ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Faculty of Engineering, Computer Science and Psychology
Databases and Information Systems Department

A Generic Engine Allowing an Automatic Evaluation of Data Collection Instruments

Bachelor's thesis at Universität Ulm

Submitted by:
Pascal Kühner
pascal.kuehner@uni-ulm.de

Reviewer:
Prof. Dr. Manfred Reichert

Supervisor:
Johannes Schobel

2018

Version from October 2, 2018

Composition: PDF-LaTeX 2_ε

Abstract

Most of the time, clinical data is still collected using paper-based questionnaires, even though this traditional approach has several limitations compared to electronic data collection. For example, each questionnaire has to be printed and handed out to each test person. Furthermore, the digitization and analysis of collected data is very time-consuming and labor-intensive. The QuestionSys project aims to solve most of these problems by providing a sophisticated framework. The latter supports the complete digital data collection process, including the creation, deployment, execution, analysis and archiving of the questionnaires. At different times, collected data of a questionnaire has to be analyzed and evaluated. For example, collected data already has to be evaluated during execution of the questionnaire, in order to determine the further course of the questionnaire. Furthermore, it has to be evaluated after the questionnaire is completed. In order to make evaluation of the data possible within the QuestionSys framework, questionnaires can contain rules, which have to be evaluated. For the purpose of evaluating these rules, a rule evaluation engine is developed in the course of this thesis. The main focus is to develop an engine, which eliminates different problems that come along with evaluation of expressions during execution time. Furthermore, this engine should be easily extensible and has to be usable on different platforms.

Acknowledgment

First of all, i want to thank my family and friends for their support and help during the time I wrote this thesis.

I particularly thank my supervisor Johannes Schobel, who always took his time, helped me and contributed with helpful suggestions when I had any questions.

Finally, i want to thank my brother Thomas Kühner for his helpful advice. I also want to thank him and my fellow student Lukas for their effort and spent time while proof-reading this thesis.

Contents

1

Introduction

As of today, clinical data collection is still mostly realized with the use of paper-based questionnaires. There are several problems with paper-based data collection instruments. With this approach, every questionnaire has to be printed and handed out to the patient. This consumes a considerable amount of resources, which makes scaling surveys difficult. Furthermore, analyzing and evaluating paper-based questionnaires is labor-intensive and time-consuming [1]. After data has been collected, it has to be digitized and entered into an electronic database, in order to perform sophisticated computer-assisted analysis on the data [2]. However, the digitization of paper-based data might result in faulty data [3]. These problems could be solved by electronic and mobile data collection instruments. Creating, updating and deploying digital questionnaires is more time-efficient, since the questionnaires are distributed via the Internet. Furthermore, faulty and erroneous data can already be prevented by input validation of the software. Thereby, digitization is also completely unnecessary, as the data is already in a digital format. Digital questionnaires have additional advantages, for example, in the context of clinical data collection. A study indicates that the immediate evaluation and data availability of mobile patient questionnaires represents a big advantage over paper-based questionnaires, where the analyzed data is not immediately available [4].

The main goal of the QuestionSys project is to build a framework to simplify data collection with questionnaires by digitization of the entire process. More specifically, the QuestionSys project focuses on mobile-data collection. Since Internet usage with mobile devices exceeded desktop usage for the first time in 2016, mobile devices are obviously a big market for such a technology [5]. The QuestionSys framework uses a

process-driven approach and allows for creating, deploying, executing, evaluating, and archiving digital questionnaires [1].

Within the QuestionSys framework collected data of a questionnaire has to be evaluated at different points. For example, in order to determine the further course of the questionnaire, collected data has to be evaluated during the execution of a questionnaire. Additionally, the data obviously should be evaluated after a questionnaire is finished and data has been collected. The framework uses rules, which contain a boolean expression, in order to enable evaluation of the data. In order to evaluate these rules, a generic rule evaluation engine, that can be used as a mobile and desktop application, is needed. Such an engine will be developed in the course of this thesis.

1.1 Objective

The goal of this thesis is to design and develop a generic rule evaluation engine in the context of the QuestionSys project. This engine should enable users to evaluate rules, which are part of questionnaires defined with the QuestionSys configurator application. These rules contain a condition, which is a boolean expression, that has to be evaluated. As a result of an evaluation the engine should indicate if a rule was evaluated to `true` or `false` in the context of the submitted results. This engine will be used in different parts of the QuestionSys framework. Thus, it should be able to work on different platforms (i.e., mobile phones, browsers). Additionally, the engine should be easily extensible and adjustable to new requirements, as the QuestionSys project is constantly evolving.

1.2 Structure of the Thesis

Chapter 2 deals with the QuestionSys project. At first, the project itself is introduced. Then, the rule engine's purpose within this project is explained in Section 2.2. Furthermore, Section 2.3 explains the structure of rules and results of the QuestionSys framework. Chapter 3 deals with requirements for the software. At that, functional and non-functional requirements are imposed on the rule engine to be developed in

the course of this thesis. Further, possible problems are introduced in Section 3.2. In Section 3.3, three different evaluation frameworks, that might be used as a solution for these problems, are introduced and compared. The following Chapter 4 illustrates the main concept of the rule engine. This includes a general architecture of the software. Afterwards, Chapter 5 deals with the implementation of the rule engine. Thus, different components are explained in detail. Then, it is explained how the engine is used and integrated into another software. Lastly, the results are summarized in Chapter 6. Thereby, the actual implementation is compared with the requirements that were imposed in Chapter 3. Furthermore, an outlook on what could be added and changed in the future is presented in Section 6.2.

2

Background

2.1 QuestionSys Framework

This chapter explains the QuestionSys project and elaborates the purpose of the rule engine within the framework.

The QuestionSys framework is a project of the University of Ulm, that launched in 2013. The goal of the project is to simplify data collection with questionnaires. As a result, a generic questionnaire framework for mobile data collection has been developed. As of now, most questionnaires are still written and evaluated by hand on paper, creating big workloads for psychologists. The QuestionSys project however takes a process-driven approach for defining, validating, deploying, processing and analyzing digital questionnaires [1].

Digital questionnaires go through a life cycle with five phases, which are all covered by the QuestionSys Framework. At first, questionnaires have to be created. Then, they can be deployed to different devices. Consequently, the deployed questionnaires can be executed on the devices. Collected data will then be analyzed and evaluated in real-time, after a questionnaire is finished. In the last phase, the collected data and the questionnaire are managed, versioned and archived [6].

2.1.1 Components of the Framework

A QuestionSys questionnaire is a executable process model, that can be executed on mobile devices with a process engine. The framework provides different components

for each phase a digital questionnaire goes through. These components are the Server, Configurator and Client.

Server The server stores deployed questionnaires and distributes them to the clients. Additionally, collected data, that originates from executed questionnaires on clients, is stored on the server and available for evaluation and analysis.

Configurator The configurator is an application, that is used to create questionnaires. Finished questionnaires are mapped to a process model and can be deployed to the server. Such a process model also contains all defined rules.

Client The client executes process models of questionnaires created with the configurator. This component uses the rule engine, in order to determine the further course of the questionnaire considering the collected data. After a questionnaire is complete, collected data of finished questionnaires can be stored on the server for further evaluation and analysis.

2.2 The Rule Engine

The rule engine is used in two phases of the questionnaire life cycle. First, it is used to determine the course of the questionnaire or respectively the next node in the process model, that is representing the questionnaire. Paper-based questionnaires might, for example, have control structures stating that one should continue on page `x` if one is 18 years or younger, otherwise one should continue on page `y`. These steps should be automatized in electronic questionnaires representing paper-based data collection instruments [7]. In order to represent such structures in the process model, XOR gateways are used [8]. Each XOR gateway has a list of branches, of whom each one has its own condition. A branch determines the next node in the process, if the condition is evaluated to `true`. Therefore, the rule engine will be used to evaluate these branches during the execution of a process model.

Furthermore, the rule engine will be used during the analysis phase. Creators of questionnaires can specify rules, that will be evaluated after data has been collected for the questionnaire. Such a rule also has a condition, which has to be evaluated by the rule engine. Additionally, headlines and descriptions are specified for both possible cases the condition can be evaluated to. These two cases are `true` and `false`. The task of the rule engine is to evaluate these rules and to present the results.

2.3 Structure of Rules and Results

Rule and branch objects, as well as other structures within the QuestionSys project are defined in the QuestionSys model. Both rules and branches of an XOR gateway contain two essential properties for rule evaluation. The first part is the condition that has to be evaluated by the rule engine. Such a condition is a boolean expression. It can contain variables starting with `$`, function calls, brackets and constant values, as well as mathematical and boolean operators.

As an example, a condition might look like this:

```
($age > 18 && $takesDrugs) || sum($mood_a, $mood_b, $mood_c) > 5
```

This condition contains the variables `$age`, `$drinksAlcohol`, `$mood_a`, `$mood_b` and `$mood_c`, the function `sum`, the brackets `()`, the constant value `5`, the boolean operators `||`, `&&`, `==` and the mathematical operator `>`.

A major problem of evaluating a rule is that all variables need to be associated with their actual values during execution time. At the creation time of a questionnaire the actual values are obviously unknown, but one can already define what answers of a questionnaire will map to which variable of the rule. To support this level of indirection, the property `variablesMapping` is used. It contains all the information, which is required to link collected data to variables of the rule during execution time. An object that performs the linking is called the context of a rule.

