Master Thesis

An approach to model reaction rules
and transform them to procedural form

Information Science
November 2014

Vincent Doppenberg
S4235002
Colophon

Author
Vincent Doppenberg

Student number
s4235002

Master programme
Information Science

University
Radboud University
Comeniuslaan 4
6525 HP Nijmegen
The Netherlands

PO Box 9102
6500 HC Nijmegen
Tel.: +31 24 361 61 61
Internet: www.ru.nl/english

Faculty
Faculty of Science
Huygens building
Heyendaalseweg 135
6525 AJ Nijmegen
The Netherlands

P.O. Box 9010
6500 GL Nijmegen
The Netherlands
Tel.: +31 (0)24-365 2661

Research institute
Institute for Computing and Information Sciences

University supervisor
Prof. dr. ir. Th.P. van der Weide
Second teacher
dr. S.J.B.A. Hoppenbrouwers
Company supervisor
ir. Jasper Kloost

Location, date
Nijmegen, November 2014
Version
Final
**Table of contents**

1  Introduction to business rules ........................................................................................................... 5  
   1.1  Defining ‘business rule’ .................................................................................................................. 5  
   1.2  The business rule rationale .......................................................................................................... 6  
       1.2.1  The ‘Business Rules Approach’ ................................................................................................. 6  
   1.3  Types of business rules ............................................................................................................... 7  
2  Research context .................................................................................................................................... 10  
   2.1  Motivation ......................................................................................................................................... 10  
       2.1.1  Introducing the Thinkwise Software Factory .............................................................................. 10  
       2.1.2  Problem statement .................................................................................................................. 10  
       2.1.3  Requirements .......................................................................................................................... 11  
   2.2  Research questions ....................................................................................................................... 11  
   2.3  Validation ......................................................................................................................................... 11  
3  Related work ......................................................................................................................................... 12  
   3.1  Levels of expression .................................................................................................................... 12  
   3.2  The external level .......................................................................................................................... 14  
       3.2.1  Structured natural language .................................................................................................... 14  
       3.2.2  Decision tables ........................................................................................................................ 14  
       3.2.3  Diagrams .................................................................................................................................... 14  
       3.2.4  Expression builders ................................................................................................................. 14  
   3.3  The conceptual level ..................................................................................................................... 15  
       3.3.1  First-order logic ....................................................................................................................... 15  
       3.3.2  SBVR – Semantics of Business Vocabulary and Business Rules ................................................. 16  
           3.3.2.1  SBVR Structured English ................................................................................................ 16  
   3.4  The internal level .......................................................................................................................... 18  
       3.4.1  Rule engines ............................................................................................................................. 18  
4  General approach ............................................................................................................................... 19  
5  Expressing reaction rules .................................................................................................................... 20  
   5.1  Defining reaction rules .................................................................................................................. 20  
       5.1.1  Event .......................................................................................................................................... 20  
       5.1.2  Condition ................................................................................................................................. 20  
       5.1.3  Action ........................................................................................................................................ 21  
       5.1.4  Bidirectional interface ............................................................................................................. 21  
   5.2  The proposed grammar .................................................................................................................. 21  
   5.3  The proposed semantics ................................................................................................................ 23  
6  Transforming reaction rules to T-SQL ............................................................................................... 24
6.1 Singular expressions............................................................................................................. 24
  6.1.1 The definition of a singular expression ............................................................................. 24
  6.1.2 Transforming a singular expression .................................................................................. 25
    6.1.2.1 The associated T-SQL constructs ............................................................................. 25
    6.1.2.2 The transformation of a singular expression to T-SQL ................................................ 26
  6.2 Query-expressions................................................................................................................ 29
    6.2.1 The definition of a query-expression .............................................................................. 29
    6.2.2 Transforming a query-expression .................................................................................... 30
      6.2.2.1 The associated T-SQL constructs of the FROM-clause .............................................. 31
      6.2.2.2 The transformation of the FROM-clause .................................................................. 32
      6.2.2.3 The associated T-SQL constructs of the WHERE-clause .......................................... 32
      6.2.2.4 The transformation of the WHERE-clause .................................................................. 34
      6.2.2.5 Transforming the action-part .................................................................................... 34
      6.2.2.6 Combining the results ............................................................................................... 35
  6.3 Events .................................................................................................................................... 35
  6.4 Semantical correctness........................................................................................................... 37
    6.4.1 Singular expressions ....................................................................................................... 37
    6.4.2 Query expressions .......................................................................................................... 37
  7 Validation – Business Case ..................................................................................................... 39
    7.1 Defining the events and actions ......................................................................................... 39
    7.2 Formalising the business rules to ECA-expressions ............................................................. 41
    7.3 Transforming the business rules .......................................................................................... 43
      7.3.1 Supplying transformations for the functions and actions used .................................... 43
      7.3.2 Transforming the ECA-expressions ............................................................................. 43
        7.3.2.1 Transforming constraint rule 1 ............................................................................. 44
        7.3.2.2 Transforming constraint rule 2 ............................................................................. 46
        7.3.2.3 Transforming reaction rule 2 ................................................................................. 47
    7.4 Conclusion .......................................................................................................................... 48
  8 Conclusions and future work .................................................................................................... 49
    8.1 Conclusions ......................................................................................................................... 49
    8.2 Future work ........................................................................................................................ 50
  9 References .................................................................................................................................. 51
    9.1 Sources for the T-SQL syntax ............................................................................................. 52
1 Introduction to business rules

The concept of ‘business rule’ has gained a lot of popularity since the early nineties of the previous century. Business rules have become increasingly more important in the field of Information Science as a means to influence the behaviour of business applications. This chapter seeks to define business rules, explain the underlying motives for using them, and distinguish between the different types of business rules.

1.1 Defining ‘business rule’

The term ‘business rule’ is a widely used term for which there is no single definition[5]. One of the most popular definitions (e.g. [7], [8], [16] and [19]), first proposed by The Business Rules Group in 1995, is the following:

“A business rule is a statement which defines or constrains some aspect of the business. It is intended to assert business structure, or to control or influence the behavior of the business.”[21]

Interestingly, The Business Rules Group has also given a definition from the business perspective, rather than the Information Science perspective:

“A business rule is guidance that there is an obligation concerning conduct, action, practice, or procedure within a particular activity or sphere.”[21]

While many have used the definition proposed by The Business Rules Group, there are others who appear to have been inspired, or have built upon these definitions:

“A business rule is a statement that aims to influence or guide behavior and information in an organization.”[10]

“A rule that can be interpreted by computers, that defines or restricts some aspects of a business, introducing obligations or needs, according to the organization policies.”[5]

Despite the many nuances, nearly all of the definitions share the notion that a business rule is a type of statement or expression which relates to a business aspect or policy. Furthermore, each of the definitions above mentions that a business rule is meant to define or constrain an aspect of the business, or to control or influence the behaviour and information of the business. The first definition explains the former as the means, and the latter as the goal.

In this paper, ‘business rule’ means a statement which controls the behaviour of the business in a given situation, or describes what is and what is not allowed given a certain aspect of the business. The goal of such business rules is to control the behaviour and information of the business.

A simple example of a business rule, which will be used throughout this paper, is the following:

An employee is not allowed to log working hours during the weekend.

This business rule is an example of a constraint regarding the days on which employees of the business are allowed to log working hours in a system.
1.2 The business rule rationale

All businesses have business rules, however, not all businesses are consciously aware of them as such. This chapter addresses the manner in which the business rules are used and managed, and how this affects the business.

Many businesses will not have an exhaustive list of business rules, readily available when asked for, and easily manageable. One can imagine rules and regulations being spread over a myriad of documents and departments, while others might be unwritten or only programmed into the business applications. In [1], a list of problems which may occur as a consequence of this has been compiled from a number of sources. The two core problems are firstly that business applications do not reflect business needs, because there is no clear and exhaustive overview of the business needs[1]. And secondly, that business applications are not flexible to changes, because the business rules are spread out over the application logic and buried within program code[1]. Similar problems have been described in [5] and [20].

As suggested by the definitions, the business rules within a business originate from the policies of a business. They can be motivated either by internal or external reasons, and can therefore be referred to as internal- and external business rules[1][11]. Internal business rules are defined by the business itself, and typically represent business strategies, while external rules are imposed by parties outside of the business, and typically represent laws or contractual agreements[1][11]. Because of this, business rules are managed by the business[1], and those who are responsible for managing them are typically not technical people[24]. Given the reasons for changing business rules, e.g. business strategies and imposed laws, business rules are volatile in nature[1][16][24]. The ability for a business to respond to its environment is therefore tied to its ability to manage its business rules, including those which affect business applications[1][14][16][19][24].

Aside from the important role of business rules in the ability of a business to respond to its environment, they also form a valuable source of information. After all, business rules describe a business’s policies, its business strategies, and what the business can and cannot do.

1.2.1 The ‘Business Rules Approach’

The realisation that business rules are dynamic and business-driven, as described in the previous paragraph, grew around the turn of the century. In [16], it can be read how it became increasingly more obvious that, for a business to be able to respond to its environment, it needs to be able to manage its business rules. Furthermore, changes in business rules should be immediately reflected by the business applications, without the need for a time-consuming and costly development process[16]. These realisations lead to the ‘Business Rules Approach’, a general methodology regarding how business rules can be used to achieve these goals[16].

The main concepts of the Business Rules Approach have been written down in the ‘Business Rules Manifesto’, like [16], written by Ronald Ross[22]. The Business Rules Manifesto forms a handbook of several dozen best practices regarding business rules, which form the cornerstones of the Business Rules Approach. A selection of these best practices can be found in Figure 1.
“Rules are not process and not procedure. They should not be contained in either of these.”

“Rules apply across processes and procedures. There should be one cohesive body of rules, enforced consistently across all relevant areas of business activity.”

“Rules build on facts, and facts build on concepts as expressed by terms.”

“Terms express business concepts; facts make assertions about these concepts; rules constrain and support these facts.”

“Rules are basic to what the business knows about itself -- that is, to basic business knowledge.”

