mathematical descriptions.
Going back to the work of McCulloch and Pitts in [20], a single neuron is represented by a scalar product of the form

$$z = w^T x = (w_0, \ldots, w_{n-1}) \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

(see [3]). Here x is the input vector, w^T contains their weights and z is the so called *inner activity* of the neuron. If a threshold for the latter value is introduced, the resulting equation

$$z = w^T x - T$$

can be interpreted as an $(n-1)$-dimensional hyperplane, dividing the n-dimensional input space. Thus, a single neuron is able to decide whether the input vector x is located in the upper ($z > 0$) or the lower half ($z \leq 0$).

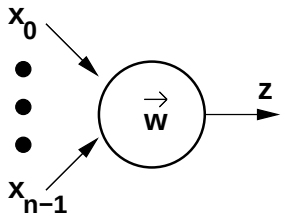

Figure 1: A single neuron

When combining several neurons to a net (fig. 2), they are usually organized in *layers*. The first layer to the left ($j = 0$) gets a set of input vectors x^{k_0}, where $k_0 = 2$ is the number of its neurons. The outputs $z^{k_{j-1}}$ of each layer $j - 1$ provide the inputs x^{k_j} for the successive layer j.

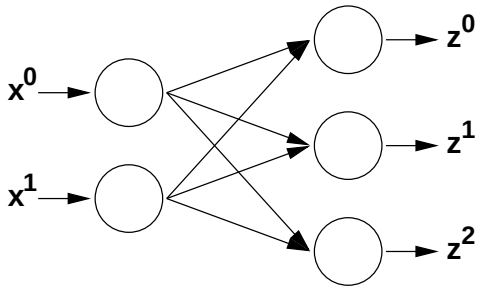

Figure 2: Two connected neuronal layers

This way, even very complex functions can be realized and also the problem of linear separability for the XOR problem may be overcome as NAUCK shows in [22, pp. 46].
Neuronal nets are widely used for pattern recognition and similar tasks, where the relations between the input parameters and the output can not exactly be specified. Instead, the weights of the ANN are adjusted by extensive training and eventually converge to a settled state.
A number of different architectures for ANN are in use today. Very common for pattern classification is the *multi-layer perceptron*, a neuronal net that can be seen as a single processing unit with many inputs but only one output. Other useful architectures are Hopfield networks, Boltzmann machines and feature maps (SOM) that are discussed in detail by NAUCK in [22].

Implementation of the example rule Now, a possible implementation for our example rule from section 2.2 is considered. Figure 3 shows a single

neuron, with the input parameters f and v. It realizes the computation of the new value for r, using the appropriate weights $w = (0, 10^{-4})$. It is assumed that a preceding check made a recommendation necessary.

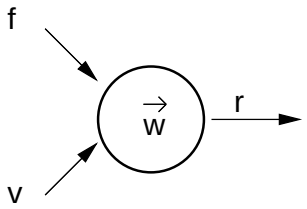

Figure 3: A single neuron computing the new value of r

Here, a first drawback of ANNs gets obvious. It is possible to use a neuronal net for implementing the AND function that decides whether the rule is applied or not. Also the single neuron from above can compute a recommendation for r, but both steps can not be combined into a single neuronal net. This would require to multiply the output t in figure 4 with v again. Unfortunately, the combination of two input parameters in a single neuron is not supported with the standard model.

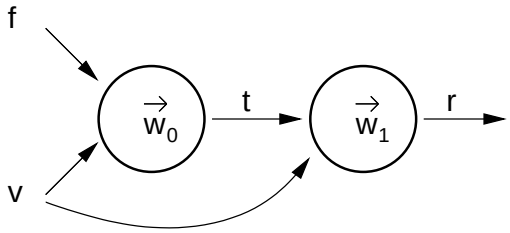

Figure 4: A first drawback: neurons can not combine their inputs

Pros and Cons

+ ANNs are well suited for vague or unknown relations between the single parameters.

+ No expert knowledge for the design of the rules—i.e. the weights—is necessary.

– ANNs show a typical black-box behaviour. They decide based on the input and their internal weights, but there is no way to extract and provide a concrete explanation for the decision process.

– The weights largely depend on the used training examples, so the test samples have to be selected with great care.

– Usually a large training set is needed to achieve an acceptable level of accuracy. In pattern recognition tasks, numbers of more than 2000 examples are used where the first half trains the ANN and the second is used for testing the final system. This is not feasible for simulation applications, where each analysis may need from an hour to several weeks of computation time.

– In simulation applications the problem may change only slightly but then a completely different parameter set is required. This behaviour is difficult to realize with the weight model of the ANNs, which always realizes a scalar product as a continuous function.

Conclusion The various drawbacks show that neuronal nets can be very time-consuming in setting up the single rules and in maintaining them. The expressiveness of an ANN is restricted to a polynomial function of the input variables, which makes the combination of several nets mandatory to implement a single rule that is more complicated than the basic example from above. All together, ANNs might be useful for detecting relations that have been unknown so far, but for a simple implementation of existing rules they do not seem to be a good choice.

2.4.3 Fuzzy logic

Prerequisites Fuzzy logic is a mathematical concept for modeling vague statements and was introduced by ZADEH in 1965 (see [29]). It is based on fuzzy sets and its according operators as an extension to the usual mathematical (sharp) sets.
Fuzzy sets generalize the concept of a characteristic function in Boolean logic. They can be used to represent vague statements like "x is small"—referring to a person's height x—by using a so-called *fuzzy membership function* $\mu_A(x) \in [0, 1]$.
For each input x this function delivers a *possibility value* between 0 and 1.
A value of `0` means that the height is not small at all, while a `1` means that the person is definitely small. The possibility value `0.7` expresses that the height is small with a possibility of `0.7` on a scale `0` to `1`. Here, it is important to not confuse the possibility or membership with a probability measure.
The characteristic functions for the different predicates are often described by building blocks like an up-ramp, a down-ramp (as in figure 5) or a triangle

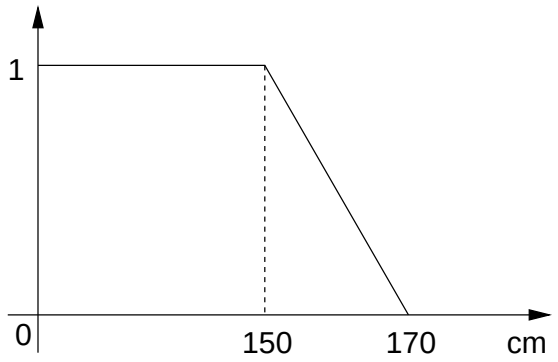

Figure 5: Membership function for "A persons height of x is small"

(Λ function). This greatly simplifies the storing of the single shapes because only some key points are needed. However, general functions are also possible and a large range of applications use B-splines or radial basis functions (RBF) for example (see [10]).
For the usage of a Fuzzy system, e.g. in an adaptive controller system, the following three basic steps are needed (see fig. 6):

1. Fuzzyfication: The output variables of the plant are transformed into fuzzy sets, which each represent a linguistic variable.

2. Inference: Based on a knowledge base that consists of if-then rules, the new plant input is computed.

3. Defuzzyfication: The linguistic output variables are transferred back to a real-valued control signal, that can be passed to the plant.

The inference step relies on an internal set of if-then rules that use the basic fuzzy operators AND and OR. Each rule of the form

```
IF angle == Z AND velocity == Z THEN current == Z
```

is evaluated and then a common fuzzy set for the result is derived by combining all intermediate results with the fuzzy OR operation.
Since the fuzzy AND and OR are a generalization of their boolean counterparts, various functions exist, that can serve as an AND or OR in fuzzy logic. Basically, every function f can be used as a fuzzy AND if it satisfies the following **t-norm** axioms:
A function $f : [0,1]^2 \to [0,1]$ is a **t-norm**, if

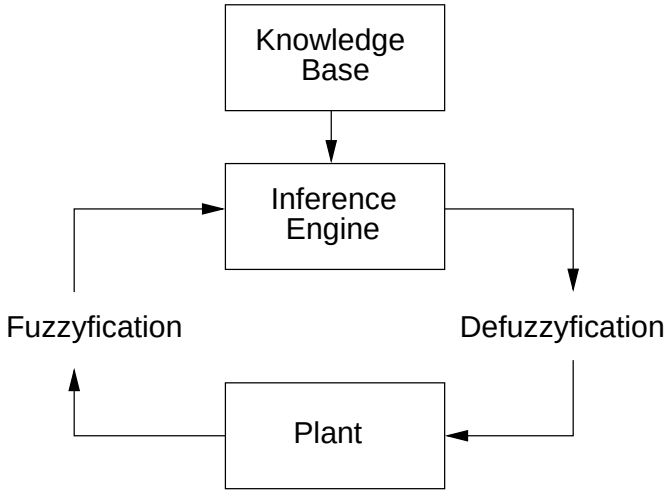

Figure 6: Basic fuzzy logic setup

$$
\begin{aligned}
(i) \quad & f(a,1) = a && (1)\\
(ii) \quad & a \leq b \Rightarrow f(a,c) \leq (b,c) && (2)\\
(iii) \quad & f(a,b) = f(b,a) && (3)\\
(iv) \quad & f(a,f(b,c)) = f(f(a,b),c) && (4)
\end{aligned}
$$

For the fuzzy OR function g, a similar set of **t-conorm** axioms has to hold: A function $g : [0,1]^2 \to [0,1]$ is a **t-conorm**, if

$$
\begin{aligned}
(i) \quad & g(a,0) = a && (5)\\
(ii) \quad & a \leq b \Rightarrow g(a,c) \leq (b,c) && (6)\\
(iii) \quad & g(a,b) = g(b,a) && (7)\\
(iv) \quad & g(a,g(b,c)) = g(g(a,b),c) && (8)\\
& && (9)
\end{aligned}
$$

Each t-norm (fuzzy AND) and t-conorm (fuzzy OR) can be expressed in terms of each other using the logical dualism:

$$f(a,b) = 1 - g(1-a, 1-b)$$

A table containing some important pairs of t-norm and t-conorm was compiled by Zeidler and can be found in [30, p. 22].