In order to understand `variablesMapping`, the result object has to be explained first. A result object stores the collected data of one finished questionnaire. This, in turn,

is achieved by storing key-value pairs, with the id of a question as the key and a list containing the data. This list contains multiple iterations of an answer to the same question, since the same question can be asked multiple times during a questionnaire. One iteration of an answer has the properties `iteration` (i.e., position of this iteration in the list or respectively the questionnaire), a `timestamp`, and `value`. This property is a list that differs for different types of questions. It only contains the specific value of an answer, if the question only allows a single user-generated answer (i.e., the person completing the questionnaire came up with the answer). For single or multiple choice questions, it contains a single object with key-value pairs that indicate if the choice associated with the key is `true` (i.e., was selected) or `false` (i.e., was not selected). For matrix questions, which are two-dimensional single or multiple choice questions, `value` contains two of these objects. The first contains all choices for the row and the second contains all choices for the column. Listing 2.1 shows a result object with a single or multiple choice question that has the id `-0123` and two possible choices `drinksAlcohol` and `consumesDrugs`.

```
{
        "results" : {
                "-0123" : [
                        {
                                "iteration" : 0,
                                "timestamp" : 123445,
                                "value" : [
                                        {
                                                "drinksAlcohol" : true,
                                                "consumesDrugs" : false,
                                        }
                                ]
                        }
                ]
        }
}
```

Listing 2.1: Simple result object containing data for a single or multiple choice question with the id `-0123`

The `variablesMapping` of a rule or branch is a list of objects (i.e, the variables) containing the properties `variableName` (i.e., identifier for the variable), `questionId` (i.e., the id of a question in the result object) and `value`. A variable always points to the answer in result, which is stored by the `questionId` of the variable. If an answer contains multiple iterations, the variable points to all of these and the implementation will have to determine how such variables are treated. The property `value` is strongly related to the property `value` in a result object. If it is empty, the variable points to the specific value (i.e., a direct user-generated answer) in `value` of an iteration of the corresponding answer, as it is visualized in Figure 2.1.

Figure 2.1: Variable points to user-generated answer in the result

Otherwise, if `value` contains a single string, it points to a single choice of a single or multiple choice question, as it is illustrated in Figure 2.2.

Figure 2.2: Variable points to choice of single or multiple choice question in the result

Lastly, if `value` contains two strings, the first points to the row and the second to the column of a matrix question. This is also demonstrated in Figure 2.3.

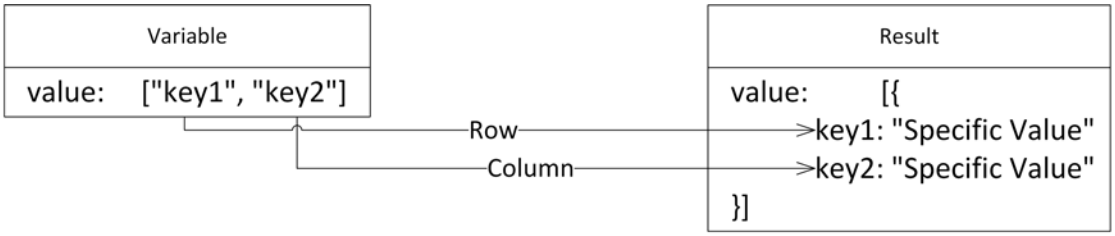

Figure 2.3: Variable points to row and column of matrix question in the result

A `variablesMapping` that maps onto the choice for the value `drinksAlcohol` of Listing 2.1 is shown in Listing 2.2. The variable `$drinksAlcohol` should consequently have the value `true`.

```
{
        variableName : "$drinksAlcohol",
        questionId : "-0123",
        value : [
                "drinksAlcohol",
        ]
}
```

Listing 2.2: Simple `variablesMapping` object mapping to the choice with the key `drinksAlcohol` of a single or multiple choice question with the id `-0123`

3

Analysis

In this chapter the requirements for the software are discussed. At first, the requirements for the software are imposed. Then, possible problems with requirements, which might be difficult to fulfill, are introduced. Lastly, different evaluation frameworks, that should eliminate these problems, are discussed.

3.1 Requirements

Taking its purpose as a part of the QuestionSys framework into consideration, the engine to be developed in the context of this thesis has to deal with the following requirements. The latter are divided into functional requirements, which are addressing the core functionality of the engine, and non functional requirements addressing the implementation and design of the software.

3.1.1 Functional Requirements

FR1 (Evaluate Rules):
The engine must correctly evaluate rules from questionnaires created by the QuestionSys configurator. The supported rules should have the structure of rule objects from the QuestionSys model.

FR2 (Evaluate XOR-Branches):
The engine must correctly evaluate branches of XOR gateways from questionnaires created by the QuestionSys configurator component. The supported branches of

XOR gateways should have the structure of branch objects from the QuestionSys model.

FR3 (Add Results of finished Questionnaires):

The rule engine must provide functionality to add context data from finished questionnaires to the engine. These results have to be correctly linked to the indicated rules and branches.

FR4 (Provide Custom Functions):

The engine should provide options to add custom functions, which then can be called during the evaluation of rules. There should be pre-defined functions of the engine itself and the possibility to add new functions when initializing the engine.

FR5 (Export Pre-Defined Functions):

It should be possible to export all pre-defined functions, which can be called during evaluation. Furthermore, the possibility to add new descriptions during run time should be present. The export format of a description should contain the name of the function, examples and explanations. Additionally, descriptions should be grouped by their purpose (e. g Math, Util, String functions).

FR6 (Present Finished Evaluations)

After one or more rules or branches have been evaluated, the engine should return objects that contain information about the evaluation process. This includes a flag indicating if the rule or branch was evaluated successfully or errors occurred. In addition information about the rule or branch, as well as the value that was evaluated should be present.

FR7 (Injection Safety):

Code that is encoded in conditions of rules, branches or the submitted results must not be executed by the engine during the evaluation of these. Solely the functions and operators which are provided by the rule engine should be used.

FR8 (Indicate Errors during Evaluation):

An error should be thrown when an evaluation fails because of an error. A detailed explanation in human-readable form should be provided.

3.1.2 Non Functional Requirements

NFR1 (Multi-Platform Support):

The engine must be developed as a NodeJS [1] module. Thereby TypeScript[2] should be used as the programming language, in order to deploy the engine to different platforms. Further, the developed module should be installable via npm[3].

NFR2 (Maintainability):

The developed code should be easily maintainable. Thus, it has to be properly documented. Additionally, all variables should have expressive names. Furthermore, the code should be structured and separated into different fields of duties.

NFR3 (Extensibility):

The developed engine has to be extensible. This implies that new requirements or a rework of old ones do not affect the architecture of the engine as a whole, but rather isolated parts. Furthermore, small changes in the overall data model should not affect the core architecture of the software developed in this thesis.

NFR4 (Stability):

The engine should not crash if errors occur. Thus, many sources of errors should be eliminated and the code should be typed as much as possible. Further, errors should be caught and properly handled.

NFR5 (Testing):

All parts of the engine must be properly tested with line coverage of 85% or more. Especially, wrong input and behavior in case of error should be tested.

3.2 Problems

This section deals with problems that have to be considered during development of the engine.

[1] `https://nodejs.org/en/`
[2] `https://www.typescriptlang.org/`
[3] `https://www.npmjs.com/`

3.2.1 Code Injection Threats

Since the rule engine will evaluate boolean expressions encoded as strings from external sources, wrong handling and execution of these can be a severe security issue because of malicious code injections. In the following, a definition of code injections from the Open Web Application Security Project (OWASP) is provided. [9].

Definition 1. *Code Injection is the general term for attack types which consist of injecting code that is then interpreted/executed by the application. This type of attack exploits poor handling of untrusted data.*

As an example a result of a questionnaire could contain code in a person's name and throw an error in the engine like this, if the expression in the condition was executed without further measures:

```
condition: "name == 'John'"
name: "throw new Error()"
```

Because the rule engine deals with private data and in order to restrict access to the machine the rule engine is running on, code injections must be dealt with.

3.2.2 Custom Functions

A very important feature of the rule engine is the possibility to use custom functions during the evaluation. These functions should provide enhanced features for rule evaluation, since it is impossible to directly execute code. Because of security issues, these functions will have to be pre-defined by the engine or added dynamically for a specific questionnaire. Additionally, the QuestionSys configurator must have information about the functions in order to properly support its users.

3.2.3 Expression Evaluation

In order to eliminate the code injection issues, the rule engine needs a safe and powerful environment for the evaluation of boolean expressions. JavaScript already offers this

functionality with the `eval` function, but the function simply executes code, encoded as a string and is thus not suited for the rule engine. Therefore, an external expression evaluation framework will have to be used by the engine.

3.3 Evaluation Frameworks

A simple solution for expression evaluation would be to use JavaScript's `eval` function (i.e. [10]), since it is simply executing code, that is encoded as a string. However, this would allow for code injections. Therefore, three different TypeScript/JavaScript frameworks to use for expression evaluation are compared in this section. For this purpose, the example from Listing 3.1 will be evaluated using each framework. The example shows a typical condition and context the engine has to evaluate.

```
1 conditionString =
2     "($age > 18 && $drinksAlcohol) || sum($mood_a, $mood_b, $mood_c) > 15";
3 context = {
4     $age: 18,
5     $drinksAlcohol: true,
6     $mood_a: 3,
7     $mood_b: 7,
8     $mood_c: 5
9 };
```

Listing 3.1: Example for a condition and context to be evaluated. With the given context, the rule should evaluate to `false`

The points of emphasis are the way the frameworks serve the purpose of evaluating expressions, whilst being safe regarding code injections. Furthermore, the possibility to add and use custom functions will be analyzed. For this purpose, the function `sum` from Listing 3.2, that is adding up all submitted values, will be added to each framework before the example is evaluated.

```
sum = new function () {
        let result = 0;
        for (let i = 0; i < arguments.length; i++) {
                result += arguments[i];
        }
        return result;
};
```

Listing 3.2: The function `sum` that sums up all submitted values

Lastly, specific advantages and disadvantages of each framework are discussed.

3.3.1 Jexl

Jexl is an expression parser and evaluator written in JavaScript [11]. With the ability to evaluate expressions, the possibility to add custom functions and code injection safety, the framework provides all the functionality that is required.