“Rules should be defined independently of responsibility for the who, where, when, or how of their enforcement.”

“Business rules should be expressed in such a way that they can be validated for correctness by business people.”

“Business rules should be expressed in such a way that they can be verified against each other for consistency.”

“Formal logics, such as predicate logic, are fundamental to well-formed expression of rules in business terms, as well as to the technologies that implement business rules.”

“Rules are about business practice and guidance; therefore, rules are motivated by business goals and objectives and are shaped by various influences.”

“Rules should arise from knowledgeable business people.”

“Business people should have tools available to help them formulate, validate, and manage rules.”

“Business rules should be organized and stored in such a way that they can be readily redeployed to new hardware/software platforms.”

“Rules, and the ability to change them effectively, are fundamental to improving business adaptability.”

---

Figure 1: A selection of best practices from the Business Rules Manifesto[22].

1.3 Types of business rules

From the business perspective, we have identified internal- and external business rules. From the information science perspective, it is not so much the origin of the business rules which is important, but their semantics and structure. Some studies consider business rules exclusively as rules which constrain actions (e.g. [9] and [14]), while others also consider business rules to be rules which produce actions or facts (e.g. [10], [17] and [23]). This paper follows the latter definition, which will be elaborated upon in this chapter.

In [10], a list of business rule types has been compiled, based on the taxonomy described in the RuleML 1.0 specification, which can be seen in Figure 2. In [17], a second list of business rule types is
can be found. The types of business rules defined in each of these lists can be mapped on each other, as will be argued below, which attests their accuracy.

Figure 2: Taxonomy of RuleML rules[2]

‘Integrity rules’[10] and ‘Restriction rules’[17] both describe statements which constrain the number of acceptable situations, however, integrity rules are specifically described as defining the acceptable relationships between data elements. In [10], ‘Transformation rules’ are also a type of constraint rule, but whether they apply or not depends on the current state of an object.

‘Derivation rules’[10] and ‘Inference rules’[17] both describe statements which can be used to infer new facts from existing facts. In [17], ‘Computation rules’ also describe statements which can be used to infer new facts from existing facts, but these specifically involve computations over the existing facts.


Given these types of business rules, at a high level, we can categorise business rules as constraint rules, derivation rules and reaction rules. A similar division of categories is made in [23], which can be seen in Figure 3. Constraint rules can be subdivided into integrity rules, which define acceptable relationships between data elements, and transformation rules, which define how specific data elements are allowed to change. Reaction rules can be sub-divided into Event-Condition-Action, Event-Action and Condition-Action rules, as is also specified in the RuleML 1.0 specification[2].
Intuitively, both constraint rules and derivation rules can be structured as reaction rules as well. A constraint rule describes a condition which must always remain true. There are two ways to achieve the same result with a reaction rule; preventing violations and reverting violations. The first method can be used if the scenarios in which the condition will be violated can be detected. In this case, the reaction rule can be expressed as follows:

**when** an action will violate the condition, **then** prohibit this action.

In the case of the second method, the violation is detected after the action is performed and the action is subsequently reverted. In this case, the reaction rule can be expressed as follows:

**when** some action occurs, **if** the condition is violated, **then** revert this action.

A derivation rule describes the value of an attribute in terms of the values of other attributes. It follows that this derived value only changes when any of the source values change. The corresponding reaction rule can therefore be expressed as follows:

**when** any of the source values change, **then** set the derived value to the new derivation result.

In Appendix A, some examples of business rules, defined within a fictional business case, can be found for each of the categories; constraint rule, derivation rule and reaction rule.

---

Figure 3: Business rule types at different levels of abstraction[23]
2 Research context

This chapter describes the research context of this paper; starting with the motivation which drives the paper, followed by the research questions derived from this motivation, and a description of the method used to validate the result.

2.1 Motivation

Before describing the problem, an introduction of the Thinkwise Software Factory, a development environment used and developed by Thinkwise Software\(^1\), will be given. This will be followed by a problem currently faced by the Thinkwise Software Factory, and a set of requirements which need to be satisfied by the solution. From this problem statement and the requirements, a more general research focus will be distilled in chapter 2.2.

2.1.1 Introducing the Thinkwise Software Factory

The Thinkwise Software Factory is a model-driven development environment in which fully functional business applications can be developed. The Thinkwise Software Factory encompasses the entirety of the development process, from designing the database and the User Interface to implementing application-specific business rules and setting up permissions for users. With the exception of implementing the business rules, there is no programming involved whilst using the Thinkwise Software Factory. Instead, by designing various aspects of the application in the Software Factory, such as the database and User Interface, a developer builds up a model of metadata which describes the application like a blueprint.

The Thinkwise Software Factory works in tandem with several generic front-end applications, one for each of the supported platforms; Windows, Web and Mobile. These front-end applications are capable of interpreting the metadata of the application and initialising themselves accordingly. These applications leverage the metadata to build up a user interface according to the ‘blueprint’ and provide a set of standard features such as adding, updating, deleting, filtering and sorting data.

Because these standard features are implemented in a generic manner in the front-end applications, only the business rules which are specific to each application need to be programmed by the developer. All of these business rules, including constraints and derivations, are in fact structured as reaction rules in a manner similar to that described in chapter 1.3. Furthermore, the business rules are not implemented at the level of the front-end applications, but either at the level of the database or a web service between the front-end application and the database, as a middle-tier architecture. To this end, the Thinkwise Software Factory exposes a set of event handlers which are triggered by the front-end applications at the appropriate time. Each business rule is implemented as a code snippet which is woven into the corresponding event handler during the generation of the database or the web service.

2.1.2 Problem statement

The business rules of an application can be implemented on several different platforms. Currently, there are three different database environments and two different web service environments which are supported by the Thinkwise Software Factory and the front-end applications. Support for different platforms continues to increase, both in depth –such as a new database environment– and in breadth –such as a new type of architecture, e.g. support for web services.

\(^1\)http://www.thinkwisesoftware.com
The largest part of an application made with the Thinkwise Software Factory has a clear separation between the design and the implementation platform, and as such can easily switch between any of the supported platforms. The business rules, however, are bound to their implementation platform and will have to be rewritten if a client wishes to change platform. This need to rewrite the same business rules, with the same underlying logic in a different syntax for a different platform, forms the core problem which motivates this paper.

2.1.3 Requirements

The method described in this paper will be a conceptual one; it is not intended to be a detailed prototype which is developed specifically for the Thinkwise Software Factory. However, this paper is written with the problem described in chapter 2.1.2 in mind, and the intention to provide a method which can be built upon by Thinkwise Software. As such, the requirements—in a broad sense of the word—which Thinkwise Software would like see fulfilled, are listed below.

1. Storing business rules in the metamodel of an application, in a platform-independent form.
2. Transforming the platform-independent business rules to the procedural form of any of the supported platforms, such that from the perspective of the front-end applications nothing will change. This is to ensure backward compatibility.
3. Fitting well within the Thinkwise Software Factory. The implementation of a business rule should be linkable to any of the event handlers exposed by the Thinkwise Software Factory, which are derived from the metadata.

2.2 Research questions

In chapter 2.1.1 it is written that all of the business rules, which are currently programmed into the Thinkwise Software Factory by developers, are structured as reaction rules. Furthermore, the second requirement in chapter 2.1.3 requires these reaction rules to be transformed to a multitude of procedural forms. The focus of this research is therefore twofold, and is described by the research questions below.

1. How can reaction rules be expressed in a platform-independent form?
2. How can these expressions be transformed to a procedural form?

The method described in this paper will propose an answer to these research questions, while keeping the requirements stated in chapter 2.1.3 in mind. Validation of the method will be performed in accordance with chapter 2.3.

2.3 Validation

To validate the method described in this paper, a small and fictional business case will be used. To show how the method provides an answer to the research questions, the business rules in this business case will first be expressed in a platform-independent manner and subsequently transformed to a procedural form.

After this, a conclusion will follow to summarise how the proposed method addresses the research questions and satisfies the requirements stated in chapter 2.1.3.
3 Related work

Before discussing any related work in the field of expressing and evaluating business rules, an interesting concept described by Ross in [15] will be introduced in chapter 3.1. In [15], Ross proposes to distinguish three separate levels with regard to the expression of business rules. This distinction serves as a very clarifying method for discussing business rules, and will therefore be adopted by this paper. The related work in this chapter will also be categorised by these levels of expression.

3.1 Levels of expression

When expressing business rules, there are a number of aspects which must be considered. According to the Business Rules Manifesto, business rules must be written and validated by business people, however, it is also stated that formal logics are fundamental to the expression of business rules. Furthermore, business rules should be stored and managed in such a way that they can be readily re-deployed to different technical platforms. There is a tension between these points which will be addressed in this chapter.

There are various methods for expressing business rules, ranging from natural languages such as English, to formal systems, and many forms in-between. Each of these methods has advantages and disadvantages, the most notable of which being the trade-off between the readability for the business people and unambiguity of the expressions. The closer a language is to natural language, the easier it is for business people to understand its expressions, but the more ambiguous these expressions become. Ambiguous expressions are subject to interpretation, and as such, the more ambiguous an expression is, the more likely it is that two people will interpret the expression differently. Formal systems, however, are entirely unambiguous and therefore not subject to interpretation. Because of this, formalised expressions can be reliably interpreted by machines. By extension, formalised business rules are both readily evaluable by machines and re-deployable to different technical platforms.

The strong tension between readability and unambiguity implies that there is no ‘one-size-fits-all’ method for expressing business rules such that both business people and machines can ‘understand’ them. However, transcribing business rules from natural language to a formal logic is prone to error and causes the need for a lot of maintenance. Ideally, a method for expressing business rules should be formal in nature with a friendly interface to bridge the gap of readability for the business people.

