Design of a platform-independent business rule authoring environment for non-technical authors

Duwaer, D.F.

Award date:
2016

Design of a Platform-Independent Business Rule Authoring Environment for Non-Technical Authors

David Duwaer

December 1, 2016

Abstract

This thesis is the result of a research conducted during the author's internship at Royal Philips N.V. In it, an authoring environment for business rules named Rash is proposed. A distinguishing feature of this authoring environment in that it is stand-alone and is made to support any number of business rule engines, target languages and authoring languages. The authoring Graphical User Interface (GUI) is a syntax-directed editor aimed to suit non-programmer users, which is achieved by optimizing simplicity [1], visually representing relevant concepts and providing corrective feedback. Where previous syntax-directed editors are made to look like a text editor [2][3], the editor proposed here distinguishes itself by moving away from text-editing paradigms, towards structure-editing paradigms. The authoring environment is suitable for language extensibility: new functions written by software engineers can easily be embedded in the authoring environment; complete with a natural language-like syntax and type checking. Finally, function libraries forming a basic authoring language are proposed. These functions read like natural language, and adhere to the principle of referential transparency [4] so that any expression written with them is also a valid natural language expression.

Contents

Chapter 1

Introduction

1.1 Introduction

The business rule approach gained massive momentum since the early 2000's [5], and is in fact a second life for the technique that was used in expert systems from the 1970s to the mid-1990s [6]. While machine-learning techniques have been the focus of advancements in artificial intelligence in the most recent decade [7], business rules are still just as relevant as discrete, well-defined and transparent logic has different application domains than the less transparent decision making typically provided by machine-learned models.

The focus of this thesis is the design of an authoring environment for business rules, which is to be used at Royal Philips N.V. for a *multitude* of different rule engines that are being used in and for various products. The aim is to make the authoring environment such that Subject Matter Experts (SMEs) for the products, who have no programming knowledge, can author the logic -without help from an IT expert.

1.2 What are business rules and why are they used?

According to the Object Management Group (OMG), within the Business Motivation Model (BMM) [8], business rules "provide specific, practicable guidance to implement Business Policies. Some Business Rules could be automated in software; some are practicable only by people." For business rules that can be automated by software, there are Business Rule Management Systems (BRMSs). BRMS's are systems that maintain a base of formally specified business rules and automatically enforces them. The core component of a BRMS is a *rule engine*. A rule engine, depicted in Figure 1.1, reads from a set of rules that reason about a model called the Business Object Model (BOM), which is an abstraction from the schema of the Business Data. When a rule condition becomes true because of a change in the business data, the rule engine executes the rule's action, which in turn may modify the business data. The business rules' only dependency on the rest of the software is through the BOM, and the

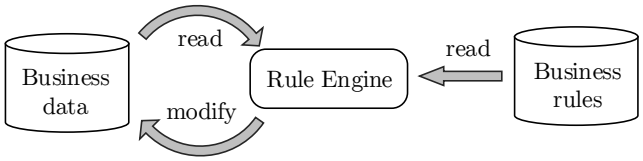

Figure 1.1: A rule engine and its context.

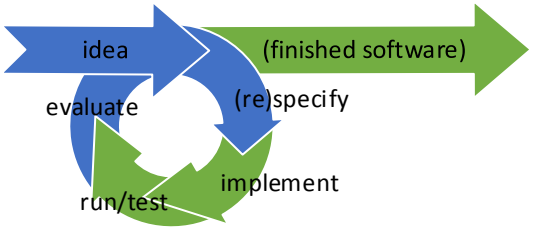

Figure 1.2: Involvement of SMEs (blue) is an integral part of the Software Development Life Cycle.

business rules are interpreted at runtime. This creates great advantages of using the business rule approach to implement business logic, over implementing it in conventional program code.

1. Business rules can be developed and deployed independently of the rest of the software. This allows for more agility [9].
2. To update business rules the application they are a part of does not have to be restarted. This is especially useful if large enterprises are using the application.
3. Individual business rules relate closely to individual functional requirements [9], making those requirements highly traceable.
4. Business rules are separated from program logic.
5. BRMS's typically provide functionality to define domain specific languages. The goal here is to have a language that trades flexibility for ease-of-use, so that for its application domain it can be used by people who have little to no skill in programming.

1.3 Business rules in practice

Some businesses need logic in their software that reasons about subjects outside the knowledge domain of their software engineers. As a consequence, SMEs and Software Engineers (SEs) need to collaborate to get this logic implemented. In Figure 1.2, it is shown how SMEs are a part of the Software Development Life Cycle (SDLC). This cycle, or a variation on it, is usually iterated through several times when a software solution is created [10]. Having multiple actors play a part in this cycle adds communication barriers that not only have the disadvantage of having people spending time on creating and interpreting communications

(e.g. requirements, documentation, ...), but can also act as funnels slowing the entire process down [11] [12]. It would therefore improve the agility of the business rule management if the collaboration between SEs and SMEs would not be not necessary.

A way this could be achieved is by having a team of people that have knowledge of both software and the domain of application. Educating an SME to be an SE or vice-versa can take long: in the Netherlands, in most fields (including software engineering) an applied sciences degree takes 4 years to obtain and a master's 5 years. The fast-changing market surrounding software makes this solution therefore unsuitable, if not unfeasible.

Another way to eliminate the necessity of SMEs and SEs collaborating is to use a *tool* that bridges the gap between the business domain and the software domain. Because the business domain differs per application, it is not feasible to make a single tool that allows software people to understand and work with *all* of these domains. Inventing with a tool that allows non-software people to write software is, however hard, a much narrower challenge. Business rules in particular —even in their formal (executable) form— have the property of being more easily understood by "business people" (people without programming knowledge) as they are relatively compact, coherent peaces of code that correspond closely to individual business requirements [5]. It is therefore not surprising that most BRMS's come with a rule authoring environment that is meant to be used by non-programmers.

1.4 Philips's case

Philips has multiple business rule executing components of varying degrees of "hard-codedness" running on varying platforms, not all of which are suitable for replacement by rule engines that comes with industry-grade BRMS's, let alone all by the same one. This raises the need for a stand-alone Business Rule Authoring Environment (BRAE) that can "export" its authored logic to any execution platform —even to source code containing hard-coded rules, if some application would require this. As Chapter 4 will point out, the industry currently does not offer stand-alone rule authoring solutions, as the tendency is towards the vertically integrated solutions known as BRMS's.

The need for building a BRAE in-house will also be leveraged for re-considering the ease-of use of rule authoring for business users: existing tools will be assessed on their usability and user tests will be employed in an effort to come up with a design that maximizes a non-programming SME's ability to author rules.

1.5 Conclusion

1.5.1 Assignment description

The goal is to design a rule authoring environment which...

1. is *independent* of the target rule engine, and can be used for multiple such rule engines or other kinds of components that execute business rules
2. is *optimal* in enabling non-programmers to author executable business rules
3. is *extensible* to incorporate external data sources, computation modules and decision makers into the decision-making power of the business rules

1.5.2 Report outline

A more detailed investigation of the requirements will take place in Chapter 3. An investigation of existing solutions and approaches is done in Chapter 4. Chapter 5 explains the main decisions that led to the initial design of Rash, the BRAE this thesis proposes. Chapter 6 describes that initial design. A prototype of the initial design was made with which user tests were conducted, described in Chapter 7. This led to additional design decisions being made and revisioning of previous design decisions, described in Chapter 8. The improved design is then presented in Chapter 9. The thesis is concluded in Chapter 10.

Chapter 2

Some basic theory and terminology

2.1 Introduction

This chapter serves to explain some basic theory and introduce some terminology that will frequently be used in the remainder of this report.

2.2 Rule engines

BRMS's have a Business Rule Engine (BRE) as their core component. This is the component that reads and interprets business rules that have been specified in some given formalism, and executes them whenever they should be executed, based on the business data, as shown in Figure 1.1. A Business Rule Engine (BRE) is mostly based on a Production system. We will first explain the more general production rule systems, and then further our focus to BREs.

2.2.1 Production rule systems

As depicted in Figure 1.1, three components are most relevant to a production system.

1. A set of *production rules*, referred to as the *ruleset*.
2. A database
3. An interpreter

A *production rule* consists of a *condition*, commonly referred to as the rule's Left-Hand Side (LHS), and an *action*, or Right-Hand Side (RHS). A rule's LHS contains a proposition about data in the database. The *interpreter* matches the rules against the database and when a rule's condition becomes true, *triggers* the rule, meaning its *action* is allowed to be executed. When the rule's action is actually executed, it is said that the rule *fires* [9].

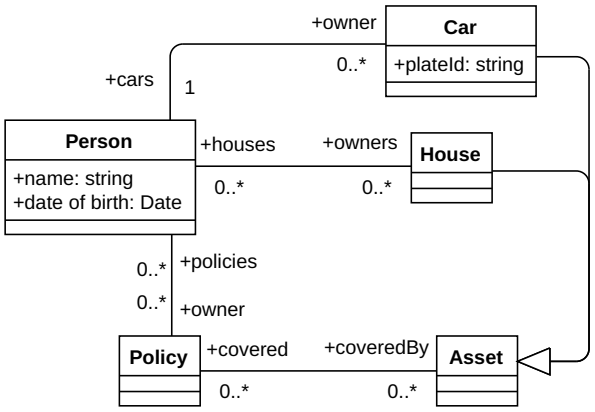

Figure 2.1: A typical BOM

2.2.2 Business Rule Engines

In most BREs, the rules reason about the database strictly by means of a model of the database called the Business Object Model (BOM). This is a a set of *classes* or *complex data types* that each contain a specific set of key-value pairs called *properties*. Relations between the classes of a model are represented by properties too. Figure 2.1 contains an example BOM. In this report we define an *entity* as being a datatype in the BOM. In BREs, it is common behavior that rules fire immediately upon being triggered [6] [13].

2.3 An example business rule

Consider a typical BOM, given in Figure 2.1. Next, consider the following business rule that reasons about the BOM from Figure 2.1.

If a car is not insured by its owner, send a notification to the owner.

This rule can be broken down into the LHS *"a car is not insured by its owner"* and the RHS *"send a notification to owner"*. The LHS consists of a fact about two *entities*: a car and a person. When this fact becomes true *for some combination of a car and a person*, the rule will trigger and fire. The RHS refers to a person, which has to be the same person that was matched with the LHS by the interpreter. This is done by using *variables* inside rules: a rule condition can assign variable names to the entities it reasons about. Upon matching, these variables will point to the entity instances that were matched. In pseudocode, the rule could look like what is shown in Listing 2.1.

Listing 2.1: Pseudocode of example rule

```
IF
    [$x: Car] is not insured by [$y: Person]
THEN
    Send notification with text "..." to [$y]
```

Here, "`[$x: Car]`" is at the same time a reference to an instance of entity `Car` and a variable assignment of the matched entity to a variable named "`$x`". "`[$x]`" is a usage of that variable.

2.4 Firing cardinality

Let us talk about *how many times* the consequence of the rule will execute when a condition becomes true. Suppose we have the following condition.

If there is a red marble in a blue bucket, do A.

And we are given the following classes.

```
class Marble {
static String shape = 'ball'; String color }
class Bucket {
String color; Collection<Marble> marbles }
```

Then we propose that we refer to the marble class by simply writing `Marble`, to a static property by writing `Marble.shape`, to an *object of the marble class* by writing `[Marble]`, to an object while at the same time assigning a variable `m` to that object by writing `[Marble m]`, and a property of that object by writing `[Marble].color` or `m.color`. Then the condition for the rule above would be expressed as follows:

$$\exists_{\texttt{[Marble m],[Bucket b]}} \left[\begin{array}{l} \texttt{m.color} = \texttt{'red'} \\ \wedge\ \texttt{b.color} = \texttt{'blue'} \\ \wedge\ \texttt{m} \in \texttt{b.marbles} \end{array} \right]$$

When this rule is fired, the consequence will only execute once: when this condition turns from false to true, even though there may be hundreds of red marbles spread over dozens of blue buckets. So how do we specify that we want a consequence to execute, say, for *every marble* for which the condition holds true? We write:

$$\exists_{\texttt{[Bucket b]}} \left[\begin{array}{l} \texttt{[Marble m].color} = \texttt{'red'} \\ \wedge\ \texttt{b.color} = \texttt{'blue'} \\ \wedge\ \texttt{m} \in \texttt{b.marbles} \end{array} \right]$$

The difference with the previous condition is, that `[Marble m]` is now a *free variable* in the condition, because it is not quantified by the *exists*-clause anymore. Similarly, if we write

$$\begin{array}{l} \texttt{[Marble m].color} = \texttt{'red'} \\ \wedge\ \texttt{b.color} = \texttt{'blue'} \\ \wedge\ \texttt{m} \in \texttt{b.marbles} \end{array}$$

then the consequence may be executed *for every combination* of a red marble and a blue bucket, where the red marble is in the blue bucket. The cardinality of a rule firing generalizes to the following.

Given the following:

- The condition of a rule, expression E contains a set V of free (i. e. not bound by a quantifier) variables, which may be objects, object properties, classes or class (static) properties.
- The subset of V that is objects is O_O.
- The properties in V belong to objects, the set making up all objects of these properties is O_P.
- The total set of objects $O = O_O \cup O_P$.
- If two objects in V are unified by a variable name in E they are considered to be one and the same object in O_O, O_P and O.

Then a rule with condition E fires for every combination of objects in O for which the condition is true. The objects in O are also called the *matches* of a rule.

Chapter 3

Requirements

3.1 Introduction

The proposed authoring environment is meant to be used for multiple applications, each with different requirements. For the investigation, four *focus applications* from within Philips were selected, each having one or more components that execute some form of business rules. The following approach has been used to work toward a valid software solution.

1. Detailed sets of requirements for every focus application were collected.
2. Identifying the *differences* and the *similarities* between the requirements from the different focus applications.
3. Proposing how the differences can be generalized and adhered to by means of configuration.
4. Specifying exactly what the functionality of the software will be, establishing the first version of the Software Requirements Document (SRD).

The following sections will investigate the requirements that those applications pose on all relevant aspects of a business rule authoring environment.

3.2 System scope

Figure 3.1 denotes the scope of the system we are proposing. Input to the system is the configuration of the rule language and the BOM for a specific application. This enables a rule author to create/model rules for that application through the authoring User Interface (UI), and the resulting rule models are output to an execution environment or compiler.

3.3 Business Object Model (BOM)

3.3.1 Occurring BOM styles

Between the rule engines of the focus projects, the data was available in several different ways. Each of these ways is elaborated in the following subsections.

Figure 3.1: The dashed line denotes the boundary of the proposed system, the arrows denote the interfaces with surrounding systems.

Object model

Some rule engines have their data available through an object model: a set of classes with properties, and relations *between* those classes represented by a subset of their properties. A simple example is given in Figure 2.1. Note that this BOM contains generalization/type inheritance; this is not important now, but it is something we will get back to in Section 5.3.

Primitive variables

When a rule engine has only one object class to match (e.g. rules can only match a set of users), it may not be necessary anymore to present data to the rules through object classes. Rules for such a rule engine only refer to plain, non-object-oriented variables. An example rule is given in Listing 3.1.

Listing 3.1: Example rule from rule engine with only simple variables

```
Claims > 0
AND RegisteredOn > TodayDate - 14
AND RegisteredOn < TodayDate - 6
```

Rules in this rule engine match a (set of) user(s) by default, so it does not need to be stated explicitly that some variables are properties of a user. This is why a simple variable suffices, and no complex data type like e. g. `User.Claims` or `User.RegisteredOn`.

Singleton objects

Some rule engines, along with a set of classes describing their object model, have global objects: objects of which there is only one instance. Consider, for instance, an insurance company "Insurance Co.", which has a data model with several entities, e. g. `Person`, `Policy`, `House`, et cetera. In its data model, the state of the company itself is also available: through a variable called "Insurance Co." of class `InsuranceCo`. This could be useful for accessing the state of company-wide policy that often changes and that may be of relevance to business rules. It does not make sense for any rule to make any other instance of this class than the available one, so the class itself should not be a part of the BOM. The pre-instantiated variable of this class, however, should be.

3.3.2 Separation of data and logic

An object model in a general-purpose object-oriented programming language can contain *methods* and *derived properties* (methods without parameters, with syntactic sugar to make it look like a normal property). The BOM available to a rule author could also contain these concepts. However, embedding logic into the data model makes this logic domain-specific because of where it is implemented. A data model without logic allows one to instantiate e.g. a data-handling back-end application for a specific application merely by soft configuration. Moreover, the logic is in a place and a format that is independent of the data model of a single application (although it *can be* made application-specific). For this reason, the BOM should contain no methods. As for derived properties: to the authoring environment they are indistinguishable from normal properties as the authoring environment itself does not deal with the implementation of the data model, and so the distinction is irrelevant.

3.3.3 Resulting requirements

The variations above are all implemented if we propose the BOM is an object model, but with a few extensions:

1. not only complex, but also primitive data types may be part of the BOM;
2. the BOM may contain global variables of any datatype, including of datatypes that are not in the BOM.

This results in the requirement set given in Table 3.1.

ID	**Description**
BOM01	The BRAE's only view on the business data is the BOM.
BOM02	The BOM consists of a set of datatypes (commonly referred to as the *entities*) and a set of variables.
BOM03	Each entity can be either complex (i.e. a class) or primitive.
BOM04	A BOM variable may be of any datatype, including datatypes that are not part of the BOM as an entity.

Table 3.1: Requirements on the BOM

3.4 Rule expressiveness

3.4.1 Introduction

This section aims to elicit what should *at least* be possible to express in the body of the requirements, as well as what classes of requirements there are. Note should be taken that the the designed authoring environment should allow for extension into more expressivity, and that the requirements elicited in this section are seen as a *minimum* to comply with the current set of rules specified for the focus projects.

We start out by classifying rules in Subsection 3.4.2, determining what classes are needed by focus projects and reducing these classes to a single class with an extended definition. Next, we start eliciting expressiveness requirements in Subsections 3.4.4 (for the LHS), 3.4.5 (for the RHS), 3.4.6 (concerning data aggregation), but not before some necessary assumptions to the BOMs of the focus projects are made (Subsection 3.4.3).

3.4.2 Types of rules

Table 3.2 shows a classification of rules that is taken from [9]. All the rules used

Rule type	**Explanation**
Mandatory constraints	A rule that prevents some actions from being enabled.
Guidelines	Similar to mandatory constraints, but instead of rejecting an action that would violate the constraint, issues a warning.
Action-enablers	Rule that tests a condition and upon finding it true, initiates an external action
Computations	Rule that generates new information based on computation. Describes the name of some value, and how this value is computed.
Event-Condition-Action (ECA)	When a specific event occurs, evaluates a condition. If condition is true, initiates an action.
Inferences	Upon finding a condition true, creates new facts for the rule engine to consider (can therefore trigger other inference, action-enabler or ECA rules)

Table 3.2: Classification of rules used in [9]

in the focus projects have been checked and are either *action-enabler*, *inference* or *ECA* rules. This classification assumes that an "action" is invoking some external procedure, distinct from making a change in the business data. If your definition of an action includes possible changes to the business data, or in other words: adding modifying or removing facts for the rule engine to consider, then *inferences* become a subclass of *action-enabler* rules. Similarly, if, by your definition, a "condition" may include the requirement that an event must occur and rules are activated instantaneously after their condition becomes true (as is standard in BRMS rule engines [6][13]), then ECA rules form a subclass of action-enabler rules too. The advantage we gain is that we only have to deal with a single definition for rules, given that they satisfy these requirements of expressive power that therefore have been added to Tables 3.3 and 3.4.