Main application areas for fuzzy systems are the fuzzy control (adaptive controllers), sensors and the decision support based on data analysis. Most systems are implemented by means of software but some microprocessors, e.g. the 68HC12 by Motorola, also offer generic support for fuzzy logic.

Implementation of the example rule For implementing our example rule using fuzzy logic, each input value would have to be split into a number of ranges which are also called *terms*. To simplify a little, the terms `small`, `medium` and `large` are used for both parameters in the following. Figure 7 depicts their membership functions, normalized to the largest X value.

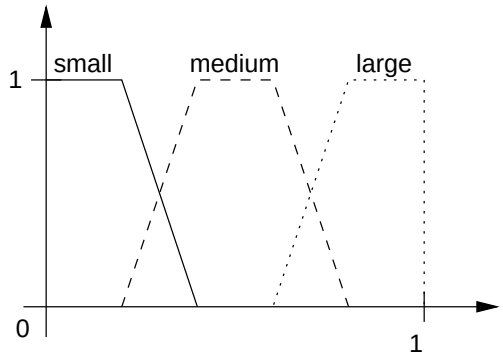

Figure 7: Membership functions for the input values

While the friction coefficient f has a maximum value of 1, for the tool velocity v a 1 corresponds to a speed of 100mm/s.
For the inference step, usually the so-called *Mamdani* controllers are used where also the output variables are separated into single terms along the X-axis. Here, the *Sugeno* model seems to be more appropriate, since the result values for the single rules can be specified as independent functions:

`if v == small and f == medium then r =` $f_1(\mu_{\text{small}}(v), \mu_{\text{medium}}(f))$
`if v == small and f == large then r =` $f_2(\mu_{\text{small}}(v), \mu_{\text{large}}(f))$

Like in [16, pp. 146] the MIN operator is used for the fuzzy AND function. For an input of $v = 15mm/s$ and $f = 0.65$, the possibility values would be

$$\begin{aligned} \mu_{\text{small}}(v) &= 1, \\ \mu_{\text{medium}}(f) &= 0.75 \quad \text{and} \\ \mu_{\text{large}}(f) &= 0.25. \end{aligned}$$

Applying the MIN operator to these values, yields the weight factor for each rule:

$$w_1 = min(1, 0.75) = 0.75$$
$$w_2 = min(1, 0.25) = 0.25$$

Remapping these weights back to the actual signal r could be done by computing the weighted sum

$$r = \sum_{j=1}^{k} w_j \cdot f_j(\vec{\mu}) / \sum_{j=1}^{k} w_j$$

where k is the number of rules that can be applied. By modeling the single functions f_j a good approximation of the wanted linearization could be achieved.
Refer to NAUCK in [22, pp. 269] for a further discussion of *Mamdani* and *Sugeno* controllers and their implementations.

Pros and Cons

+ Like ANNs they are designed to deal with un-precise information, here in the form of un-sharp input variables.
+ The knowledge is stored in simple rules and can be easily maintained and extended.
– Different rules for the same parameter influence each other.
– The membership functions and the defuzzyfication method have to be designed very carefully since minor changes can have a great impact on the output variables. Hence, like for ANNs, a lot of analysis examples might be needed to get useful results.

Conclusion Compared to the ANNs, fuzzy systems appear to be much easier to maintain. Once the system is set up, single rules can be added or changed with ease. Here, the composition of the single membership functions and the defuzzyfication are major drawbacks. The outputs of all rules for a parameter are weighted and accumulated to be transferred back to a real variable by the defuzzyfication step. This means that the final result always depends on all single rule outputs. As a consequence, the weights w_j and the

remap functions f_j for all previous rules might have to be changed in order to get similar results as before, if a single rule is added to a working system. This behaviour makes it difficult to predict the outcome of an added or changed rule. It aggravates the maintenance of the whole system and thereby lowers the attractiveness of fuzzy logic as a solution to the given problem.

2.4.4 Production rules

Prerequisites *Production rules* usually have the following form:

```
<condition> -> <action>
```

They basically consist of a left-hand side (LHS) that specifies the condition upon which the rule gets active. The equivalent right-hand side (RHS) describes the action that should be taken once the condition is true.
Production rules are mainly used by inference-based systems like expert or recommender systems. An inference-based system is built by assembling a knowledge base which is then interpreted by a separate program or module called *inference engine*. The end user of the application interacts with the inference engine, which uses the data put in the knowledge base to answer questions, solve problems, or offer advice.

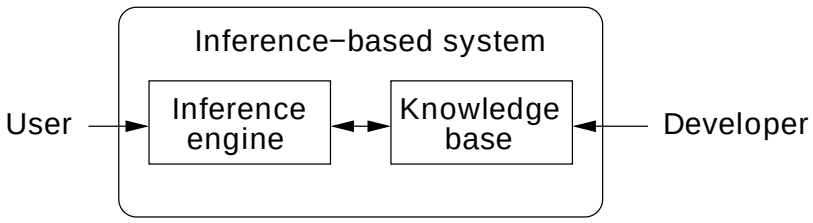

Figure 8: Basic structure of an inference-based system

The typical inference-based system carries one or more of the following features:

- Can represent and manipulate knowledge data.
- Is able to effectively solve problems by using inference.
- Can justify its decisions, to a certain extent.
- Extends its knowledge automatically or at least supports the acquisition.

- Offers a user-friendly interface.

Inference-based systems are used for a wide range of tasks like diagnosis, configuration and construction. For example for expert systems, HELBIG has compiled a more or less complete list of application areas that can be found in [11, pp. 245].
At the start of the inference process a number of facts/conditions are set that represent the start state. Then the inference engine starts to look for rules that match their conditions and can be 'fired', i.e. activated. The actions of a rule may set new conditions or remove unneeded ones and so the inference engine continues until no further rule can be activated. Finally, the end state of the inference engine represents the answer/decision of the system.
Rule systems can be processed using *forward chaining*—like described before—or *backward-chaining*. The former is mainly used for diagnosis and simulation tasks, while the latter is important for planning and configuration problems.
For very large and complex tasks, rule-based knowledge systems are sometimes complemented by factual knowledge about the objects, using frames, semantic nets or predicate logic. A detailed description and discussion of these data representation techniques was done by HELBIG and can be found in [11].

Implementing the example rule Implementing the basic rule from section 2.1 with a rule-based inference system can be done straightforward. As an example, the syntax of the expert system CLIPS (see [14] for a short introduction) is selected that would specify the example rule as follows:

```
(defrule basic_example
 (ToolVelocity ?v)
 (FrictionCoefficient ?f)
 (test (and (< ?v 20)
            (> ?f 0.3)))
=>
 (modify (RelSlidingVelocity (* ?v 0.0005)))
```

It is assumed that the necessary facts `ToolVelocity`, `FrictionCoefficient` and `RelSlidingVelocity` have been assigned using the CLIPS command `assert`. The left-hand side above the `=>` ensures that the necessary condition is met, while the action below modifies the relative sliding velocity according to the example rule. The prefix syntax of CLIPS is a little bit unusual and may be hard to read at first sight. Nevertheless, in a similar manner

additional rules could be specified and existing ones could be changed or even removed, with the desired results.

Pros and Cons

+ The rules of the knowledge base are easy to maintain.
+ A large variety of rules of thumb can be implemented, regarding the available arithmetic expressions.
– The task of data acquisition requires an expert that is needed for creating the rules.
– A special interpreter for the rules is needed.
– Added rules may be ill-formulated, e.g. lead to circular dependencies, and affect the overall performance of the system.

Conclusion From all the three discussed methods, the production rules are the best approach. Since they are represented by arbitrary arithmetic expressions—limited only by the capabilities of the used interpreter—a large set of recommendation needs can be covered.
The rules can directly be 'cast' into the database and are processed as specified. This high transparency—between the input in form of the rules and the output in form of the recommended values—is certainly the most appealing advantage.