To this end, Ross proposes to split the expression of business rules into an analysis-level (external) and a design-level (internal)\(^ {[15]} \). The external level represents where the rules are analysed and written, while the internal level represents expressions which can be evaluated by machines. In addition, Ross proposes a conceptual level which sits between this man-machine boundary, and argues that the ultimate power lies in this layer. Indeed, one can imagine that with a strong conceptual level, business rules can be interchangeably expressed in a multitude of external and internal representation techniques, as has been illustrated by Figure 4, and in a similar fashion by Figure 3. Furthermore, the conceptual level would remove the disconnect between the external and internal representations, thus removing the need for transcribing one into the other.
It follows that the method used for the external level must be such that the business rules are relatively easy to understand and create, while at the same time mapping onto a formal conceptual level. Furthermore, the method used for the internal level must be readily evaluable by a machine, and it must be possible to reliably transform the expressions between the external level and the conceptual level, and the conceptual level and the internal level.
3.2 The external level

The external level can be seen as an interface which maps onto the conceptual level, offering a friendly method which guides users in creating well-formed expressions of business rules. Because the implementation of the external level maps onto the conceptual level, it follows that different implementations of the external level are interchangeable. In this paragraph, existing examples of such implementations will be discussed.

3.2.1 Structured natural language

In [15], Ross gives several examples of methods for expressing rules at the external level, one of which being Structured English. Structured English is one example within a group of methods called ‘structured’ or ‘controlled’ natural language. As the name implies, these languages are in principle natural language, but they are bound by a formal grammar which restricts the language to a subset of the natural language. This formal grammar is in fact the conceptual level, and the structured natural language is the interface which maps onto it. Being a subset of a natural language, a structured natural language is less expressive, however, understandable by business people whilst remaining unambiguous[6].

3.2.2 Decision tables

Another example of a method for expressing business rules at the external level, given by Ross in [15], is the decision table. Decision tables are in essence tabular representations of decision logic[13]. In a decision table, each row represents a condition or an action, while each column represents an implication between combinations of these conditions and actions. The decision tables themselves form the interface for the underlying, formal decision logic, which is the conceptual level.

One of the strengths of decision tables is the concise manner in which they can display a multitude of alternative paths towards actions, given a group of conditions[13].

3.2.3 Diagrams

The decision logic mentioned in 3.2.2 can also be expressed in the form a diagram, such as a flowchart, which represents a decision path. Without going into too much detail, every decision point in a flowchart represents a condition, while every arrow leaving a decision point represents a possible outcome. Much like with decision tables, a condition point in a decision diagram can be followed by another condition point, or an action. In comparison with a decision table, each unique decision path in a decision diagram represents a column in a decision table.

Decision tables and decision diagrams such as a flowchart form an example of two external levels that are interchangeable due to their shared conceptual level.

3.2.4 Expression builders

A fourth example of a method for expressing business rules at the external level is the expression builder, or ‘point-and-click expression builder’ as referred to by Ross in [15]. Expression builders offer users a controlled method for building expressions, such that the expressions are always well-formed according to the grammar of a formal logic. The expressions can be built via the expression builder in a building block fashion.
This guidance towards well-formed expressions makes it easier for users to write well-formed expressions, but it would still require knowledge of the formal logic that is used. However, it is conceivable for an expression builder to be used in tandem with a structured natural language. The structure of the natural language can then be enforced through the expression builder, and it will be easier for users to understand what the building blocks mean, and understand how they can be combined.

3.3 The conceptual level

The conceptual level is the layer which sits between the external level and the internal level. The conceptual level must be formal in nature, in order to allow evaluation, parsing or transcription by machines. Furthermore, the conceptual level must be both independent from the external level and the internal level. This will allow the business rules to be readily re-deployed to different technical platforms, as well as representing the business rules with a different method at the external level.

3.3.1 First-order logic

For the conceptual level, Ross states that the representation language must at least be as powerful as predicate logic\(^{15}\), also known as first-order logic. If we consider the example of a business rule which was raised in chapter 1.1, we can express its most obvious interpretation in first-order logic in the following way

\[
\forall e \forall d \left[ \text{Weekend}(d) \implies \neg \text{HoursLogged}(e, d) \right]
\]

Or equivalently

\[
\forall e \neg \exists d \left[ \text{Weekend}(d) \land \text{HoursLogged}(e, d) \right]
\]

The examples above assume two predicates; ‘Weekend’ and ‘HoursLogged’, which respectively evaluate if a day is part of the weekend and if an employee has logged working hours on a day. Literally, the expressions above would translate to:

*For every employee ‘e’, and for every day ‘d’, if ‘d’ is part of the weekend, then ‘e’ has not logged working hours on ‘d’.*

And

*For every employee ‘e’, there does not exist a day ‘d’, for which ‘d’ is part of the weekend and ‘e’ has logged working hours on ‘d’.*
3.3.2 SBVR – Semantics of Business Vocabulary and Business Rules

In 2008, the Object Management Group adopted a specification called SBVR, which describes both a conceptual level and an external level for expressing business rules\[^{12}\]. SBVR stands for Semantics of Business Vocabulary and Business Rules, and has been employed and built upon by many researchers (e.g. [3], [4], [5], [6], [7], [9], [10] and [23]). Some researchers even consider SBVR to be the leading standard in the field of business rules\[^{7}\].

At the conceptual level, SBVR describes a rich, formal logic which supports full first-order logic, as well as the alethic and deontic modalities\[^{12}\]. At the external level, SBVR describes a structured natural language approach called Structured English\[^{12}\]. Furthermore, the specification describes how the external level maps onto the conceptual level, creating a very rich and powerful method for expressing business rules\[^{12}\].

3.3.2.1 SBVR Structured English

The SBVR specification provides an English vocabulary for describing vocabularies and stating rules\[^{12}\]. As such, the SBVR specification is a metamodel which can be instantiated to form the business vocabulary of a specific business application. The metamodel describes the types of Structured English components and how they can be used and combined in the Structured English expressions. Since the structure of the expressions is determined by the grammar of the formal logic, the Structured English components, or combinations thereof, map onto the components of the formal logic. Because the structure of the expressions is enforced in the metamodel, and the mapping of Structured English components on formal logic components is defined in the metamodel, all instances of the metamodel produce well-formed expressions in the language of the business, which can be transcribed into formal logic automatically. Some examples of the mapping of Structured English onto the formal logic can be seen in Figure 5.

<table>
<thead>
<tr>
<th>A.2.1.1 Quantification</th>
<th>A.2.1.2 Logical Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>each</td>
<td>universal quantification</td>
</tr>
<tr>
<td>some</td>
<td>existential quantification</td>
</tr>
<tr>
<td>at least one</td>
<td>existential quantification</td>
</tr>
<tr>
<td>at least ( n )</td>
<td>at-least-( n ) quantification</td>
</tr>
<tr>
<td>at most one</td>
<td>at-most-one quantification</td>
</tr>
<tr>
<td>at most ( n )</td>
<td>at-most-( n ) quantification</td>
</tr>
<tr>
<td>exactly one</td>
<td>exactly-one quantification</td>
</tr>
<tr>
<td>exactly ( n )</td>
<td>exactly-( n ) quantification</td>
</tr>
<tr>
<td>at least ( n ) and at most ( m )</td>
<td>numeric range quantification</td>
</tr>
<tr>
<td>more than one</td>
<td>at-least-( n ) quantification ( n = 2 )</td>
</tr>
</tbody>
</table>

Figure 5: Examples of Structured English mapping onto formal logic\[^{12}\]
In Figure 6, an example from the SBVR specification can be seen which shows a business rule from a business case. In this example, ‘rental car’ and ‘branch’ are instantiations of the general concept, and ‘is owned by’ is an instantiation of the binary verb concept. The two general concepts are in this case verb concept roles of the verb concept ‘is owned by’. This shows the relationship between the metamodel that is the SBVR specification, and a plausible instantiation of the metamodel regarding the rental of cars.

It is obligatory that each rental car is owned by exactly one branch.

Figure 6: Example of a Structured English expression

By transcribing this Structured English expression to SBVR’s formal logic at the conceptual level, we would get the following expression:

\[ O \forall r : RentalCar \exists^1 b : Branch \text{OwnedBy}(r, b) \]

In the expression above, O means ‘Obligatory’ and \( \exists^1 \) means ‘exactly one’.

When looking back to the running example of a business rule, it can be expressed using SBVR Structured English as follows:

**An employee must not log working hours on weekend days.**

The underlined words are instantiations of noun-concepts. The cursive words are verb concepts, which express a state of affairs regarding one or more noun concepts. Finally, the remainder of the words are keywords, which are for instance used for quantification, logical and modal operations.

In the example above, the word ‘an’ indicates an existential quantification over the general concept ‘employee’. The words ‘must not’ indicate an obligation formulation with a negation component. The words ‘log ... on’ indicate a ternary verb concept between the noun-concepts ‘employee’, ‘weekend days’ and ‘working hours’.

At SBVR’s conceptual level, the same business rule would look like this:

\[ \forall e : Employee \exists w : WorkingHours \exists d : WeekendDays \text{LogOn}(e, w, d) \]

Or equivalently

\[ \neg P \forall e : Employee \exists w : WorkingHours \exists d : WeekendDays \text{LogOn}(e, w, d) \]

Where \( F \) (Forbidden) and \( \neg P \) (not Permitted) are logical equivalents for expressing a prohibition.
3.4 The internal level

The internal level is the level at which a machine can ‘understand’ and execute business rules. An obvious example of an internal level of a business rule is a piece of program code, written in a certain programming language for a certain platform. This piece of program code would contain instructions which will make the computer behave such that it satisfies the business rule as it was specified at the external and conceptual level.

3.4.1 Rule engines

Another example of an internal level is the rule engine, which is a popular method for the evaluation and execution of business rules.

In short, a rule engine is a program that can evaluate a given input against rules to determine the output\textsuperscript{18}. A rule engine is supplied with a set of formal rules, e.g. business rules, which formulate decisions at an abstract level\textsuperscript{18}. These rules are then used to evaluate input data, which contains specific instances of the concepts used in the rules, and make decisions about the required action to be taken\textsuperscript{18}.