One could instantiate Jexl and then add the function `sum` like this:

```
let jexl = require('Jexl');
let sum = function () {...};
jexl.addTransform('sum', sum);
```

Listing 3.3: Instantiating and adding functions to Jexl

After that, the example from Listing 3.1 can be parsed and evaluated in the way it is illustrated in Listing 3.4. Note that functions in Jexl can't be called without the pipe symbol `|`, which signals that the variable in front of the pipe is the first parameter of the function call. This is very uncommon and makes it rather difficult to automatically parse the example's format to Jexl's internal format of function calls. In addition, Jexl doesn't support a `$` character as the start of variable names, which is the first character of variable names in conditions of the QuestionSys framework.

Other advantages of Jexl are the possibility to add custom unary operators and a powerful query language for arrays.

```
conditionString =
        "(age > 18 && drinksAlcohol) || mood_a|sum(mood_b, mood_c) > 15";
context = {
        age: 27,
        drinksAlcohol: false,
        mood_a: 3,
        mood_b: 7,
        mood_c: 5
};
jexl.eval(conditionString, context);
};
```

Listing 3.4: Condition string and context parsed into correct format and evaluated with Jexl

3.3.2 vm2

vm2 is a sandbox module for NodeJs, that is specifically designed for running untrusted code [12]. The idea with vm2 would be to use JavaScript's built-in `eval` function to evaluate rules, with vm2's sandbox eliminating possible code injection threats. In vm2 one can specify exactly which node modules, objects or functions are usable. Thus, vm2 creates a safe environment to use the JavaScript `eval` function, that runs JavaScript code encoded as a string.

Listing 3.5 shows the creation of an instance of vm2 as well as the adding of the function `sum`.

```
let result = {};
const vm = new NodeVM({
        sandbox: {result}
});
let sum = function () {...};
vm.freeze(sum, 'sum'); //Adds Sum Function to vm2
```

Listing 3.5: Instantiating vm2 and adding the function `sum`

In order to have access to the result of an evaluation both inside and outside of vm2, the object `result`, where the results should be stored, is submitted to vm2 when it is instantiated. If objects have to be added to vm2 during run time, `freeze` can be used. Additionally, with `freeze` the objects are read-only and can not be modified inside vm2's sandbox.

In vm2 you could evaluate the example from Listing 3.1 like this:

```
conditionString = "(context.$age > 18 && context.$drinksAlcohol) ||
        sum($mood_a, $mood_b, $mood_c) > 15";
context = {
        $age: 27,
        $drinksAlcohol: false,
        $mood_a: 3,
        $mood_b: 7,
        $mood_c: 5
};

vm.freeze(context, "context"); //Adds context to vm2
vm.run(`eval(evaluatedValue = ${conditionString})`);
```

Listing 3.6: Condition string and context parsed into correct format for vm2 and evaluated with vm2

In detail, only the variable references in the condition string were changed to point towards the context variable, which is an easy-to-automate procedure.

In addition, vm2 is the most powerful one of the discussed frameworks, because it can execute plain JavaScript code. However, it is not as simple to use as the others and misuse might lead to security threats, due to the fact that JavaScript's `eval` function is used within the sandbox.

3.3.3 expr-eval

expr-eval is a mathematical expression parser and evaluator, specifically designed as a safe alternative for JavaScript's `eval` function [13].

In the following it is illustrated how expr-eval can be instantiated and how a function can be added.

```
let sum = function(){...};
parser = new Parser();
parser.functions['sum'] = sum;
```

Listing 3.7: Instantiating expr-eval and adding the function `sum`

As one can see, it is very easy to add custom functions to expr-eval. In addition, there are already many pre-defined functions and mathematical operators available.

In expr-eval you could evaluate the example from Listing 3.1 like this:

```
conditionString =
    "($age > 18 and $drinksAlcohol) or sum($mood_a, $mood_b, $mood_c) > 15";
context = {
    $age: 27,
    $drinksAlcohol: false,
    $mood_a: 3,
    $mood_b: 7,
    $mood_c: 5
};
parser.evaluate(conditionString, context);
```

Listing 3.8: Condition string and context parsed into correct format for expr-eval and evaluated with expr-eval

Unfortunately, since expr-eval has different uses for `!` and `||`, logical operators have to be parsed into `and`, `or` and `not`. Another disadvantage of the framework is the missing possibility to access arrays in JavaScript's usual way (i.e., with the operators `[]`).

3.3.4 Comparison

In the following, the frameworks will be compared regarding their functionality, security, parsing work needed for the condition and handling of custom functions.

Functionality The most powerful of the frameworks is vm2, as it can execute plain JavaScript code, unlike the expr-eval and Jexl. Yet, all the functionality of all three frameworks is sufficient for the demanded tasks.

Security vm2's use of `eval` comes along with security issues (i.e code injection threats), when it is used wrongly. In contrary, Jexl and expr-eval eliminate these completely.

Parsing the Condition Parsing the condition string into the correct format for Jexl is difficult and complex because of the design of function calls. In contrast, it is simple for vm2 and expr-eval.

Custom Functions Jexl has severe disadvantages regarding the handling of custom functions, as these can only be called with the pipe operator. However, with vm2 and expr-eval, functions can be used exactly as it is already done within the conditions of rules and branches of the QuestionSys framework.

After all, Jexl seems to be the least suitable of the three frameworks, because of it's disadvantages regarding custom functions, which are very important the rule engine. The differences between expr-eval and vm2 are not that significant. Although vm2 is more powerful, since `eval` can be used, expr-eval is sufficient for the demanded tasks. Furthermore, expr-eval eliminates the security issues that come along with vm2. Ultimately, expr-eval will be used for expression evaluation, because of the simple handling of custom functions and lack of security issues.

4

Concept

This chapter addresses the concept of the rule engine. At first, the main concept is explained. Then, the architecture of the rule engine is introduced. Note that, rules and branches are not distinguished in the whole concept and thus both will be referred to with rule.

4.1 Main Concept

The main concept is divided into two parts. First, the evaluation process is explained in general. Second, a way to deal with custom and pre-defined functions is introduced.

4.1.1 Evaluation Process

The main purpose of the rule engine is to evaluate rules of questionnaires created with the QuestionSys configurator application. When a rule is evaluated, the rule engine follows the evaluation process that is illustrated in Figure 4.1

In the first step of the process, the rules that should be evaluated have to be submitted to the rule engine. Since rules and results are independent objects, the results (i.e, the collected data of completed questionnaires), have to be linked to the corresponding rules. Therefore, a context is created for each rule when it is submitted to the rule engine. Such a context represents the collected results for the variables used within the rule, by linking the specific value of an answer in a result to the corresponding variable. However, the contexts are not yet "filled" with values at this point of the process. Consequently, results

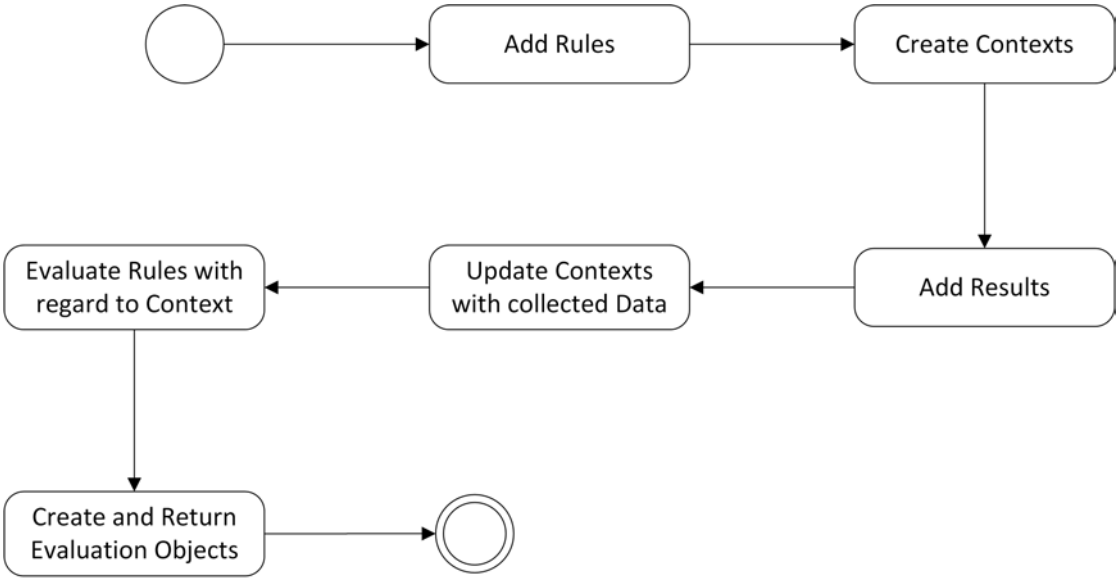

Figure 4.1: Evaluation Process

have to be added to the engine. The contexts of the rules are then updated and "filled" with the collected answers of the results. Figure 4.2 shows an example for a context and illustrates, in a simplified way (i.e., without `value` and `iteration` in the result), how variables are "linked" to the corresponding answers by using the mapping of a rule.

Next, these rules are evaluated with regard to their context. Thereby, the conditions are first parsed into the correct format for the rule engine. Then, the `expr-eval` framework, that was discussed in Section 3.3.3 is used for evaluating the conditions.

Finally, after a rule has been evaluated, the results of evaluations have to be presented with additional information, as stated in *FR6*. Therefore, evaluation objects are created and returned. These indicate the value a rule was evaluated to and contain further information about the rule and evaluation process.

4.1.2 Custom Functions

As stated in *FR4*, the rule engine has to offer pre-defined functions, in order to provide enhanced possibilities for evaluations. These functions may be part of a condition.