2.5 Cognitive psychological aspects

As the latest research in the area of cognitive science shows, providing a functional system is sometimes not enough. In the past, numerous systems were developed that cared about all the intricate technical details but forgot an important point: the user. The human-computer interface (HCI) is what the user sees when he works with the system. His interpretations and expectations, probably based on former experience with similar programs, determine whether the system is useful to him and he will continue to work with it.
Some points of HCI are discussed by Carroll and Rosson in [6]. Their early work focuses on people that are absolutely new to computers, learning to use them like typewriters. The situation of these newbies and their confusion may be well compared to a learner of FEM software. Carroll and Rosson carried out several tests to find out how the learning success

of single persons could be improved. They detected that a lot of existing interfaces—this includes the program's manuals and online help—make significant errors:

1. Similarities and metaphors like "A computer is a super-typewriter" may help, but sometimes lead to wrong conclusions. Hence, they must be used wisely and points where the comparison breaks need to be clearly specified.

2. Beginners have little desire in trying out new functionality. They try to avoid reading lengthy manuals and stick to old methods as long as they can, even if the application of theses methods leads to a decrease of overall efficiency. This also holds for advanced users that try to get every job done with the basic expert knowledge they acquired 'accidentally'. Very seldom they refer to the manual to detect new functions that could help them in solving their tasks more quickly.

3. Restricting the functionality of a program or interface greatly helps the beginners to focus on the basic tasks only. They make less errors and learn faster how to perform daily routine work. The knowledge is better portioned and does not overwhelm the user. Since he has less to digest at once he is better motivated for the next 'baby step' towards expertise.

Some of the above mentioned points are picked up by SUH and SUH in [28]. In their article they discussed the reasons for the failure of several large expert systems and summarized them as follows:

a.) User neglect: The user does not accept the new technology, mainly because he does not like the idea of being replaced or overruled by a 'machine'.

b.) Bad application domain: The case in which the expert system is used, could be solved better and more efficiently by a different approach.

c.) Mismatch of procedure: Management and the workers have a different view on the working procedure that is to be supported by the expert system.

d.) Poor knowledge acquisition: The tedious task of collecting knowledge is delayed, either because no expert user is available or the transfer to the knowledge base is difficult to accomplish.

They proposed to improve expert systems by taking a closer look at the mentioned points and to counteract where it is possible.
Although some time has past and research in the area of HCI progressed, most of the points above are still the foundation for state-of-the-art HCI methods and approaches (see [8, chap. 5,7]).
They are not directly connected to the implementation of the rule engine, but they influence some of the following design decisions. Basically, the rule engine is an independent object that does not know much about its environment. On the other hand, it is embedded in a certain GUI context, which requires the rule engine to offer the needed interfaces for supporting its parent application.
Apart from item 3, which is already covered by the general approach of this work, the following design directives are taken into account:

- Additional help about the rules and parameters should be offered at the fingertip of the user and not in manuals (item 2). In a first approach, text descriptions should be available for rules and parameters, respectively.
- The rule engine should show a passive behaviour, i.e. it does not start any actions like setting parameters or displaying recommendations on its own but on request only. The user should be integrated to the decision making process as much as possible such that he still feels to be 'in control' (item a).
- The language for the rule engine and the database format should be kept simple, such that adding rules is not too complicated (item d).

2.6 Derived structure

Now that the final decision fell for production rules as the most promising approach, an attack plan for the implementation is to be derived.
A survey of existing expert systems revealed that none of them can be directly applied. They are designed for a variety of purposes like biotechnological process control [27], diagnosis [5], construction [15, 13, 4] and configuration [21, 23, 18]. Together with the general expert systems like CLIPS or Jess, they are based on true inference, where new facts can be added during processing. In our case only the data values of the facts (= parameters) change at runtime. Additionally, this work mainly leans upon the development of the two programs MSC SuperForm and MSC SuperForge that are implemented in C++. For interfacing an existing system to their GUIs a lot of extra effort would be required anyway.

As a result, an independent system is implemented from scratch that can be easily integrated into an existing C++ GUI application and is not overfraught with unneeded functionality. By introducing two simplifications that do not restrict the functionality of the rule engine, the implementation is kept simple:

1. For the condition and the action of a rule, only functions as arithmetic expressions are used. This means that they can always be reduced to a single value, which is of the type `bool` for the condition.

2. Since the arithmetic expressions for a rule may use other parameters, a dependency graph is implicitly given. It contains all parameters as vertices, while each directed edge from a vertex u to v denotes that parameter u has to be computed before v. For this graph no cyclic dependencies are allowed, i.e. it has to be a *directed acyclic graph* (DAG). Such a DAG always has at least one schedule (see [24]) which is an important property regarding the ordering of parameters in 3.4.

Figure 9 shows the general operation of the rule engine as a whole.

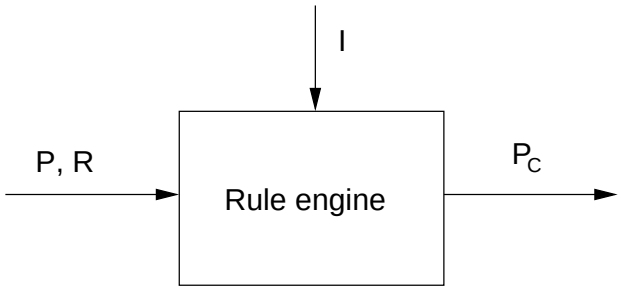

Figure 9: General operation of the rule engine

P is the parameter set, R the rule set. Both are read by the rule engine that interacts with the user/application (I) and suggests better values. After the user accepts the made changes, the set P_C of corrected parameters may be read out. The available data types of parameters get restricted to `bool`, `double` and `integer` for now. The C++ type `double` might also stand for any other floating point data type (like `REAL` in PASCAL).
A rule, like in most *production-rule* systems, consists of two parts, namely the *condition* (if-clause) and the *production* (then-clause). The rule engine checks whether the *condition* is true and then computes a single value that can be recommended to the user. The rules are used to store expert knowledge in

the form of single statements like our introductory example from section 2.2. In the following, two variants of the rule concept are used that only differ in the way their result value is interpreted:

- A *recommendation* rule is used to compute a better value for a parameter. This value may be returned on request by the application. Usually, the user decides when a recommendation should be given and also has the choice to neglect the recommended value.

- A rule can be defined as 'critical'. Critical rules are stored separately and can be used to detect range violations for the single parameters. Everytime a parameter changes by a user entry, the *rule engine* should be employed to check all of its range rules. If one of them 'fires', the application should warn the user about the range violation and enforce proper correction of the mischosen value. This feature is very beneficial for the cross-platform GUI library Qt, since it does not provide its own special method for the range-checking of parameters as the MFC does with its DDX mechanism.

If several rules are defined for a parameter, it is often the case that more than one could be applied. For this frequent occurrence, some form of conflict resolution has to be provided. The rule engine uses two of the many strategies described by JACKSON in [14, pp. 85]:

1. Priority (salience): By setting an explicit priority value a single rule can be fired first, independent of the number of dependencies. The higher priority value wins, the default value is zero. If several rules have the same priority, the complexity of the rules decides.

2. Complexity: A rule with more parameter dependencies is preferred to one that has lesser values it depends on.

If neither of these give a clue, the decision is based on the appearance order of the single rules in the input file.
For storing the rules and parameters in external files, a simple XML format will be used. This flexible format makes the addition of new data fields much easier and its well-formed structure, based on single tags, facilitates the parsing. Additionally, a lot of commercial and free XML editors like `Jaxe` or `Pollo` exist, that can be used to maintain the rule databases.
Figure 10 already shows a more detailed view on the parts that the rule engine consists of. The two lists for parameters and rules are complemented by a small stack processor. This class is responsible for evaluating the current

rules and, if necessary, calculating a recommendation value. All of these subparts can be accessed via the rule engine that connects them to the world outside.

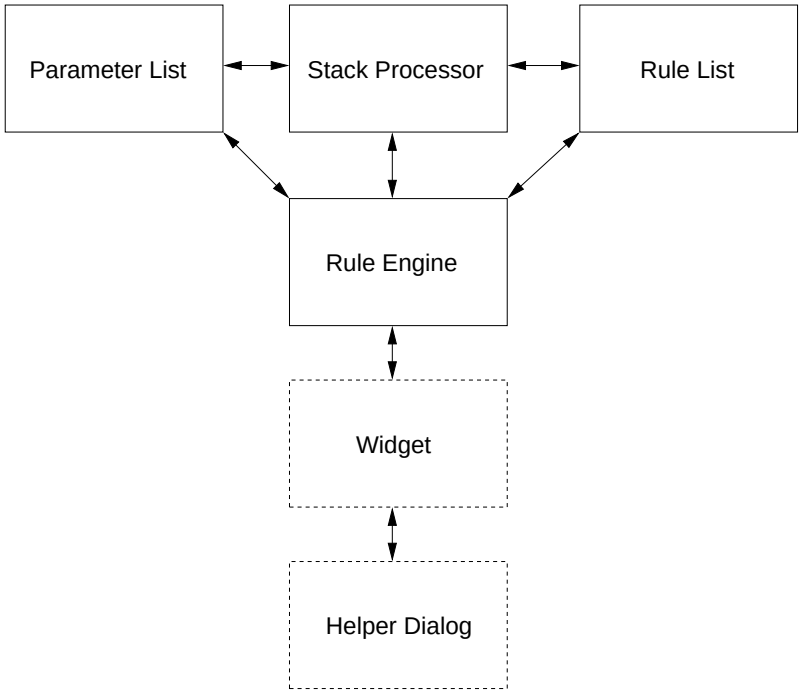

Figure 10: Interaction of the single parts

The Widget and the Helper Dialog are depicted with dashed box lines because their concrete implementations are not really part of this work.
From this basic design, a working prototype has been devised, whose implementation is described in the following section.