3.4.3 Dependency of expressiveness assessment on the BOM structure

If the BOM contains methods, these methods can provide (part of the) calculations that need to be expressed in the rule body. To find out what logic really needs to be expressed in the rules currently specified for the focus projects, we have assumed for each project a *dumb* BOM (see Section 3.3) representing the business data. Moreover, every concept mentioned in the rules that would by reasonable assumption have a finite number of instances (not, for instance, a date object) was assumed to be an *entity*, meaning that —conceptually— all instances of it are loaded into the rule engine memory and can be matched against. This provides a fairly normalized data representation for the analysis performed in the next subsection.

3.4.4 Rule condition expressiveness

To get an idea of what expressive power will be required for the rule conditions, we used the following approach.

1. Collect all sets of rules currently specified for the focus projects
2. Write out these rules using constructs from *first order logic*, *arithmetic*, *comparison operators* and *set theory*
3. List which constructs are used

From this, it was learned that these rules could all be written by using exclusively the following concepts. Concepts needed for data aggregation are left out here, and are specifically addressed in Subsection 3.4.6.

1. Propositional logic operators, i. e. AND, OR and NOT
2. Comparison operators, i. e. $=, \neq, <, >, \leq, \geq$
3. Basic number arithmetic, i. e. $+, -, *, /, pow(), \sqrt{}$
4. Combining multiple constraints on variables by *unification*, i. e. by simply referring to the same variable in these multiple constraints
5. Set membership, i. e. "$\in$"
6. Set builder, i. e. defining a set by stating the properties that its members must satisfy [14].
7. Existential quantification
8. Universal quantification

An explicit set of requirements is given in Table 3.3.

3.4.5 Rule consequence expressiveness

A rule consequence contains one or more actions. As described in Subsection 3.4.2, a rule action may be a database update or an external procedure call.

ID	Description
RC1	It is possible to require the occurrence of specific events in a rule's condition
RC2	It is possible to express the semantical equivalent of mathematical parenthesis that control the order of calculation inside a condition.
RC3	It is possible to express the semantical equivalents of logical AND, OR and NOT inside a condition.
RC4	It is possible to express the semantical equivalents of $=, \neq, <, >, \leq$ and $\geq$ inside a condition.
RC5	It is possible to express the semantical equivalents of $+, -, *, /, pow()$ and $\sqrt{}$ inside a condition.
RC6	It is possible to express the semantical equivalent of unification inside a condition.
RC7	It is possible to express the semantical equivalent of defining a set using set builder notation in the condition. This means that a datatype and a condition is specified by the author, and the resulting set is the selections of all items of that datatype for which the condition holds.
RC8	It is possible to express the semantical equivalent of logical *exists* inside a condition.
RC9	It is possible to express the semantical equivalent of logical *forall* inside a condition.

Table 3.3: User requirements on the expressive power for a rule's condition

External procedure calls

The four focus projects have little variation in the kinds of actions that were required by the specified rules. These actions were all in one of the following.

1. Sending a notification/showing a card to a person that was mentioned in the rule condition
2. Setting a flag on a person that was mentioned in the rule condition

A card is a small rectangular Graphical User Interface (GUI) item providing an insight to a user; it is like a notification, but displayed in a feed. It should be possible for software engineers to define actions that the rule author can then use. This is a very feasible job for the software engineers, given the small amount of actions that projects (at least the focus projects) use.

A side note: there were many rules that had some time constraint with respect to how often a card or notification of a specific type was to be shown, e.g. "don't show this card twice in one month". For the rules to "know" this, a user's collection of notifications must be part of the BOM. This provides another way of "sending" notifications to the user, i.e. by simply updating the notifications in the business data, and letting the notification client discover the change in the data by itself.

Database updates

Doing database updates entails the following standard actions.

ID	Description
RA1	The RHS of a rule consists of one or more actions
RA2	An action may be the creation of a new instance of an entity.
RA3	An action may be a value update of a direct or indirect (i.e. the property of a property, etc.) property of an entity instance.
RA4	An action may be the removal of an instance of an entity from the business data.
RA5	An action may be an external procedure call pre-defined/configured by a software engineer (the rule author needs only to enter function parameters relevant to the business).

Table 3.4: User requirements on the expressive power for a rule's consequence

1. Creating entities
2. Updating entities
   (a) Updating values
   (b) Inserting new values into a list
   (c) Removing values from a list
   (d) Updating values in a list
3. Removing entities

Note the recurring nature of this list: updating values in a list may entail updating a list that is a property of that list item - and so on. A syntax has to be devised to deal with this.

Requirements

The resulting set of requirements is given in Table 3.4.

3.4.6 Aggregation expressiveness

Aggregation is the combining of multiple data points —from multiple points in time and/or belonging to multiple objects— in a calculation, to obtain a new data point. Here are some typical expressions occurring in rule conditions that contain data aggregation.

1. *"the app has been in use daily for one week"*
   Here one has to confirm the existence of usage during every day of the past seven days.
2. *"the app has been used 20 times"*
   One has to count the amount of usage sessions.
3. *"the app has been used for 30 hours in total"*
   One has to sum the durations of all usage sessions.
4. *"the user's heart rate is 20bpm higher than the user's average"*
   One has to calculate the average user heart rate over time.
5. *"the average age of the customers of shop a"*
   One has to calculate the average age of all customers.

ID	Description
RC7	It is possible to express the semantical equivalent of defining a set using set builder notation in the condition. This means that a datatype and a condition is specified by the author, and the resulting set is the selections of all items of that datatype for which the condition holds.
ST1	There is a datatype, in further requirements referred to as —but in the application not necessarily named— a DateTime, representing a date specification.
ST2	There is a datatype, in further requirements referred to as —but in the application not necessarily named— a TimeCriterium, representing a selection on time.
ST3	A TimeCriterium can be defined by specifying conditions on the date, time and/or timezone attributes of timestamps that will be considered a part of the time selection.
ST4	There is a way to cluster a TimeCriterium into a collection of TimeCriteriums based on the value of a property of the DateTime datatype.
ST5	It is possible to check if a DateTime adheres to a TimeCriterium.

Table 3.5: User requirements on the selection power for a rule's condition

Assuming the BOMs as described in the previous paragraph, we have formulated, collected, generalized and structured the aggregation used by the existing rules. Aggregation was split into two steps: *selection* of a collection of objects, and doing the aggregation itself on that collection. A set of requirements on what expressiveness the authoring environment should provide with respect to selecting objects is given in Table 3.5 and with respect to the aggregation in Table 3.6. Requirement RC7 from Table 3.3 is also featured in Table 3.5 for cohesiveness.

3.5 Deployment

Assume there is a single Rash deployment per application. Consider one of the applications that have multiple rule engines. This application has a rule engine on the server-side and one on the client-side. Despite the fact that both engines use the same application, both have a different model scope: the client-side rule engine only sees a single user. Moreover, data is accessed differently on the server and the client side: on the server side, all data is available in the BOM entities, while on the client side, much data is available through *service* classes. While ideally, the application is adapted so that the data is available through BOM entities as much as possible, it is not known if this will be possible or feasible for all future applications. Hence, you have to assume completely different BOMs for the different rule engines within an application. Because different rule engines may also support different sets of functions, you cannot assume any similarity between rule engines of the same application. *If* there is to be a single instance of Rash per application, it will take only a marginal extra effort to support multiple applications. This leaves the following two options.

1. One Rash instance per rule engine (so sometimes multiple Rash instances

ID	Description
A1	A rule author can express that a specific condition should hold for a specific fraction of the elements in a collection.
A2	A rule author can express that a specific condition should hold for a specific number of elements in a collection.
A3	A user can get the average value of a function of a collection member over that collection.
A4	A user can get the sum of the value of a function of a collection member over that collection.
A5	A user can get the minimum value of a function of a collection member over that collection.
A6	A user can get the maximum value of a function of a collection member over that collection.
A7	A user can get the number of items in a collection.
S8	A user can obtain a list of the first n elements of a list. In case of elements with begin and end timestamps, the user may specify whether to use the begin or the end timestamp for the ordering; by default the beginning timestamp is used.
S9	A user can obtain a list of the last n elements of a list. In case of elements with begin and end timestamps, the user must specify whether to use the begin or the end timestamp for the ordering, otherwise the ending time stamp is used.

Table 3.6: User requirements on the aggregation power for a rule's condition

per application)

2. One RASH instance for all applications

While there are no *guarantees* about similarities of the BOMs and the sets of supported functions between the rule engines of an application, there will be an *aim* to make them similar, so that rules can be reused. If this is the case, then there is a big advantage in letting a single authoring environment handle all the rule engines in one application. A rule that is dependent on functions and entities that appear in more than one rule engine only needs to be written once, and an author can control in which engines the rule is deployed. This leads us to the conclusion that the second option, a single RASH instance for all applications, is a requirement.

ID	Description
DP1	RASH will be centrally deployed
DP2	RASH will be accessed through a portal
DP3	A single instance of RASH can manage multiple applications

Table 3.7: User requirements on how, when and where RASH is deployed

3.6 UI Integration

Currently, the focus projects each have their own approach to rule authoring. The following UIs types are found.

1. No GUI — the rule engine is Drools and rules are authored only by software engineers in **.drl* files.

2. A text-box for entering rule conditions in an extremely simple condition language. This text-box has no language support of any kind. The condition always selects a set of users and the action is always to send a notification to these users. A second text-box serves to enter the notification text.

3. A modular form (see Figure 3.2) that lets the author build a rule by selecting a template for a condition, then fill in the blanks, followed by selecting a template for an action, then fill in the blanks. The templates are fixed; users cannot create custom templates.

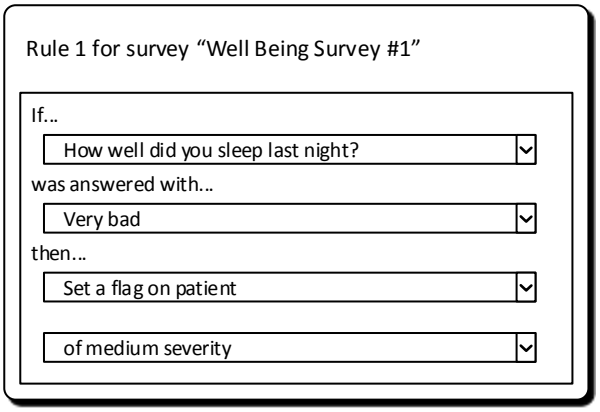

Figure 3.2: A modular form for building a template-based rule. In this screenshot: the choices made in the 1st and 3rd inputs change the available options in the 2nd and 4th inputs, respectively.

All interfaces that occur (options (2) and (3)) are web interfaces. Their UI's have different look-and-feels. Moreover, some applications use forms to build a rule, and these forms have intermediate database lookups. Designing a reusable, generic interface that would accommodate this last feature would require formalizing what constraints to the presented editing options and to the authored rule the interface is aiming to achieve. This is an advanced challenge with uncertain outcome, but certainly interesting for future work. A simpler solution, for now, is to design the system such that different rule authoring UI's can be created for it. The resulting set of requirements is given in Table 3.8.

ID	Description
UI1	The authoring UI should be web-based (i.e. should run in a web browser).
UI2	The UI for editing the body of a single rule can be embedded in another application's web-based UI.
UI4	The architecture should be such that additional UIs can be created that only work for specific applications.

Table 3.8: Requirements on the authoring environment's UI

3.7 Rule testing

Should the author test?

A few of the focus projects provide rule authoring for end-users, which are presumed to be non-programmers. In both these projects, rules are deployed into the actual rule engine directly when the user "saves" a rule. In BRAEs that come with industry-grade BRMS's —IBM Operational Decision Manager [3], inRule [15], Drools [16], etc.— rules can be tested by feeding them with mock data. The reason for this difference is that while the rules authored in the former case are meant for fairly predictable, elemental use cases, and there are no cases of *inference rules* (see Table 3.2), while the BRAEs in the latter case are meant to be ready for any kind of rules. Because we are designing a general-purpose BRAE we should see testing as an integral part of authoring, and users should therefore be able to test rules before employing them in the live application.

Testing on real or mock data

It seems like a convenient choice to allow the author to test rules on the real business data, without harming that data. This is easily possible by simply not persisting any changes made by the rules to the data during test runs. The problem with testing on real data, however, is privacy. A rule author can write rules that set highly specific conditions (e.g. in a rule condition requiring a person's address to be that of his neighbors). From the test results, the author can then elicit private information. This could especially be harmful if the business data covers sensitive information like medical patients' records. For this reason, rules should be tested against mock instances of the business database.

Requirements

The resulting set of requirements is given in Table 3.9.

3.8 Conclusion

Many requirements topics were omitted in order to keep the research effort focused on the core problem of the assignment. For instance, security wasn't

ID	Description
VC1	Per rule engine the BRAE can be configured to deploy rules immediately upon clicking save.
VC2	Per rule engine the BRAE can be configured to bundle saved changes into a *commit*. A commit has to be explicitly *pushed* by the rule author for the changes to the rule set to be deployed.

Table 3.9: User requirements on the integration concerning rule deployment

covered, and version control was only covered to a limited extent in the *testing* section. The requirements that *have* been covered should provide a sound motivation for the design that is presented in Chapter 6.

Chapter 4

Existing solutions

4.1 Introduction

When devising any enterprise software solution, the *build or buy* question is one of the first and important ones to answer [17]. Also, it is crucial to be aware of the state of the art. Therefore, for this research, a wide range of existing solutions for business rule authoring have been assessed in various industry niches.

The industry offers a range of BRMS's that offer a rule engine, a rule authoring environment and rule management capabilities including rule approval workflows. Unfortunately, the authoring environments offered with these products are not offered separately, and there are no commercial or open-source stand-alone BRAEs available. However, there are lessons that can be learned from the BRAEs that come with existing BRMS's for building a stand-alone BRAE in-house.

First, industry-leading BRMS's were explored for their rule authoring environments. In addition to this, authoring environments for database queries were explored. This is because rule conditions have a lot in common with database queries: making a selection on a set of database instances (rows) is similar to making a selection on a set of entity instances for which a rule should fire. All the UIs for rule- and query-authoring that can be found in the industry can be broken down into six categories, which are elaborated in Section 4.2 and 4.3. Of these categories, one was chosen to be essential for Philips's purposes (Section 4.4). Existing tools that applied this editing approach —but were not specifically built for *rules*— were looked at next, assessing for reuse and harvesting the concepts they apply (Section 4.5). Finally, in Section 4.6, the choice is made to build the solution in-house.

```
18 rule "Hello World"
19     salience 0
20     when
21         m : Message( status == Message.HELLO, myMessage : message )
22     then
23         System.out.println( myMessage );
24         m.setMessage( "Goodbye cruel world" );
25         m.setStatus( Message.GOODBYE );
26         update( m );
27 end
28
29 rule "GoodBye"
30     salience 0
31     when
32         Message( status == Message.GOODBYE, myMessage : message )
33     then
34         System.out.println( myMessage );
35 end
```

Figure 4.1: A text editor, enhanced with features specifically useful for editing code

4.2 Existing business rule authoring environments

4.2.1 Introduction

The authoring environments of InRule® [15], Drools [16], XpertRule® [18], Bosch Visual Rules® [19], Progress® Corticon® [20].

4.2.2 Code

Drools is an example of a rule engine that has a language, Drools Rule Language (DRL), in which entire rule bases can be written. This can be edited in Eclipse (see Figure 4.1) or in Workbench, Drools's web-based rule authoring and management environment. Unfortunately, neither environment has type-checking and auto-complete for the DRL. A BRMS like InRule has no language exposed to the user to do this, but it does have a language (an 'Excel®-like syntax' [15]) for entering only the condition of a rule.

4.2.3 Flow/decision tree

Most BRMS's support decision trees, sometimes referred to as flows. A decision tree is depicted in Figure 4.2. Every leaf in the decision tree is equivalent to a rule's consequence. The condition of that rule is then equivalent to the conjunction of all conditions appearing on the *root path* of that leaf [21].

4.2.4 Decision table

The decision table is semantically similar to the decision tree, but differently visualized. Every row in a decision table corresponds to a root path to a leaf in a decision tree. Every column in a decision table corresponds to a variable and the rows contain conditions about these variables, see Figure 4.3.

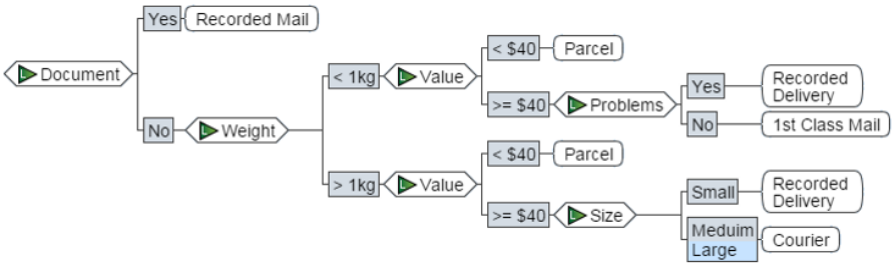

Figure 4.2: A decision tree encodes several rules. Angular boxes contain variables, rectangular boxes contain conditions and round boxes (leafs) contain actions.

	Value	Document	Size	Problems	Weight	**Shipping**
1	*	Yes	*	Yes	*	Recorded Delivery
2	>= $40	No	*	Yes	< 1kg	Recorded Delivery
3	*	No	*	*	> 1kg	Courier
4	*	No	*	*	< 1kg	Parcel
5	>= $40	No	Small	*	< 1kg	Recorded Delivery
6	>= $40	No	Meduim Large	*	> 1kg	Courier
7	*	Yes	*	No	*	1st Class Mail

Figure 4.3: A decision table encodes several rules. Each column is about a variable, each row contains conditions on each of those variables (or a "*" if no condition has to hold) and the last column contains actions.

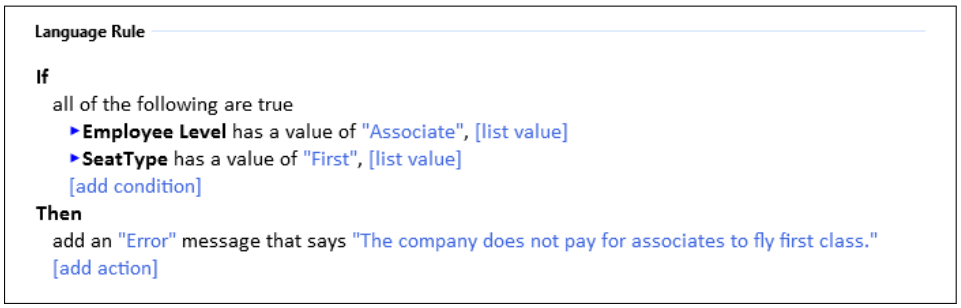

Figure 4.4: Editing natural-language code by not typing text, but editing the syntax tree directly, filling up fields by selecting possible content options from a menu's.

4.2.5 Syntax-directed editor

A projectional editor like the one of InRule displayed in Figure 4.4 lets the user manipulate some language's Abstract Syntax Tree (AST) directly. By choosing constructs and values from drop-down boxes, one builds up a syntax tree. Because this bypasses parsing, the syntax of the edited language may look like natural language.

4.3 Existing database query authoring environments

When a rule becomes enabled, it does so for a selection of entities, described in Section 2.3. If every entity corresponds to a database table, the condition of a rule has the same function as a database selection query. For this reason we have also looked at authoring solutions for database queries for non-programmers.

4.3.1 Visual query builder

A visual query builder, shown in Figure 4.5 is one that shows the classes of a database in schema view (an Entity-Relationship (ER) diagram), and attempts to visualize the conditions the query imposes on the displayed classes in this view. Relations between properties of distinct classes are drawn as lines, and conditions in which only a single class's property is involved are drawn as a seperate item connected with a line to said property.

4.3.2 Template query builders

Another common way the industry lets non-technical users author queries is by letting authors move directly towards building a visualization, see Figure 4.6. The author chooses what has to be on the x-axis, on the y-axis, what to group by and what to filter by, or whatever is relevant to the type of visualization that is being built. The underlying query is a fixed template for a visualization type, and the author selects (lists of) database columns for the blank spots. Filters can be added too, which are fixed templates with open parameters too.