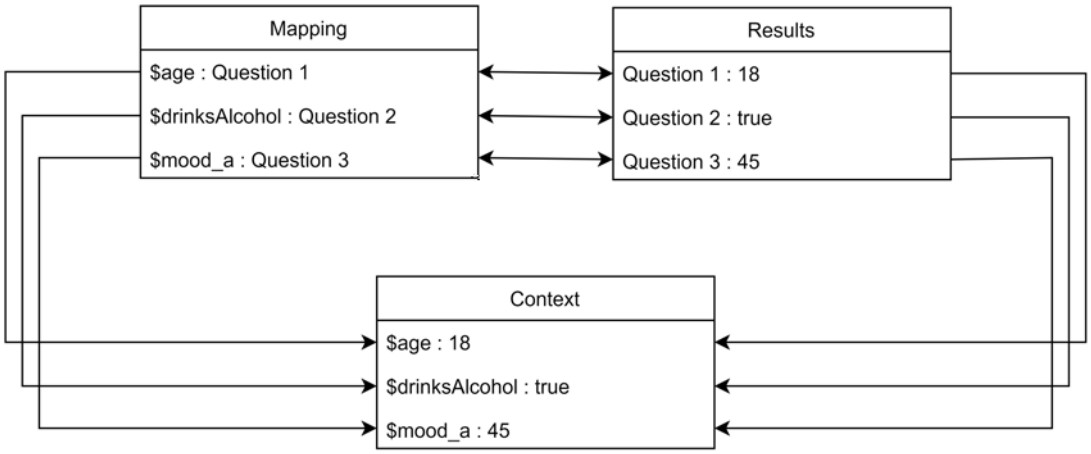

Figure 4.2: Linking rules and results in a context

Thus, the engine contains a repository that manages these pre-defined functions. Since the rule engine cannot supply fitting functions for every problem, it is also possible to add custom pre-defined functions during run time. These functions can be used by `expr-eval` as well, when rules are evaluated. As described in *FR5*, the repository also contains descriptions of the provided functions, that will be used by the QuestionSys configurator application.

4.2 Architecture

This section deals with the general architecture of the rule engine, that is extracted from the main concept.

In order to create a maintainable (*NFR2*) and extensible (*NFR3*) software component, the rule engine is divided into four components, as shown in Figure 4.3. These components are `Manager`, `Functions`, `Model` and `Evaluator`.

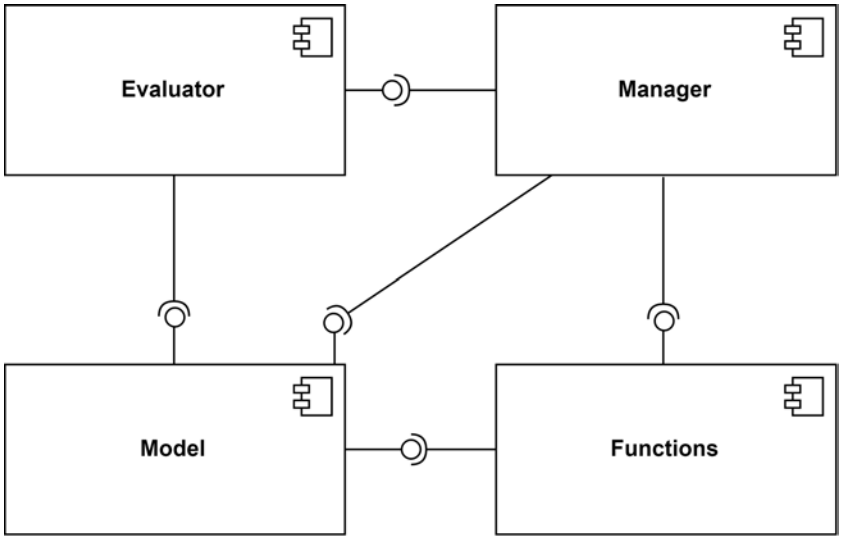

Figure 4.3: Architecture of the rule engine

4.2.1 Manager

The `Manager` is the entry point and the only component of the engine, which is used for communicating with external applications. It provides functionality to add and evaluate rules and results from the QuestionSys framework. Additionally, the whole evaluation process is managed by this component. Thus, it makes use of all other components. Additionally, it enables external applications to communicate with the `Functions` component, in order to enable the QuestionSys configurator application to access the descriptions of functions.

4.2.2 Functions

The `Functions` component is used for dealing with custom functions. It supplies predefined functions to the rule engine. However, since the rule engine cannot supply fitting functions for every problem, the component allows for adding custom functions defined and implemented by third-party developers.

Furthermore, descriptions of functions, that should help creators of questionnaires at creating new rules, are managed by this component.

4.2.3 Evaluator

The `Evaluator` component evaluates the conditions of rules. For this purpose it parses the conditions into the correct format used by the rule engine. Then, conditions are evaluated with regard to the corresponding context object of the rule. For evaluation of the conditions, the framework `expr-eval`, that was discussed in Section 3.3.3, is used. The component gets access to all functions from the `Functions` component by the `Manager` and makes them available to `expr-eval`, so that they can be used during the evaluation process.

4.2.4 Model

The `Model` component consists of different classes, interfaces and builders used by the engine. It is for example used to create contexts for rules. Furthermore, after a rule was evaluated, this component is used to create evaluation objects.

4.2.5 Communication between Components

In order to the illustrate relations between components, the communication of components during the evaluation process is shown in Figure 4.4. As one can see, the communication to the external application exclusively takes place via the `Manager`. Furthermore, the single components, outside of the `Manager`, do not communicate with each other.

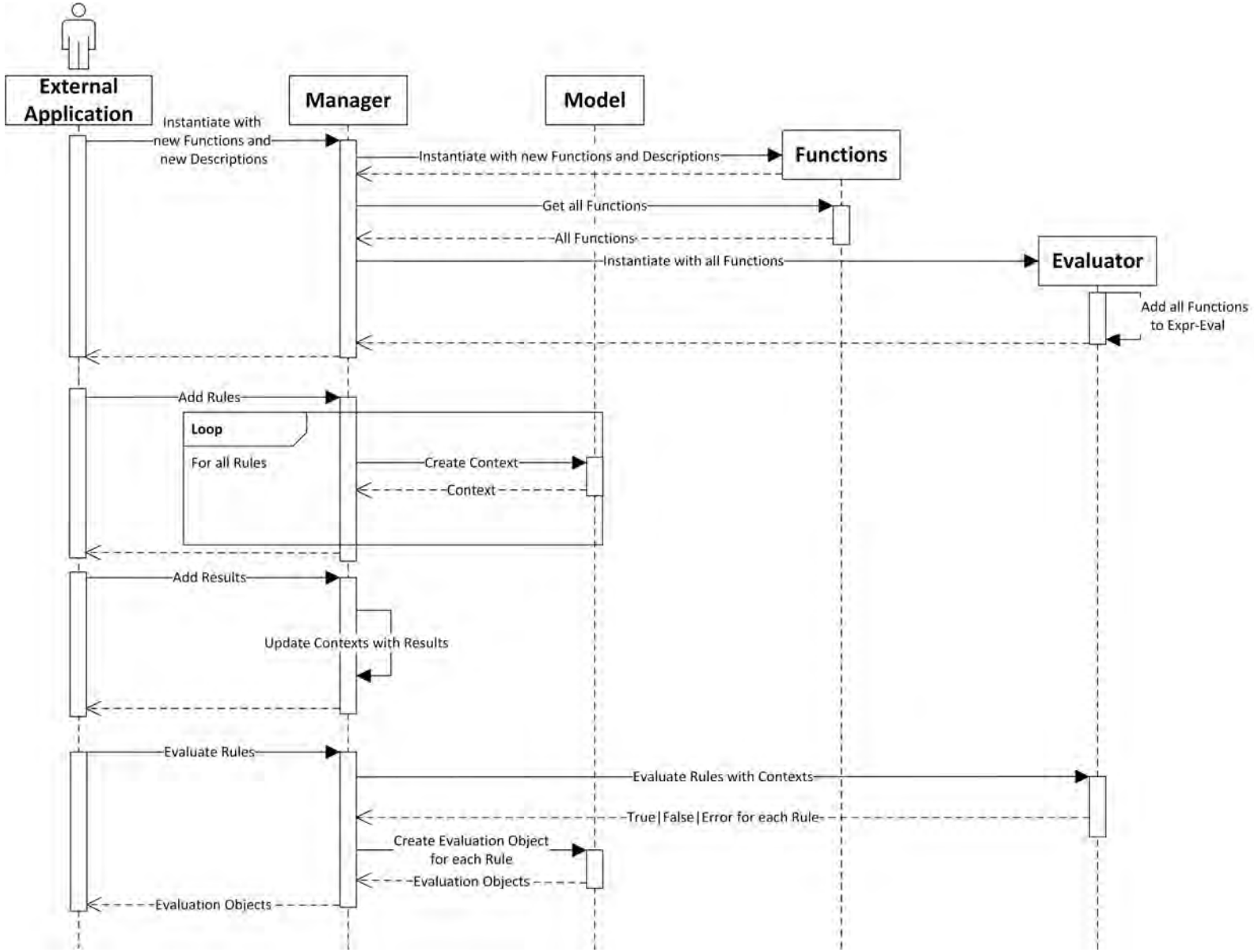

Figure 4.4: Communication between components

5 Implementation

This chapter deals with the implementation of the rule engine. First, some details about the implementation are explained. Second, the implementation of selected components is introduced and illustrated. Then, it will be explained how the engine is used and integrated into another application.

5.1 Implementation Details

This section introduces the package manager npm, as well as details regarding the testing of the rule engine.

5.1.1 npm

npm is a package manager for JavaScript, that contains over 600.000 public packages. It can be used for sharing packages of code, managing multiple versions of code and code dependencies, as well as integrating other packages into the developed package or software [14]. However, npm can also be used to manage private packages, that are not shared with the community. npm can for example install packages from private GitHub repositories, that represent a npm package [15]. Such a package contains a `package.json` file with all relevant meta information [16]. This file also contains the software's `dependencies` to other npm packages and the `developerDependencies` to packages, which are only needed during development.

npm is used to share packages of JavaScript code. However, in order to support TypeScript developers, type definition files can be part of a package. Such files contain

all relevant typings for corresponding JavaScript files. Thus, TypeScript developers can use these packages with full type functionality.