3 Implementation

This section introduces the internal design of the rule engine in greater detail. It concentrates on the parts and interfaces that are required for its basic work and leaves out the minor details. At the end, the used language for arithmetic expressions is specified, while putting an emphasis on the available operators and their properties.

3.1 Rule engine

To the world outside, the `RuleEngine` is a monolithic object but it is built by several cooperating classes as the UML (Unified Modeling Language) diagram in figure 11 shows.

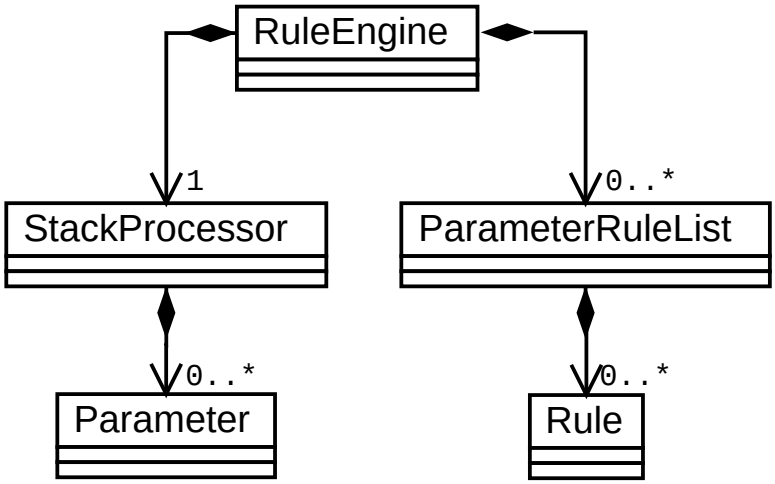

Figure 11: UML overview of the `RuleEngine`

Each `RuleEngine` has a single instance of the class `StackProcessor`. The `StackProcessor` is responsible for evaluating the arithmetic expressions of rules. For this, it keeps track of the available `Parameter`s by storing them in a list. A `RuleEngine` also holds a list of `ParameterRuleList`s. In a `ParameterRuleList` all `Rule`s—recommendations as well as range violations—for a single parameter are combined.
The basic interface of the `RuleEngine` is depicted in figure 12.
An application that employs the rule engine, can command it to read in data from an external file and `Parameter` values can be set or read out.
While reading a RuleEngine XML file it is usually assumed that the single rule entries are unsorted, regarding their precedence. Thus after all data was read, the parameter rule lists are sorted by the algorithm described later in section 3.4. For efficiency reasons this step can be omitted by setting the variable `Precompiled` to `true`.
The additional variable `Postfix` serves a similar purpose. The arithmetical expressions for the rules are usually entered in infix notation. During the read-in process they get converted to postfix once because this is the only format the `StackProcessor` can evaluate. For speedup or for the benefit of users that are used to RPN (Reverse Polish Notation), `Postfix` can be set

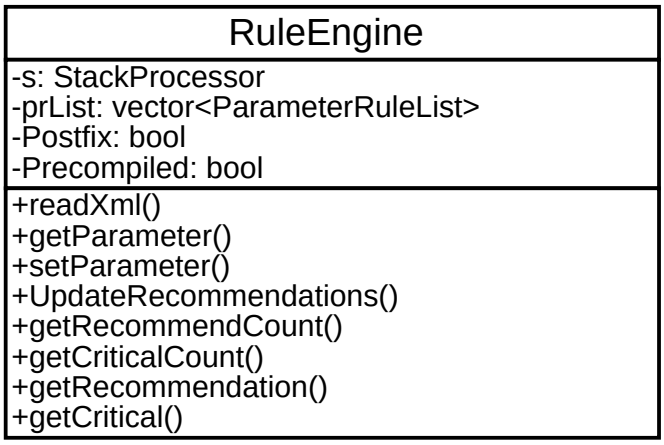

Figure 12: Basic interface of the `RuleEngine`

to `true` and the conversion is skipped.
The most important method is `UpdateRecommendations`, which loops through the rules and tries to detect new range violations or recommendations. If this update is successful the 'fired' rules are marked and can be retrieved with either `getRecommendation` or `getCritical`.
For a 'recommendation update' a number of basic steps are performed. At start, the shadow values for all parameters are reset to the current value of each parameter (see section 3.5 for further explanations). Then in a loop all `ParameterRuleList`s are processed: At first, the critical rules are scanned. As soon as the condition of a critical rule for a parameter is found to be true, the index of the rule is stored and the loop stops. The update process then continues with the next `ParameterRuleList`. Rules whose condition can not be evaluated—an undefined parameter or a division by zero might be reasons—are simply skipped. If no range violation for a parameter could be found, the recommendation rules are checked in a similar manner.
Like this, a single parameter can have either a range violation or a recommendation at maximum, but not both. The total number k of recommendations can be retrieved by the functions `getRecommendedCount` and `getCriticalCount`. Via the indices 0–$(k-1)$ the single rules may then be addressed with the functions `getRecommendation` and `getCritical`.
Finally, the syntax for a valid `RuleEngine` XML file is given:

```
<RuleEngine>
      [<Name></Name>]
      [<Precompiled></Precompiled>]
      [<Postfix></Postfix>]
```

```
        <Parameters>
                List of parameters
        </Parameters>
        <Rules>
                List of parameter rule lists
        </Rules>
</RuleEngine>
```

Tags that are enclosed in brackets are optional, the lists of parameters and rules may have an arbitrary length both—including the zero.

3.2 Rules

The class `Rule` is simply a container for all its describing data (see fig. 13).

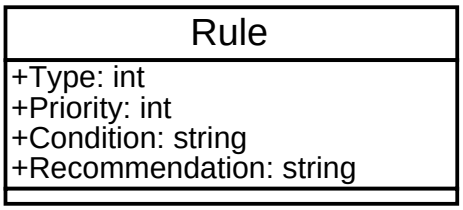

Figure 13: Basic interface of a `Rule`

Via the `Priority` parameter, a rule can be prejudiced against others. Rules with higher numbers are processed first within a `ParameterRuleList`, the default priority value is 0.
If two rules have the same priority, then the dependencies of the `Condition` and `Recommendation` expressions regarding other parameters decide which of them comes first. They are measured by the in-degree of the parameter vertex in the precedence DAG, i.e. the number of incoming edges—the self-reference does not count. A rule that uses four other parameters is always preferred against one that refers to only three.
During read-in, critical rules are distinguished from normal recommendations by the `Type` field. It has to be set to `Critical` for a range violation, which ensures that the rule is added to the right list in the `ParameterRuleList`.
The XML syntax for a `Rule` is:

```
<Rule>
        [<Name></Name>]
        [<Type></Type>]
```

```
      [<Priority></Priority>]
      [<Description></Description>]
      [<Condition></Condition>]
      <Recommendation></Recommendation>
</Rule>
```

3.3 List of rules for a parameter

The class `RuleParameterList` keeps track of all rules for a single parameter. As the interface (see fig. 14) shows, the rules are divided into critical ones that signal range violations and normal recommendations.

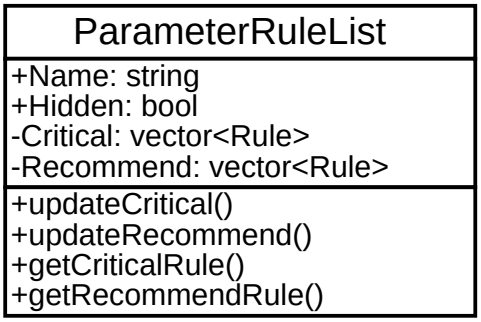

Figure 14: Basic interface of a `ParameterRuleList`

Each `RuleParameterList` needs to have the name of the `Parameter` assigned that the rules are applied to. Within each of the rule lists `Recommend` and `Critical`, the entries are sorted as described in the previous section 3.2. The first rule that is processed has the highest priority and the most dependencies to other parameters.
It will often be the case that several rules are defined for a `Parameter`, using the same subexpressions. For example, the two `Conditions`

```
(temp <= 20) AND (FE = TRUE) AND (friction < 0.3)   /* 1 */
(temp <= 20) AND (FE = TRUE) AND (friction >= 0.3)  /* 2 */
```

share the expression `(temp <= 20) AND (FE = TRUE)`.
In this case, the processing speed and the readability in the XML file can be enhanced by defining the subexpression as a 'hidden' parameter. Its `Condition` is left empty and the `Recommendation` is set to the subexpression. Setting the `Hidden` field to `TRUE`, marks the parameter as internal and avoids that it gets listed in the recommendations after an update.