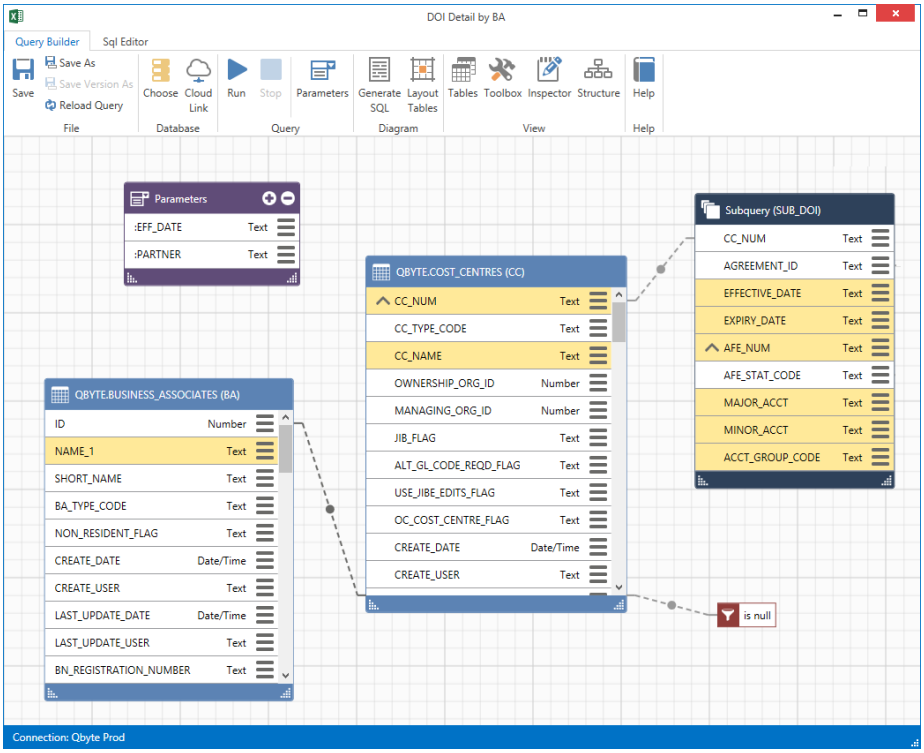

Figure 4.5: A visual query builder, based on the attempt to visualize a query in an ER diagram.

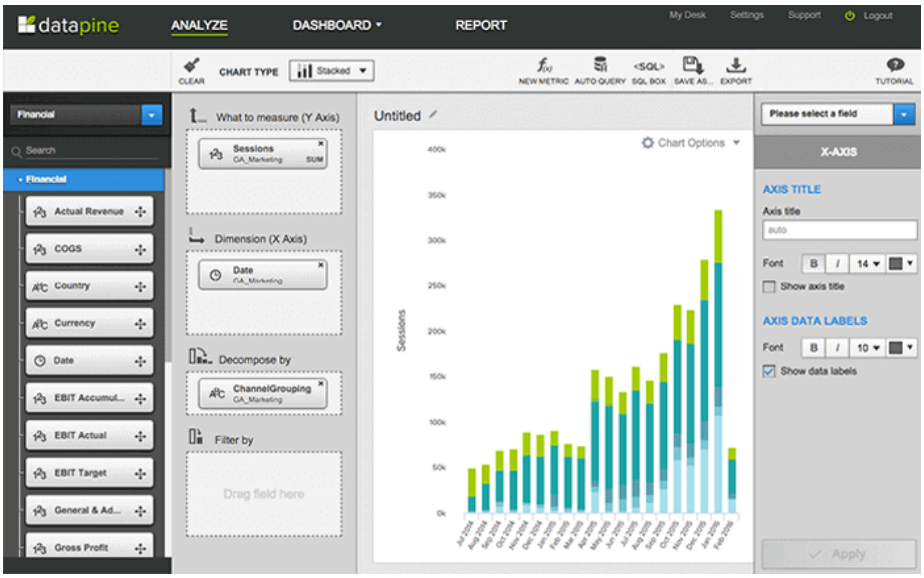

Figure 4.6: A template query builder lets a user “fill in the blanks” in a mostly pre-formed query

4.4 Assessment of most useful authoring concepts

4.4.1 By comparison with found alternatives

Template query builders

Template query builders are included in the previous Subsection for completion, but they are not suitable for expressive diversity demanded by rules specified at Philips; see Chapter 3.

Visual query builders

Visual query builders visualize entities and relations between them; a relation about a single entity is shown as a 'filter'. If used for editing rule conditions these relations would be the facts about one or more entities. When using more complex datatypes that are not entities like selections of time or timestamps, the meaning of the visual query builder view becomes unclear: should these datatypes be displayed in the same way as an entity? And how should it be displayed if a datatype like a time selection is in fact the result of a calculation? Moreover, the visual query view natively provides no way to visualize what the logical relationship between the many facts are: if they should hold in conjunction, disjunction, or more complicated nested structures.

Workflow / decision tree editors

Workflow and decision tree editors are also not suitable. Workflow editors, contrary to visual query builders, *only* provide a visualization of logical relations; all other expression constructs are hidden within the nodes. For decision tables a similar issue holds, but there the calculations have to be hidden in the column headers. This raises the issue of how those constructs are then edited inside the nodes. For example, neither a decision table or decision tree can in a trivial way represent a for-all construct, meaning another way of editing that and other expressions would still have to be developed.

Syntax-directed editors

Syntax-directed editors work with a view that encodes all information at once.

Conclusion

The syntax-directed editor seems the clear winner here, but let us back up a bit and consider combinations of the alternatives. One could think of combining a workflow or decision tree editor with a visual query builder, because they are partially complementary. Unfortunately, even the combination of these editors could not, for instance, visualize a for-all construct. Therefore they would still have to incorporate code editing in their nodes or column definitions, *or* incorporate a syntax-directed editor for doing this. Moreover, having multiple different views increases the complexity of the editor and thereby the learning curve. The syntax-directed editor is the only catch-all solution even when combinations are considered, and is therefore the solution of our choice.

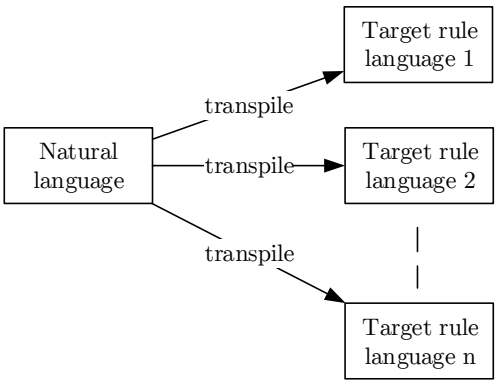

Figure 4.7: Hypothetical solution: transpiling from natural language to target languages that can be loaded into a rule engine.

4.4.2 By comparison with natural language

Another way to explain why the syntax-directed editor is the best solution is to see it as the next-best alternative to using natural language to program. One could see natural language as the ideal way for non-programmers to communicate their wishes to a computer, if it were not for the fact that that is, in its most direct sense, impossible. Natural language is, after all, used with great ease by all of us to communicate concepts of high complexity, this report being an example of that. The situation is displayed in Figure 4.7. As is to be expected, there are a number of issues with this. Firstly, the semantics of natural language are often ambiguous. Secondly, if it would not be ambiguous, capturing its vocabulary and grammar in a working parser is a world-class challenge.

A solution is to have an intermediary, non-ambiguous rule language. Each construct of this rule language can be parsed to natural language and displayed to the user. This fits the syntax-directed editor paradigm: the user can select from a menu of natural-language constructs that fit, thereby understanding what is going on the one hand, while keeping the program executable on the other hand. The structure is depicted in Figure 4.8.

4.5 Existing syntax-directed editors

The syntax-directed editor is not only being used for business-rules. It is a fairly old concept, and in the past decades several incarnations have popped up, notable examples being the Intentional Programming Workbench, SmartTools, Jetbrains Meta-Programming System (MPS) (see Figure 4.9) and Mbeddr (Figure 4.10). A study was conducted in 2014 by M. Voelter et al. [22], which investigated the usability of such systems, and Jetbrains MPS specifically. This study highlighted a set of usability issues with projectional editors, see Table 4.1. This list of issues needs to be considered when aiming to improve the usability of syntax-directed editors for rules. Some of the issues need some explanation.

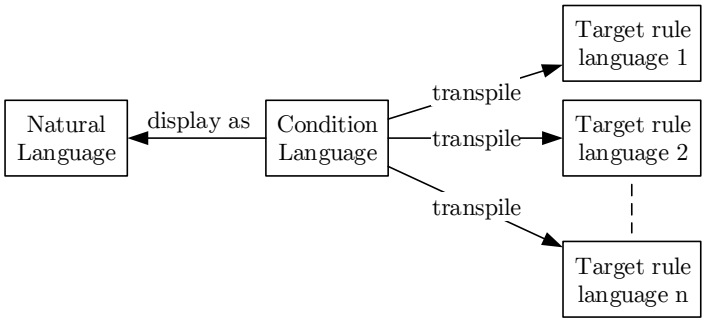

Figure 4.8: Transpiling from a machine-readable condition language to target languages and natural language.

Category	**ID**	**Issue**
Efficiently Entering (Textual) Code	EE1	Requires manual, user-based disambiguation
	EE2	Cannot establish references to non-existing nodes
	EE3	Requires structure-aware typing
	EE4	What you see is not what you type
	EE5	Requires notation-specific editor support
Selecting and Modifying Code	SM1	Selection is based on the tree structure
	SM2	Hard to perform cross-tree modifications
	SM3	Requires structure-aware copy/paste
	SM4	Does not support free-floating comments
	SM5	Requires dedicated support for commenting code
	SM6	Does not support custom layout
Infrastructure Integration	II1	Requires tool support for diff/merge
	II2	Text-based shell-scripting tools cannot be used
	II3	Requires tool support to export/import textual syntax

Table 4.1: Usability issues with projectional editors, found by Voelter et al. [22]

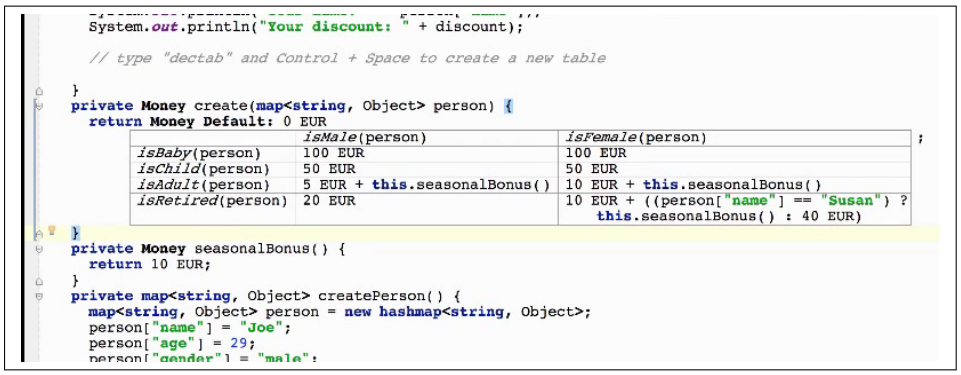

Figure 4.9: Example of projectional editing in JetBrains MPS

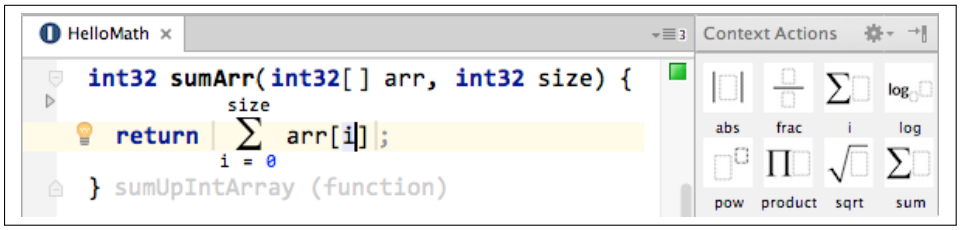

Figure 4.10: Example of projectional editing in Embeddr

Issue EE1 refers to the fact that constructs are chosen by text-input: the user can start typing code at some vacant node in the syntax tree, and the editor generates a subtree from the typed code. When the syntax of this code is ambiguous, as it is sometimes in projectional editors because projectional editors *allow* it to be, the editor has to prompt the user to disambiguate. EE4 and EE5 do not apply so much to a projectional editor as shown in Figure 4.4, where all projections are text-centered, as they do to more general projectional editors as shown in Figure 4.9 and Figure 4.10 that have non-textual projections of language constructs.

A more general conclusion Voelter et al. made in their research, from an enquiry among Jetbrains MPS users, was that projectional editing was generally seen as an efficient technique for every-day work, but that the effort of getting used to it is high. This provides another challenge for the design of Rash.

4.6 Conclusion

Several kinds of editor paradigms are used to make non-programmer users author business logic. However, available tools either are not stand-alone or separable from the software package they belong to, which is the case with the authoring environments that come with BRMS's, or they are tools that are very general-purpose and therefore providing a lot of configurability, making it hard to learn for an end user. Theoretically, one could separate Drools Workbench from the rest of Drools, but this likely requires a large reverse engineering

effort and adaptation on one hand, and much adaptation of Drools-specific code on the other hand.

The choice therefore falls on developing a new tool from the ground up. Doing this will provide us with full flexibility to address all user requirements in an optimal way, and to fully optimize the tool for usability by non-programmers.

Chapter 5

Main design decisions

This chapter aims to explicitly list which design decisions were made in order to address the challenges set out in Section 1.5. In the previous chapter a choice was made to design a syntax-directed editor, a decision recounted in Section 5.1. The subsequent decisions that were made to optimize usability for non-programmers are given in Section 5.2. Decisions regarding the type system, which is an important system both in guiding the user towards and in ensuring the user can only do correct solutions, are described in Section 5.3. How the languages of the rule's condition and consequence relate to each other is inferred in Sectin 5.4. Finally we discuss language extensibility and what challenges it addresses in Section 5.5.

5.1 Syntax-guided editor

As discussed in Chapter 4, but repeated for completeness here, examination of existing solutions led us to the decision for taking the *syntax-guided editor* approach to let non-programmers author rules.

5.2 Usability for non-programmer

5.2.1 Maximization of simplicity

When a rule author logs into the system, he only sees information that has relevance to his role as rule author for the applications he is entitled to author for; all other information is left out.

5.2.2 From a piece of text to a syntax tree visualization

In a syntax-directed editor, you are essentially not creating a text, but you are manipulating the abstract syntax tree directly. The text is generated *from* that. Existing syntax-directed editors for rules still look very much like a piece of text is being created [3][15][16]. From Table 4.1, issues EE1, EE3, EE4, SM1, SM2 and SM3 all originate from the fact that the editor still looks like a text editor, leading the user to want to treat it as text: from wanting to *type* the construct (s)he wants to use (EE1, EE3, EE4) to wanting to make text

selections disregarding the syntax tree structure (SM1, SM2, SM3). Because the *projectional editor* still wants to be similar to a *code* editor, problems arise, and the true potential of directly editing the underlying structure is not leveraged.

What a syntax-directed editor should do instead of trying to look like text is to communicate the structure that is actually being edited to the user: the syntax tree. The experience of editing textual code should be completely dropped in favor of editing a syntax tree, so instead of hiding the tree structure we will bring it out visually as clear as possible while keeping in mind that the author can still *read* the resulting projected text. Because a syntax tree or tree data structure in general is similar to a physical structure, for building ("authoring") one, lessons can be taken from building physical structures on a desk: it relies heavily on moving parts around with your hands (e.g. moving Lego blocks or jigsaw pieces around). This can be translated to a heavy reliance on dragging and dropping.

5.2.3 Type system awareness

We show the datatypes of expressions to the user. Not just at the empty leafs of the syntax tree (where input is required), but through the entire tree, so that the type of each expression is known before it is dragged-and-dropped elsewhere. This is not done in existing syntax-directed editors [3][15][16][2], instead, they only show the data type of an empty leaf requiring input. To continue the example of the last subsection, not showing the datatypes of expression to the user is like giving the user a jigsaw puzzle where all interconnects have identical shape: there is going to be a lot more trial and error, because the number of correct solutions has not changed, but the number of clues to direct the user toward them has decreased.

5.2.4 Automatically creating variables

Of Table 4.1, issue EE2 states that one cannot establish references to non-existing nodes. With rule editing variable scopes are fairly simple: there is the global scope, containing all global variables and variables representing the matches of a rule (the instances of entities against which a rule will match). Then if there is a function with a *lambda expression*, then that lambda expression introduces a new variable scope and one or more variables that are only defined within that scope. For instance, if we have an expression "... and there exists a person x such that x's name is 'John"', then x is the new variable and the part that came after the words "such that" is the new scope outside of which x is not defined. E.g., x is not defined in the part of the sentence denoted by "...".

Because of the simple variable scope system, creation of variables can easily be automated: global scope is assumed by default. If a variable is used in the variable-declaration part of a lambda expression (i.e. the "person x" part of "there exists a person x such that ..."), its scope is set to the corresponding function-body part of that lambda expression instead (i.e. the "x's name is 'John"' part of the example). The first action that is done (i.e. the first usage of a variable) that violates this scope constraint is the one that gets rejected:

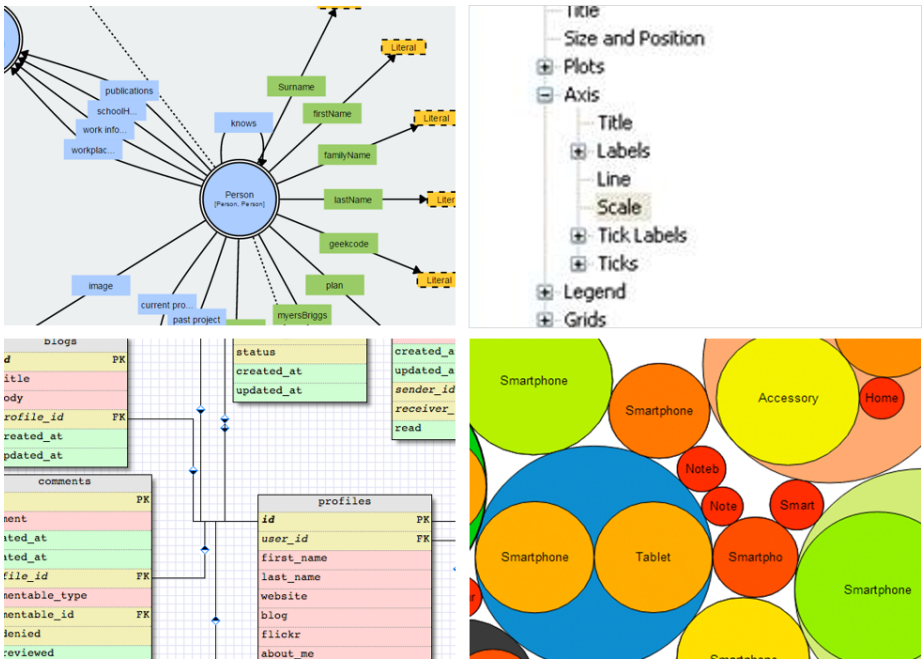

Figure 5.1: Ways to visualize the BOM: graph (t.l.), list (t.r.), schema (b.l.) and nested shapes (b.r.)

this can be a variable usage outside of that variable's scope, or this can be a variable usage that *sets its scope* such that some usages would fall outside of it.

If the creation of variables is automated, the user can move right on to referencing them. We can let the user do this by, in whichever place (s)he can choose a variable for usage, always presenting an unnamed alternative for every datatype. This nameless alternative can be used in the same ways as the named, already-created variables, with the sole difference that using it will give it a name and "create" it.