Rule engines are advocated by the Business Rules Approach as a method for evaluating business rules\textsuperscript{16}. This makes sense, because rule engines suit the practices described by the Business Rules Approach very well. By using a rule engine, the business rules can be changed easily, the rules are defined in a formal manner, and are stored in one cohesive body in such a way that they can easily be redeployed to a different technical platform. Furthermore, changes made to the business rules will be reflected by the business applications immediately.
4 General approach

Recall that the focus of this paper is twofold. Firstly, expressing reaction rules in a platform-independent manner, and secondly, transforming these expressions to procedural form. These points will be handled in chapter 5 and chapter 6 respectively. Some choices regarding the approach in these chapters will be discussed here.

In chapter 1.3, we saw that reaction rules have a distinct Event-Condition-Action structure. The SBVR standard, despite having a very rich and powerful conceptual level, cannot be used to express these reaction rules. The reason for this is that it offers no means by which the semantics of events and actions can be captured. However, because the conceptual level of SBVR is largely based on first-order logic, it is very well suited for expressing the conditions of reaction rules. After all, the conditions of reaction rules are expressions that must evaluate to true or false, which is something that is inherently true for all expressions in first-order logic. This is also demonstrated by SBVR’s ability to express constraint rules and derivation rules.

Chapter 5 will propose a method to express reaction rules. Even though SBVR is considered one of the leading standards in the field of business rules, the proposed method is not intended to be an extension of the SBVR standard. The reason for this is that the SBVR standard places particular emphasis upon the external level and the conceptual level, and a systematic transformation between the two, whereas this paper will focus on the conceptual level and the transformation to the internal level. That being said, it is conceivable for the method proposed in this paper to be built upon in order to form an extension to SBVR for reaction rules. This will be further discussed in chapter 8.2. Instead of extending SBVR, the proposed method will extend first-order logic, in order to leverage its strength for expressing conditions, and add a means for expressing events and actions.

Chapter 6 will demonstrate how expressions which follow the conceptual level, proposed in chapter 5, can be transformed to a specific form of program code. This paper will not consider a rule engine for the evaluation and execution of the expressions, due to the requirements stated in chapter 2.1.3. However, recall that both program code and rule engines can be placed at the internal level. This means that from the perspective of the conceptual level, they are not mutually exclusive. As such, a rule engine can be used as well, which will be further discussed in chapter 8.2.

When transforming the expressions to program code, a transformation method must be created for each form of program code. However, many of these transformation methods will prove to be similar. For example, the transformation methods for two procedural dialects of SQL will likely have many similarities, as will the transformation methods for two object-oriented programming languages. In order to prove the concept of transforming a platform-independent reaction rule to a piece of readily executable program code, transformation steps to T-SQL have been worked out in chapter 6.
5 Expressing reaction rules

This chapter proposes a formal, conceptual level for the expression of reaction rules. Before discussing the syntax and semantics of this conceptual level, however, a detailed definition of a reaction rule in the context of this paper is given.

5.1 Defining reaction rules

As was shown in chapter 1.3, reaction rules have an Event-Condition-Action structure. The precise meaning of the concepts ‘Event’, ‘Condition’ and ‘Action’ in this paper, as well as their respective relationships, will be explained in this chapter.

5.1.1 Event

An event is the named occurrence of a specific action in an application, which corresponds to an event handler in order to provide a handle for a reaction to this action. An event is often an action which is initiated by a user in an application and of such relevance that the action may require a reaction in accordance with the specification of a business rule. Whenever an event occurs, the application will call the corresponding event handler, which is a type of procedure in which the reaction to this event is specified.

An Event-Condition-Action expression is the conceptual representation of an event handler. An event handler may take arguments which will be supplied by the application and provide context regarding the action which triggered the event handler. Because of this, the event construct can be seen as the signature of the procedure, while the Condition-Action expressions form the body of the procedure and specify the reaction according to the specification of the business rule.

5.1.2 Condition

A condition is an expression, built up out of predicates and logical connectives, which is always evaluable to either true or false. This makes first-order logic very suitable for expressing these conditions, as was already argued in chapter 4. The grammar that defines first-order logic can be seen in Figure 7.

In reaction rules, conditions are used to constrain in which situations an action is executed, often based upon the provided arguments in the event handler. A condition and an action therefore share an antecedent-consequent relationship, similar to that of a logical implication.
<domain> ::= s where s ∈ SET
<variable> ::= x for any x ∈ VAR
<argument> ::= a for any a ∈ F₀
<argument> ::= f( <argument-list> ) for any f ∈ F₁, F₂, ...
<argument> ::= <variable>
<argument-list> ::= <argument>
<argument-list> ::= <argument>, <argument-list>

<atomic-formula> ::= p | T | ⊥ where p ∈ P₀
<atomic-formula> ::= p( <argument-list> ) for any p ∈ P₁, P₂, ...
<formula> ::= <atomic-formula>
<formula> ::= ¬<formula>
<formula> ::= ( <quantifier><variable> : <domain> <formula> )
<formula> ::= ( <formula> <op> <formula> )

<op> ::= ∧ | ∨ | ⇒ | ⇔
<quantifier> ::= ∀ | ∃

Figure 7: From https://proofwiki.org/wiki/Definition:Predicate_Calculus/Formal_Grammar/Backus-Naur, with the minor addition of <domain> to use quantifiers on specific domains, i.e. a specific set.

5.1.3 Action

Much like an event, an action can be seen as the signature of a procedure for which the implementation is predefined for all supported platforms. Unlike an event, however, an action will not be called directly by the application, but will be called from within an event handler. Actions are used to request the application to perform a certain predefined operation, based upon the specification of a business rule.

Actions can be predefined in a development environment such as the Thinkwise Software Factory to achieve a common action in the application, but could also be added manually by a developer to achieve a custom action.

5.1.4 Bidirectional interface

To give an overview of the previous paragraphs, an event is a predefined means by which the application calls upon a selection of business rules to evaluate a certain situation. An action is a means by which the business rules can call upon the application to perform a certain action. The events and actions therefore form a bidirectional API (Application Programming Interface) which can be used to implement reaction rules.

5.2 The proposed grammar

Since first-order logic, shown in Figure 7, is very suitable for expressing conditions, it will be incorporated in the definition of Event-Condition-Action expressions. The conditions in these expressions can be any well-formed expression in first-order logic.

By adding a definition for the concepts event and action, and specifying how event, condition and action may be used in tandem, a grammar is given which describes the language of well-formed Event-Condition-Action expressions. The definition of this grammar can be seen in Figure 8.
\[
\begin{array}{ll}
<\text{arg}> & ::= \text{<variable>} \\
<\text{proc-arg}> & ::= \text{arg-type <arg>} \\
              & \mid \text{arg-type <arg> output} \\
<\text{proc-arg-list}> & ::= \text{<proc-arg>} \\
              & \mid \text{<proc-arg>, <proc-arg-list>} \\
<\text{event}> & ::= \varepsilon \\
              & \mid \varepsilon( \text{<proc-arg-list>}) \\
<\text{action}> & ::= \alpha \\
              & \mid \alpha( \text{<argument-list>}) \\
<\text{CA-expr}> & ::= \text{each <variable> : <domain> [ <CA-expr> ]} \\
              & \mid \text{<formula> }\rightarrow\text{ <CA-expr>} \\
              & \mid \text{<action>} \\
<\text{CA-expr-list}> & ::= \text{<CA-expr>} \\
              & \mid \text{<CA-expr>, <CA-expr-list>} \\
<\text{ECA-expr}> & ::= \text{<event> }\rightarrow\text{ <CA-expr-list>} \\
\end{array}
\]

Figure 8: The grammar of well-formed Event-Condition-Action expressions

In order to avoid confusion between the syntax of implications in first-order logic and the connectives between events, conditions and actions, a different type of arrow is used: →.

\( T, \varepsilon \) and \( A \) describe the sets of instantiations of the metatypes type, event and action respectively. The meta-type type has not been mentioned before, because its role is trivial. The instantiations of type are the types which can be used for the arguments of events. These types can be predefined as they are not specific to an application, but they may be required in the transformation process.

The instantiations of events and actions are specific to a business application and represent the events that can occur in the application and the actions that can be used in the expressions. In a model-driven environment, however, the metadata of the application can be leveraged to generate all of the events, and all of the commonly used actions. Both events and actions can have zero or more arguments, respectively providing information the regarding the event that has occurred, and the action that is to be executed.

The following conditions with regard to the structure of the reaction rules, which are enforced by the definitions in Figure 8, must hold:

- An event is always the antecedent of a set of one or more Condition-Action expressions.
- A formula which expresses a condition is always the antecedent of an action.
- An action is always the consequent of a formula which expresses a condition, or an event.

With the addition given by Figure 8, it will become possible to describe, in a platform-independent manner, conditional reactions to event occurrences in a business application in the form of actions on that business application.
5.3 The proposed semantics

In this chapter, the semantics of the reaction rules will be illustrated by describing in a step-by-step manner what happens when a sequence of events is triggered.

Let $\mathcal{E}$ be the set of instances of events which can occur, and $R$ the set of rules which have been defined on the system. Note that in this context, ‘rule’ refers to a reaction rule, i.e. an $<$ECA-expr$>$.

For every event $\varepsilon \in \mathcal{E}$ there exists a sequence of rules $R(\varepsilon) \in R^*$ which will be triggered by event $\varepsilon$ and will be evaluated sequentially.

Let $S$ be the set of possible states of the system.

Every rule $r \in R$ has, when executed, an effect on the state of the system: $S_i \xrightarrow{r} S_{i+1}$. It is, however, possible that $S_i = S_{i+1}$.

The state of the system after evaluation of rule $r$ in state $s \in S$ is given by the function $\text{evalRule}(s, r)$.

Let $E \subseteq \mathcal{E}$ be the sequence of events which have occurred. The state of the system after the occurrence of these events is $\text{evalRule}(S_0, E)$, in accordance with the definitions below.