Assuming the new internal parameter is named '`coldFE`', the two conditions could be rewritten as:

```
coldFE AND (friction < 0.3)   /* 1 */
coldFE AND (friction >= 0.3)  /* 2 */
```

This way, rule sets can be optimized—either manually or by an external program that issues an appropriate algorithm, like the *Rete* algorithm by FORGY (see [9]) that is used in OPS5 and CLIPS for example.
In XML, a `ParameterRuleList` is specified as follows:

```
<Parameter>
      <Name></Name>
      [<Hidden></Hidden>]
      List of rules
</Parameter>
```

Again, the list of rules may have an arbitrary length or be empty.

3.4 Ordering of rules

As has already been stated in section 2.6, the rules have to be ordered just once after they are read in. The goal of this process is that the rules can be evaluated in the order they are stored in memory, which speeds up the evaluation. For this, all the necessary information has to be drawn out of the `Condition`s and `Recommendation`s of the single rules, that implicitly define the directed acyclic graph (DAG) of dependencies for the parameters.
As a first step towards a correct ordering, each `ParameterRuleList` gets his set of dependencies updated. This is achieved by processing the single rules one after the other. For each `Condition` and `Recommendation`, a special function of the `StackProcessor` parses the arithmetic expressions and returns the set of used parameters. This set is then added to the internal list of dependencies for the parameter. In the same step, the number of dependencies for the single rule is updated, which is important for their ordering within a `ParameterRuleList` (see 3.2).
Once all single parameters have an updated set of their dependencies, the problem is to get them in the right order. Here a common graph problem called *topological sorting* is encountered. It arises as a natural subproblem in most algorithms on directed acyclic graphs. Topological sorting orders the vertices and edges of a DAG in a simple and consistent way and hence plays the same role for DAGs that depth-first search does for general graphs. It can

be used to schedule tasks under precedence constraints and any topological sort (also known as a linear extension) defines an order to do these tasks such that each is performed only after all of its constraints are satisfied.
The problem can be stated in a more mathematical way as follows:

> **Input**: A directed, acyclic graph $G = (V, E)$ (also known as a partial order or poset).
>
> **Problem**: Find a linear ordering of the vertices of V such that for each edge $(i, j) \in E$, vertex i is to the left of vertex j.

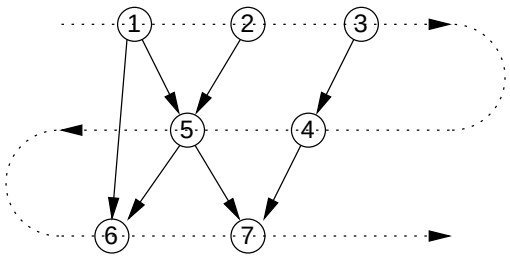

Figure 15: The problem of *topological sorting*

The usually applied basic algorithm (see [2, pp. 497] for example) performs a depth-first search of the DAG to identify the complete set of source vertices, where source vertices are vertices without incoming edges.
At least one such source must exist in any DAG. Note that source vertices can appear at the start of any schedule without violating any constraints. After deleting all the outgoing edges of the source vertices, new source vertices are created, which can sit to the immediate right of the first set. This is repeated until all vertices have been accounted for.
The problem is that the DAG for the `RuleEngine` is given only implicitly by the rules. Instead of creating one first, the algorithm is slightly modified and operates on two sets. A set of already accounted vertices A and a set R for the rest.
At the start of the algorithm, A is empty while R is initialized with all `ParameterRuleLists`. Then it loops over the entries in R and tries to find a parameter rule list whose dependencies are met by A. If an element has been found, it is removed from R, added to A and gets the current list index assigned. This is repeated until R is empty or no more elements can be added. The latter is an indicator for a cyclic dependency in the precedence graph. In this case all further elements of R are deleted, i.e. the according parameter rule lists are removed.

Once the algorithm has finished, the list of `ParameterRuleLists` is sorted with a less-than operator, based on the assigned list index only.

3.5 Parameter

A `Parameter` basically consists of a Name and its two data values (see fig. 16).

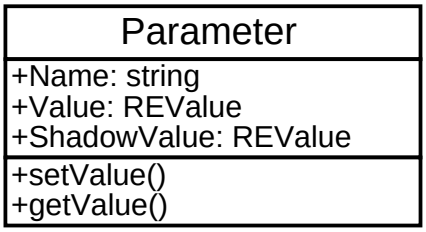

Figure 16: Basic interface of a `Parameter`

All parameters keep track of their actual value as it is currently set in the GUI. Additionally, the shadow value is used to store the recommended value. At the beginning of an update for all recommendations, the shadow value is set to the actual value for all parameters. This step is necessary because during the update process only the shadowed values are read and written while evaluating the rules. It provides a kind of 'latch', since it is allowed for a parameter to refer to itself in a rule (reflexivity).
Values are encapsulated in their own class `REValue`, which is a simple container that stores the type (integer, double or boolean) and the data itself.
Parameters are usually set by the GUI, whenever the user has entered a new value for it. After a recommendation update, either the current or the shadow value can be fetched for further comparisons or display purposes like for the example in section 4.2.
The XML syntax for a `Parameter` is:

```
<Parameter>
      [<Name></Name>]
      <Value>
            <Type></Type>
            <Data></Data>
      </Value>
</Parameter>
```

The `Name` has to start with a letter, followed by an arbitrary combination of letters and digits (see 3.7.7). With the variable `Type` the data type of

the value can be specified. Accepted get `Int`, `Bool` and `Double` for integer, boolean and floating-point data, respectively. The field `Data` is used to set the default value for the parameter.

3.6 Stack processor

The class `StackProcessor` has three main tasks:

- It directly manages the list of `Parameter`s,
- converts infix expressions to postfix during the read-in process and
- evaluates postfix expressions.

The basic interface as depicted in figure 17 lists the names of the according functions.

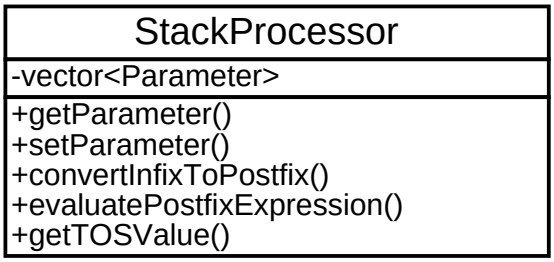

Figure 17: Basic interface of the StackProcessor

After an expression has successfully been computed with `evaluatePostfixExpression`, the function `getTOSValue` returns the result from the top of the stack (TOS).

3.6.1 Converting Infix to Postfix

With 'paper and pencil' one can convert infix to postfix very easily. The steps to perform are:

1. For each operator symbol in the expression, put a pair of parenthesis around the operator and operands. The resulting expression is called fully parenthesized.
2. For each operator symbol find its right parenthesis in the fully parenthesized form. Replace that right parenthesis by the operator.

3. Remove all of the left parentheses from the expression.

This algorithm converts the complicated expression

```
a/b^c+d*e-a*c
```

to its postfix equivalent

```
a b c ^ / d e * + a c * -
```

The second rule indicates that the only things that 'move' are the operator symbols and the right parenthesis tells where to move them. Thus, the operands in the postfix form occur in the same order as they do in the infix form. In converting infix to postfix, only the operator symbols are of major interest, the operands can just be copied from the infix string to the postfix string.
To handle the operators correctly, they are pushed onto a stack until the 'location of the correct right parenthesis' is found. To do it right, each operator needs to be assigned two precedence values—one when it is on the infix expression string (the 'in-coming precedence') and one when it is on the stack (the 'stack precedence'). Usually an operator will carry the same value for both, but the left parenthesis is an exception. It gets the highest possible value for incoming and the lowest for stack precedence.
The reason for this special assignment gets clear while looking at the pseudo-code for the used algorithm:

```
Stack s;
Token t;
while (!endOfInput())
{
  getNextToken(t);
  if (t == operand)
    cout << operand;
  else if (t == rightParen)
  {
     while (s.top != leftParen)
     {
       cout << s.top;
       s.pop();
     }
     s.pop();
  }
  else
```

```
  {
    while (StackPrec(s.top) >= InPrec(t))
    {
      cout << s.top;
      s.pop();
    }
    s.push(t);
  }
}

while (!s.empty)
{
  cout << s.top;
  s.pop();
}
```

While parsing a string, the scanner is built by a single function `getNextToken` that returns the next token in the input. Encountered operands—numbers as well as variable identifiers—are directly output. On a right parenthesis all stacked operators, up to the left matching parenthesis, are popped from the stack. Before a new operator is pushed, all stack members that have a higher precedence are popped. At the end of the input, the stack is cleared by popping the operators that remained.
If the left parenthesis would have a non-zero stack precedence, it could be popped from the stack by another operator that comes in and has a higher precedence. This has to be avoided; a left parenthesis may only be removed whenever a matching right parenthesis is found.
Due to the algorithm, all operators are *left-associative* (see [1, p. 30]), i.e. the expression

```
2-4-5
```

is evaluated as

```
(2-4)-5
```

3.6.2 Evaluating postfix expressions

For the evaluation of postfix expressions, the same scanner function `getNextToken` is used. This time, not the operators but the operands are put on a stack.