5.2.5 Visual object hierarchy guidance

Visualization style

The business object model can be viewed as a forest-like structure: it is a set of classes that have zero (in case of a primitive) or more properties, which besides having a property name are classes too. Properties can therefore have properties, creating multiple tree structures, i.e. a forest. We want to give the user a comprehensive and intuitive insight into the BOM and we can use the power of visualization to achieve this. For a forest structure, several visualizations are available, see Figure 5.1.

The multi-level list is a suitable way to encode a forest structure like the BOM, but has a tendency to become much higher than it is wide when it gets expanded, become harder to fit on a screen in its entirety. It also does not evoke

the analogy in the viewer's mind that the list items are physical objects that can be pointed to and that have properties, an analogy that is often used as a narrative when teaching the object-oriented programming paradigm.

The schema visualization, when being used for displaying entities and their properties, including relations, do not display the data structure as a forest. For instance, when we have entities Person and Car, and a car has a property "owner" of type Person, the schema visualization puts an edge named "owner" between the Car and the Person tableau's for that, instead of drawing a second Person-tableau and drawing a line towards that one. If it *would*, the view would get clogged very quickly. Also, properties that have complex datatypes, displayed as edges, and of properties that have primitive datatypes, displayed as items in a list, are treated completely non-homogeneously by the visualization. On the upside, the schema visualization does evoke some notion of physical objects.

The nested shapes visualization does evoke some notion of physical objects and the fact that objects and primitives that are properties are drawn inside the owner's shape confirms the notion of ownership. The latter however has a downside: it somewhat strongly suggests containment with every property, while properties can also merely be references. All properties are treated homogeneously, while the visualization of complex properties/entities extends seamlessly from the visualization of a primitive property/entity.

The graph-style visualization shows a shape for each entity or property like the nested shapes visualization does. A property's shape is displayed outside of the owner's shape, with a directed edge (e.g. with an arrow head) denoting the relation between the two. It therefore has the same advantages as the nested shapes visualization, but it lacks the disadvantage of suggesting containment. Therefore, our choice falls on this visualization.

Interactivity

The forest of a BOM can be infinite. Consider, for instance, a Person having a property "spouse" that is also of type Person. This recurrence creates an infinite path. Hence, the BOM can never be visualized in its entirety. Even if it could be, showing all of the BOM at once can overwhelm the user. For this reason, we use interactivity: properties are only shown when an object is clicked. When a primitive property or entity is clicked, nothing happens. When the user clicks the background, the visualization goes back to only showing the top nodes.

Force layout

A physical force layout is employed to let the nodes float around in, partially attracting and repelling each other, to strengthen the physical analogy.

5.2.6 User-defined functions

Authors can reuse their own and other author's creations with a user-defined functions feature. Pieces of AST can be diverted to a new function with the

help of a user friendly mechanism. A user-defined function will automatically become available at any other application if its dependencies (i.e. the functions and datatypes that are in its body) are also available.

5.3 Type system

5.3.1 Entity or no entity

Consider a rule engine for an insurance company with the BOM depicted in Figure 2.1. This BOM contains a set of complex types (classes), e.g. `Person` and `House`. It also contains at least one primitive type, e.g. `String`, the type of a property of `Person`. This type, besides being primitive, has another difference with `Person` and `House`: it is not an *entity* in the data model; one way of thinking of it is that there is no separate database table for all `string` values. As another example, consider the property `date of birth` of class `Person`, which is itself of type `Date`. This type is a complex datatype, but it is *not* an entity. Each type has a flag that says whether it is or is not an entity. Only if a datatype is an entity, it will be featured as a top-level node in the BOM visualization.

5.3.2 Constructibility

As has been concluded in Subsection 3.4.5, it should be possible to construct new objects. Classes have no methods in Rash, so new objects can only be returned by functions that have no object-oriented context —functions that are only dependent on their arguments. An object's constructor is also a function that depends on nothing but the class that it is creating and the constructor arguments; because of this similarity, we refer to the functions that create objects as *constructors*.

Some complex datatypes may have attributes that are all allowed to have undefined values, while others have properties that are not. The constructors of some objects may also be "smart", i.e., may not be a straightforward mapping between (a subset of) the entity's properties and the constructor's arguments. Therefore, Rash cannot automatically generate constructors for all objects. Hence, both kinds of entities can be accommodated by simply specifying in a datatype definition *if* the datatype is allowed to have a no-argument constructor. Only if this is specified to be *true*, Rash will provide the author with such a constructor for that datatype.

5.3.3 Generic types

In BOMs, collections are a common thing. Consider the BOM of Figure 2.1, which contains several collections: `Person.cars`, `Person.houses` and `Person.policies`, for instance. A collection is a datatype, but it is a *generic* datatype: it also has an item type. `Person.cars` is a collection with item type `Car` and `House.owners` is a collection with item type `Person`. In common object-oriented programming languages Java, C++ and C#, a generic type is denoted `Type<Parameter>`, e.g. `Collection<Car>`.

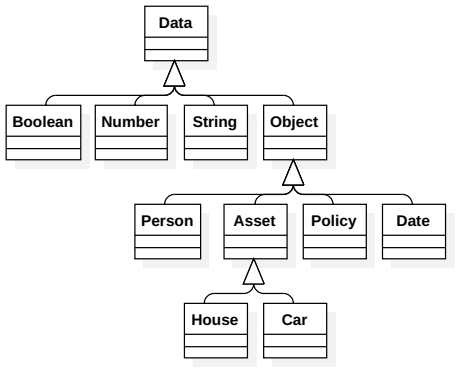

Figure 5.2: An example datatype inheritance tree

5.3.4 Enum types

BOMs, it often occurs that a property of some type —a number, or a string— can only take a finite set of values. For instance, a House may have an integer property 'carStorage' that may take only the values 0 (meaning none/public), 1 (parallel), 2 (angled), 3 (perpendicular), 4 (driveway) or 5 (carport). It would be ideal if not only the authoring-environments type checking system could prevent the user from entering faulty values (e.g. -1 or 6), but also only let the user interact the names/meanings of the possible values, instead of with the integer values, e.g. with "driveway" instead of "4".

5.3.5 Subtyping

Should subtyping be part of the type system supported by the BRAE? Suppose you want to have a function that checks if two objects are equal. Because every application has different datatypes coming with its BOM, this function would have to be respecified for every one of these datatypes. If you have a type hierarchy, however, you can define the equality function to work with an abstract type called, say, 'object'. Now, every datatype introduced can simply be specified to be a *subtype* of 'object', and is therefore understood to work as an input to the object equality function. Some sort of type hierarchy is considered essential for usability of functions.

Another design choice is whether to allow *multiple supertypes* to one datatype. The reason to do this is to allow polymorphism across the inheritance tree. Consider the example inheritance tree depicted in Figure 5.2. Suppose that we want a "greater than" function to work for both `Number` and `Date` parameters (provided that at each individual usage, both parameters are the same type). There is two ways to solve this.

1. Define the "greater than" function to work on some abstract type called, for example, `Comparable`. Both `Number` and `Date` datatypes, besides being subtypes of `Data` and `Object`, respectively, will *also* be subtypes of `Comparable`.

2. Define the 'greater than' function as requiring arguments of type `Number`. *Overload* this function to also accept `Date` arguments.

There are several problems with the first approach that illustrate why we should *not* support multiple supertypes.

1. One should consider the *time* functions are implemented. One would expect that the *greater than* function and the `Number` type would be reused by almost all applications. If *one* of those applications needs to make *greater than* work on a `Date` object as well, then the *greater than* function definition would need to be adapted to support an abstract `Comparable` type, and the `Number` type definition would have to be adapted to have `Comparable` as a supertype. To prevent every possible case of this, one would have to create an abstract supertype for every function.

2. Multiple inheritance is harder to communicate to the author. If we consider the inheritance graph, where the nodes are types and the directed edges correspond to direct subtype-supertype relations, then with multiple inheritance this graph is a directed acyclic graph, while with single inheritance it is a (poly)tree.

With function overloading, the downsides are

1. Functions that are overloaded support multiple distinct types, which may be more complicated to communicate to the user. However, this is likely to be less complicated to communicate than the myriads of abstract types likely to result from multiple inheritance.

2. It is not possible to *demand* that some methods are implemented. With multiple inheritance, this is possible. For example: suppose for `Comparable` both a *greater than* and a *less than* function are specified. Then it is impossible for a `Date` object to inherit `Comparable`, while *only* implementing the *greater than*, but *not* the *less than* function.

These issues are considered less problematic than the issues associated with multiple inheritance. Hence, we decide *against* multiple supertyping, but add the requirement that it should be possible to overload the types of function parameters.

5.4 Two languages per rule: condition and consequence

In a rule editor one needs to edit rule conditions and rule consequences: essentially two "programs" per rule that have distinct purposes, begging the question how these languages are related. After all, there are expressions that have meaning in both the condition and the consequence (e.g. "1 + 2") and there are expressions that do not (e.g. an action like "send notification" does not fit in the consequence). The namespace of the condition and the consequence are also related, as any variable used in the condition can be used in the consequence. As variables can be of all types this entails that all types in the condition language should be part of the consequence language.

Considering constructs, we should realize that the execution engine handles the code in a consequence differently. Consider for instance Drools as the execution engine. If someone would want to do a for-all quantification to calculate a boolean value, the syntax for this is different in the condition and in the consequence. Generally, for each authoring language construct you have to write separate compilers for the condition and for the consequence. However, we want to drag e.g. a "+" construct freely between the condition and the consequence in the authoring interface, even though if the rule engine implementation differs depending on its place. For this reason, for some more advanced condition constructs it may be considered not worth the effort to support them in the consequence, or some consequence constructs may be completely meaningless in the condition (e. g. any construct of type "procedure"). Note that the latter does not happen vice-versa: all condition constructs "make sense" in the consequence, since the top node of the condition is a boolean, and one can think of procedure constructs in the consequence that take a boolean as an operand. Anyhow, one wants to define constructs that are supported for both the condition and the consequence, and one wants to define constructs that are only supported by either one of them.

Now the next question is: should both consequences use the same set of types, or should there be consequence-only types? The answer is to use the same set of types, because a type that has no meaning in either part of the rule will never end up there if there are no constructs supported in this rule that make use of it. For example: if the condition's top node type is "boolean", an element of output type "procedure" will never end up in the syntax tree of the condition if there is no condition construct with an input type "procedure" or any supertype of that. Because it also simplifies things, we let the type sets for the condition and the consequence be equal.

5.5 Extensibility of the syntax

We aim to ensure that the language for rule writing used in our authoring environment can provide the highest possible level of abstraction while staying formal/exact. Consider Figure 5.3. The line denotes the level of abstraction of concept specifications (everything is a concept, including a business rule). At the top are a person's thoughts: the concepts are very abstract. If users could nevertheless program with their thoughts, e.g. program simply by imagining the designed solution, it is safe to say that their productivity would be very high relative to the productivity of a modern-day programmer. With natural language this would still be very high. However, computers work with formally specified, discrete concepts, not the fuzzy concepts that humans can work with. The optimal point for users to be programming in would therefore be the most abstract concept specification that is still formal. Figure 5.3 denotes this with the green dot.

Current-day high-level general purpose programming languages are probably the highest level of abstraction we can achieve without getting domain specific. If you want to go further up, you have to make assumptions about the context of the application, and a language is not suitable for all domains anymore. Because

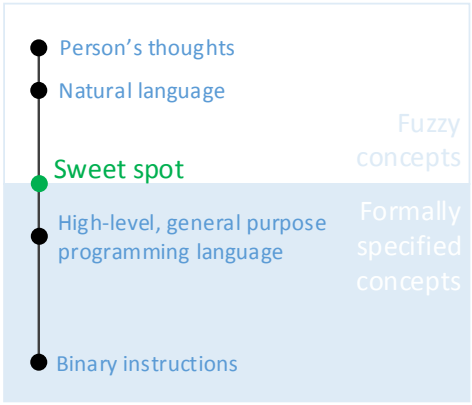

Figure 5.3: Range of abstractness, from instructions in the brain (highly abstract) to binary instructions on a processor (highly concrete). The sweet spot is where would want rule authoring to be.

in Rash an author manipulates the syntax tree directly, the need for parsing is bypassed. This allows it to provide a functional language in which function calls can have a *free form* syntax specified per-function. This allows maximum flexibility in specifying domain-specific syntax, with the only limitation being that the syntax has to be context-free. This is not a big problem, as the vast majority of natural languages have been found to be context-free [23].

Suppose a business wants to program on a high level abstraction to achieve the most productivity. Firstly, they use the highest-level generic purpose programming language out there to get their logic implemented. If a business can specify its own functions and add them to the rule language for general use or for specific applications, it would allow business to create a language more tailored to the application: they go up from generic-purpose towards the "sweet spot" in Figure 5.3, but only as much as their programmer's time would allow them to. To get closer, one could let rule authors introduce abstraction. Rule authors should therefore be able to combine expressions built with these functions to define more functions in a quick, user-friendly and easy-to-understand way, so they can do it autonomously and flexibly allocate their time between *introducing* and *using* these abstractions. In Figure 5.3, this brings us as close to the "sweet spot" as the time of a business's programmers and rule authors combined allows.

Summarizing:

- Software engineers can for any project at any time add new functions by creating a Rash definition for this function, and specify in it which rule engines support it, and implementing the parser for this function for every rule engine that is supposed to support it.

- Rule authors, in turn, can create new functions from expression trees he's created in a user-friendly way, which will be explained further in Chapter 6.

The former provides extensibility to the authoring environment: for any data source that might become available in the future, functions that use it can be added to the rule language. The latter improves the usability of the authoring environment.

Chapter 6

Initial design

6.1 Introduction

6.1.1 Chapter outline

The goal of this chapter is to propose the initial design of a BRAE named Rash. Consider Figure 6.1, which was featured earlier in the requirements section of this report. The system has three interfaces with its surroundings, and this chapter specifies all three of them: the editor that forms the authoring UI (Section 6.2), the language and BOM configuration (Section 6.3) and the rule *model* that comes out of the editor (Section 6.4). The language and BOM configuration is organized in modules referred to as *libraries*, and some essential libraries are proposed in Section 6.5. We conclude the chapter in Section 6.6.

Figure 6.1: The dashed line denotes the boundary of the proposed system, the arrows denote the interfaces with surrounding systems.

6.1.2 Choice of narration: constructs become functions

Whereas in the previous chapters we have repeatedly referred to constructs as the building blocks of a language, the authoring language that is part of the system presented in this chapter has *functions* as its building blocks. This is because it is a functional language; one that of which the syntax solely consists of calling/using functions; even *defining* functions is done outside the language.

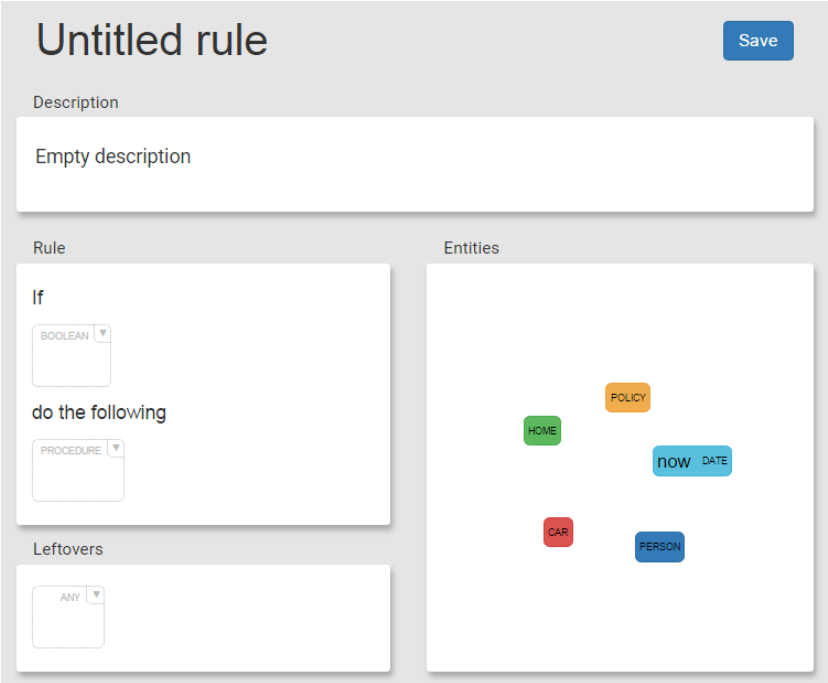

Figure 6.2: The Rash rule editor when starting with a new rule

6.2 Editor

6.2.1 Overview

Upon opening a newly created, unedited rule, the rule editor looks like is shown in Figure 6.2. The rule editor contains the following areas, with the section numbers that describe them.

- **Title**, **description** and **save button**. The title and description are editable in-place. The description serves as a quick and easy way to write the purpose of the rule down informally, and serves to make sure the original intention of the rule stays clear while authoring; and for quick and detailed identification of the rule to people who did not author it.
- **Rule panel**, described in Subsection 6.2.2
- **Entities panel**, described in Subsection 6.2.3
- **Leftovers panel**, described in Subsection 6.2.4

After explaining these GUI elements, the ways to interact with them in order to build rules are explained (Subsection 6.2.5).

6.2.2 Rule panel

The panel titled "Rule", contains the rule "code": in here we will build expressions for the condition (an expression of type "boolean") and the consequence

(an expression of type "procedure") of a rule that are formal enough to be automatically transformed to executable form. In Rash, expressions are built by manipulating the Abstract Syntax Tree directly. An example of a rule as it appears in Rash to the rule author while editing is shown in Figure 6.3. From this figure, you should note that the entire rule reads like an instruction in natural language; it can be read like a text from the top left to the bottom right.

Figure 6.3: A rule as it appears in Rash. This rule sends a notification to users on their birthday.

The parts of the rule that can be changed by the author are the two syntax trees corresponding to the condition and the consequence. They can be recognized by the two hierarchies of nested wireframe boxes. Each box represents an expression of the datatype that is specified in its top right corner in all caps font, a style that is maintained for datatypes throughout the application. Its contents may be either a function (a nonterminal, i.e. having child expressions), a literal value (terminal) or a variable (terminal). For example: the outermost box of this rule's condition is of datatype boolean, and it contains a logical conjunction ("... and ...") function. This function demands two child expressions of datatype "boolean". In this example, they have both been filled in with the equality ("... = ...") function. This equality function requires two child expressions of datatype "data". The types "number" and "month" are subtypes of "data". Therefore, the attribute of the "now" variable (which is —not visible in this partial screenshot— of type "date") called "dayInMonth" which is of type "month" could be used as a child expression. Similarly the other variables could be used too.

The top row displaying the "*For every [x Person] such that*" is automatically generated based on what the current syntax tree for the condition contains. It is an explicit recounting of the condition's *free variables* (in this case "x" of type Person).

Authoring language

Though the formulation of the instruction in Figure 6.3 is more verbose than how humans would normally say that someone's birthday is today, the sentence is correct and the rule is displayed in a *self explanatory* way. The ambiguity

and vagueness in the way humans communicate with each other means that this relative verboseness in *formally* specifying a rule can never be fully eliminated.

The *authoring language* is the language that the author sees. It is a *functional programming language* [24]. Because by building a syntax tree the parser is bypassed, the grammar may be ambiguous and does not need to be suitable for parsing. The tree hierarchy is displayed with help of boxes around the text, the user won't make parsing errors either. This together allows the *text* or *syntax* describing a function call (each box contains either a function call, a variable or a literal value) to take any form. In the case of Figure 6.3 it takes the form of natural language sentence describing what the function does, which is the intended use for Rash, though other syntaxes are possible. In order to make the assembly of nested natural expressions read like proper natural language too, the sentences that form the syntax of each individual function have to be *referentially transparent*. This boils down to the requirement that any entity in the sentence can be replaced by its definition and vice versa. Hence in Rash the natural language syntax of a function should be a definition or description of its output.