The rule engine, in turn, is developed as an npm package. The only dependency of the rule engine is the expr-eval framework, which is also an npm package. The QuestionSys model (i.e., the npm package containing types for rules, branches and other structures of the QuestionSys framework) is deliberately not included in `dependencies`, in order to avoid multiple dependencies and possibly mismatching versions of the QuestionSys model package in an application and the rule engine. Since an application that works with the rule engine should also be depending on the model, the rule engine can simply use this version of the model package, without having its own dependency. Thus, the rule engine uses the QuestionSys model package, that is provided by the application. Yet, if there should be the need for a rule engine package with an included dependency on the model, this could be achieved with a second package, that has dependencies on the rule engine and the model. Figure 5.1 illustrates this issue.

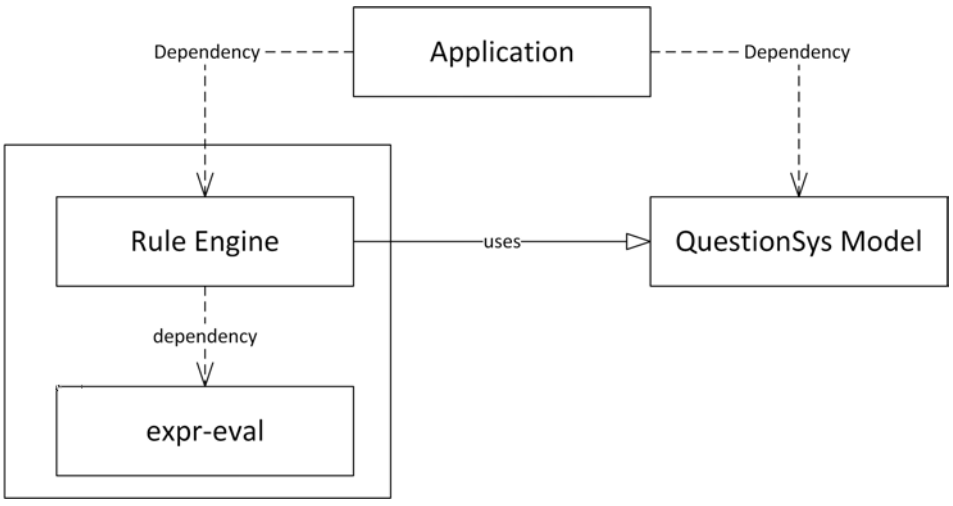

Figure 5.1: Dependencies between the rule engine and the QuestionSys model

5.1.2 Testing

The rule engine is tested with the frameworks mocha[1] and chai[2], that allow for unit-testing of NodeJS applications. All components and classes are tested on their own. Additionally, the functionality of the rule engine is also tested, when all components are integrated into the software. The npm package nyc[3], which can detect test coverage, implies, that the rule engine's tests provide a line coverage of 89%.

5.2 Implementation of Selected Components

In this chapter the implementation of different components and important parts of the rule engine will be explained in detail.

5.2.1 Model

The `Model` component consists of different structures and builders used by the rule engine.

Internal Rules and Branches

The QuestionSys model has two independent objects rule and branch, which represent rules and branches of XOR gateways. Yet, since both of these objects share the same essential properties for evaluation, which were already explained in Section 2.3, the engine internally uses a `BasicRule` object, that represents both of these properties. The properties of a `BasicRule` are:

conditionString The condition of a rule or branch encoded as a string.

variablesMapping The `variablesMapping` property of a rule or branch.

[1] https://www.npmjs.com/package/mocha
[2] https://www.npmjs.com/package/chai
[3] https://www.npmjs.com/package/nyc

Note, that `BasicRule` does not contain all overlapping properties of rules and branches, because both branches and rules share the property `name`, which isn't part of `BasicRule`. Since this property is not necessary for evaluation, it would be more complex to support new objects for evaluation, which might contain all necessary properties but not `name`. Thus, it would lower the extensibility of the rule engine. Yet, the properties of rules and branches are useful at a later stage and should not get lost. Hence, the objects `InternalBranch` and `InternalRule` extend `BasicRule` with the remaining properties of rules and branches. This structure is visualized in Figure 5.2. Thus, the `BasicRule` can be used as a generalized type for `InternalBranch` and `InternalRule` objects, as long as the additional information for the single objects are not relevant.

Another advantage is, that an `InternalRule` object is structurally identical with the respective rule object of the QuestionSys model (i.e, they share the same properties with the same type). The same situation applies for `InternalBranch` objects and the branch object of the QuestionSys model. Since TypeScript uses structural typing, different structures with the same properties are compatible, meaning that, for example, a rule object from the QuestionSys model can be used as `InternalRule` without any restrictions or type casts [17].

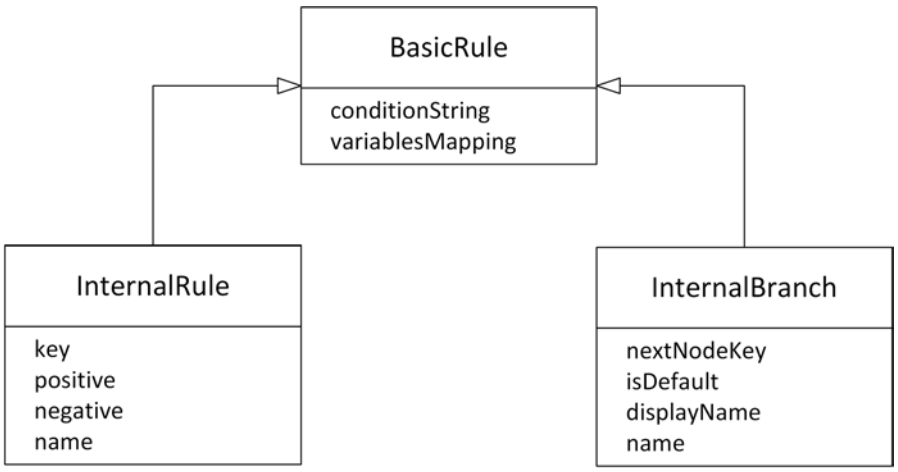

Figure 5.2: Relationships between `BasicRule`, `InternalRule` and `InternalBranch`

Results and Context

The rule engine uses a `Context` object to represent the context of a rule (i.e., an object linking variables to their actual value). The relevant properties of a `Context` object are:

variablesMapping	`variablesMapping` property of the corresponding `BasicRule` object.
variables	Maps variable names to their actual value. In detail, it is a key-value map, with variable names as the key and their value as value.

A `Context` must be initialized by submitting the variablesMapping of a `BasicRule` object. Results can be added to the context with the `addResult` function (i.e, Listing 5.1). Thereby, the `variables` property will be filled with the results (i.e., the variables will be associated with the value that is specified in the result). In order to achieve this, the function `fillVariable`, which associates the variable's name with its actual value, is called for each variable (i.e., object in the List `variablesMapping`). Yet, if `result` doesn't contain answers to the question with the `questionId` of a variable, `fillVariable` is not called for this variable.

```
1 addResult(result: Result) {
2         for (let variable of this.variablesMapping) { //For each variable
3                 let answers = results.result[variable.questionId];
4                 if (!isNullOrUndefined(answers)) {
5                 //If result contains answer to the question
6                         this.fillVariable(variable.questionId, variable.
                              variableName, variable.value, answers);
7                 }
8         }
9 }
```

Listing 5.1: The `addResult` function from the `Context` class

The function `fillVariable` (i.e, Listing 5.2) adds a key-value pair to `variables`, which maps `variableName` to the actual value of the variable. Therefore, it first creates the actual value for the variable out of the result with `createVariableValue` for each iteration of the question. If there is only one interation, the variable will be directly

associated with that value, otherwise the variable will be associated with a list of values (i.e., one for each iteration). In order to get a single iteration within a condition, the function `getIteration($variableName, iteration)` can be used.

```
1  private fillVariable(questionId: string, variableName: string, value: any,
       answers: any[]) {
2          let variableValue = [];
3          for (let answer of answers) { //For each iteration
4                  variableValue.push(this.createVariableValue(answer, value));
5          }
6          if (answers.length == 1) {
7                  variableValue = variableValue[0];
8          }
9          //Association of value with variableName
10         this.variables[variableName] = variableValue;
11 }
```

Listing 5.2: The `fillVariable` function from the `Context` class

The actual value of the variable is extracted with `createVariableValue` (i.e., Listing 5.3). The submitted property `answer` should be a single iteration of an answer to a question in a result object and `value` is the identically named property of a variable in the list `variablesMapping`. The function resolves the relationships between the submitted property `value` of the variable and the property `value` of the submitted `answer`, in the way it is described in Section 2.3. In detail, if `value` is empty, the specific value in `value` of the submitted `answer` is assigned to the variable's value. If it contains a single string (i.e., pointing to a single or multiple choice question), the string will be matched to the value of the corresponding choice in the `value` property of `answer`, which is then returned as the variable's value. Yet, if it is pointing to a matrix question (i.e., `value` contains two strings) the variable will be further divided into `row` and `column`. Then, the first key in `value` is matched with the row (i.e, the first object in `value` of the `answer`) in the same way as it is done for single or multiple choice questions. Equally, the second key is matched with the column (i.e, the second object in `value` of the `answer`). The properties `row` and `column` of a such a variable can

be accessed within a condition by using the functions `getRow($variableName)` and `getColumn($variableName)`.

```
1  private createVariableValue(answer: any, value: any): any {
2          ..
3          switch (value.length) {
4                  case 0: //Direct answer to question
5                  variable = answer.value[0];
6                  break;
7                  case 1: //Answer is choice of single or multiple choice
                       question
8                  let temp = answer.value[0];
9                  ..
10                 variable = temp[value[0]];
11                 ..
12                 case 2: //Answer to Matrix question
13                 let tempRow = answer.value[0];
14                 let tempCol = answer.value[1];
15                 variable = {};
16                 ..
17                 variable.row = tempRow[value[0]];
18                 variable.column = tempCol[value[1]];
19                 ..
20         }
21         return variable;
22 }
```

Listing 5.3: The `createVariableValue` function from the `Context` class

Evaluations

After a rule has been evaluated, the results of the evaluation are represented by an evaluation object. The `BasicEvaluation` is the most general evaluation object and only contains the essential properties, which are available after a `BasicRule` object has been evaluated. These properties are:

evaluationType Provides information about the evaluation process (i.e Success, FunctionError, ParserError, ValueError).

evaluatedValue This is the value, the rule was evaluated to (e.g., `true`, `false` or `Error`).

evaluationKind Unique identifier of the type of the evaluation object (either `Basic`, `Branch` or `Rule`).