If an unary or binary operator is encountered one or two values are popped from the stack, respectively. They are combined, according to the requested operation, and the result is put on the stack again.
If a variable identifier is found in the input, the value of the corresponding parameter is fetched from the internal list of `Parameter`s. The `StackProcessor` always uses the shadow values, representing the last recommendation for this parameter.

3.7 Rule expression language (REL)

Now, the set of allowed operators and expressions for the REL is introduced in detail. The operators are grouped, according to what parameters they expect and their output. Eventually, the operator precedences and the grammar for the arithmetic expressions in infix notation are described.

3.7.1 Computational operators

Computational operators are the four normal binary operators for addition, subtraction, multiplication and division. They get complemented by the `MIN` and `MAX` function for returning the lowest and highest of two values, respectively. They all map a 2-tuple to a single scalar.
The data type `type` may stand for a `double` or an `int` in the following. For example, an addition between a `double` and an `int` is supported as well as an addition between two `int`s. The result for the first will be a `double`, while the latter gives an `int` again.

$$\text{compop:} \quad \text{type} \times \text{type} \to \text{type}$$

$$\text{Available:} \quad +, -, \cdot, /, \texttt{MIN}, \texttt{MAX}$$

3.7.2 Equational operators

Equational operators map two `type` values to a `bool` value. Like for the computational operators, an `int` value is automatically coerced to a `double` if the other operand is a floating point number.

$$\text{equivop:} \quad \text{type} \times \text{type} \to \text{bool}$$

$$\text{Available:} \quad <, >, =, \neq, \geq, \leq$$

3.7.3 Coercions

For the two preceding operator types, coercions are performed automatically as soon as one of the two operands is a `double`. The result of the operation is then coerced to a `double`, too.
The normal coercion chain can be broken by the following explicit coercions, that are unary and cast the given `double` to an `int` again:

- ROUND : Normal rounding operation
- FLOOR : Rounds to the greatest `int` that is smaller than the `double`
- CEIL : Rounds to the smallest `int` that is greater than the `double`

coerceop: double → int

Available: `ROUND`, `FLOOR`, `CEIL`

3.7.4 Boolean operators

In order to combine single logical terms, the REL supports the usual boolean operators `AND` and `OR`. They map two values of type `bool`, to a single one.

boolop: bool × bool → bool

Available: `AND`, `OR`

3.7.5 Unary operators

So far only a subset of the usual functions like tanh, arccos or x^y have been implemented. These are the `SIN` and `COS` function, the natural logarithm `LOG` as well as the e^x function `EXP`. The `NEG` operator changes the sign of the current number and can also be applied to integer values.

unaryop: double → double

Available: `SIN`, `COS`, `LOG`, `EXP`, `NEG`

Apart from the functions that operate on double values, the boolean `NOT` maps a `bool` to a `bool`.

boolnot: bool → bool

It can be replaced by a single exclamation mark '`!`' in the input.

3.7.6 Operator precedence

Now that all the operators have been introduced, the question about their priorities in relation to each other arises. Which have a higher precedence than others? The answer can be partly derived from the REL grammar that is given in the following section. For the sake of completeness, table 1 shows the precedences of the single operator groups—from high priority at the top to the lowest precedence.

Precedence	Operators
16	'(', ')'
15	'NEG'
14	'MIN', 'MAX', all unary ops
13	'*', '/'
12	'+', '-'
10	'<=', '<', '>=', '>'
9	'!=', '=='
8	'NOT'
7	'AND'
6	'OR'

Table 1: Operator precedences

The higher the precedence is, the stronger surrounding operands are bound to the operator. This means that the expression

```
A <= 4 AND 6 + 3 * 2 != B
```

for example is evaluated as

```
(A <= 4) AND ((6 + (3 * 2) != B)
```

3.7.7 Grammar

The grammar for the infix format of the REL follows the PASCAL syntax for arithmetic expressions (see [12]). It got enhanced by the unary operators `COS`, `SIN` a.s.o., the boolean values `TRUE` and `FALSE` and the `MIN` and `MAX` functions that were appended to the non-terminal `factor`.

```
expression : simple_expr
           | simple_expr relop simple_expr

simple_expr : term_list
```

```
            | '+' term_list
            | '-' term_list

term_list : term
          | term '+' term_list
          | term '-' term_list
          | term 'OR' term_list

term : factor
     | factor '*' term
     | factor '/' term
     | factor 'AND' term

factor : varIdentifier
       | signlessConstant
       | '(' expression ')'
       | 'NOT' factor
       | unaryOperator factor
       | binaryFunction '(' expression ',' expression ')'
       | 'TRUE' | 'FALSE'

unaryOperator : 'SIN'   | 'COS'
              | 'EXP'   | 'LOG'
              | '-'     | 'NEG'
              | 'ROUND' | 'FLOOR' | 'CEIL'

binaryFunction : 'MIN'
               | 'MAX'

relop : '<'
      | '<='
      | '>='
      | '='
      | '!='

varIdentifier : letter [letter|digit]*

signlessConstant : signlessDouble
                 | signlessInt

signlessDouble : signlessInt '.' signlessInt
               | signlessInt '.' signlessInt exponent
               | signlessInt exponent
```

```
exponent : 'E' signlessInt
         | 'E' '+' signlessInt
         | 'E' '-' signlessInt

signlessInt : digit
            | digit signlessInt

digit : ['0'-'9']

letter : ['a'-'z' 'A'-'Z']
```

As can be easily derived from the grammar, a variable identifier `varIdentifier` always starts with a letter, followed by an arbitrary number of letters or digits. It is important to note that this syntactical restriction also applies to parameter names.
Since usual XML editors will be the preferred source for RuleEngine files, the `StackProcessor` is able to cope with the replacements `<` and `>` for < and >, respectively.
Finally, a few examples for correct syntax:

```
(A < B) AND NOT ((2*D) >= F)
2*3.1415
4*cutLength + (2.5/cutWidth) * 27.0 / 5.0
```

as well as incorrect expressions:

```
(A << B)
(A < B) AND NOT ((2*D >= F)
```

4 Examples

This section roughly sketches the development and usage of two example dialogs that employ the RuleEngine in different ways. Both examples were created using the C++GUI library Qt under Linux.

4.1 Basic example rule

The first example implements the basic rule from section 2.2. Figure 18 shows the initial setup of the dialog.

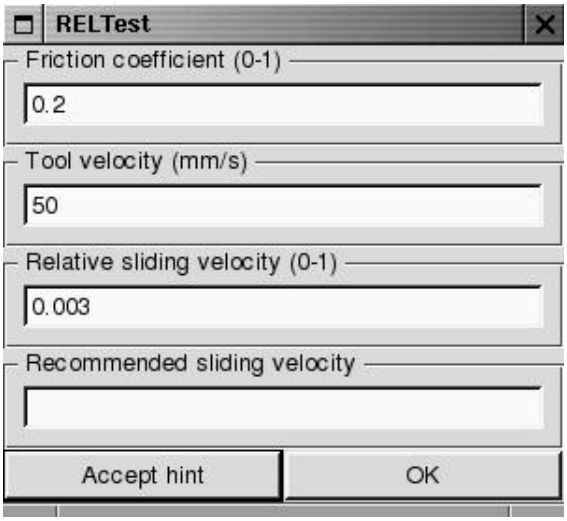

Figure 18: Initial dialog setup

Apart from the needed entry fields for the parameters, the hint and the OK button, a button has been added that accepts the current recommendation as the new relative sliding velocity.
While the user changes values, the rule engine is continuously employed to compute a better value for the relative sliding velocity, if possible. In figure 19 the user has changed the friction coefficient to `0.5` and the tool velocity to `5`, which means that the example rule gets active. Consequently, a recommendation is displayed in the text field at the bottom.
If the user clicks the 'Accept hint' button, the recommended value is set as new parameter value within the rule engine and the dialog itself. Figure 20 shows the dialog after the suggestion for the sliding velocity was accepted.
To reach this functionality, only a few changes to the original code were necessary. In the following code snippets they are enclosed by an '`#ifdef USE_RULE_ENGINE`' construct. First an instance `rEngine` of the RuleEngine was added to the dialog, that reads the data from an external file in the constructor of the dialog class:

```
/** Constructor for the \a basicruledlg class. */
basicruledlg::basicruledlg( QWidget *parent, QString name )
    : QDialog( parent, name, true)
{
#ifdef USE_RULE_ENGINE
    ifstream f("BasicRuleExample.xml");
    if (f)
```

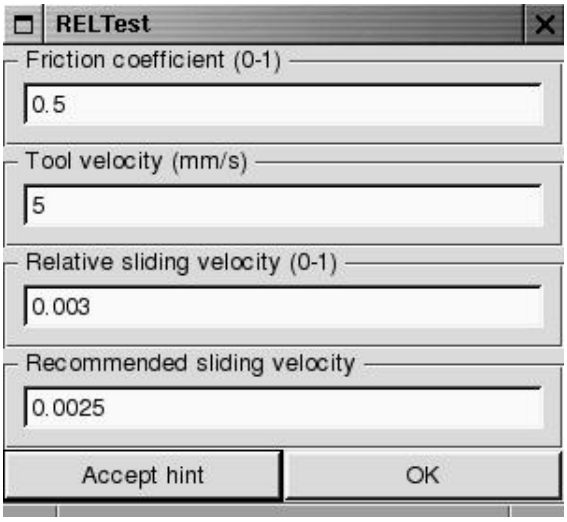

Figure 19: A recommendation is available...