6.2.3 Entities panel

In the card titled "Entities" the types and variables making up the BOM are shown. Banners with a name represent variables; banners without a name represent objects that are not instantiated yet. Uninstantiated objects become instantiated upon using them (creating a reference to them in one of the syntax trees), meaning another banner with an automatically generated name will appear in the domain browser, along with the nameless banner, and the reference created in the syntax tree will refer to the newly instantiated object. Note that there is one object called `now` of type `Date` already instantiated. This is an object that comes with the Time library, something that was already implied by Figure 6.10, and it contains the current date and time. The result of instantiating an object is shown in Figure 6.4. This object was instantiated by dragging the nameless `Car` object to the empty input field on the leftovers card. Notice that a new variable with name "carA" appeared on the entities card, how the `carA` object is handled as an expression of datatype `Car`, and how a new empty field has appeared at the bottom of the leftovers card.

6.2.4 Leftovers panel

The panel titled "Leftovers", featured on the bottom left of Figure 6.2 and 6.4, can house any expression you drag into it —from single variables to entire expression trees—, and it has no upper limit on the number of expressions it can hold. You can also create new expressions in it. The sketchpad serves as "table space" to put things aside or to write down parts of the implementation of the rule. It allows the user to work in any order —if the sketchpad had not been there, the user would be forced to write the correct rule implementation top-down in one go in the Rule panel.

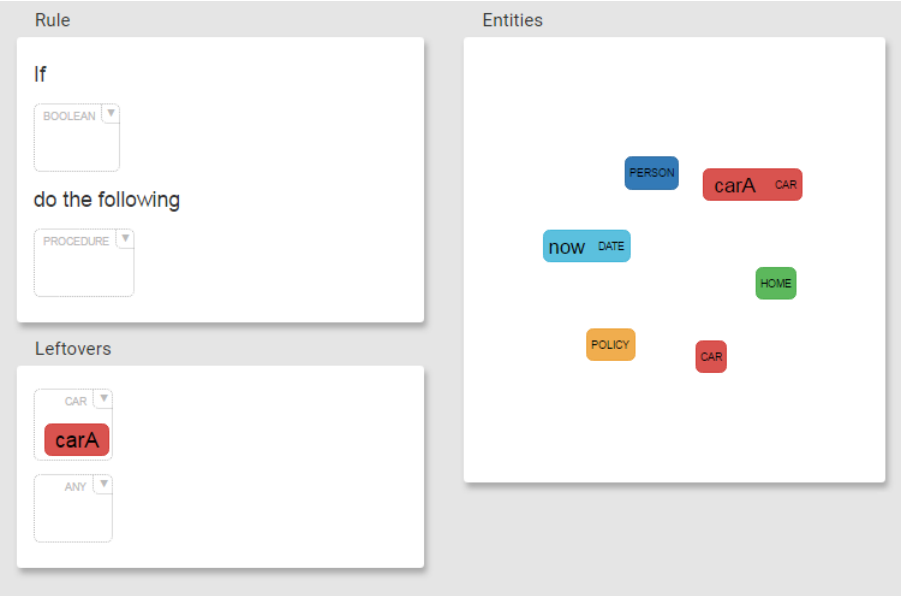

Figure 6.4: A `Car` object has been instantiated

6.2.5 Editing methods

Expression selection menu

Each expression box in the Rule and Leftovers panel features a small down-pointing arrow in the top right corner. When clicked, a dropdown menu appears in which an expression can be chosen as content for that box, see Figure 6.5. It shows all the functions with a fitting output datatype, the option to enter a literal value if allowed for the datatype and, if available, a few likely choices for variables that fit the datatype. Because variable attributes can be infinitely recurrent (e. g. if a `Person` has an attribute of type `Person`), it is often impossible to show a *complete* list of variables.

The top right corner of an empty expression box states the *allowed datatype*. In order for an expression to fit the box its (output) datatype must be a subtype (direct or indirect) of the allowed datatype of that box. Additionally, some functions also require the child expression to be of a specific nature (besides a specific datatype). The dropdown menu of the expression box adapts to this; e.g. when a function needs a root variable as a child (a variable that is not another variable's attribute), the drop-down of that child expression shows possible types to instantiate a variable, as shown in Figure 6.6 (it is still possible to drag a root variable into the expression box from the *entities* card).

Drag and drop

Expressions, variables and uninstantiated objects can be dragged and dropped. One can drag variables (so also attributes of variables) or uninstantiated objects (and also attributes of uninstantiated objects) to an expression in the syntax

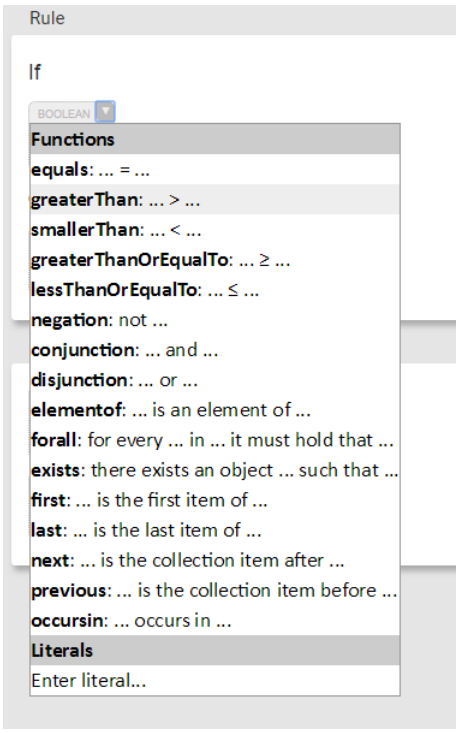

Figure 6.5: Clicking the drop-down button in the corner opens the expression selection menu.

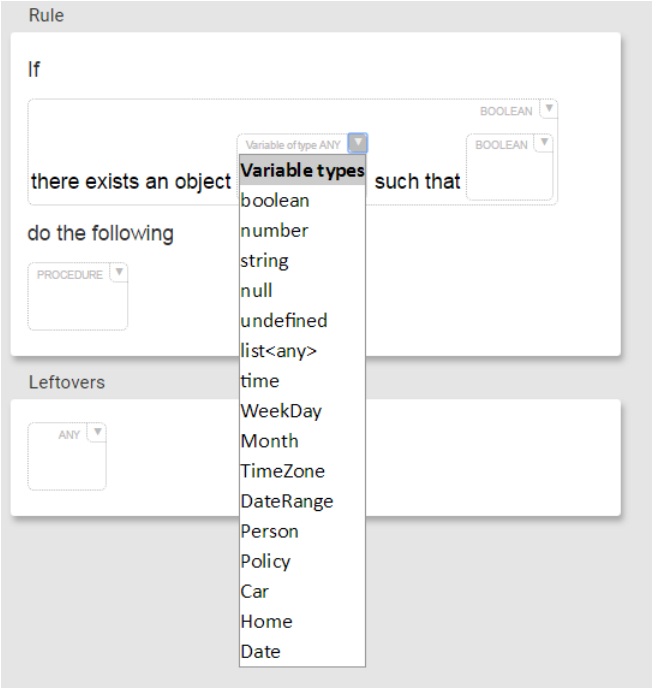

Figure 6.6: A drop-down for an expression that *has* to be a reference to a root variable.

tree. This replaces the target expression, but only if the type of the dragged variable inherits the allowed type of the target expression.

One can also drag expressions to other expressions. This removes the expression from its original location, and replaces the expression on which it is dropped. Again, the move is rejected if the dragged expression does not inherit the allowed type of the target expression.

Select, copy, cut, paste and delete

Selecting expressions is done by clicking on them. By holding CTRL or SHIFT, the user can select multiple expressions. Selecting an expression means one selects the entire subtree that has that expression as a root node. Selecting an expression and then one of its (indirect) children is therefore the same as selecting only the first expression. Deleting the selection is done by pressing DELETE. If one deletes the last usage of a particular variable that was instantiated by the rule, this variable also disappears from the *entities* card. Using CTRL+C or CTRL+X, one can copy or cut the selected expressions, respectively, and the expressions are hereby placed on the clipboard. When using CTRL+V subsequently, the expressions (meaning, the expressions including all their children) get appended to the bottom of the *leftovers* card.

Undo and redo

For any UI where the user makes some kind of creation in which mistakes can be made, undo and redo are two essential operations. They are available under CTRL+Z and CTRL+SHIFT+Z, respectively, or as buttons in the top right corner, left of the save button. To keep the GUI minimal, they are only visible when applicable, i.e. the undo and redo buttons are only visible if and only if there is at least one action on the undo or the redo stack, respectively. The undo and redo stacks are not saved along with the rule.

Repeating parts of a function

When an expression is selected that contains a function, operand-repetition buttons become available where they are applicable. Consider for instance the "... and ..." function. For this function it makes sense to have more than two operands. When selecting an expression that uses it, it appears as in Figure 6.7. Clicking the "+" button next to the 2nd operand will expand the function the read "... and ... and ...".

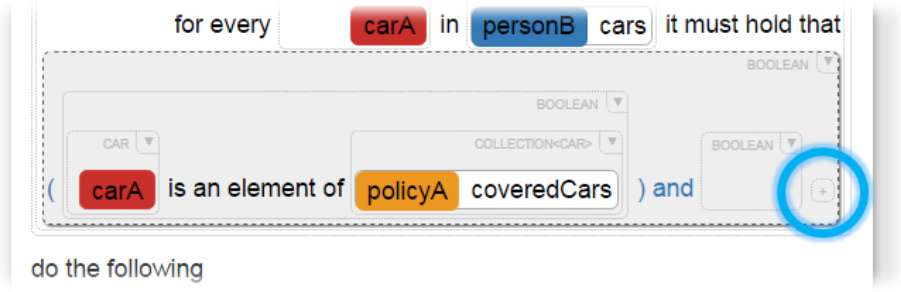

Figure 6.7: Function expansion button (encircled in blue), visible when expandable function (here: a conjunction) is selected.

6.2.6 User-defined functions

After evoking the context menu on an expression box (e.g. by right-clicking), one can select "Define function...", see Figure 6.8. By selecting this option, a pop-in opens in which the author can define a function out of the entire subtree of the right-clicked expression, that is, the expression and all its child expressions. We will refer to this subtree as the *function body*. In Figure 6.8, an expression is selected that says that variable `personB` of type `Person` has his or her birthday today. Indeed, this could be said in a shorter way, and so we want to define a function for this.

Figure 6.9 shows the pop-in for defining the function. The right-hand column shows a list of arguments. Here you can select the argument you wish to edit. Clicking the bottommost, transparently printed argument will allow you to create a new argument.

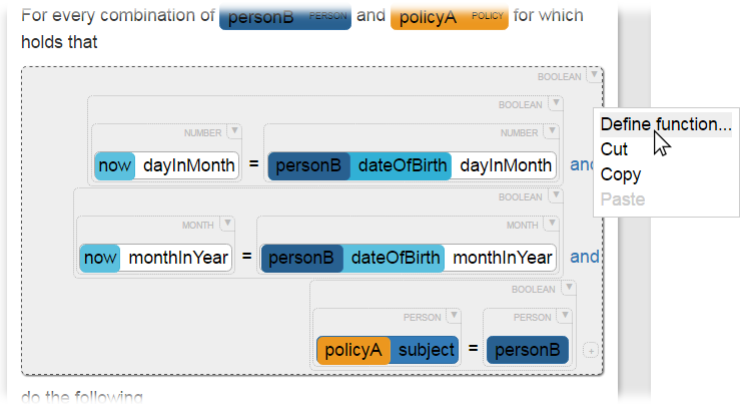

Figure 6.8: The context menu is used for defining a new function from a subtree

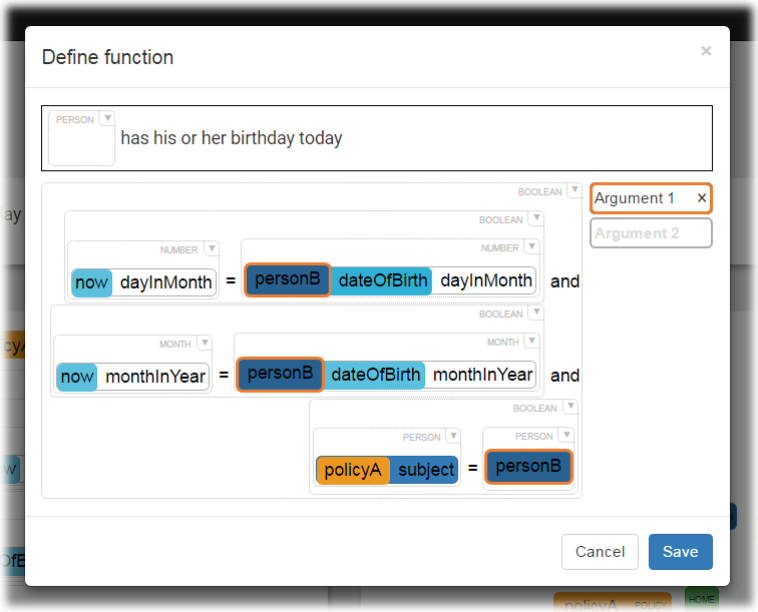

Figure 6.9: Pop-in in defining a new function.

After selecting an argument, you can click in the function subtree what child `Expression` or `RootVariable` you wish the argument to correspond to. Not all choices you can make are allowed. If you make an illegal choice, your action will be rejection and you will get a pop-in with the reason why. Table 6.1 lists cases when arguments are illegal, and the reasons why. Furthermore, sometimes there are `Expressions` or `RootVariables` that *have* to be in an argument in the final function. To this end, arguments satisfying this are already instantiated when the function definition pop-in opens. When you try to remove or change these such that the constraint is no longer met, your action is rejected and you get a pop-in with the reason why. Table 6.2 lists when arguments are obligated and the reason why.

Argument corresponds to...	Reason for rejection
Subtree that contains a variable that is also used elsewhere in function body.	If the function would be reused with different content in this argument, the unification link between the variable now used in the argument and in the other place(s) in the function body will be broken, leading the reused function to behave differently than the original subtree from which it was defined.

Table 6.1: Restrictions on the parameters of author-defined functions

When	Reason for obligation
When a variable is free with regard to the function body, i.e. it is not a bounded variable of which the corresponding function is in the function body, this variable *has* to be contained in (the subtree of) an argument of the function.	If this is not enforced, replacing the original subtree with the function that is based on it will cause the usage of this variable to disappear, leading to breakage of unification links with other uses of the variable and/or to the disappearing of a match of the rule.

Table 6.2: Restrictions on the parameters of author-defined functions

6.3 Language & BOM configuration

This chapter describes how the rule language and BOM are configured. First, it is argued that configuring either of them involves instantiating objects from the same set of classes in Subsection 6.3.1. There it is also explained that these instantiations are grouped into *libraries*. Subsequently, the classes that need to be instantiated to configure the language and the BOM are presented. These are `Function`, elaborated in Subsection 6.3.2, `Type`, elaborated in Subsection 6.3.3 and `Variable`, elaborated in Subsection 6.3.4.

6.3.1 Organization of the language & BOM configuration

Figure 6.10 shows the complete content creation workflow of Rash. Let us start by focusing on the rightmost part of the chart. The arrows from the Domain-Specific Rule Metamodel, or simply Rule Metamodel, to the Rule Model denotes the process of authoring the condition and consequence syntax trees. Instantiating variables happens automatically by simply referring to them as described in Subsection 5.2.4 and 6.2.5.

Before there can be authored, however, there needs to be a business rule language and a BOM, as they are part of the Rule Metamodel. The term *rule language* is a somewhat misleading shorthand, because technically the rule *language* is fixed: it is a functional language in which the *functions* may vary per application domain. It is not possible to define these functions in by using

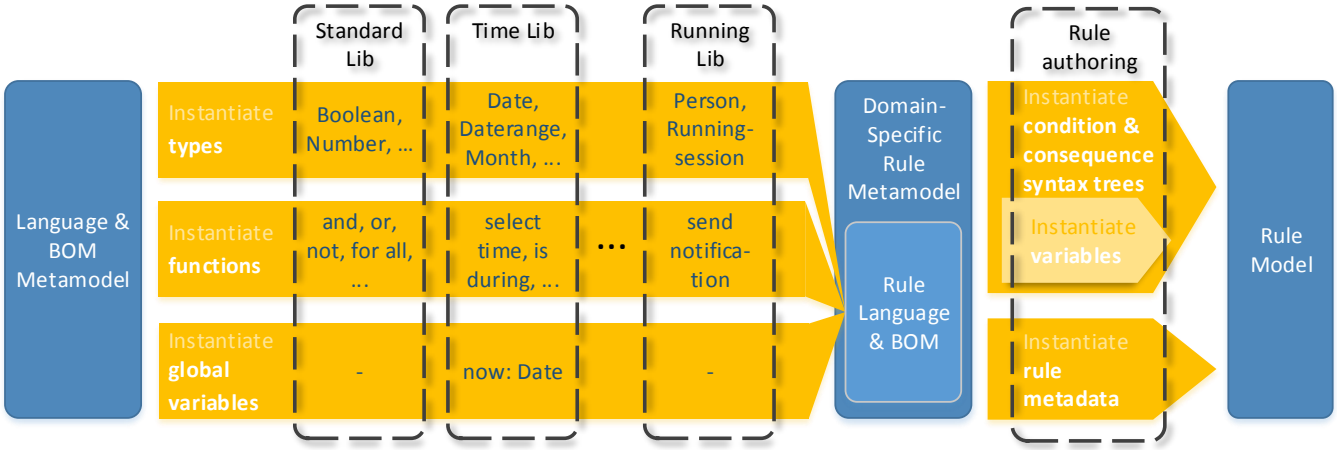

Figure 6.10: As part of the Domain-Specific Rule Metamodel (or simply: Rule Model) the rule language and BOM need to be defined. One does this by defining a set of libraries, each being a set of types, functions and globals. The libraries detailed in this figure serve as examples. From left to right, they increase in domain-specificness.

the rule language itself; the rule language is for writing rules and in rules it is only possible to *use* functions. Therefore, the syntax is completely shaped by the *functions*, *types* and *variables* that form the Rule Language & BOM. These three classes, `Type`, `Function` and `Variable`, therefore form the core of the Language & BOM Metamodel, featured at the left of Figure 6.10.

The BOM usually consists of a set of business object classes, which are *types*. It is, however, not impossible for a business to have a singleton or a service, i. e. a class of which there is only one instance in their business object model. For this reason, besides classes, a business object model may consist of *variables* too, and it may also have classes defined that the rule author cannot instantiate. Furthermore, one may want to define domain-specific functions. Common bundles of functionality that are of interests to many applications —like first-order-logic and number arithmetic— can be composed of a set of types and functions. Some bundles, like reasoning about time, may need global variables too: for instance, a "current time" instantiation of the a date-time datatype. The bottom line is: a "common bundle of functionality" is not all that different from the definition of a BOM, as they may both need functions, types and variables.

For this reason, the notion of *libraries* is used. Each library consists of a set of *types*, *functions* and *variables*. A few libraries have been recommended as a part of this work, they are described in Section 6.5. The BOM for an application is provided by just defining another library that contains the necessary types. Figure 6.10 displays this process. For each application, or for each rule engine if

an application has multiple rule engines (e. g. server-side and client-side), before rules are being authored for it, one can decide which libraries will be supported. Also, which rules depend on which libraries completely defines for which other rule engines the rules can be reused.

6.3.2 The `Function` class

Functions appear in the user interface as something similar to what is depicted in Figure 6.11.

Figure 6.11: Example of the way a function should look. This is a *for-all* function.