Definition of $\text{evalRule}(s, E)$:
- $\text{evalRule}(s, \emptyset) = s$
- $\text{evalRule}(s, e + +E) = \text{evalRule}(\text{evalRule}(s, R(e)), E)$

Definition of $\text{evalRule}(s, R)$:
- $\text{evalRule}(s, \emptyset) = s$
- $\text{evalRule}(s, r + +R) = \text{evalRule}(\text{evalRule}(s, r), R)$

Given these definitions, any sequence of occurred events will yield a sequence of zero or more nested $\text{evalRule}(s, r)$ statements. The first rule is evaluated against the initial state $S_0$ and every subsequent rule is evaluated against the result of the previous evaluation. Accordingly, the history of the system is defined by:

1. The initial state $S_0$ of the system.
2. The sequence of events $E$ which have occurred.
6 Transforming reaction rules to T-SQL

Using the grammar of Condition-Action expressions in chapter 5.2, two types of T-SQL expressions can be defined. Namely, those expressions which perform a single action when a certain condition is met, and those expressions which perform an action for each record in a set which satisfies a certain condition. The distinction between these two expressions is entirely and respectively determined by the absence or presence of ‘each <variable> : <domain> [ <CA-expr> ]’ expression-parts, which introduce iterations over sets. The former type of expression will be referred to as a singular expression, while the latter type of expression will be referred to as a query-expression.

It must be noted that because of this distinction, two types of conditions can be distinguished as well. Namely, those conditions which constrain an action or inner-expression, and those conditions which constrain the set upon which an action is performed. This distinction is entirely determined by whether the condition is nested inside an ‘each <variable> : <domain> [ <CA-expr> ]’ expression-part, or not. In the former case, it constrains the set upon which an action is performed, while in the latter case it constrains when its inner-expression is evaluated.

To make the transformation to T-SQL easier, we will reflect this distinction by creating two separate definitions; one for singular expressions and one for query-expressions. Each of these definitions is a subset of the definition which was given earlier, while together describing the whole definition. As such, the expressivity of the expressions neither increases nor decreases.

6.1 Singular expressions

This paragraph will handle the definition of a singular expression, argue what the semantically equivalent T-SQL constructs are, and describe a transformation from the definition of a singular expression to said T-SQL construct.

6.1.1 The definition of a singular expression

The singular expressions can be described in accordance with the definition below. Note that a singular expression can contain a query-expression as an inner-expression. The definition of <query-expr> will be given in chapter 6.2.1.

\[
<\text{singular-expr}> ::= \text{<formula>} \rightarrow <\text{singular-expr-part}> \\
| \text{<action>}
\]

\[
<\text{singular-expr-part}> ::= <\text{singular-expr}> \\
| <\text{query-expr}>
\]

The syntax diagram corresponding to the definition above can be seen in Figure 9.
6.1.2 Transforming a singular expression

As can be seen in Figure 9, a <singular-expr> is either an <action> or an inner-expression which is evaluated when a certain condition is satisfied. An <action> represents a predefined T-SQL statement or statement-block which is assumed to be a valid. The inner-expression can in turn be another <singular-expr> or a <query-expr>.

In chapter 6.4.1, it is argued that an inner-expression, which is only evaluated when a certain condition holds, is equivalent to an if-statement in T-SQL. So in order to transform a <singular-expr>, we need to know how to correctly transform a <singular-expr> to a T-SQL if-statement and how to transform a <query-expr>. The latter point will be handled in chapter 6.2.2.

6.1.2.1 The associated T-SQL constructs

The syntax-definition of the T-SQL if-statement can be seen in Figure 10.

```
IF Boolean_expression
    { sql_statement | statement_block } 
[ ELSE
    { sql_statement | statement_block } ]
```

*Boolean_expression*

Is an expression that returns TRUE or FALSE. If the Boolean expression contains a SELECT statement, the SELECT statement must be enclosed in parentheses.

*{ sql_statement | statement_block }*

Is any Transact-SQL statement or statement grouping as defined by using a statement block. Unless a statement block is used, the IF or ELSE condition can affect the performance of only one Transact-SQL statement. To define a statement block, use the control-of-flow keywords BEGIN and END.

This definition includes an optional ‘ELSE-part’, however, it is not possible to describe an if-else statement in one singular expression. Instead, such a result would be achieved by creating two singular expressions in which the first has a condition and the second has the inverse condition. Furthermore, the inner-expression can be a T-SQL ‘statement_block’, therefore the inner-expression will always wrapped by the ‘BEGIN’ and ‘END’ keywords.
Figure 11 shows the syntax diagram of the T-SQL if-statement after having stripped away the optional and irrelevant parts.

**T-SQL if-statement:**

```
IF Boolean_expression BEGIN statement_block END
```

Figure 11: The syntax diagram of the simplified definition of the T-SQL if-statement.

### 6.1.2.2 The transformation of a singular expression to T-SQL

Looking back at the syntax diagram of the `<singular-expr>`, we can transform these expressions to valid T-SQL statements quite easily, as can be seen in Figure 13. Note that ‘T..formula’ and similar expressions in the syntax diagrams must be read as T(formula), i.e. the transformation of formula expression-part.

**singular-expr:**

```
formula -> singular-expr
```

Figure 12: The syntax diagram of `<singular-expr>`.

**T(singular-expr):**

```
IF T..formula BEGIN T..singular-expr T..query-expr END
```

Figure 13: The T-SQL transformation of `<singular-expr>`, shown as a syntax diagram.

For this transformation to be valid, two conditions must hold.

Firstly, T(singular-expr) and T(query-expr) must both be of the type ‘sql_statement’. According to the T-SQL definition, ‘sql_statement’ can be any valid T-SQL statement, which includes an if- or query-statement. The transformation of `<singular-expr>` can be either T(action) or a T-SQL if-statement. It follows from the premise that the T-SQL representation of an action is a valid T-SQL statement, and it follows from this proof that the transformed T-SQL if-statement is valid. The condition that T(query-expr) is a valid T-SQL query-statement will be proven later on in chapter 6.2.2.

Secondly, T(formula) must be of the type ‘Boolean_expression’. According to the T-SQL definition, ‘Boolean_expression’ can be any expression that returns true or false. This is inherently true for expressions in first-order logic, however, the expressions must be transformed to a valid and equivalent syntax in T-SQL.
Since T-SQL supports negations, conjunctions and disjunctions in its Boolean expressions, each of these will be transformed explicitly. The implication and equivalence operators can be expressed in terms of negation and conjunction, so these formulas will not be given further attention. Furthermore, the universal quantifier can be expressed in terms of an existential quantifier by using a double negation, so this formula will be omitted from the transformation as well. The definition of <formula> can therefore be simplified to what is seen in Figure 14.

Transforming these expressions to valid and equivalent T-SQL statements is trivial and will not be handled in detail. The results can be seen in Figure 15.

The <atomic-formula> is a predicate metatype, which, much like <action>, has user-defined instances. T-SQL supports various basic predicates for which instances of <atomic-formula> can be defined. This too, is quite trivial and will not be handled in more detail. A list of common T-SQL predicates can be found in Figure 16.
In the T-SQL transformation of <atomic-formula>, <domain> can be an existing table, view, table-valued function or table variable. A <value> can be a constant, a variable, a scalar function or the value of a column from a single record of a <domain>. A <string_expression> can be a string of characters which may include certain wildcard characters in accordance with the specifications\[26\]. These restrictions must be enforced in the corresponding instances of atomic-formula at the external level.

Using these transformations, <formula> can be transformed to a valid T-SQL ‘Boolean_expression’. This means that T(singular-expr) is a valid T-SQL statement if the condition that T(query-expr) is a valid T-SQL statement holds.
6.2 Query-expressions

This paragraph will describe the definition of a query-expression, and the transformation from this definition to a T-SQL query-statement.

6.2.1 The definition of a query-expression

The query-expressions are those expressions which cannot be described by the definition for singular expressions, but can be described by the definition which can be seen below.

\(<\text{query-expr}> ::= \text{each} \ <\text{variable}> : \ <\text{domain}> \ [ \ <\text{query-expr-part}> ]\)

\(<\text{query-expr-part}> ::= \text{each} \ <\text{variable}> : \ <\text{domain}> \ [ \ <\text{query-expr-part}> ]\)

\| \ <\text{formula}> \rightarrow \ <\text{query-expr-part}>\)

\| \ <\text{action}>\)

Figure 17 and Figure 18 respectively show a syntax diagram of <query-expr> and <query-expr-part>. Together they form the definition of the query-expressions.

query-expr:

\[
\text{each} \ \text{variable} \ : \ \text{domain} \ [ \ <\text{query-expr-part}> ]
\]

query-expr-part:

\[
\text{each} \ \text{variable} \ : \ \text{domain} \ [ \ <\text{query-expr-part}> ] \ \text{formula} \rightarrow \ <\text{query-expr-part}>
\]

\[
\text{action}
\]

Figure 17: The syntax diagram of <query-expr>; the root of a query-expression.

Figure 18: The syntax diagram of <query-expr-part>; a recursive part of a query-expression.

As can be seen in Figure 17 and Figure 18, a <query-expr> is a nested and interleaved sequence of one or more iterations over a set and zero or more condition-parts, with eventually an <action> as the inner-most expression-part. As such, a <query-expr> is an <action> which is applied to each item that meets certain conditions in the Cartesian product of a group of sets. This is the same thing as executing a T-SQL statement on the result-set given by a T-SQL query which uses cross joins. T-SQL allows for a number of T-SQL statements to be applied as an action upon such a set in a query-statement. In chapter 6.4.2 is it argued in more detail that a <query-expr> may be considered to be semantically equivalent to a T-SQL query-statement which uses cross joins.

However, the query-expressions cannot be transformed to T-SQL query-statements yet in a mechanical manner. One problem arises from the nature of the T-SQL query-statements, compared to the nature of the query-expressions. In T-SQL query-statements, all of the source-sets are grouped in the FROM-clause, and all of the set-constraining conditions are grouped in the WHERE-clause. By contrast, we have seen that the iterations over sets and the set-constraining conditions in query-expressions can be interleaved. To make it possible to mechanically transform query-expressions to
T-SQL query-statements, the iterations over sets and the set-constraining conditions must be grouped together. This can be achieved by applying the simple transformation step below.