Note that `evaluationType` contains very important information about the evaluation process. In detail, if it is `Error`, `FunctionError` or `ParserError` the evaluation failed during execution. In this case, `evaluatedValue` contains the error that occurred. However, `ValueError` signals that the evaluation finished without errors, but the type of `evaluatedValue` is not `boolean`. In that case, the rule must be corrupt in some way, since the engine focuses on `true`/`false` evaluations. `Success` means that the evaluation finished without errors and `evaluatedValue` is of the type `boolean`. Due to these different cases, the function `getValue`, that either returns the `boolean` value of `evaluatedValue` or throws the given error, is offered.

In order to supply evaluation objects with additional information of `InternalRule` and `InternalBranch` objects, two additional classes `BranchEvaluation` and `RuleEvaluation`, that extend `BasicEvaluation` with additional properties of the evaluated `InternalBranch` (i.e., with `name`, `isDefault`, `nextNodeKey` and `displayName`) or `InternalRule` (i.e., with `name`, `key`, `positive` and `negative`), are provided. The structure of evaluations can be seen in Figure 5.3.

The rule engine additionally provides an `EvaluationBuilder`, whose task is to simplify the process of creating an evaluation object. Hence, it selects the `evaluationType` for a submitted evaluated value with the function `getEvaluationType` (i.e., Listing 5.4) and instantiates `BasicEvaluation`, `RuleEvaluation` or `BranchEvaluation` objects with additional information of the submitted `BasicRule`, `InternalRule` or `InternalBranch` object. For this purpose, it provides the static function `buildEvaluation`. This function always instantiates the most specific evaluation object, that is possible. If for example an object with all properties of `InternalBranch` is submitted to `buildEvaluation`, it will return a `BranchEvaluation` object.

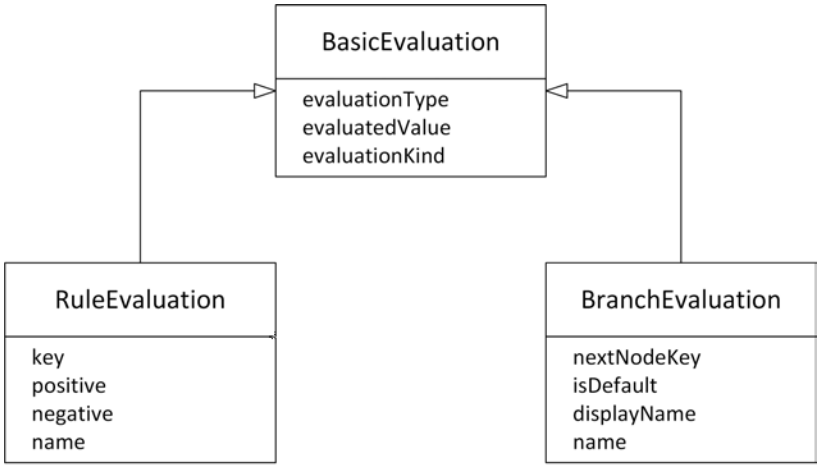

Figure 5.3: Relationships between `BasicEvaluation`, `RuleEvaluation` and `BranchEvaluation`

```
1  private static getEvaluationType(evaluatedValue: any): EvaluationType {
2          if (isBoolean(evaluatedValue)) {
3                  return EvaluationType.Success;
4          }
5          if (isFunctionError(evaluatedValue)) {
6                  return EvaluationType.FunctionError;
7          }
8          if (isError(evaluatedValue)) {
9                  return EvaluationType.ParserError;
10         }
11         return EvaluationType.ValueError;
12 }
```

Listing 5.4: The `getEvaluationType` function from the `EvaluationBuilder`

As an example, a simple `BasicEvaluation` object is shown in Listing 5.5

```
1  {
2                  evaluationType: "Success",
3                  evaluatedValue: true,
4                  evaluationKind: "Basic"
5  }
```

Listing 5.5: Simple `BasicEvaluation` Object

5.2.2 Functions

The `Functions` component consists of the `FunctionRepository`, the `FunctionDescriptions` JSON file and the `Functions` object containing pre-defined functions.

FunctionDescriptions As described in *FR5*, pre-defined functions should be exported, in order to help creators of questionnaires. The export format of a function is a `FunctionDescription`. The relevant properties of such a `FunctionDescription` are:

key	The name the function can be called by in a condition.
displayName	The name of the function, that will be presented to users of the QuestionSys configurator.
group	The group to which the function can be attributed to.
explanation	A detailed explanation of the function's purpose and usage.
example	An example showing how the function can be used in a condition.

As an example, a description for a `sumList` function that sums up all values of a submitted list can be seen in Listing 5.6.

```
description = {
        "key": "sumList",
        "displayName": "Sum of a List",
        "group": "Math",
        "explanation": "Sums all values of the submitted list",
        "example": "sumList($valsToSum)"
}
```

Listing 5.6: Simple `FunctionDescription` object

All pre-defined `FunctionDescription` objects are stored in a JSON File called `FunctionDescriptions`. In the JSON file, the descriptions are further grouped by their property `group` and stored by their property `key`, as one can, for example, see in Listing 5.7.

```
{
  "Math": {
    "sumList": {
          "key": "sumList",
          "displayName": "Sum of a List",
          "group": "Math",
          "explanation": "Sums all values of the submitted list",
          "example": "sumList($valsToSum)"
        },
        ...
  },
  "Util": {
        ...
  },
        ...
}
```

Listing 5.7: Example for `FunctionDescription` objects stored in the `Function-Desriptions` JSON file

Functions `Functions` on the other hand is a TypeScript object containing the engine's pre-defined functions. These are stored by the key, that is identifying the function also used to call it within a condition. An example for such an object can be seen in Listing 5.8.

```
functions = {
        "sumList": function (vals: number[]) {
                return vals.reduce((sum, currentValue) =>
                sum + currentValue);
        }, ...
}
```

Listing 5.8: Example of pre-defined functions stored in the `Functions` object

FunctionRepository Both the `FunctionDescriptions` file and the `Functions` object are combined in the `FunctionRepository`. The relevant properties of the `FunctionRepository` are:

functions Object that contains all pre-defined functions of the `Functions` object. Additionally, it can contain custom functions which were added during run time. The structure is the same as the structure of the `Functions` object.

descriptions Instance of the `FunctionDescriptions` JSON file. It can also contain descriptions of custom functions, that are added during run time.

The main purpose of the `FunctionRepository` is to manage all functions that can be used in a condition and their corresponding descriptions. Therefore, it allows for adding and deleting custom functions. Single functions can be added with `addFunction`, where also the `key` of the function has to be submitted. Multiple functions can be added in the structure of a `Functions` object with `addFunctions`. Obviously, `FunctionDescription` objects can also be added with `addDescription` (i.e., a single object) and `addDescriptions` (i.e., a list of objects). It has to be mentioned, that existing functions and descriptions will be overwritten by new functions and descriptions with the same key.

Since the descriptions of functions would be useless, without the possibility to access them, functionality to query `descriptions` is provided. In order to get all groups, whom at least one description is attributed to, as a list, the method `getAllGroups` has to be called. Then, `getDescriptions` can be used to get all `FunctionDescription` objects of one submitted group. If no group is submitted, all `FunctionDescription` objects are returned.

5.2.3 Evaluator

The `Evaluator` component consists of the `RuleEvaluator`, `ConditionStringParser` and `FunctionParser`.

ConditionStringParser The `ConditionStringParser` parses conditions of rules into the correct format for evaluation.

expr-eval, the framework that is used for evaluating expressions, only supports `and`, `or` and `not` as boolean operators. However, in the preferred format of conditions in the QuestionSys project `&&`, `||`, which is used for string concatenation in `expr-eval`, and `!`, that is used for calculating the factorial of a number, are used. Thus, the `ConditionStringParser` parses these operators into the format of `expr-eval` (i.e., it replaces `&&` with `and`, `||` with `or` and `!` with `not`). However, if an application already uses the correct format for conditions, but still wants to use the symbols that would be replaced, different options can be adjusted with `ConditionStringParserOptions`. This is an object containing three boolean values `and`, `not` and `or` that determine if the respective operators `&&`, `||` and `!` should be changed to the format of `expr-eval`. If no `ConditionStringParserOptions` are submitted or boolean values are missing, they default to true and the operators are automatically replaced. However, this is handled by the `RuleEngineManager`.