The information we need for a function like this is the type signature (e.g. "`Any`" or "`Collection<Any>`") of every parameter, and what text comes in between the parameters (e.g. "for every", "in", ...). Moreover, some functions (also the one in Figure 6.11) have pairs of input fields that combined function as a single lambda function being passed as input to the function. Also, some operands may be any kind of expression with the right datatype, while other operands may *have* to be a reference to a type, a classproperty, a variable or a root variable (i.e. not a property of some other variable). A UML class diagram of the metamodel of a function is given in Figure 6.12.

In the following paragraphs we will first cover type signatures in more detail, then the repeatability of operands, then the surrounding text forming the syntax of the function and finally we cover lambda relations.

IC0	`LambdaRelation.lambdaVar.expressionType = 'rootvar'`
IC1	`LambdaRelation.lambdaExpression.expressionType = 'expression'`
IC2	$\forall_{o \in \texttt{Function.operands}}$ `|Construct.templates| = |o.templates|`

Table 6.3: Invariants concerning function definitions in Rash

Type templates

Consider the function depicted in Figure 6.11. If someone would fill in the first parameter with an object of, say, type `Car`, the allowed type in the second parameter would become more specific, namely `Collection<Car>` instead of `Collection<Any>`. This is because the "any" types in the type signatures of the first and the second parameter are coupled. In Figure 6.12, this is modeled by having a single `TypeEqClass` with a property `max: Type` pointing to the `Type`

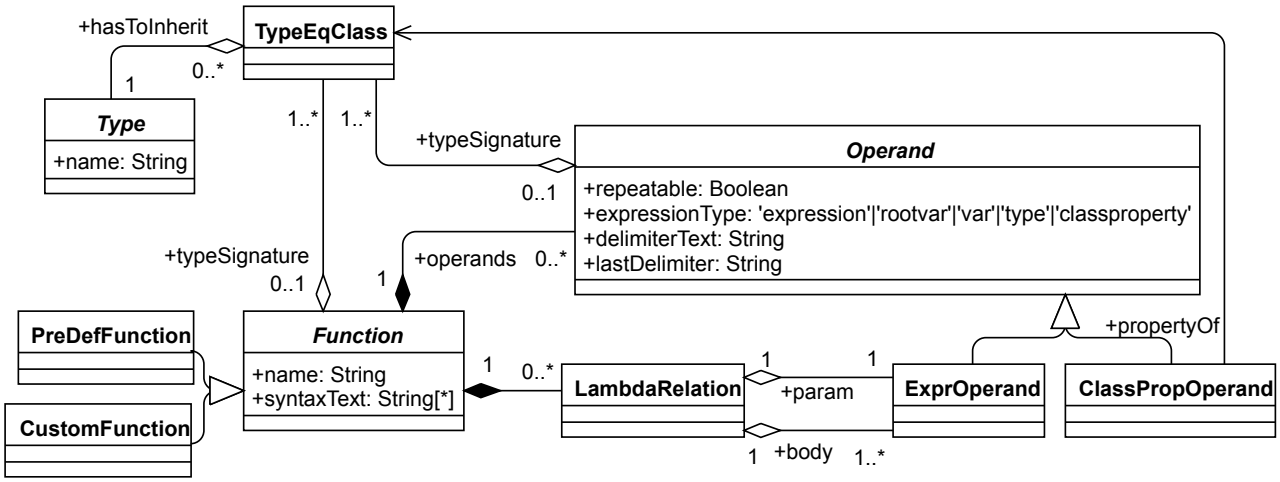

Figure 6.12: UML class diagram of the metamodel of a `Function`. Invariants from Table 6.3 hold.

with name `Any`. This `TypeEqClass` is referenced from an array of `TypeEqClass`es called *typeSignature* of both the first and the second `Operand`. When functions are assembled into a syntax tree, the `TypeEqClass`'s of neighboring functions are hooked up so that similarity constraints such as this can propagate through the syntax tree. Rash makes sure the *allowed* and *current* datatype of every expression in the syntax tree is kept up to date accordingly.

Type overloading

Notice from Figure 6.12 that the class properties `Function.templates` and `Operand.templates` are plural: `Function` and `Operand` may have multiple templates. This is useful for *overloading* a function. For example consider a function *greater than* from a standard library. This function is only defined on numbers. Suppose that for some application, a software engineer has a datatype named `Date` representing the date and time, and wants the *greater than* function to be defined on that datatype. He can then *overload* the *greater than* function, and specify a second set of templates for it (i.e. a template for the output and a one for each of its operands). The function will then appear as in Figure 6.13.

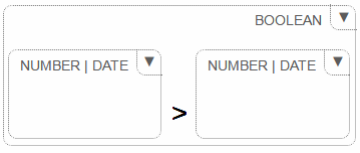

Figure 6.13: GUI appearance of a function that is overloaded to support a type named `Date`

IT0	`GenericTypeInstance.param` equals, or is a direct or indirect subtype of `GenericTypeInstance.instanceOf.param`

Table 6.4: Invariants concerning types in Rash

Repeatability

Sometimes, one might want to define a function in which operands are repeatable. Consider, for instance, a conjunction. The syntax may be "... and ...", with only two operands. But if one were to make the first or the second operand "repeatable", with " and " as the delimiter text, then the function could take from 2 up to infinite operands, e. g. "... and ... and ..." would be possible too. In the user interface, if applicable to an operand, the user gets the option to add another operand, or remove one.

Text surrounding the parameters

Before the first operand, between every operand and after the last operand, text can be configured to appear. A function that has two operands and a `Function.syntaxText` array of `['there exists', 'such that', '']` yields the syntax "there exists ... such that ...". If the first operand's `repeatable` attribute is set to TRUE, its `delimiterText` and `lastDelimiter` left to their default values, then the syntax "there exists ..., ... and ... such that ..." becomes possible.

Lambda relations

Notice that in Figure 6.11, the variable that is specified in the first parameter can only be used in the expression that will be placed in the third parameter, and not outside. This is because this variable is a *bounded variable*. In fact, parameter 1 and 3 together specify an *anonymous function* or *lambda function*, with an input variable (parameter 1) and an expression calculating the output (parameter 3). In the execution engine, this lambda function itself, instead of just its calculated output, is passed to the for-all function as a whole to apply it to every element of the collection given in parameter 2. The restriction that lambda functions use variable definitions (parameter 1) which may only be used in the lambda *expression* (parameter 3) must be enforced in the GUI and therefore the presence of a lambda relation must be modeled explicitly.

6.3.3 The `Type` class

A class diagram of the datatype metamodel is shown in Figure 6.14.

6.3.4 The `Variable` class

A class diagram of the variable metamodel is shown in Figure 6.15. Variables can be either `RootVariable` or `ObjectProperty`. If a variable is an `ObjectProperty` is does not have a name of itself, but it is owned by another variable.

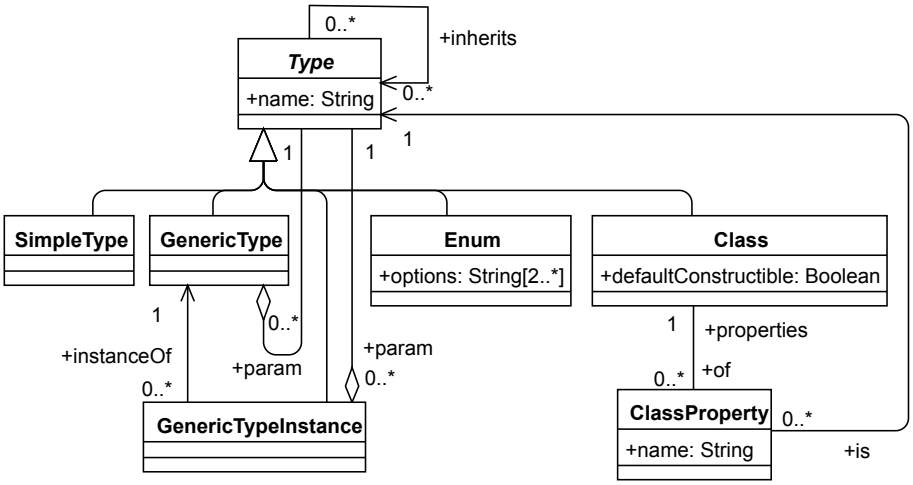

Figure 6.14: Metamodel for `Types` in Rash. Invariants of Table 6.4 hold

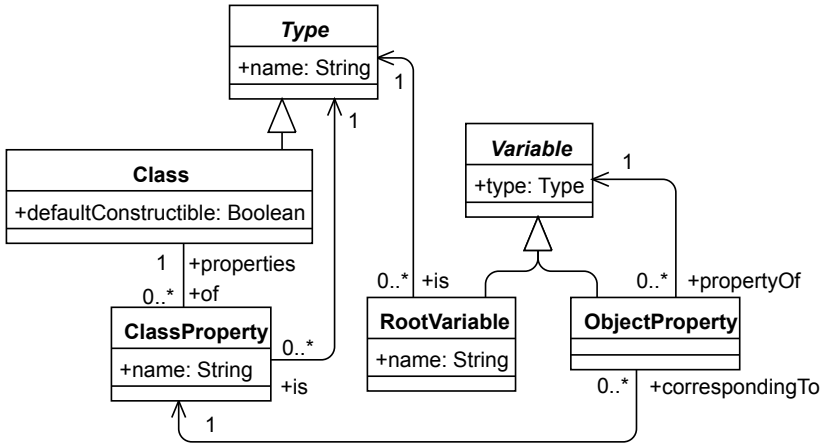

Figure 6.15: Variables in Rash. Invariants from Table 6.5 hold.

IV0	`ObjectProperty.propertyOf.type =` `ObjectProperty.correspondingTo.has`

Table 6.5: Invariants concerning variables in Rash

6.4 The Rule Metamodel

A UML diagram of the `Rule` class is given in Figure 6.16. A rule has a title, e. g. "Birthday notification" and a textual description, which the author can optionally use to write an initial, informal specification of the rule, e. g. "Send a birthday notification to a person on his or her birthday". The rule has one expression for its condition, one for its consequence and any number of expressions as "sketches", which correspond to the expressions built in the leftovers panel (see Section 6.2.4). The expressions of a rule are in a context-free grammar.

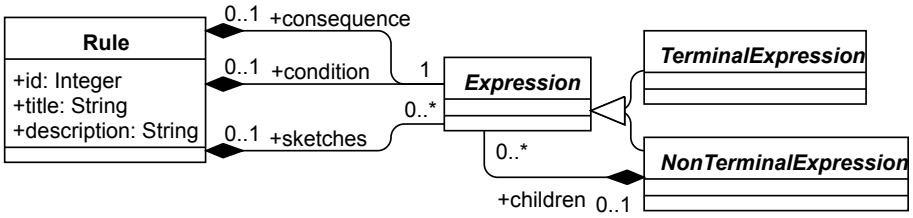

Figure 6.16: Class diagram of the metamodel of a rule

Therefore, an expression consists of a syntax tree, structered as denoted in the right half of Figure 6.16. We distinguish between a `NonTerminalExpression`, which is an expression that has child expressions, and a `TerminalExpression`. Any `Expression` exist as a child of some `NonTerminalExpression` or of the rule, as a sketch, the consequence or the condition. Now consider the metamodel of a Rash syntax tree in Figure 6.17. It is essentially the same model as in the right half of Figure 6.16, but more elaborate.

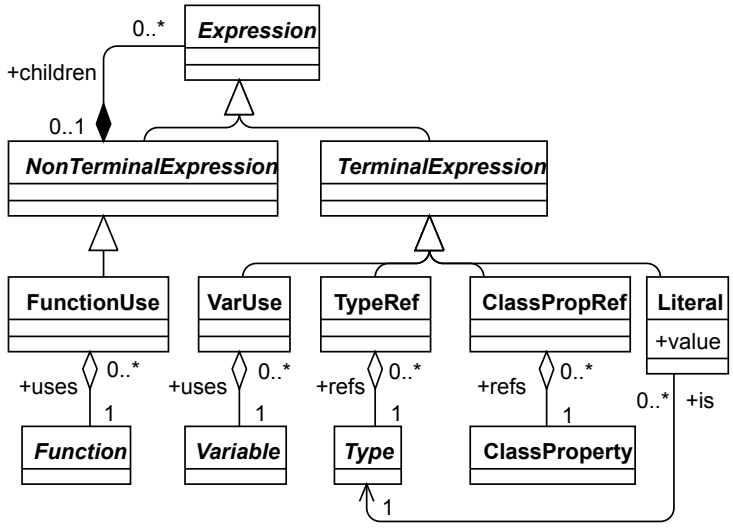

Figure 6.17: More elaborate metamodel of a syntax tree in Rash, showing all possible kinds of expressions and the things they refer to.

This figure shows a `NonTerminalExpression` is always a `FunctionUse` which, as the name says, uses a `Function`. `TerminalExpressions` can refer to any other elements of the language metamodel: a `VarUse` can refer to variables which are *instances* of types or their (in case of classes), while `TypeRef` and `ClassPropRef` can refer to the types and their properties themselves. A `Literal`, as the name says, refers to a literal occurrence of an instance of a type. Note that all of the classes that can be referenced by the syntax tree, the classes on the bottom row of Figure 6.17, have already been covered in Section 6.3.

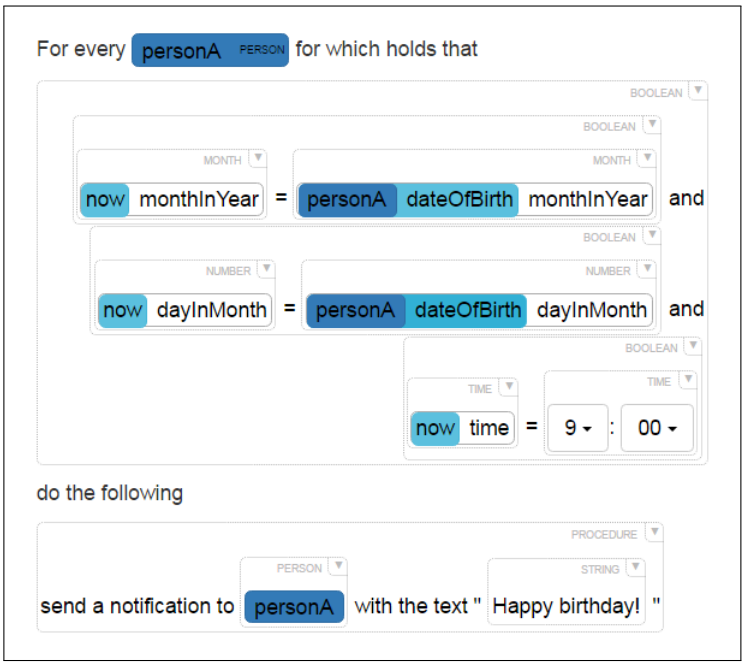

Figure 6.18: An example rule

An example instantiation of the rule metamodel is given in Figure 6.19. It depicts part of the rule of which the GUI-equivalent is given in Figure 6.18.

6.5 Standard libraries

6.5.1 First order logic library

Because a condition of a rule is essentially a proposition that should evaluate to TRUE for some selection of entities, a language that comes to mind is First Order Logic extended with arithmetic for comparing numbers, strings and other types (i. e. with $>$, $<$, $=$, etc.), and sets (with $\in$, $\subseteq$, etc.). The set of functions for first-order logic included for RASH is given in Table 6.6.

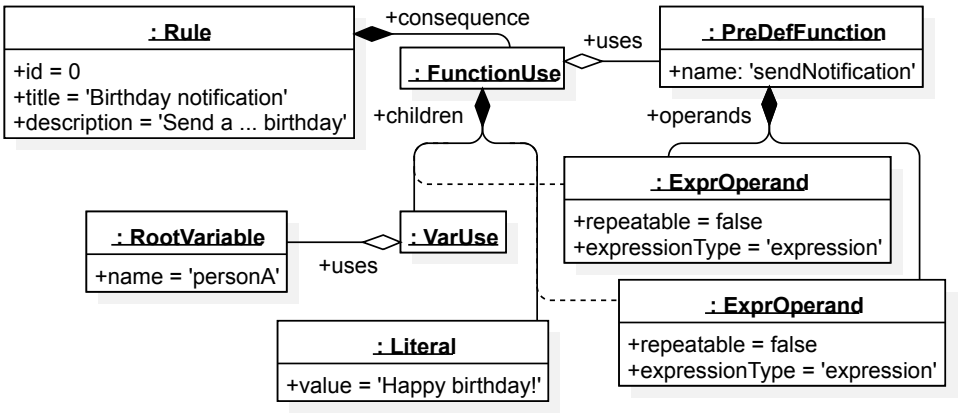

Figure 6.19: Object diagram depicting a partial (i.e. without the condition) model of the rule of Figure 6.18.

Function name	**Function signature**
negation	not `bool` $\longrightarrow$ `bool`
conjunction	`bool` and `bool` $\longrightarrow$ `bool`
disjunction (inclusive)	at least one of `bool` or `bool` is true $\longrightarrow$ `bool`
for all	for every `data` in `collection<data>` it must hold that `bool` $\longrightarrow$ `bool`
exists	there exists a `object` such that `bool` $\longrightarrow$ `bool`
equals	`data` $=$ `data` $\longrightarrow$ `bool`
greater than	`number` $>$ `number` $\longrightarrow$ `bool`
lower than	`number` $<$ `number` $\longrightarrow$ `bool`
greater than or equal to	`number` $\geq$ `number` $\longrightarrow$ `bool`
lower than or equal to	`number` $\leq$ `number` $\longrightarrow$ `bool`

Table 6.6: First-order-logic functions

Function name	Function signature
addition	`number` + `number` ⟶ `number`
subtraction	`number` - `number` ⟶ `number`
multiplication	`number` × `number` ⟶ `number`
division	`number` / `number` ⟶ `number`
power	`number` ∧ `number` ⟶ `number`

Table 6.7: Arithmetic functions for the rule condition language

6.5.2 Number arithmetic library

From the requirements it is clear that we need basic number arithmetic, so we provide this with the functions in Table 6.7.

6.5.3 Aggregation library

Based on the requirements posed in Section 3.4.6, a set of Rash functions is proposed, given in Table 6.8.

6.5.4 Time library

A big part of language is reasoning about time. A good example is in a rule condition like "*If a user brushes at least 2 minutes, 2 times a day, every day, for 2 weeks.*" To elegantly specify this expression, one would need to use a "for all" on a "collection of days". More generally, one might want to make a selection of time based on any criterium, and then cluster it to get a collection of these time selections. A set of requirements to reason about time has been defined, given in Table 3.5. Based on these requirements, a set of Rash functions is defined, given in Table 6.9.

6.5.5 Event library

In ECA rules, the condition, besides containing facts about business objects, contains a statement *that an event happens.* The moment an event happens this statement is true, so it makes sense to make the output type of such a function a `bool`. For the input type, the event, we introduce a new datatype `event`. The functions of this library are given in Table 6.10.

6.5.6 Action library

Table 6.11 lists functions that allow basic data alteration actions, and a function that allows you to do multiple actions in sequence.

6.6 Conclusion

We have proposed a design for an authoring environment that tackles the challenges outlined in Chapter 1. There is plenty of design aspects that has not been treated here —user rights management, version control, interfacing with rule engines, to name a few— because they are too far from the core problems that are the focus of this research.