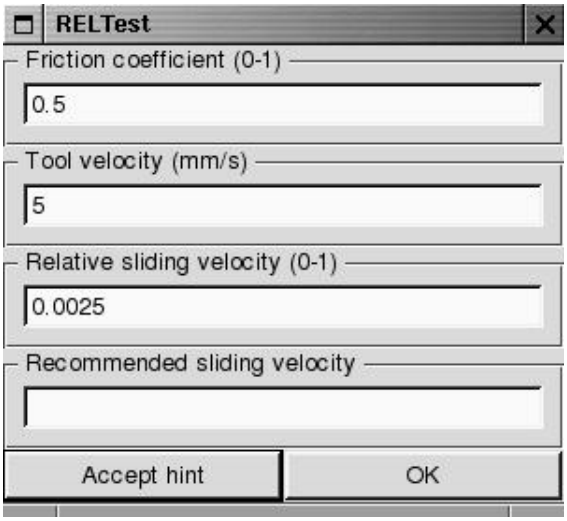

Figure 20: ... and got accepted.

```
      rEngine.readXml(f);
#endif

    QBoxLayout *vb = new QVBoxLayout( this );

    ...
```

Whenever a text field changes, a special function is called that gets the new string as argument. Such a function exists for each of the three parameters and checks whether the string does not contain any invalid characters. These functions were complemented by a few lines to support the rule engine in its task:

```
void basicruledlg::slotFrictionChanged(const QString &text)
{
  if (text.isEmpty())
    return;

  bool success = false;
  double frictionNew = text.toDouble(&success);

  if (true == success)
  {
    friction = frictionNew;
#ifdef USE_RULE_ENGINE
    rEngine.setParameterValue("FrictionCoefficient", friction);
    rEngine.UpdateRecommendations();
    updateRecommendation();
#endif
  }
  else
  {
    // Restore old value
    leFriction->setText(QString::number(friction));
  }
}
```

After checking the input as before, the new value for the friction is set in the rule engine. Then the function `UpdateRecommendations` is called that processes the rules, stored in `rEngine`. The function `updateRecommendation` checks whether a recommendation value for the parameter sliding velocity is available. If yes, it displays the value in the fourth text field:

```
#ifdef USE_RULE_ENGINE
void basicruledlg::updateRecommendation()
{
  REValue curVal;
  REString curString;

  if (true == rEngine.hasRecommendation("SlidingVelocity"))
  {
    REValue curVal;
    REString curString;

    curVal = rEngine.getParameterShadowValue("SlidingVelocity");
    curVal.toString(curString);
    leRecommended->setText(curString.c_str());
  }
  else
  {
    leRecommended->setText("");
  }
}
#endif
```

Whenever the 'Accept hint' button is clicked the function `slotAcceptRecommendation` is called that sets the recommended value, first for the RuleEngine and then in the GUI:

```
void basicruledlg::slotAcceptRecommendation()
{
  // Accept the made recommendation
  rEngine.acceptRecommendedValue("SlidingVelocity");
  if (leRecommended->text() != "")
    leSlidingVelocity->setText(leRecommended->text());
}
```

4.2 Baking a cake

The second example asks the user to enter some main ingredients for a dough recipe (see figure 21)
Apart from the text entries and the OK button that accepts the made settings, the lower left of the dialog offers a 'Rule info' button. It changes its appearance, depending on whether recommendations are available or not. The basic behaviour of the dialog and the necessary changes in the source code are similar to the previous example. Upon each change of a text field

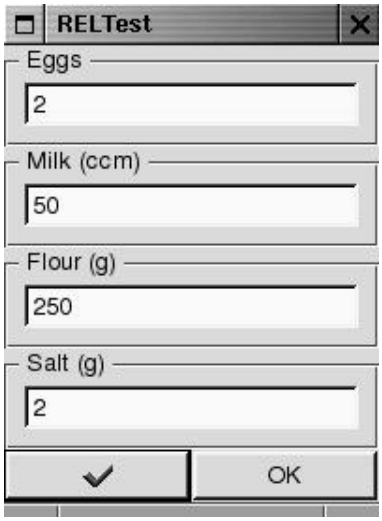

Figure 21: Initial dialog setup

the RuleEngine processes all rules and computes new recommendations. In figure 22 the user has entered a new value for the number of eggs, which leads to a number of recommendations as is signaled by the little symbol on the 'Rule info' button.
If the user presses this button, another dialog comes up that delivers further infos about the recommendations. The evoked dialog is shown in figure 23.
It was designed as a kind of simple 'recommendation browser' that informs the user about which values should be set and why. With the upper line of 'arrow buttons' he may step back and forth in the list of fired rules or jump to the first and last entry, respectively. The affected parameter, as well as its current and recommended value, are displayed in the 'Info' text field. Additionally, the description of the rule is shown and tries to justify the decision. In the bottom line, the user can select to accept all necessary changes or only the current recommendation.
Finally, figure 24 shows how the separation between normal recommendations and critical rules can be exploited. Following an invalid entry of `-3` eggs by the user, a 'stop' sign is displayed in the lower left.
If the user clicks the 'Rule info' button again, the same browser dialog is shown as for normal recommendations. The only difference is that the word 'Recommendation' in the 'Info' field is replaced by 'Range violation'.

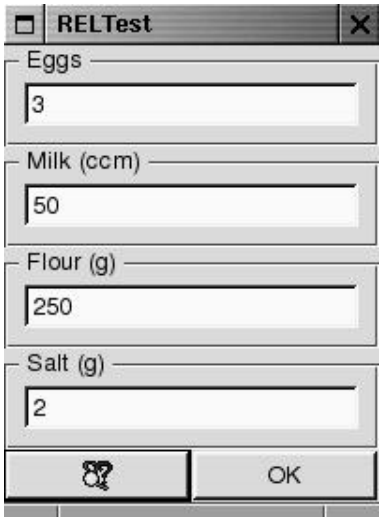

Figure 22: Recommendations are available

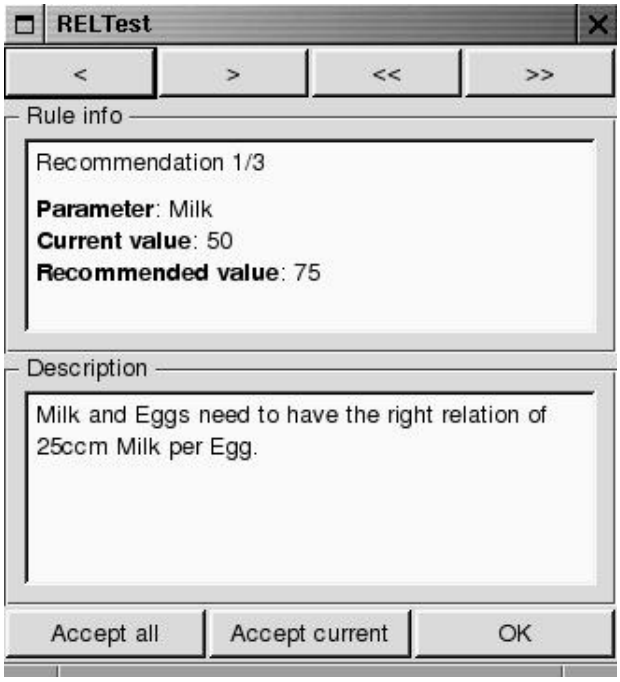

Figure 23: A simple 'recommendation browser'

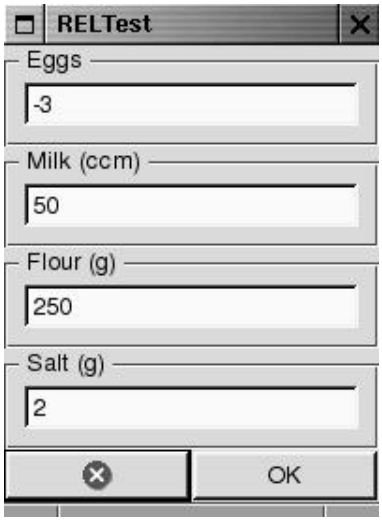

Figure 24: A range violation was detected

5 Conclusion

In this work a small rule engine system was designed and built. Its task is to deliver recommendation values for a set of parameters, based on rules that express certain relations between them. Written in C++ and providing a useful set of interface functions, it can be fitted into existing applications without a lot of extra work.
It manages two separate lists of rules and parameters, that are read in from an external text file in XML format. Right at the start, all rules are reordered such that they can be processed in a row without hurting any precedence constraints. Upon request, the conditions for the single rules are checked, based on the current parameter settings. These conditions—as well as the formulas for calculating the corrected value—are specified in the *Rule Expression Language* (REL), a simple language for arithmetic expressions. If single rules 'fire' and recommendations are available the corrected parameter values can be read out. Apart from normal recommendations, the rule engine also knows the so-called 'critical' rules that can be used to detect range violations.
The rule engine is not able to process rule sets with cyclic dependencies between the single parameters. It is expected that the given precedence graph—derived from the relations between the single parameters—is cycle-free. This implies that some hierarchy for the watched parameters has to be provided.