All expression-part occurrences of the form ‘<formula> \(\rightarrow\) each <variable> : <domain> [ <query-expr-part> ]’ must be transformed to ‘each <variable> : <domain> [ <formula> \(\rightarrow\) <query-expr-part> ]’. Even though the latter expression appears to be less efficient because it evaluates the condition for each iteration, these expressions are effectively equivalent. More details regarding this can be found in chapter 6.4.2.

After applying this transformation step, all query-expressions can be described by the following definition:

\[
\begin{align*}
<query\text{-expr}> &::= \text{each } <variable> : <domain> [ <query\text{-expr-part}> ] \\
<query\text{-expr-part}> &::= <next\text{-from-part}> \\
&\quad | <where-part> \\
&\quad | <action> \\
<next\text{-from-part}> &::= \text{each } <variable> : <domain> [ <query\text{-expr-part}> ] \\
<where-part> &::= <formula> \rightarrow <where-part> \\
&\quad | <formula> \rightarrow <action>
\end{align*}
\]

Figure 19 and Figure 20 show the revised syntax diagrams of the query-expressions.

**query-expr:**

```
  each  variable  :  domain  [  query-expr-part  ]
```

*Figure 19: The syntax diagram of <query-expr>; the root of a query-expression.*

**query-expr-part:**

```
  each  variable  :  domain  [  query-expr-part  ]
    \rightarrow  \text{action}
```

*Figure 20: The revised syntax diagram of <query-expr-part>; a recursive part of a query-expression.*

This transformation gives us query-expressions which contain a sequence of one or more expression-parts which introduce an iteration over a set, followed by a sequence of zero or more expression-parts which introduce a condition, followed by an <action>. From a T-SQL perspective, we can already see the FROM- and WHERE-clause emerge, which will make the transformation much easier.

### 6.2.2 Transforming a query-expression

To transform <query-expr> to a valid T-SQL query-statement, we will first look at what T-SQL query-statements are. T-SQL has three types of query-statements, namely the select-statement, the update-statement and the delete-statement[27]. Each of these query-statements can be split into two
distinct parts; a query that returns a set of data which can be combined from several sources and constrained by several conditions, and an action that is applied to each record of this result.

Each of these query-statements can be structured in the same manner; the action-part is written first, followed by the query-part. Each query-statement also supports the same syntax for the query-part, which will allow us to transform all query-statements in the same way.

The query-part of a query-statement is built up out of several clauses. In order, the query-statement may contain a FROM-, WHERE-, GROUP BY-, HAVING- and ORDER BY-clause\[^{28}\]. Out of these five clauses, only the FROM-clause is mandatory in a query-statement\[^{28}\] and only the FROM- and WHERE-clause will be considered in this paper. The other clauses can be used to achieve more advanced results and support for these clauses can be added in the future, but they cannot yet be described by the current query-expressions. The FROM- and WHERE-clause of a query-statement can be used to indicate source sets of which the Cartesian product is taken and the conditions which constrain this result to all records which satisfy the conditions. This is true for both inner-joins and cross-joins, but not for outer-joins\[^{29}\]. Consequently, outer-joins are not yet supported either, but can also be supported by extending the definition of the query-expressions.

The action-part of each type of query-statement is bound to a certain syntax, but this part is the result of the transformation of <action>. As such, the action-part is assumed to have a valid transformation.

### 6.2.2.1 The associated T-SQL constructs of the FROM-clause

The syntax of the FROM-clause is quite complex, but will be limited to supported constructs. Furthermore, the source-sets which can be used will be limited to tables, views, table-valued functions and table variables, as these are the most common and share a similar syntax. Given these simplifications, the syntax of the FROM-clause can be simplified to what is seen in Figure 21.

\[
\text{FROM} \{ \text{<table_source>} \} [\ldots n ] \\
<\text{table_source}> ::= \\
\{ \\
\text{table_or_view_name} [\ [\text{AS} ] \text{table_alias} ] \\
\mid \text{user_defined_function} [\ [\text{AS} ] \text{table_alias} ] \\
\mid \text{@variable} [\ [\text{AS} ] \text{table_alias} ] \\
\}
\]

**Figure 21:** The syntax definition of the T-SQL FROM-clause\[^{30}\].

The syntax diagram of the definition above can be seen in Figure 22.

**Figure 22:** The syntax diagram of the simplified version of the T-SQL FROM-clause.
In this definition, ‘table_or_view_name’, ‘user_defined_function’ and ‘variable’ all represent the source of a tabular set of data in the query-statement. In turn, ‘table-alias’ is a unique identifier of that set and can be used to access the column values of each record in the set, which can be used in actions and in predicates. A comma-separated list in the FROM-clause can be used to describe both inner-joins and cross-joins, as an inner-join can be seen as a cross-join with at least one condition.

6.2.2.2 The transformation of the FROM-clause

The FROM-clause of the query-statement is built up out of the ‘each <variable> : <domain> [ <query-expr-part> ]’ expression-parts from <query-expr> and <next-from-part>. In this expression-part, <domain> represents the source of a set of data, and <variable> binds to each item in the set, providing access to its values. As such, the transformation can be made given that the conditions below are satisfied. Both of these conditions should be enforced at the external level.

- The <domain> refers to the name of an existing table, view or table variable. Or a valid call of an existing table-valued function.
- The <variable> is unique within the FROM-clause.

The transformations of <query-expr> and <next-from-part> can be seen in Figure 24 and Figure 26.

query-expr:

```
[each] variable : domain [ query-expr-part ]
```

Figure 23: The syntax diagram of <query-expr>.

T(query-expr):

```
FROM domain variable T..query-expr-part
```

Figure 24: The T-SQL transformation of <query-expr>, shown as a syntax diagram.

next-from-part:

```
[each] variable : domain [ query-expr-part ]
```

Figure 25: The syntax diagram of <next-from-part>.

T(next-from-part):

```
domain variable T..query-expr-part
```

Figure 26: The T-SQL transformation of <next-from-part>, shown as a syntax diagram.

Note that <query-expr> and <next-from-part> are the only expression-parts which contain a <query-expr-part>, and that <next-from-part> is one of the three definitions of <query-expr-part>. Because of this, <query-expr> is always followed by zero or more <next-from-part>‘s. The resulting FROM-clause is therefore always in accordance with Figure 22.

6.2.2.3 The associated T-SQL constructs of the WHERE-clause

The syntax-definition of the WHERE-clause is much simpler than that of the FROM-clause, and can be seen in its entirety in Figure 27.
WHERE <search condition>
<search_condition>::=
  { [ NOT ] <predicate> | ( <search_condition> ) }
  { [ AND | OR ] [ NOT ] <predicate> | ( <search_condition> ) } 
  [...n]
<predicate>::=
  { expression { = | < > | != | > | >= | != | <= | <> } expression
  | string_expression [ NOT ] LIKE string_expression [ ESCAPE 'escape_character' ]
  | expression [ NOT ] BETWEEN expression AND expression
  | expression IS [ NOT ] NULL
  | CONTAINS ( { column | * } , '<contains_search_condition>' )
  | FREETEXT ( { column | * } , 'freetext_string' )
  | expression [ NOT ] IN ( subquery | expression [...n] )
  | expression { = | < > | != | > | >= | != | <= | <> } { ALL | SOME | ANY } ( subquery )
  | EXISTS ( subquery )

Figure 27: The syntax definition of the T-SQL WHERE-clause\cite{11\cite{13}.}

Many of the T-SQL predicates which were listed earlier can be seen in the definition, including a few extra predicates. Since all of these predicates can be combined in the same way in ‘search_condition’, they can all implemented as instances of the <atomic-formula> metatype.

The syntax diagram of the definition of the WHERE-clause can be seen in Figure 28.

T-SQL WHERE-clause:

Figure 28: The syntax diagram of the T-SQL WHERE-clause.

search-condition:

Figure 29: The syntax diagram of ‘search-condition’ as seen in Figure 27.

However, the syntax diagram of ‘search-condition’ can be greatly simplified to what is seen in Figure 30.

search-condition:

Figure 30: A simplified version of the syntax diagram of 'search-condition'.
6.2.2.4 The transformation of the WHERE-clause

After the simplification of ‘search-condition’, it is equivalent to $T(formula)$, with the exception of the last line in the syntax diagram of $T(formula)$. This last line describes an expression of the form “$EXISTS(subquery)$”. However, as can be seen in the definition of ‘predicate’, this expression can be described by the ‘predicate’ type. As such, $T(formula)$ is equivalent to ‘search-condition’.

This means that <where-part> can be transformed to a valid WHERE-clause without the need for more transformations. This transformation can be seen in Figure 32.

where-part:

Figure 31: The syntax diagram of <where-part>.

$T(where-part)$:

Figure 32: The T-SQL transformation of <where-part>.

Since the <where-part> can contain nested conditions, which must all be true in order for the <action> to be evaluated, these conditions are strung together with a T-SQL AND-operator. Note that the <action> is not transformed here, because it will be transformed and placed before the FROM-clause, in the action-part of the query-statement. More about this can be read in the next paragraph.

6.2.2.5 Transforming the action-part

Despite the fact that it is not the responsibility of this transformation process to ensure the validity of a transformed <action>, it is the responsibility of this transformation process to ensure that <action> will be transformed and placed at the correct location.

An <action> is always the inner-most expression-part of a set-based expression, while it is always the first clause in a query-statement. $T(action)$ must therefore be placed before the FROM-clause. To achieve this, we will define a function which, given a <query-expr>, returns $T(action)$ of that <query-expr>.

\[
\begin{align*}
\text{action-part(query-expr)} & ::= \text{action-part(query-expr-part)} \\
\text{action-part(query-expr-part)} & ::= \text{action-part(next-from-part)} \\
& | \text{action-part(where-part)} \\
& | T(action) \\
\text{action-part(next-from-part)} & ::= \text{action-part(query-expr-part)} \\
\text{action-part(where-part)} & ::= \text{action-part(where-part)} \\
& | T(action)
\end{align*}
\]

When transforming a <query-expr>, the result of this function for that <query-expr> must simply be placed before the FROM-clause. The results can be seen in Figure 33.
6.2.2.6 Combining the results

At this point, the entire set-based expression can be transformed to a valid T-SQL query-statement. The complete transformation can be seen in Figure 33 and Figure 34.