Function name	Function signature
set builder	the set of all variables `object` for which holds that `bool` ⟶ `collection<object>`
element of	`data` is an element of `collection<data>` ⟶ `bool`
count	the number of items in `collection<data>` ⟶ `number`
max	the maximum value in `collection<number>` ⟶ `number`
min	the minimum value in `collection<number>` ⟶ `number`
average	the average value in `collection<number>` ⟶ `number`
sum	the sum of all values in `collection<number>` ⟶ `number`
sum of a property	the sum of `attribute of object` of all the objects in `collection<object>` ⟶ `number`
average of a property	the average `attribute of object` of all the objects in `collection<object>` ⟶ `number`
min of a property	the minimum `attribute of object` of all the objects in `collection<object>` ⟶ `number`
max of a property	the maximum `attribute of object` of all the objects in `collection<object>` ⟶ `number`
first	`data` is the first item in `collection<data>` ⟶ `bool`
last	`data` is the last item in `collection<data>` ⟶ `bool`
next	`data` is the collection item after `data` ⟶ `bool`
previous	`data` is the collection item before `data` ⟶ `bool`
sort by attribute	sort `collection<object>` by `attribute of object` ⟶ `collection<object>`
sort	sort `collection<sortable>` ⟶ `collection<sortable>`
n first items	the first `number` elements of `collection<data>` ⟶ `collection<data>`
n last items	the last `number` elements of `collection<data>` ⟶ `collection<data>`
cluster collection	cluster `collection<object>` by `attribute of object` ⟶ `collection<collection<object>>`

Table 6.8: Aggregation functions for the rule condition language

Function name	Function signature
time selection	all moments `date` between `date` and `date` for which holds that `bool` ⟶ `daterange`
occurs during	`date` occurs during `daterange` ⟶ `bool`
cluster time selection	cluster `daterange` by `attribute of date` ⟶ `collection<daterange>`

Table 6.9: Time functions for the rule condition language

Function name	Function signature
occurs	`event` occurs ⟶ `bool`

Table 6.10: Functions for the event library

Function name	Function signature
sequence	`Action`; `Action` ⟶ `Action`
create	create a new object of type `Object` ⟶ `Action`
update	set `data` to `data` ⟶ `Action`
delete	delete `Object` ⟶ `Action`
append	append `data` to `Collection<Data>` ⟶ `Action`
remove	remove `data` from `Collection<Data>` ⟶ `Action`

Table 6.11: Functions for the event library

A prototype with core features was built, focusing on the GUI front-end. With this prototype, a series of user tests were conducted, and the next chapter describes the tests along with their findings.

Chapter 7

Prototype and user testing of initial design

7.1 Introduction

Testing is an integral part of a scalable software development process [25] [10]. User testing specifically is important in the development of a valid user interface. It is, contrary to testing of most components, both part of the software's *verification* and *validation* [26] processes. This means that the requirements to the software change as a result of the findings from the user tests. This thesis contains one such iteration.

The structure of this chapter is as follows. We first clarify what parts of the initial design were included in the prototype in Section 7.2. The testing method is explained in Section 7.3. Events pointing towards a hindered usability that occurred during the tests, and the usability problems that were deduced from them, are given in Section 7.4.

7.2 Prototype

A prototype was made of the initial design proposed in Chapter 6. Some features could be implemented only partially and some not at all, but here is a list of features that made it into the prototype.

7.2.1 List view

When the prototype is opened, a rule list view is seen. The application domain can be selected in a drop down menu in the right corner of the screen in the title bar. When opening the application, the first application domain in the menu is selected.

The rule list shows each rule's title with its description below it. Clicking on a rule opens it, navigating the user to the rule editor. When hovering a rule's table row a red delete button becomes visible in the right end of the row. In

the bottom right corner of the screen, with fixed position (i.e. relative to the browser window), a yellow "+" button appears. Clicking it creates a new rule titled "Untitled rule" and with description "This rule's description goes here" and immediately adds it to the bottom of the list.

7.2.2 Editor view

All panels and editing methods described in Chapter 6 were implemented. However, not all features of the metamodel were implemented, which manifests itself in the editor:

- Overloaded function type signatures
- Repetition of individual operands (only one repeating operand per function was supported)
- Constructing complex datatypes
- For a `ClassPropOperand` only the type template of the *owner class* could be specified, and not the template for the type of the property itself
- The constraint that a variable usage in a subtree of a variable that has a scope larger than that subtree *has* to be part of one of the operands if that subtree were to be used to define a function, has not yet been implemented.

7.2.3 Persistence

Rules were persistent, i.e. they could be saved. Custom function definitions could not be saved. This also led to an error if someone tried to load a rule that was saved with a custom function in its body.

7.3 Testing method

The subject was given a laptop with a prototype of Rash running in full screen mode. A video capture was made featuring only the screen (video), the computer audio and audio in the room (through the laptop's built-in microphone). Assignments that the user had to carry out in Rash were written down on paper, and not revealed before prior assignments were completed. Prior to all tests, the user was...

- asked for permission to record the screen, the computer sound and the sound in the room;
- reminded that the test is about the application's performance, and not theirs;
- instructed to vocalize their thoughts whenever they can, especially if they would not know how to proceed;
- reminded that they can ask questions to the tester, even if the tester cannot answer all of them for the sake simulating the situation that the user will be on his own;

#	Subject's relevant experience	Example rule walkthrough
1	None	-
2	None	+
3	None	+
4	Professional software engineer	+
5	Specifying rules to programmers part of job; had a few programming assignments in University	+

Table 7.1: User tests

- explained the background and purpose of business rules, as well as what they are and how they are structured.

A total of 5 user tests were conducted, see Table 7.1. The table also features if beforehand a walkthrough of building an actual business rule was given.

7.4 Usability issues found in user tests

The user test analysis is as follows. First, events that are testifying of a hindered usability are collected, which is done in Subsection 7.4.1. A set of problems with the application that were causing these hindrances were deducted from this in Subsection 7.4.2. Solutions to these problems are considered not in this chapter but in Chapter 8.

7.4.1 Events

All events that manifested problems with the usability —some of which occurred often, others only once— are given in Table 7.2. A thin layer of interpretation has already been applied before writing this down —this is necessary because otherwise all events that related to a subject saying something would be unique, and this list would be impractically long.

7.4.2 Problems with the design

A list of problems was compiled that are likely to be the cause of the events listed in Table 7.2, and this list is given in Table 7.3. For these problems, a set of solutions was proposed in Chapter 8. The extra step of defining problems first, instead of finding solutions for unwanted events directly, was done because solving broad, conceptual problems leads to a more coherent product that finding individual solutions for many tiny problems.

Event description	**Problem**
S shows misunderstanding of type name	1, 2
S shows misunderstanding of variable name	1, 2
S shows misunderstanding of function name/description	1, 2, 22
S shows misunderstanding purpose of 'leftovers' panel	6
Automatic move of first non-empty boolean expression from leftovers to condition confuses S	5, 23
S shows misunderstanding relationship between objects and their properties	7
S shows he had insufficient experience in business rule specification	8
leading S trying to find solution by trying random combinations	9
S expects expression can be deleted by dragging it somewhere but finds out it cannot	10
S wants to select the matches first	11
S retries action after it got rejected	12
S tries to select variable box instead of expression box surrounding it; this is not possible	4
S shows insufficient experience in programming	8
S unclear on what constraints decide what expressions are allowed where (typing, variable scoping, expression-type)	13
S unaware that 'new rule' button has worked after having clicked	14
S unclear on when mouse cursor is on 'delete rule' button and when not	14
S unclear on fact that rule title is editable	14
S can't find way to enter literal; does not know meaning of the word 'literal'	16
S unclear on value conventions of certain database fields (e.g. `Car.color`: "red", "Red", "#FF0000"?)	2
S unclear on difference between hovered and selected boxes	14
S unclear on how to add another proposition in conjunction with the current one	8, 19
S wants to create a new object but cannot	23
S unclear on that the content selection menu *replaces* the current content (including all child expressions) of a box	4, 5
S finds application tiresome to use	3
S unclear on meaning of the different operands of some function	2
S expresses insufficient experience in declarative programming	8
S unclear on fact that all text in Rule panel forms a single sentence	1, 9
S attempts to unify variables by giving them identical names	20
S does not see that a box with the variable name in it is a manifestation of the object it refers to	21

Table 7.2: Problem manifestations during user tests, with ID's of underlying problem from Table 7.3. "S" is a shorthand for "subject".

ID	Problem description	Solution
1	Naming of types, functions, properties not properly targeted to non-programmer	4
2	No documentation available for types, properties, functions and their operands	3
3	Application suboptimal regarding maximizing keyboard liveness, button size and minimizing command-depth (maximum amount of steps until command completion)	1, 7
4	Syntax tree visually inconsistent	1
5	Automatic actions are being done of which it is not certain that the user had intended them	1
6	Function of Leftovers panel is not self-explanatory	5
7	Relations between objects and their properties not self-explanatory	6
8	Insufficient experience	
9	The subject is insufficiently thinking about the semantics	7
10	Of three methods of moving expressions around: dragging, context menu based and keyboard-shortcut based, it is unpredictable which action can be performed by which method.	2
11	The application imposes a working order that is not strictly necessary.	8
12	Illegal moves are rejected without any feedback as to why; as a result the user has no clue what went wrong.	7
13	Some constraints are invisible	7, 8
14	Not always clear enough visually whether something is a clickable, what its click function is and when it is successfully clicked.	1
16	The command for entering a literal, as one of the three main categories of expressions, is much too hidden.	9
19	Editing only happens top-down, and working bottom-up requires a 3 step process	11
20	Upon moving focus off the variable name text box, no validation is done	7
21	Most users understand the concept of variables from middle school mathematics, but the variable names are simply being identitied as being variable names.	10
22	Expression selection menu only features textual entries, and does not employ any visual cues as to what its options look like when employed in the syntax tree.	12
23	Not a problem with the initial design but only with the prototype.	

Table 7.3: Supposed problems causing hindrance during user tests with ID's of the solutions from Chapter 8 that addressed them

Chapter 8

Additional design decisions

8.1 Introduction

In this chapter, a number of design decisions motivated by the problems exposed in the user tests are presented in Section 8.2. Each of the design decisions has a "solution ID" for traceability to the problems given in Table 7.3. In this table, the last column refers to the solutions of this chapter by these ID numbers.

8.2 Design decisions

8.2.1 Improving the syntax tree display

Solution ID: 1

Consistency

Functions and literals are represented by a single box in the tree view, see Figure 6.3. Variables, on the other hand, are a box in a box, the inner being a box of solid color. Also, the entire rule supposedly reads like natural language, but this mantra does not descent into the display of variables or object properties. The top row of Figure 6.3 reads "*For every x person*", while natural language would have "*For every person x*". Object properties read more unlike natural language.

Instead, when a variable appears in the rule with its type it should feature the type first, the name second, and when it appears in the rule its solid-colored box should be the *only* box representing the variable as an expression —no seperate wireframe box should appear around it. To emphasize that the variable is an expression like literals and functions, the solid variable box will take the height of an expression box, and feature its type in the top right corner as with other expression boxes.

Finally, empty expressions will get a look as if they are shallow chambers in the table —as if they lie a few millimeters deeper—, and they will also look a bit darker, as if the surface of the editing panel has a slightly different color than

the material thats underneath. Besides make it far more obvious when a rule's syntax tree is *not finished*, it creates the suggestion that when content is put into it, this content *lays over* the hole like a puzzle piece. This is important because of the meaning of the datatype label. On empty expressions, this datatype label states the *allowed type*, while on expressions that have content, it states the *actual* type, the output type of the content that is in it. The overlay suggestion supports this difference; the label in the hole is part of the hole, hence part of the function around it that featured that hole, and the label on a nonempty expression is part of the puzzle piece that lays on top.

Content selection menu only on empty expressions

The content selection menu should be featured only on empty expressions, for three reasons:

1. Test subjects were repeatedly confused when they selected an expression for an expression box that already had content in it, and upon selecting it found out that what they selected replaced the original content. Only allowing them to do this operation on expressions that are *already empty* forces them to explicitly delete or move aside the original content if they attempt to replace it, addressing problem 5 of Table 7.3. This does create an extra action for the user, but it pushes them more into the intended, dragging-oriented editing method described in Section 5.2.2.

2. The entire empty expression can be turned into a button, addressing problem 3 of Table 7.3.

3. Losing the small button in the top right corner of every expression makes for a far less clogged view of the syntax tree.

8.2.2 Support as many actions as possible through all editing methods

Solution ID: 2

Table 8.1 shows which commands are supported by which editing method. Firstly, the expression selection menu is only for creating new expressions. This is a well-defined cause and can therefore be deemed predictable enough. The context menu is currently only used for copy, cut, paste and function definition. When right-clicking on an empty expression, we could let it show the expression selection menu extended with a *paste* option so that it would support creation, but this would make the difference between left and right mouse buttons vague, and the right mouse button would not have a uniform response across all expressions like it does now. The delete function could and should however be added to the context menu. Moving to keyboard functions; it is hard to do creation fully with the keyboard —e.g. open the expression selection menu with a keyboard key— as it is impossible to select an empty expression, and if you cannot select it, it is not clear to which expression the keyboard key applies. Defining functions however is only done on nonempty —selectable— expressions, so a keyboard key, say, F can be designated for the define function command on the selection. If the selection consists of multiple expressions, pressing F will do nothing. Moving on to drag-and-drop: creation of functions and literals is

	Create			Delete	Copy-paste	Cut-paste	Define function
	Function	Literal	Variable				
Expression selection menu	+	+	+				
Context menu (right-click)				a	+	+	+
Keyboard				+	+	+	a
Drag-and-drop			+	a	a	+	

Table 8.1: Supported commands per editing method. A +-sign means the command was supported in the initial design. An "a" means the command will be added in the improved design.

not possible as there is nowhere to drag them from. Deletion, however, can be added; this is described in the designated paragraph below. Furthermore, dragging an expression from one place to another is inherently cut-paste. This however could easily be extended to be copy-paste: if the user holds CTRL when he starts the drag, a copy of the expression is being dragged off instead of the original.

Deletion by dragging

From user tests is became apparent that users, in an environment that allowed them to building things by dragging and dropping, expected things to also be *removable* by dragging them to some place. This is a sign that users start to see logic in the way the application work, since they are *extrapolating* this logic. They should feel that expressions should be seen as "things" that they can hold and move around like Lego blocks —and by this logic, similarly to Lego blocks, the way to remove them from the playing field is by picking them up and moving them off the table. This insight should be encouraged, so it is important to feature a drop-point for removing expressions. This drop point should both feel "off the playing field" and be clearly visible, so it will be hard for users to not see it. This is achieved by only showing the drop point while the user is dragging something, so that the drop point can be large, positioned in the bottom center, hovering in front of the "playing field" (it is positioned relative to the browser window, and not the document). It features a trashcan icon as it is the universally accepted icon for removing things.

8.2.3 Room for documentation everywhere

Solution ID: 3

The following features will be added:

- Hovering over a type gives documentation about the type, which is an optional property of the function metamodel
- Hovering over a property of a variable gives documentation about the property, which is an optional property of the class property metamodel.

- Hovering over an empty expression box gives the documentation specific to that operand, which is an optional property of the operand metamodel.

8.2.4 Non-technical names

Solution ID: 4

Types, functions and properties need to have names that are solely targeted to non-programmers. Functions and properties will all get an optional property called "display name" which may contain spaces. Datatypes will keep only a single name, but this name should be made to reflect what a non-programmer understands. A display name with spaces is unsuitable for datatypes (e.g. with generic types, spaces would look confusing).

8.2.5 Making the leftovers self-explanatory

Solution ID: 5

The "Leftovers" panel will be renamed to the "Sketchpad" panel, i.e. the title above the panel will be changed to "Sketchpad". This sketchpad will feature the set of boxes that have been dragged and/or created there, and in the bottom there is always a big 'drop area' surrounded by a dashed line, with in the center the text "Drag any piece of content here, or build a new expression by selecting a function", with 'selecting a function' being obviously clickable. Clicking it will create a new empty expression box to the pool with the content selection menu open and the cursor focus in the search box, as when a function is selected in the normal way according to the new design.

8.2.6 Promoting the object-property relationship

Solution ID: 6

The following will be implemented to promote the object-property relationship.

- In the tooltip when hovering an object property, the phrase "<PROPERTY NAME> of <CLASS NAME/OBJECT NAME(IF AVAILABLE)>" must be the most clearly visible of the whole tooltip
- When an object property appears in the syntax tree, it must form the phrase "<PROPERTY NAME> of <OBJECT NAME>".

In both cases, the object name in question can itself be a property of another object. In this case, the object name is, recursively, "<PROPERTY NAME> of <OBJECT NAME>", so that the entire chain of ownership is displayed.

8.2.7 Constructive feedback

Solution ID: 7

Most actions that are not allowed cannot be initialized by the user. For instance, when clicking the dropdown button on a field of type "boolean", the "addition" function does not appear because it has an incompatible output type. For other actions however, it can only be clear after the user has initialized them

that they are illegal. Think of drag-and-drop actions. If they are illegal, these actions get rejected. In RASH, no action gets rejected without providing constructive feedback to the user. When a drag-and-drop action gets rejected, it is usually because of a type mismatch. RASH shows a message that says what the type mismatch is, making novice users aware of the type system, but also providing insight into the type system (what types are subtypes of what other types). Furthermore, RASH can present a list of functions that would form the connection between the attempted input type and the demanded input type. Upon clicking one of these functions, the user's attempted drag-and-drop action is completed by adding the selected intermediary function in between.

8.2.8 Possibility to explicitly add matches

Solution ID: 8

The author can drag entities into the top row of the rule (the row where the matches are listed) to add it to the rule matches. Upon hovering over there while an entity is being dragged, the row will look as it would if that entity would be a match. This way it is shown to the user that dragging something over there does something.

Semantically, this changes something about the rule model: the user can add an entity to the matches without that entity being used in the rule's condition. It simply means that the rule will match against every new instance of that entity.

One advantage is that the Event library from Subsection 6.5.5 becomes obsolete if events are objects from entity classes. Another advantage is that the constraint that all variables used in the consequence must be globals or matches will no longer have to lead to a user action being rejected if he tries to instantiate a variable in the consequence. Instead, when the user does this this variable simply also gets added to the rule matches, causing it to appear in the top row.

8.2.9 Improved literal entering

Solution ID: 9

The following should be implemented to make entering of literals go better.

- If the datatype of the expression box is primitive, show option reading "Enter value..." on the top of the expression selection menu
- If the datatype of the expression box is complex, show option reading "Create new <TYPE NAME>..." on top of the content selection menu —if constructing is turned on for that datatype
- The form for entering a literal must be in focus after the option for entering a literal is chosen. The form focus must be clearly visible for users that do not know where to look

8.2.10 Conventional variable names

Solution ID: 10

Variable names currently are currently the type name with a letter behind it, starting from “a”, e.g. “carA”, “personC”. These strings were often not recognized by the user as variable names. Hence, the improved design will adopt a convention widely used in scholarly environments and single-letter variable names starting from ’x’, then ’y’, et cetera. The entire alphabet is still used, but the order is just different, and the type name is dropped.

8.2.11 Rejected: Bottom-up syntax tree building

Solution ID: 11

This feature was considered for the initial design, and is again reconsidered. It means that, for an expression, a menu can be opened in which a function can be picked that is placed *around* that expression. So if someone wrote a condition and wants to add a second condition in conjunction, all he needs to do is select conjunction from the bottom-up function selection menu and the conjunction will be made around the condition that was already there.

The problem is that it is very hard to give the button that opens this menu a place in the GUI such that it is, from its appearance and placement, immediately understood what its function is. Beside this, disambiguation on *which* operand the user wants the current expression to become of the the future parent expression makes the UI even more confusing. It is decided it is safer to omit this functionality, as bottom-up editing can still be done by a few more moves on the author’s part, but these moves are all much easier to understand.

8.2.12 Improvement of content selection menu

Solution ID: 12

The content selection menu, in the initial design, does not employ any visual cues, see Figure 6.5. In the improved design, previews of what each function looks like in the syntax tree are given, so that the user immediately makes the connection between the options presented in the menu, and the expressions he sees in the syntax tree. The user will understand the application more quickly. Also, suggested variable names will be shown in the menu exactly as they look in the syntax tree.

8.3 Conclusion

Table 7.3 shows that all issues that have been identified have been addressed in some way. For now, it is not a goal to address insufficient experience of rule authors in any other way than having them practice with the application, which does not require additional features. The improved design will be explained in the next chapter, Chapter 9.