FunctionParser The `FunctionParser` should wrap pre-defined functions into a new safer function. The return values of functions called during run time could, for example, be `undefined` or `NaN`. Thus, a generic function should deal with errors, undefined return values and other problems when custom or pre-defined functions are executed, in order to make the evaluation process safer and indicate problems during function calls. If this check wasn't implemented, the evaluated value of the whole condition might, for example, be `undefined` because a function call returned `undefined`, but there would be no possibility to locate that erroneous function call in the condition. Therefore, the method `wrapperFunction`, that can be seen in Listing 5.9, creates a new safer function based on a submitted function. In case of a problem, a `FunctionError`, which is an `Error` with an additional `errorType` property of the type `FunctionErrorType`, is thrown. This property can be either `ExecutionError`, meaning that the function threw an error during execution, or `ReturnValueError`, indicating that the return value is not usable. Thereby, the submitted function is called first and occurring errors are caught. If an error is caught at this point, it is converted to a `FunctionError` with

ExecutionError as the type. Otherwise, if the returned value is undefined or NaN, a FunctionError with ReturnValueError as type is thrown. The message of the error further specifies which function failed.

```
1  static wrapperFunction<T extends Function>(key: string, func: T): Function {
2          return function () {
3                  let retValue: any;
4                  try {
5                          //Call of submitted function
6                          retValue = func.apply(this, arguments);
7                  } catch (e) {
8                          //Function threw error => Convert to FunctionError
9                          let f = new FunctionError(FunctionErrorType.
                               ExecutionError, e.message);
10                         f.stack = e.stack;
11                         throw f;
12                 }
13                 //Function returned a value
14                 if (isNullOrUndefined(retValue)) {
15                         //Value is null or undefined => ReturnValueError
16                         throw new FunctionError(FunctionErrorType.
                               ReturnValueError, `...`);
17                 } else if (isNumber(retValue) && isNaN(retValue)) {
18                         //Value is NaN => ReturnValueError
19                         throw new FunctionError(FunctionErrorType.
                               ReturnValueError, `...`);
20                 }
21                 return retValue;
22         }
23 }
```

Listing 5.9: The wrapperFunction from the FunctionParser

Since TypeScript's type information is lost after its compilation to JavaScript [18], the FunctionParser should at first also provide type checking of the submitted parameters and returned values of the pre-defined functions during run time. However, this functionality is not completely working and safe. Therefore, it is not part of the final software. It is implemented with a regular expression that extracts the head of a function

in `Functions` by key. Then, each parameter's type is extracted as a string. If the type is known to the parser and there is a function that allows for type checking of this type, the parameter will be checked. Otherwise, the parameter is simply not checked.

RuleEvaluator The most important part of the `Evaluation` component is the `RuleEvaluator`, that is used for evaluating conditions with regard to a given context. The class uses `parser`, an instance of the expr-eval framework, and an instance of `ConditionStringParser`, as well as the static `wrapperFunction` from the `FunctionParser`.

It allows for adding of custom functions with the method `addFunction`, that is shown in Listing 5.10. However, the submitted function is not directly added to `expr-eval`. The function is first wrapped into a new safer function with the `wrapperFunction`, which is then added to `parser`, as it is shown in Listing 5.10.

```
1 addFunction(key: string, func: (...args: any[]) => any) {
2         //Adds wrapped function to expr-eval parser
3         this.parser.functions[key] =
4          FunctionParser.wrapperFunction(key, func);
5 }
```

Listing 5.10: The `addFunction` from the `RuleEvaluator`

The `evaluate` function (i.e., Listing 5.11) is designed very simple. The submitted property `conditionString` should be the identically named property of a `BasicRule` and `contextVariables` should be the `variables` field of a `Context` object, although it is typed as `any`, and it is assumed to be correctly submitted by the `RuleEngineManager`. At first, `conditionString` is parsed with the instance of `ConditionStringParser`. Then, the condition is evaluated with `parser` and the evaluated value is returned. Yet, if an error is thrown during evaluation, it is caught and returned.

```
1  evaluate(conditionString: string, contextVariables: any): any {
2          try {
3                  //Parsing conditionString and evaluating it with expr-eval
4                  let returnValue = this.parser.evaluate(this.
                       conditionStringParser.parse(conditionString),
                       contextVariables);
5                  return returnValue;
6          }
7          catch (Error) {
8                  //Return Error if one occured
9                  let returnValue = Error;
10                 return returnValue;
11         }
12 }
```

Listing 5.11: The `evaluate` function from the `RuleEvaluator`

5.2.4 Manager

The `Manager` component is the main component. It consists of the `RuleEngineManager`, which manages and communicates with all other components during run time. However, it is not very complex, since there is not much new logic in the component. The evaluation process is managed by this component and it "wires" the single components together into the working rule engine. The `RuleEngineManager` contains an instance of `RuleEvaluator` named `ruleEvaluator`, as well as `functionRepository`, which is an instance of `FunctionRepository`.

The `RuleEngineManager` allows for communicating with `functionRepository` by passing through all relevant functions. Additionally, functions and `FunctionDescription` objects can be added when `RuleEngineManager` is initialized.

In order to provide extensibility, functions for managing contexts and evaluating rules are provided for `BasicRule` objects. Additional functions, specifically designed for users of the rule engine, then provide additional support for branches and rules, but in the background the basic functions are still used.

The general evaluation function `evaluateBasicRule`, that can be seen in Listing 5.12, evaluates `BasicRule` objects. The submitted object `basicRule` is evaluated with the `evaluate` function of `RuleEvaluator`. Then, an evaluation is built by the `EvaluationBuilder`. Since, `InternalRule` and `InternalBranch` objects extend `BasicRule`, they can also be evaluated by this function. Thus, the returned objects do not have to be `BasicEvaluation` objects, as the `EvaluationBuilder` always creates the most specific object for each evaluation. It is merely assured, that the returned objects contain at least the properties of a `BasicEvaluation`.

```
1 private evaluateBasicRule(basicRule: BasicRule, context: Context):
      BasicEvaluation {
2         let evaluatedValue: any = this.ruleEvaluator.evaluate(basicRule.
              conditionString, context.variables);
3         return EvaluationBuilder.buildEvaluation(evaluatedValue, basicRule);
4 }
```

Listing 5.12: The `evaluateBasicRule` function from the `RuleEngineManager`

Contexts are created with the function `createContext`, that builds a `Context` object based on a submitted `BasicRule` object. However, the created `Context` is not directly "filled" with results. Thus, `BasicRule` objects could be added to the engine without the results being collected yet. Yet, in the current state of the engine such a functionality is not implemented because it is not necessary.

In the next step, the `Context` objects have to be "filled" with the actual results. For this purpose, the function `fillContext`, which can be seen in Listing 5.13, is provided.

```
1 private fillContext(context: Context, results: Result) {
2         context.addResult(results);
3 }
```

Listing 5.13: The `fillContext` function from the `RuleEngineManager`

For all three of these basic functions, an additional function (i.e., `createContexts`, `fillContexts`, `evaluateBasicRules`), with the same functionality, that deals with multiple `BasicRule` or `Context` objects, is provided.

The more sophisticated functions `evaluateRules` and `evaluateXORBranches` (for evaluating rules and branches) use these basic functions, and work as an interface for external applications. Since there is no big difference between `evaluateXORBranches` and `evaluateRules`, only the `evaluateXORBranches` function will be explained in the following. This function creates one `BranchEvaluation` object for each submitted branch. When multiple branches are submitted, the position of the branches in the submitted list matches with the position of the `BranchEvaluation` objects in the returned list. The function itself simply uses `createContexts`, in order to create the contexts. Afterwards, the `Context` objects are "filled" with the results. Then, the branches are evaluated with the method `evaluatedBasicRules`. Thereby, the function `evaluateBasicRule` returns `BasicEvaluation` objects, although it is certain, that the objects are in fact `BranchEvaluation` objects, because of the implementation of `EvaluationBuilder`. Obviously, it would be better to assure this to the users, by declaring the return type of the function to `BranchEvaluation`. For this purpose, the type is overwritten to `BranchEvaluation`, by using TypeScript's type assurances, that override the analyzed type [19].

```
1  evaluateXORBranches(xorBranches: Branch[] | Branch, results: Result):
       BranchEvaluation[] | BranchEvaluation {
2          let rulesIsArray: boolean = true;
3          //Temporary transformation into list
4          if (!isArray(xorBranches)) {
5                  rulesIsArray = false;
6                  xorBranches = [xorBranches];
7          }
8          let contexts: Context[] = this.createContexts(xorBranches);
9          this.fillContexts(contexts, resultSet);
10         let evaluations = this.evaluateBasicRules(xorBranches, contexts);

11         return rulesIsArray ?
12                 <BranchEvaluation[]>evaluations : <BranchEvaluation>
                       evaluations[0];
13 }
```

Listing 5.14: The `evaluateXORBranches` function from the `RuleEngineManager`

5.3 Integration

In the following it is explained how the rule engine can be used.

Instantiating the Rule Engine In order to use the rule engine, an instance of `RuleEngineManager` has to be created. At this point, custom functions, which can then be used in conditions, and their descriptions, as well as options for the `ConditionStringParser` can be passed to the engine. These objects are all optional and also typed. The structure of the submitted functions exactly matches the structure of the `Functions` object with pre-defined functions. Furthermore, the descriptions are a list of `FunctionDescription` objects and the options are of the type `ConditionStringParserOptions`, that was explained in Section 5.2.3. Listing 5.15 demonstrates the instantiation of a `RuleEngineManager` object.

```
let functions: FunctionsObject = {...};
let functionDescriptions: FunctionDescription[] = [...];
let options: ConditionStringParserOptions = {...};

//Instantiating the rule engine
let ruleEngine = new RuleEngineManager(functions, functionDescriptions,
    options);
```

Listing 5.15: Instantiating the rule engine

Evaluating Rules and Branches After the rule engine was instantiated, lists of rules or branches and corresponding results can be evaluated with the functions `evaluateRules` and `evaluateBranches`. Yet, these functions also allow for evaluating single rule or branch objects. Listing 5.16 demonstrates how a list of rules can be evaluated.

```
let rules: Rule[] = [...];
let results: Result = {...};
let evaluations: RuleEvaluation[] = ruleEngine.evaluateRules(rules, results);
```

Listing 5.16: Evaluating rules with the rule engine

The returned value of these functions usually is a list of `BranchEvaluation` or `RuleEvaluation` objects. However, if only a single rule or branch is evaluated, only a single evaluation object is returned. Both `BranchEvaluation` and `RuleEvaluation` are more specialized `BasicEvaluation` objects. Each one of these contains three important properties. First, the property `evaluationType` indicates if any errors or problems occured during the evaluation of the corresponding rule. Second, `evaluatedValue` is the value that was evaluated for the rule (e.g., `true`, `false`, `Error`). Third, the type of an evaluation object is displayed by `evaluationKind`, since there are more specialized evaluation objects, which extend `BasicEvaluation` with the important properties of the evaluated objects (i.e rule or branch). However, the provided functions `evaluateRules` and `evaluateXORBranches` already indicate the value of this property with their return type. Furthermore, dealing with `evaluationType` and `evaluatedValue` is not necessary, since all evaluation objects possess the function `getValue`. This function returns `evaluatedValue`, if the evaluation was successful and otherwise throws the corresponding error. Thus, if `getValue` doesn't throw an error, it is certain, that it returns a `boolean` value. Section 5.2.1 explains the evaluation objects in more detail.