T(query-expr):

```
FROM domain variable T.query-expr-part
```

Figure 33: The final T-SQL transformation of <query-expr>.

T(query-expr-part):

```
WHERE T.formula
AND
```

Figure 34: The final T-SQL transformation of <query-expr-part>.

6.3 Events

Chapters 6.1 and 6.2 show how a Condition-Action expression can be transformed to a T-SQL statement. This chapter will show how an entire Event-Condition-Action expression can be transformed to a T-SQL procedure, to form the event handler which can be called upon by an application.

Since the transformation of the event construct to a T-SQL procedure which wraps around the transformed Condition-Action expression is quite trivial, it will not be described in great detail.

The syntax diagram of <ECA-expr> can be seen in Figure 35. The syntax definition of a T-SQL procedure, stripped of a few optional parts, can be seen in Figure 36. The result of the transformation can be seen in Figure 37.

```
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name
[ { @parameter [ type_schema_name. ] data_type }
[ OUT | OUTPUT ] [ ,...n ]
AS { [ BEGIN ] sql_statement ; [ ,...n ] [ END ] }
```

Figure 36: The syntax definition of a T-SQL procedure.
**T(eca-expr):**

![Diagram of T(eca-expr)](image)

*Figure 37: The T-SQL transformation of <ECA-expr>.*

The syntax diagrams of <arg> and T(arg) can be found in Figure 38 and Figure 39. As was mentioned earlier, the transformation of ‘arg-type’ is predefined.

![Diagram of <arg>](image)

*Figure 38: The syntax diagram of <arg>.*

![Diagram of the transformation of <arg>](image)

*Figure 39: The syntax diagram of the transformation of <arg>.*

The syntax diagrams of <CA-expr-list> and T(CA-expr-list) can be found in Figure 40 and Figure 41.

![Diagram of <CA-expr-list>](image)

*Figure 40: The syntax diagram of <CA-expr-list>.*

![Diagram of the transformation of <CA-expr-list>](image)

*Figure 41: The syntax diagram of the transformation of <CA-expr-list>.*
6.4 Semantical correctness

Chapters 6.1 and 6.2 have proven the syntactical correctness of the transformations they describe by matching the transformation on the syntax diagrams of the official T-SQL syntax definitions. This chapter will argue the semantical correctness of the transformations, that is, whether the result of the transformation achieves the same behaviour in T-SQL as was intended by the original expression.

The semantical correctness will be argued by making it plausible that a T-SQL if-statement is the semantical equivalent of a singular Condition-Action expression, and a T-SQL query-statement is the semantical equivalent of a query Condition-Action expression. Both points were touched upon earlier, but will be handled in more detail in this chapter.

6.4.1 Singular expressions

A singular expression can have one of three forms:

1. An action.
2. A singular expression which should only be evaluated when a condition is satisfied.
3. A query-expression which should only be evaluated when a condition is satisfied.

In the first case, the transformation will simply be the transformation of the action, which will be a T-SQL statement or statement-block. The transformations of actions are predefined, so the transformation process cannot change the semantics of the expression. In the second case, the transformation will be an if-statement which uses the transformation of the condition as its condition and the transformation of the singular expression as its inner T-SQL statement. The inner T-SQL statement will only be evaluated if the transformation of the condition evaluates to true. The transformation of the condition has been shown to be trivial and syntactically correct, and we can assume that T-SQL’s boolean semantics are the same as that of first-order logic. As such, the inner T-SQL expression will only be evaluated in the same situations as intended by the original expression. It follows that if the transformation of the singular expression is semantically correct, which follows from this chapter, that result of the transformation has the same meaning as the original expression.

In the third case, the transformation will be an if-statement which uses the transformation of the condition as its condition and the transformation of the query-expression as its inner T-SQL expression. For this case, the same argument as for the second case can be made, with the addition that the transformation of the query-expression is semantically correct. This will be proven in chapter 6.4.2.

6.4.2 Query expressions

A query expression can contain the following constructs:

1. One or more `‘each <variable> : <domain> …’` parts.
2. Zero or more `‘<formula> → …’` parts.
3. Exactly one `<action>`.

The first construct represents an iteration over a set and evaluates its inner-expression for each occurrence in the set, where `<variable>` binds to each occurrence. Nesting such constructs will cause the inner-expression to be evaluated for the combination of each occurrence in each set, that is, each occurrence in the result of the Cartesian product of these sets.
The second construct describes an inner-expression which should only be evaluated if the first-order logic expression represented by <formula> evaluates to true. This construct is directly or indirectly contained within one or more instances of the first construct. This construct therefore restricts for which iterations the inner-expression is evaluated.

The third construct is always the inner-most component of a query-expression and represents an action that is to be executed. The action will be executed for all iterations described by components of the first type, except those which do not satisfy the conditions described by components of the second type.

The query-expressions are transformed to T-SQL query-statements. Every component of first type introduces a source of tabular data of which the Cartesian product is taken (cross join). Every component of the second type introduces a condition which removes all records of the Cartesian product which do not satisfy the condition. Finally, the action will be performed for each remaining record. It can therefore be argued that the semantics of the query-statement is equivalent to the semantics of the original expression.

Furthermore, the order in which the conditions are applied and the Cartesian products are taken is independent of the order in which these components are specified in the query-statement, but is decided by the query-optimizer instead. This is why reordering these components in the original expression, to promote the ease of transformation, will not affect the semantics of the transformed expression.
7 Validation – Business Case

In this chapter, it will be demonstrated that the method proposed in this paper answers the research questions and satisfies the validation described in the chapters 2.2 and 2.3 respectively. To this end, the small, fictional business case which can be found in Appendix A is used.

7.1 Defining the events and actions

Before translating the business rules from the business case into platform-independent reaction rules, we will first define a set of plausible events and actions which could be exposed by an application. As was stated earlier, events and action are not produced by the reaction rules, but are predefined.

The way in which business rules can be implemented depends on the available events and actions. Since the motivation for this paper is a problem faced by the Thinkwise Software Factory, the events and actions used in this example will be inspired by those used in the Thinkwise Software Factory.

Events

- **Entity_ValueChanged** – An event that is fired when a user has changed the value of one of the attributes of an entity, whilst in the process of inserting or updating a record. An output argument is supplied for each attribute of the entity in question, containing the current value of that attribute and allowing for a new value to be specified. Furthermore, an output argument is supplied with which can be controlled if the changes are allowed to be saved.

- **Entity_Inserted** – An event that is fired when a user has inserted a record into an entity. An argument is supplied for each attribute of the entity in question, containing the inserted value of that attribute.

- **Entity_Updated** – An event that is fired when a user has updated a record of an entity. Two arguments are supplied for each attribute of the entity in question, containing the old value and the new value of that attribute.

- **Entity_Deleted** – An event that is fired when a user has deleted a record of an entity. An argument is supplied for each attribute of the entity, containing the deleted value of that attribute.

- **Entity_ContextChanged** – An event that is fired when the context of an entity is changed. For instance, Project and ProjectHours have a master-detail relationship on the project_id attribute. ProjectHours are shown in the context of a certain Project and selecting a different Project will change the context of ProjectHours. An argument is supplied for each attribute which determines the context, containing the new value of that attribute. Furthermore, output arguments are supplied with which can be controlled if inserting or updating is allowed.

The events listed above are merely a selection which will suffice for expressing the business rules in the business case. Many more events can be considered plausible.
The set of events which will be used for this business case can be found below. The events below are written with all of their arguments, however these will be omitted later on for the sake of brevity.

\[ E = \{ \]

**Project_Inserted** (int projectId, int clientId, date startDate, date finishDate, bool finished, decimal hourlyRate, decimal cost, int duration),

**Project_Updated** (int old_projectId, int new_projectId, int old_clientId, int new_clientId, date old_startDate, date new_startDate, date old_finishDate, date new_finishDate, bool old_finished, bool new_finished, dec old_hourlyRate, dec new_hourlyRate, dec old_cost, dec new_cost, int old_duration, int new_duration),

**Project_ValueChanged** (int projectId out output, int clientId out output, date startDate out output, date finishDate out output, bool finished out output, decimal hourlyRate out output, decimal cost out output, int duration out output, bool allowSave out output),

**ProjectHours_ValueChanged** (int projectId out output, int personId out output, date date out output, int nrOfHours out output, bool allowSave out output),

**ProjectHours_ContextChanged** (int projectId, bool allowInsert out output, bool allowUpdate out output),

**Person_Deleted** (int personId, string firstName, string lastName, int age, string emailAddress)

\[ \}

**Actions**

- **Entity_Delete** – An action which will delete an instance of an entity. This action takes one argument; the instance of the entity to be deleted. It must be noted that, because the argument requires an instance of an entity to be supplied, the action can only be used inside an ‘each <var> : <domain> [ <CA-expr> ]’ expression, where <var> will be bound to such an instance. Such constraints are specific to certain actions and will therefore need to be enforced at the external level.

- **SetArgumentValue** – An action which will set the value of an output argument of an event. This action takes two arguments; the argument and the value.

- **ShowMessage** – An action which will cause the application to show a message to the user. This action takes one argument; the message which needs to be shown.

Just like with the events, the entity action will be available for all entities. The set of actions which will be used for this business case can be found below.

\[ A = \{ \]

**Project_Delete** (instance),

**ProjectHours_Delete** (instance),

**SetArgumentValue** (argName, argValue),

**ShowMessage** (text)

\[ \}

Predicate functions and other basic functions, with implementations for every platform, will also need to be defined in order to build up conditions and, for instance, perform basic arithmetic actions.
However, these functions and their implementations are so trivial that they will not be given individual attention. They will be used in the formalisation of the business rules, and their transformations can be found in chapter 7.3.1.