Chapter 9

Improved design

9.1 Introduction

This chapter proposes the improved design, on which lessons learned from user tests on a prototype of the initial design have been applied. We go through the upgrades in the design step by step, starting with the editing GUI in Section 9.2, the Rule Language & BOM Metamodel in Section 9.3, the Rule Metamodel in Section 9.4 and finally the standard libraries in Section 9.5.

9.2 Editor of improved design

9.2.1 Syntax tree view

The improved syntax tree appearance is shown in Figure 9.1. Note

- how the dropdown menu button in the corner of each expression has disappeared,
- how variables now take up their entire expression box,
- how the object-property relationship is written out in natural language
- how a variable reads e.g. "person x" now, instead of "x person" and
- variable names are now instantly recognizable as such as they follow a scholarly convention.

9.2.2 Expression selection menu

See Figure 9.2. An empty expression box appears as a shallow hole in the rule or sketchpad panel. When clicked, a dropdown menu appears in which an expression can be chosen as content for the box, see Figure 6.5. It shows the option to enter a literal value if allowed for the datatype, a few likely choices for variables that fit the datatype and all the functions with a fitting output datatype. Because variable attributes can be infinitely recurrent (e. g. if a `Person` has an attribute of type `Person`), it is often impossible to show a *complete* list of variables. A maximum is set of 8 variables, and a maximum recurrence depth of 2

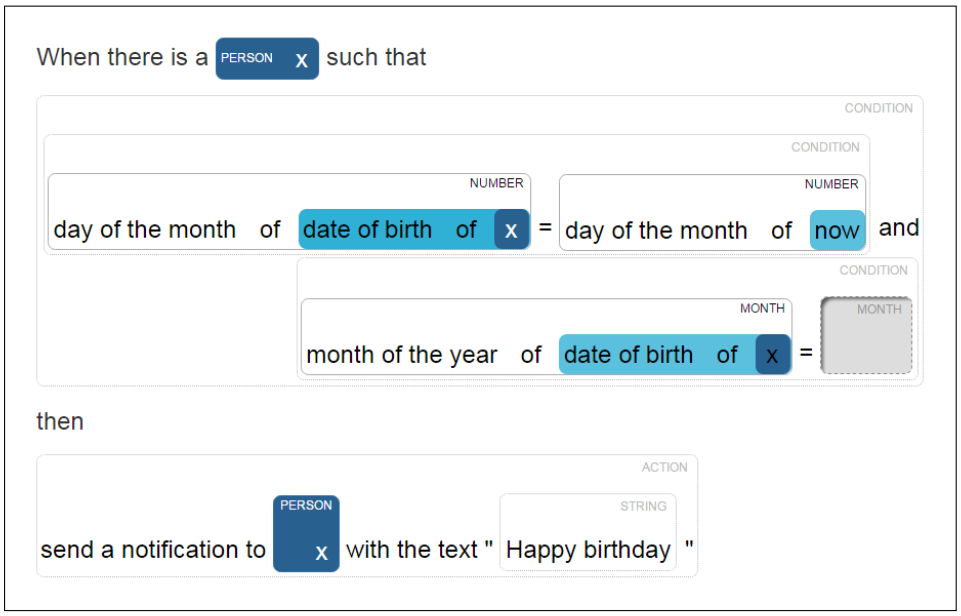

Figure 9.1: Rule appearance in the improved design RASH, replacing the original design shown in Figure 6.3.

(a property of a property). Upon opening the menu, the options can be filtered by typing keywords with the keyword: the empty expression box temporarily functions as the text input field and the option list updates with every keystroke.

Notice the improvements:

- The command for enter literal used to read "Enter literal..." and appeared at the bottom of the menu. Now, it appears at the top and reads "Enter value...", a choice of words more targeted to non-programmers.
- Variables look in the menu as they do when used in the syntax tree.
- Functions look in the menu as they do when used in the syntax tree, and they have more descriptive "display names" that may contain spaces, and that are completely targeted to the user.

9.2.3 Entities panel

Figure 9.3 features the improved entities panel. There are three things that should be noticed: the relation between objects and properties is now promoted by a bold printed text in the tooltip of a property. Secondly, this tooltip can feature property-specific notification, something that can be optionally defined on a class property. Thirdly, author-friendly display names of properties are supported, e.g. "date of birth" instead of "dateOfBirth"; these are optional as well. To be consistent with the syntax tree, the type name is given before the variable name.

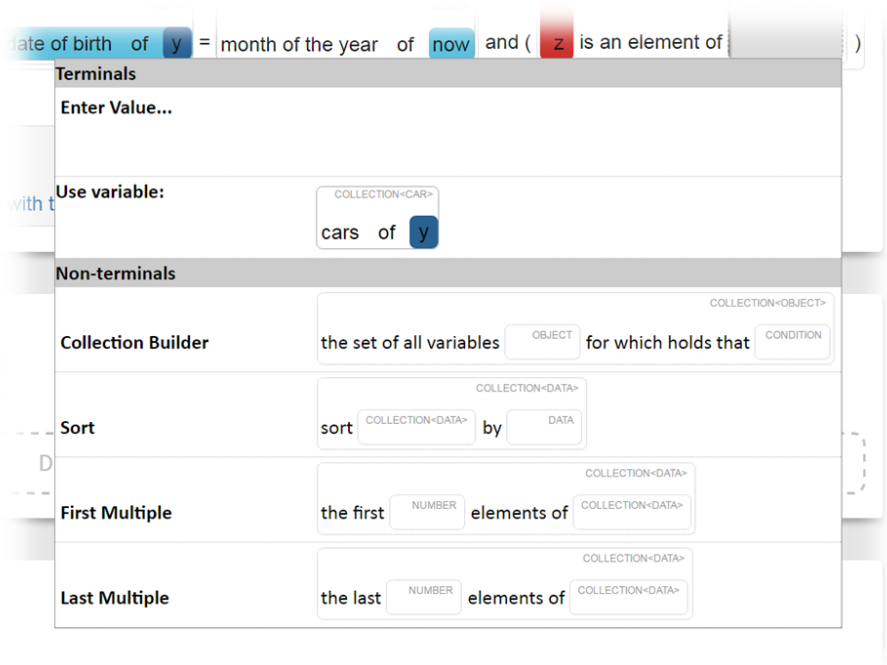

Figure 9.2: The expression selection menu of the improved design, replacing the old one shown in Figure 6.5. Provides the user with visual cues and more natural text, among other things.

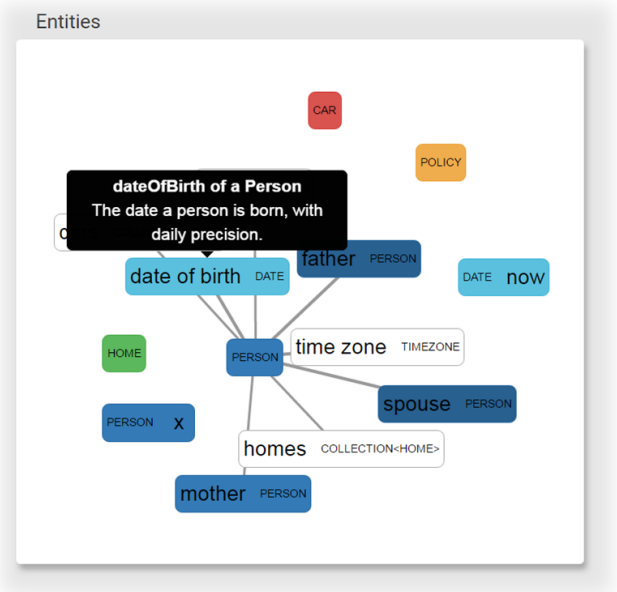

Figure 9.3: The entities panel in the improved design, showing improved property naming and property-specific documentation in tooltips.

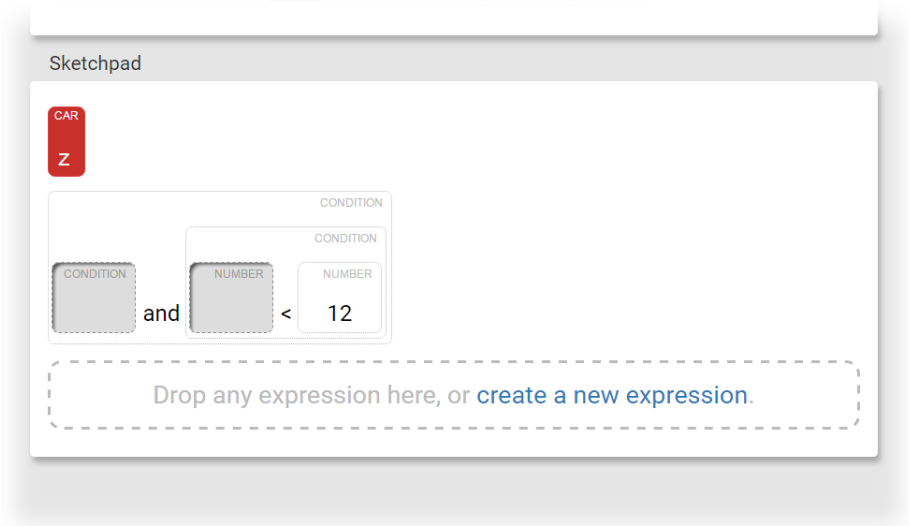

Figure 9.4: The improved appearance of the sketchpad, populated with two arbitrary expressions.

9.2.4 Sketchpad panel (previously: Leftovers panel)

The improved leftovers panel, now called sketchpad, is shown in Figure 9.4. The drop area now explains the functionality of the panel, as with a newly opened rule this drop area is all that the sketchpad panel contains.

9.2.5 Editing methods

Four controls have been added, as reflected by Table 8.1.

1. Deletion with the context menu
2. Defining a function by pressing the F key when exactly one expression is selected
3. Deletion by dragging towards a "trashcan"
4. Copy-paste by hold CTRL while starting a drag

Furthermore, it is now possible to instantiate variables anywhere in the rule, which was previously only possible in the condition. Also, entities can be dragged to the top row of text in the rule panel, which will add them as a match.

9.3 Rule Language & BOM Metamodel of improved design

In the rule metamodel, optional display name properties have been added to the `Function` and `ClassProperty` classes. Optional documentation text properties have been added to the `Type`, `ClassProperty`, `Function` and `Operand` classes.

9.4 Rule Metamodel of improved design

In the rule model, a property named `unmentionedMatches` is added to the `Rule` class, to list matches that have not been mentioned once in the rule syntax trees. In the initial design, matches were by definition free variables of the rule's condition. Now, a match can be explicitly added to the rule without it ever being referred to in the condition or consequence of the rule, hence the necessity for this new property of the `Rule` class. The union of the free variables of the condition, the free variables of the consequence and the variables in `unmentionedMatches` is now the set of matches of the rule.

9.5 Libraries of improved design

The Event library has been dropped, and display names have been added to all functions. Beside this, the libraries have not been changed.

9.6 Conclusion

The majority of the features of the improved design presented in this chapter have been implemented in the prototype, partially for demonstration purposes, and partially to see how design decisions turn out in reality. However, a new set of user tests is deferred to future work. A product is not made with a single feedback loop, but with multiple. The design proposed in this chapter addresses issues that occurred with the initial design of Chapter 6. Finding out if it actually solves them will be part of the third design iteration.

Chapter 10

Conclusion

With this thesis, we have proposed a Platform-Independent BRAE for Non-Technical Authors that is suitable for use at Royal Philips N.V. In academic sense, we have achieved something that is

1. Unique in being the only stand-alone general purpose rule authoring environment
2. An on-the-fly domain-specific language creator, by letting authors define functions of their own within the authoring GUI itself
3. Moving syntax-directed editing away from text editing paradigms towards structure editing paradigms

There are some reservations to keep in mind. When an application has large sets of rules that have many clauses in common, decision tree editing becomes a useful addition to syntax-directed editing. The main disadvantage of decision trees is that they need programmatic expressions in their edges and/or nodes. To make editing these user-friendly something like Rash is still needed. However, for the intended use by Philips there is no need for decision trees in sight.

Future work includes, first and foremost, a rule testing mechanism. Furthermore, more development cycles would make the product better, i.e. make more prototypes and do more user tests. Other work is building the back-end that will fit this rule engine: parsers to existing rule engines, or an entirely new rule engine. Also, automatic configuration of the the rule metamodel, i.e. automatic elicitation of the relevant types, functions and variables from this rule-executing back-end so that these will not have to be configured by hand will be valuable work. Finally the editor would be suitable for an automated training program, in which a subject is taught to write rules by being given a series of textual rule specifications of increasing complexity that have to be turned into a formal specification.

Acronyms

AI Artificial Intelligence. 81

AST Abstract Syntax Tree. 25, 35, 44

BMM Business Motivation Model. 3

BOM Business Object Model. 1–3, 8, 10–15, 17, 18, 34–37, 42, 45, 51, 52, 74, 77

BRAE Business Rule Authoring Environment. 1, 5, 6, 12, 20–22, 37, 42, 79

BRE Business Rule Engine. 7, 8

BRMS Business Rule Management System. 3–5, 7, 13, 20, 22, 23, 30

DRL Drools Rule Language. 23

ECA Event-Condition-Action. 13, 60

ER Entity-Relationship. 25, 26

GUI Graphical User Interface. 1, 15, 19, 43, 49, 54, 58, 62, 73, 74, 79

LHS Left-Hand Side. 7, 8, 13

MPS Meta-Programming System. 28, 30

OMG Object Management Group. 3

RHS Right-Hand Side. 7, 8, 13, 16

SDLC Software Development Life Cycle. 4

SE Software Engineer. 4, 5

SME Subject Matter Expert. 3–5

SRD Software Requirements Document. 10

UI User Interface. 10, 19, 20, 22, 42, 49

Glossary

authoring language The language that appears to the author in Rash's front end. 45

C# C# is a multi-paradigm programming language encompassing strong typing, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines [27]. 37

C++ C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation [28]. 37

card A small rectangular GUI container containing multiple related elements that may appear multiple times on a page [29][30]. 15

functional programming language In computer science, functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data [24]. 45

Java Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented and specifically designed to have as few implementation dependencies as possible [31]. 37

natural language Language used by humans. 1

Production system A production system is a program that provides some form of Artificial Intelligence (AI) by executing a set of rules about behavior. 7

Bibliography

[1] Wilbert O Galitz. *The essential guide to user interface design: an introduction to GUI design principles and techniques.* John Wiley & Sons, 2007.

[2] Jetbrains. Meta programming system - dsl development environment. URL `https://www.jetbrains.com/mps/`. [Online; accessed 9-October-2016].

[3] IBM. Operational decision management, . URL `http://www-03.ibm.com/software/products/en/category/operational-decision-management`. [Online; accessed 21-October-2016].

[4] J.C. Mitchell. *Concepts in Programming Languages.* Cambridge University Press, 2003. ISBN 9780521780988. URL `https://books.google.nl/books?id=7Uh8XGfJbEIC`.

[5] B. Von Halle. *The business rule revolution : running business the right way ; [fundamental issues: business approach, technology approach].* Happy About, 2006. ISBN 9781600050138. URL `https://books.google.nl/books?id=KrTGV2Xf7V8C`.

[6] Drools. Drools documentation, . URL `https://docs.jboss.org/drools/release/6.5.0.Final/drools-docs/html_single/index.html`. [Online; accessed 8-October-2016].

[7] N. Bostrom. *Superintelligence: Paths, Dangers, Strategies.* Oxford University Press, 2014. ISBN 9780199678112. URL `https://books.google.nl/books?id=7_H8AwAAQBAJ`.

[8] Object Management Group. Business motivation model. URL `http://www.omg.org/spec/BMM/1.3/`. [Online; accessed 29-September-2016].

[9] J. Boyer and H. Mili. *Agile Business Rule Development: Process, Architecture, and JRules Examples.* Springer Berlin Heidelberg, 2011. ISBN 9783642190414. URL `https://books.google.nl/books?id=nccgSywG2Y8C`.

[10] Wikipedia. Systems development life cycle — wikipedia, the free encyclopedia, 2016. URL `https://en.wikipedia.org/w/index.php?title=Systems_development_life_cycle&oldid=746335442`. [Online; accessed 26-October-2016].

[11] Oscar Hauptman. The different roles of communication in software development and hardware r&d: Phenomenologic paradox or atheoretical empiricism? *Journal of Engineering and Technology Management*, 7(1): 49–71, 1990.

[12] James D Herbsleb, Audris Mockus, Thomas A Finholt, and Rebecca E Grinter. An empirical study of global software development: distance and speed. In *Proceedings of the 23rd international conference on software engineering*, pages 81–90. IEEE Computer Society, 2001.

[13] IBM. *Operational Decision Management For Dummies*. –For dummies. . URL `https://www-01.ibm.com/marketing/iwm/iwm/web/signup.do?source=sw-app&S_PKG=ov30443&dynform=14874&lang=en_US`.

[14] Wikipedia. Set-builder notation — wikipedia, the free encyclopedia, 2016. URL `https://en.wikipedia.org/w/index.php?title=Set-builder_notation&oldid=742772024`. [Online; accessed 5-October-2016].

[15] InRule. Write business rules with irauthor. URL `http://www.inrule.com/products/inrule-components/irauthor/`. [Online; accessed 29-September-2016].

[16] Drools. Drools - overview, . URL `http://www.drools.org/`. [Online; accessed 8-October-2016].

[17] Arthur M Langer. Build vs. buy. In *Guide to Software Development*, pages 37–48. Springer, 2011.

[18] XpertRule. Expert systems - graphical rules authoring. URL `http://xpertrule.com/expert-systems-graphical-rules-authoring/`. [Online; accessed 8-October-2016].

[19] Bosch. Business rule management with visual rules - components. URL `https://www.bosch-si.com/products/business-rules-management/brm-components/tools-platforms.html`. [Online; accessed 8-October-2016].

[20] Progress. Corticon business rule studio. URL `https://www.progress.com/corticon/components/studio`. [Online; accessed 9-October-2016].

[21] J. Ross Quinlan. Simplifying decision trees. *International journal of man-machine studies*, 27(3):221–234, 1987.

[22] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering*, pages 41–61. Springer, 2014.

[23] Geoffrey K. Pullum and Gerald Gazdar. Natural languages and context-free languages. *Linguistics and Philosophy*, 4(4):471–504, 1982. ISSN 1573-0549. doi: 10.1007/BF00360802. URL `http://dx.doi.org/10.1007/BF00360802`.

[24] Wikipedia. Functional programming — wikipedia, the free encyclopedia, 2016. URL https://en.wikipedia.org/w/index.php?title=Functional_programming&oldid=741227841. [Online; accessed 28-September-2016].

[25] Wikipedia. V-model (software development) — wikipedia, the free encyclopedia, 2016. URL https://en.wikipedia.org/w/index.php?title=V-Model_(software_development)&oldid=731736893. [Online; accessed 27-July-2016].

[26] Dolores R Wallace and Roger U Fujii. Software verification and validation: an overview. *Ieee Software*, 6(3):10, 1989.

[27] Wikipedia. C sharp (programming language) — wikipedia, the free encyclopedia, 2016. URL https://en.wikipedia.org/w/index.php?title=C_Sharp_(programming_language)&oldid=745154334. [Online; accessed 19-October-2016].

[28] Wikipedia. C++ — wikipedia, the free encyclopedia, 2016. URL https://en.wikipedia.org/w/index.php?title=C%2B%2B&oldid=745154329. [Online; accessed 19-October-2016].

[29] UI-Patterns.com. Cards design pattern. [Online; accessed 29-September-2016].

[30] Intercom.com. Cards are fast becoming the best design pattern for mobile devices.

[31] Wikipedia. Java (programming language) — wikipedia, the free encyclopedia, 2016. URL https://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=743173085. [Online; accessed 8-October-2016].