By adding a rule engine to an existing application the different parameters of a dialog can be watched. If the user sets an inappropriate value and a defined rule gets 'activated' a better setting can be recommended. Keeping the rules in an external 'knowledge base' helps the developer and the ardent user to adjust the rule set if needed. For this, no recompilation of the whole program is required.
This functionality outdoes the range checking of existing GUI frameworks like the DDX mechanism of the MFC. There, values can only be checked to lie in a specified interval, which is established at compile-time. The added features of the rule engine can help the inexperienced user to set reasonable values for special parameters in his first try. In addition, the rule system is able to tell the user why this setting is reasonable. The possibility to request descriptions of parameters and rules can have a positive learning effect on the newcomer.
So far, a working prototype of the rule engine is provided, together with a simple external check program called RuleChecker (see Appendix B) that can be used to verify the integrity of rule sets. The defined rule expression language (REL) is powerful enough to cover a large range of arithmetic expressions. Nevertheless, there is still room for some improvements:

1. The set of operators for the rule expression language could be enhanced by additional functions like x^y, arcsin or tanh.

2. A set of special comments or sub-tags for the descriptions of parameters and rules could be introduced—delimiting a link to a parameter, for example. Combined with a properly designed browser dialog they would enable the user to skim through the internal structure of the loaded rule set.

3. Based on the rule engine and the RuleChecker program, a special GUI application for designing and testing rule sets would be helpful.

The simple examples from section 4 show that the rule engine can be easily molded into an already existing GUI environment. As a next step, a version of the recommender system is planned to be integrated to the development of the FE-FV simulation application MSC SuperForge in cooperation with the Femutec GmbH, Hamburg. This practical test will hopefully deliver some good results as well as further insights and help to support and guide the current and future users of the program.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques and Tools.* Addison Wesley, second edition, 1988.

[2] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science.* Addison Wesley, 1995.

[3] Rüdiger Brause. *Neuronale Netze. Eine Einführung in die Neuroinformatik.* Leitfäden und Monographien der Informatik. Teubner Stuttgart, 1991.

[4] Thorsten Breitfeld. *Entwicklung von Expertensystemen zur Unterstützung konstruktionsbegleitender Finite-Elemente Berechnungen.* PhD thesis, Institut für Statik und Dynamik der Luft- und Raumfahrtkonstruktionen, Universität Stuttgart, 1999.

[5] Bruce G. Buchanan and Edward H. Shortliffe. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley, 1849.

[6] J. Carroll and M. Rosson. Paradox of the active user. In *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction.* Bradford Books/MIT Press, 1987.

[7] C.C. Chen and S. Kobayashi. Rigid-plastic finite element analysis of ring compression. *Appl. Num. Method. Form. Proc. ASME AMD*, 28:163, 1978.

[8] Alan Dix, Janet Finlay, Gregory D. Abowd, and Russell Beale. *Human-Computer Interaction.* Pearson Education Limited, third edition, 2004.

[9] C.L. Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[10] Chris J. Harris, Jonathan M. Roberts, and P. Edgar An. An intelligent driver warning system. In L.C. Jain, R.P. Johnson, Y. Takefuji, and L.A. Zadeh, editors, *Knowledge-based Intelligent Techniques in Industry*, pages 1–51. CRC, 1999.

[11] Hermann Helbig. *Künstliche Intelligenz und Wissensverarbeitung.* Verlag Technik, Berlin, 2 edition, 1996.

[12] Rudolf Herschel. *Standard-Pascal : Systematische Darstellung für den Anwender nach DIN 66256.* Oldenbourg, München, 1991.

[13] Volker Hirsch. *Ein Expertensystem als Konstruktionswerkzeug zur aktiven Geräuschminderung an Getriebegehäusen.* PhD thesis, Fachgebiet Maschinenelemente und Maschinenakustik, Technische Universität Darmstadt, 1998.

[14] Peter Jackson. *Introduction to Expert Systems.* Addison Wesley, third edition, 1999.

[15] Thomas Jüttner. *Der Entwurf adaptiver Regler durch Einsatz eines Expertensystems.* Number 672 in 8: Meß-, Steuerungs- und Regelungstechnik. VDI, 1997.

[16] Dimitris Karagiannis and Rainer Telesko. *Konzepte der Künstlichen Intelligenz und des Softcomputing.* Lehrbücher Wirtschaftsinformatik. Oldenbourg, 2001.

[17] Benjamin Kuipers. *Qualitative reasoning: modeling and simulation with incomplete knowledge.* MIT, 1994.

[18] Chung-Yu Liu, C.R. Emerson, and K. Srihari. An expert system approach to surface mount pick-and-place machine selection. pages 303–320. Chapman & Hall, 1994. Structures and Rules.

[19] Jan Lunze. *Künstliche Intelligenz für Ingenieure*, volume 1. Oldenbourg, 1994.

[20] Walter McCulloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[21] P. Mertens, L. Ludwig, and Th. Wedel. Knowledge-based parameter configuration in MRP packages. In Singh and Travé-Massuyès [26], pages 355–360. Proceedings of the IMACS International Workshop on Decision Support Systems and Qualitative Reasoning, Toulouse, France, 13-15 March, 1991.

[22] Detlef Nauck, Frank Klawonn, and Rudolf Kruse. *Neuronale Netze und Fuzzy-Systeme.* Computational Intelligence. Vieweg, 1996.

[23] Sisir K. Padhy and Suren N. Dwivedi. Pcaad — an object-oriented expert system for assembly of printed circuit boards. *Expert Systems*, 9(1):11–23, February 1992.

[24] J. K. Pearson and P. G. Jeavons. A survey of tractable constraint satisfaction problems. Technical Report CSD-TR-97-15, 1997.

[25] Hendrik Schafstall. *Verbesserung der Simulationsgenauigkeit ausgewählter Massivumformverfahren durch eine adaptive Reibwertvorgabe.* PhD thesis, Universität der Bundeswehr Hamburg, 1999.

[26] M. G. Singh and L. Travé-Massuyès, editors. *Decision Support Systems and Qualitative Reasoning.* IMACS, 1991. Proceedings of the IMACS International Workshop on Decision Support Systems and Qualitative Reasoning, Toulouse, France, 13-15 March, 1991.

[27] J-Ph. Steyer, J-B. Pourciel, D. Simoes, and J-L. Uribelarrea. Qualitative knowledge modeling in a real time expert system for biotechnological process control. In Singh and Travé-Massuyès [26], pages 387–393. Proceedings of the IMACS International Workshop on Decision Support Systems and Qualitative Reasoning, Toulouse, France, 13-15 March, 1991.

[28] Chang-Kyo Suh and Eui-Ho Suh. Using human factor guidelines for developing expert systems. *Expert Systems*, 10(3):151–156, August 1993.

[29] L. Zadeh. Fuzzy sets. *Information and Control 8*, pages 338–353, 1965.

[30] Jens Zeidler. *Unscharfe Entscheidungsbäume.* PhD thesis, Technische Universität Chemnitz, Fakultät für Informatik, 1999.

Appendix A: XML file for the basic rule example

```
<RuleEngine>
<Name>Basic Rule Example</Name>
<Parameters>
 <Parameter>
  <Name>FrictionCoefficient</Name>
  <Value>
   <Type>Double</Type><Data>0.2</Data>
  </Value>
 </Parameter>
 <Parameter>
  <Name>ToolVelocity</Name>
  <Value>
   <Type>Double</Type><Data>50</Data>
  </Value>
 </Parameter>
 <Parameter>
  <Name>SlidingVelocity</Name>
  <Value>
   <Type>Double</Type><Data>0.003</Data>
  </Value>
 </Parameter>
</Parameters>

<Rules>
 <Parameter>
 <Name>SlidingVelocity</Name>
  <Rule>
   <Name>Basic Rule Example</Name>
   <Description>Approximation for the sliding velocity
by Chen/Kobayashi.</Description>
   <Condition>(FrictionCoefficient > 0.3) AND (ToolVelocity < 20) AND
              ((SlidingVelocity < 0.0005*ToolVelocity*0.9) OR
              (SlidingVelocity > 0.0005*ToolVelocity*1.1))</Condition>
   <Recommendation>0.0005*ToolVelocity</Recommendation>
  </Rule>
 </Parameter>
</Rules>
</RuleEngine>
```

Appendix B: CD

This CD contains the source code for the RuleEngine and the Qt example, as well as the Windows executable for the latter. It also provides the program RuleChecker—a command-line tool, based on the rule engine classes—that can be used to check the integrity of rule sets.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich meine Diplomarbeit "Recommending Values for Parameter Sets in Simulation Applications" selbständig ohne fremde Hilfe angefertigt habe und daß ich alle von anderen Autoren wörtlich übernommenen Stellen, wie auch die sich an die Gedanken anderer Autoren eng anlehnenden Ausführungen meiner Arbeit besonders gekennzeichnet und die Quellen nach den mir vom Prüfungsamt angegebenen Richtlinien zitiert habe.

Hamburg, den 14.01.2005

(Unterschrift)