Custom Functions Custom functions and their descriptions are managed by the `FunctionRepository`. Yet, communication with the `FunctionRepository` also happens through the `RuleEngineManager`. As it was discussed earlier, functions and descriptions can be added during the setup of the `RuleEngineManager`. If new functions have to be added during run time, the function `addFunction`, that adds a submitted function with a submitted key to the `FunctionRepository`, can be used. The key acts as the name, by which the function can be called by within a rule. If there already is a function defined with this key, it is overwritten with the new function. Multiple functions, structured as the `Functions` object, can be added with `addFunctions`.

Functions are described by `FunctionDescription` objects, which are described in Section 5.2.2. These contain meta information about the custom functions, which should help creators of questionnaires. `FunctionDescription` objects can also be added during execution time with the functions `addFunctionDescription` and

`addFunctionDescriptions` of the `RuleEngineManager`. These descriptions, in turn, can be queried with different functions. The method `getAllGroups`, for example, returns all groups that contain at least one function, as a list. Now, all descriptions of one group can be received by calling the function `getDescriptions` and submitting the name of the group. If no group is submitted, all descriptions are returned. One can also query for single descriptions by their key, with the function `getDescription`.

Variables within Conditions Multiple things have to be considered when dealing with variables within a condition. First, if the corresponding answer in the result has multiple iterations, the single iterations have to be accessed first, for example with the pre-defined function `getIteration($variableName, iteration)`. Furthermore, if the variable is bound to a matrix question (i.e, the property `value` of the variable in `variablesMapping` has two keys), the value of the variable is further divided into `row` and `column`. In order to access these properties, the functions `getRow($variableName)` and `getColumn($variableName)` can be used. Properties of lists and objects can be accessed by calling the pre-defined function `getProperty($variableName, propertyName)`.

6

Summary

The main goal of this thesis was to design and develop an engine for evaluating rules and branches of the QuestionSys framework. At first, requirements to the software were imposed. With these requirements in mind, possible problems regarding the fulfillment of them were analyzed. These problems included code-injection safety, expressions evaluation and the ability to use custom functions within the conditions of a rule or branch. They were eliminated by the use of an evaluation framework. In order to find a good solution, the frameworks vm2, expr-eval and Jexl were compared. Ultimately, expr-eval was used as evaluation framework, as it was the best fit for the rule engine. Then, a general concept of the rule engine was developed. This concept included a general illustration of the evaluation process and architecture of the rule engine. Furthermore, the main communication flow between the different components of the architecture was presented. Afterwards, a working implementation of the rule engine, that is based on the general concept was introduced and explained. Thereby, the usage of the rule engine was also illustrated.

6.1 Fulfillment of Requirements

In the following for each functional and selected non-functional requirements, it is discussed whether the requirement is fulfilled by the developed rule engine.

FR1 and *FR2* are fulfilled by the rule engine. The `RuleEngineManager` offers tailored functions that correctly evaluate rule and branch objects of the QuestionSys model.

FR3 is fulfilled by the rule engine. In order to link results with rules or branches, a `Context` object, is created and "filled" with the collected results for each rule or branch.

FR4 and *FR5* are completely covered by the rule engine. Functions that can be used during evaluation are stored and managed by the `FunctionRepository`. Thereby, a file with pre-defined functions of the engine is already included. However, this file is only filled with essential functions for dealing with variables and probably has to be extended. Furthermore, functions can be managed during run time (i.e., adding, deleting, overwriting). In order to export these functions, every function can and should have an additional description, that contains the key, name, group, explanation and an example of how the function can be used. These descriptions can also be managed during run time and there is the possibility to query these.

FR6 is covered by evaluation objects. The rule engine creates an evaluation object for each rule or branch that is evaluated. These objects contain the field `evaluationType` that indicates if the evaluation was successful or which kind of error occurred. Additionally, the evaluation objects contain all relevant information of the evaluated rule or branch.

FR7 demands the rule engine to eliminate code injection threats. With the usage of the framework expr-eval for evaluating conditions and expressions, there is no possiblity for code injections unless a custom function is unsafe (i.e., it is, for example, using JavaScript's `eval` function).

Lastly, *FR8* is covered by the rule engine. The evaluation objects include any errors that occur during evaluation of a rule or branch. Additionally, there is the `FunctionError` which particularly indicates errors that occured during the execution of a custom function.

Now the non-functional requirements are discussed.

Obviously, *NFR1* is fulfilled, as the engine is developed with TypeScript as an npm package.

The developed rule engine is easily extensible (i.e., *NFR3*), since the core parts, that deal with evaluation conditions, are designed to work with the minimum amount of information, which is needed for evaluation. Thus, new objects that contain the essential properties

for evaluation can easily be added without any changes to the core functionalities of the software.

The rule engine is properly tested with the frameworks mocha and chai. The line coverage of the different tests amounts to 89%. Hence, *NFR5* is fulfilled.

6.2 Outlook

The rule engine in its current state is limited to evaluations of rules and branches of the QuestionSys model. With the generic implementation of the evaluation and rule objects, new objects could be supported quite easily, as long as the essential properties for evaluation (i.e., the properties of a `BasicRule`) do not change.

A great feature would also be the support of the `[]` characters for accessing arrays. Currently properties of arrays have to be accessed with the function `getProperty`, which is uncommon and sometimes confusing. Such a feature could be added to the `ConditionStringParser`.

The rule engine already possesses the ability to deal with pre-defined functions. However, the rule engine doesn't come with many pre-defined functions. In the future, lots of useful functions could be added to the engine, making the rules easier to create and understand for the creators of questionnaires. Furthermore, there are no type checks for parameters and return types of functions yet. A functionality that automatically injects run-time type checks into custom functions was already developed but it is not part of the engine, since the current design and implementation is immature. However, it could be redesigned or updated in the future. Alternatively, type checks could, for example, be added for each function itself.

Due to the continuous development of the QuestionSys framework, the engine will have to be updated and adapted to the changes. At the moment, for example, the result objects are being reworked and updated. Unfortunately, the rule engine was not yet used as a component of the QuestionSys project. When the engine is actually in use, new problems and missing functionalities might show up. Users of the engine can leave important feedback and contribute to the further development of the rule engine.

Bibliography

[1] DBIS, U.o.U.: Questionsys - a generic and flexible questionnaire system enabling process-driven mobile data collection. (https://www.uni-ulm.de/in/iui-dbis/forschung/laufende-projekte/questionsys/) Accessed: 2018-10-01.

[2] Pavlović, I., Kern, T., Miklavcic, D.: Comparison of paper-based and electronic data collection process in clinical trials: Costs simulation study. **30** (2009) 300–16

[3] Paul, J., Seib, R., Prescott, T.: The internet and clinical trials: Background, online resources, examples and issues. J Med Internet Res **7** (2005) e5

[4] Fritz, F., Balhorn, S., Riek, M., Breil, B., Dugas, M.: Qualitative and quantitative evaluation of ehr-integrated mobile patient questionnaires regarding usability and cost-efficiency. International Journal of Medical Informatics **81** (2012) 303 – 313

[5] Etherington, D.: Mobile internet use passes desktop for the first time, study finds. (http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide) Accessed: 2018-10-01.

[6] Schobel, J., Pryss, R., Wipp, W., Schickler, M., Reichert, M.: A mobile service engine enabling complex data collection applications. In: 14th International Conference on Service Oriented Computing (ICSOC 2016). Number 9936 in LNCS (2016) 626–633

[7] Brandt, C.A., Argraves, S., Money, R., Ananth, G., Trocky, N.M., Nadkarni, P.M.: Informatics tools to improve clinical research study implementation. Contemporary Clinical Trials **27** (2006) 112 – 122

[8] Schobel, J., Schickler, M., Pryss, R., Reichert, M.: Process-driven data collection with smart mobile devices. In: 10th International Conference on Web Information Systems and Technologies (Revised Selected Papers). Number 226 in LNBIP. Springer (2015) 347–362

[9] OWASP: Code injection. (https://www.owasp.org/index.php/Code_Injection) Accessed: 2018-10-01.

[10] Mozilla: Mdn web docs - javascript eval. (https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/eval) Accessed: 2018-10-01.

[11] TomFrost: Javascript expression language: Powerful context-based expression parser and evaluator. (https://github.com/TomFrost/Jexl) Accessed: 2018-10-01.

[12] Simek, P.: Advanced vm/sandbox for node.js. (https://github.com/patriksimek/vm2) Accessed: 2018-10-01.

[13] Crumley, M.: Javascript expression evaluator. (https://github.com/silentmatt/expr-eval) Accessed: 2018-10-01.

[14] npm Inc.: What is npm. (https://docs.npmjs.com/getting-started/what-is-npm) Accessed: 2018-10-01.

[15] npm Inc.: npm docs - npm-install. (https://docs.npmjs.com/cli/install) Accessed: 2018-10-01.

[16] npm Inc.: npm docs - package.json. (https://docs.npmjs.com/files/package.json) Accessed: 2018-10-01.

[17] Microsoft: Type compatibility in typescript. (https://www.typescriptlang.org/docs/handbook/type-compatibility.html) Accessed: 2018-10-01.

[18] Bierman, G., Abadi, M., Torgersen, M.: Understanding typescript. In: European Conference on Object-Oriented Programming, Springer (2014) 257–281

[19] Syed, B.A.: Typescript - type assertion. (https://basarat.gitbooks.io/typescript/docs/types/type-assertion.html) Accessed: 2018-10-01.

List of Figures

Name: Pascal Kühner Matriculation number: 916513

Honesty disclaimer

I hereby affirm that I wrote this thesis independently and that I did not use any other sources or tools than the ones specified.

Ulm, ...

Pascal Kühner