7.2 Formalising the business rules to ECA-expressions

Now that the actions and events are known, the business rules can be formalised. The business rules in Appendix A are written in natural language, and as such are inherently open to interpretation. In this chapter, the business rules will be formalised with a plausible interpretation. Ascertaining that this formalisation is correct, is outside of the focus of this paper. However, as touched upon earlier, and described in more detail in chapter 8.2, it is possible to extend the SBVR specification to this end.

Constraint rule 1

\[
\begin{align*}
\text{Project_ValueChanged (\ldots)} & \\
\rightarrow & \\
\rightarrow & \neg \text{IsNull (startDate)} \land \neg \text{IsNull (finishDate)} \land \text{SmallerThan (finishDate, startDate)} \\
\rightarrow & \text{SetArgumentValue (allowSave, \bot)} \\
\rightarrow & \neg \text{IsNull (startDate)} \land \neg \text{IsNull (finishDate)} \land \text{SmallerThan (finishDate, startDate)} \\
\rightarrow & \text{ShowMessage ('Finish-date cannot be before start-date.' )}
\end{align*}
\]

Constraint rule 2

\[
\begin{align*}
\text{ProjectHours_ContextChanged (\ldots)} & \\
\rightarrow & \exists p : \text{Project} [ \text{Equals (p.project_id, projectId)} \land \text{Equals (p.finished, } \top)] \\
\rightarrow & \text{SetArgumentValue (allowInsert, } \bot)
\end{align*}
\]

Constraint rule 3

\[
\begin{align*}
\text{ProjectHours_ValueChanged (\ldots)} & \\
\rightarrow & \text{LargerThan (nrOfHours, 12)} \\
\rightarrow & \text{SetArgumentValue (allowSave, } \bot) \\
\rightarrow & \text{LargerThan (nrOfHours, 12)} \\
\rightarrow & \text{ShowMessage ('No more than 12 hours can be booked per day')}
\end{align*}
\]

Derivation rule 1

\[
\begin{align*}
\text{Project_ValueChanged (\ldots)} & \\
\rightarrow & \text{IsNull (finishDate)} \\
\rightarrow & \text{SetArgumentValue (finished, } \bot) \\
\rightarrow & \neg \text{IsNull (finishDate)} \\
\rightarrow & \text{SetArgumentValue (finished, } \top)
\end{align*}
\]
**Derivation rule 2**

\[ \text{Project_ValueChanged(...) \implies } \neg \text{IsNull (startDate)} \land \neg \text{IsNull (finishDate)} \implies \text{SetArgumentValue (duration, Minus(finishDate, startDate))} \]

**Reaction rule 1**

\[ \text{Project_Inserted(...) \implies } \text{IsNull (startDate)} \implies \text{Project_SetValue ('start-date', CurrentDate())} \]

**Reaction rule 2**

\[ \text{Person_Deleted(...) \implies } \text{each ph : ProjectHours [ each pr : Project [ \text{Equals (pr.client_id, personId)} \land \text{Equals (pr.project_id, ph.project_id)} \implies \text{ProjectHours_Delete (ph)} ] ] } \]

\[ \text{each pr : Project [ \text{Equals (pr.client_id, personId)} \implies \text{Project_Delete (pr)} ] } \]
7.3 Transforming the business rules

Since many of the expressions use the same constructs and are similar in complexity, only the first of each such case will be transformed in this chapter in order to prove the concept. Constraint rule 1, Constraint rule 2 and Reaction rule 2 together cover all types of expressions used and will be transformed in this chapter.

7.3.1 Supplying transformations for the functions and actions used

The transformations of the predicate functions, the miscellaneous functions and the actions that were used in the formalisation of the business rules can be found in the following tables.

<table>
<thead>
<tr>
<th>Predicate function</th>
<th>T-SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>1</td>
</tr>
<tr>
<td>( \perp )</td>
<td>0</td>
</tr>
<tr>
<td>Equals ( (x, y) )</td>
<td>( x = y )</td>
</tr>
<tr>
<td>SmallerThan ( (x, y) )</td>
<td>( x &lt; y )</td>
</tr>
<tr>
<td>LargerThan ( (x, y) )</td>
<td>( x &gt; y )</td>
</tr>
<tr>
<td>IsNull ( (x) )</td>
<td>( x ) is null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Miscellaneous function</th>
<th>T-SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply ( (x, y) )</td>
<td>( x \ast y )</td>
</tr>
<tr>
<td>Minus ( (x, y) )</td>
<td>( x - y )</td>
</tr>
<tr>
<td>CurrentDate ()</td>
<td>GetDate()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>T-SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project_Delete ( \text{instance} )</td>
<td>DELETE instance</td>
</tr>
<tr>
<td>ProjectHours_Delete ( \text{instance} )</td>
<td>DELETE instance</td>
</tr>
<tr>
<td>SetArgValue ( \text{argName, argValue} )</td>
<td>SELECT argName = argValue</td>
</tr>
<tr>
<td>ShowMessage ( \text{caption, text} )</td>
<td>PRINT text</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>T-SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>int</td>
</tr>
<tr>
<td>bool</td>
<td>bit</td>
</tr>
<tr>
<td>date</td>
<td>date</td>
</tr>
<tr>
<td>Dec</td>
<td>numeric ( (5, 2) )*</td>
</tr>
<tr>
<td>String</td>
<td>varchar ( \text{max} )*</td>
</tr>
</tbody>
</table>

* More precise mappings for length can be made, these types merely demonstrate this example.

7.3.2 Transforming the ECA-expressions

The first step of transforming an ECA-expression to T-SQL is transforming the event-part to a T-SQL procedure which will serve as an event handler. This transformation step is illustrated in Figure 35 and Figure 37 in chapter 6.3.

The next step is the transformation of \( \text{<CA-expr-list>} \), that is, the list of Condition-Action expressions that is to be evaluated when the event occurs. The transformation of \( \text{<CA-expr-list>} \) is shown in Figure 41 to simply be the sequential transformation of each \( \text{<CA-expr>} \).

When transforming a \( \text{<CA-expr>} \) to T-SQL it is important to determine whether the expression is a singular expression or a query expression. The distinction between these two expressions has
been explained in chapter 6 and the transformation steps have been illustrated in Figure 13 for singular expressions and Figure 33 and Figure 34 for query expressions. It must be noted that a query expression can be contained within a singular expression as well.

Finally, all of the functions and actions used in the Condition-Action expressions will need to be replaced by their corresponding transformation. The transformations of these functions and actions has been given in chapter 7.3.1.

7.3.2.1 Transforming constraint rule 1

Constraint rule 1 uses the Project_ValueChanged event, for which the definition can be seen below.

**Project_ValueChanged** (int projectId output, int clientId output, date startDate output, date finishDate output, bool finished output, decimal hourlyRate output, decimal cost output, int duration output, bool allowSave output)

Transforming this event in accordance with Figure 37 and Figure 39 gives the following result.

```
CREATE PROCEDURE Project_ValueChanged
    @projectId int output,
    @clientId int output,
    @startDate date output,
    @finishDate date output,
    @finished bit output,
    @hourlyRate numeric(5, 2) output,
    @cost numeric(5, 2) output,
    @duration int output,
    @allowSave int output
AS
BEGIN
    T (CA-expr-list)
END
```

The list of Condition-Action expressions in constraint rule 1 contains two Condition-Action expressions, namely:

\[ \neg \text{IsNull (startDate)} \land \neg \text{IsNull (finishDate)} \land \text{SmallerThan (finishDate, startDate)} \]

\[ \text{SetArgumentValue (allowSave, } \bot \text{)} \]

And

\[ \neg \text{IsNull (startDate)} \land \neg \text{IsNull (finishDate)} \land \text{SmallerThan (finishDate, startDate)} \]

\[ \text{ShowMessage ('Finish-date cannot be before start-date.' )} \]
Both of these expressions are singular expressions, and transforming them in accordance with Figure 13 gives the following result.

\[
\text{IF } \neg \text{IsNull}(T(\text{startDate})) \land \neg \text{IsNull}(T(\text{finishDate})) \land T(\text{SmallerThan}(T(\text{finishDate}), T(\text{startDate})))
\]

\[
\text{BEGIN}
\text{T(SetArgumentValue (T(allowSave), T(⊥)))}
\text{END}
\]

\[
\text{IF } \neg \text{IsNull}(T(\text{startDate})) \land \neg \text{IsNull}(T(\text{finishDate})) \land T(\text{SmallerThan}(T(\text{finishDate}), T(\text{startDate})))
\]

\[
\text{BEGIN}
\text{T(ShowMessage(‘Finish-date cannot be before start-date.’))}
\text{END}
\]

The first-order logic expressions which form the conditions of the if-statements, and the event arguments which are used in the functions, still need to transformed. These transformation steps are described by Figure 15 and Figure 39 respectively. Applying these steps gives the following result.

\[
\text{IF } (\neg (T(\text{IsNull (@startDate)}) \land \neg (T(\text{IsNull (@finishDate)}) \land T(\text{SmallerThan (@finishDate, @startDate)})))
\]

\[
\text{BEGIN}
\text{T(SetArgumentValue (@allowSave, T(⊥))})
\text{END}
\]

\[
\text{IF } (\neg (T(\text{IsNull (@startDate)}) \land \neg (T(\text{IsNull (@finishDate)}) \land T(\text{SmallerThan (@finishDate, @startDate)})))
\]

\[
\text{BEGIN}
\text{T(ShowMessage (‘Finish-date cannot be before start-date.’))}
\text{END}
\]

Finally, the functions and actions can be replaced by their T-SQL representations which were given in chapter 7.3.1. This gives the following, final result.

\[
\text{CREATE PROCEDURE Project_ValueChanged}
\text{ @projectId int output,}
\text{ @clientId int output,}
\text{ @startDate date output,}
\text{ @finishDate date output,}
\text{ @finished bit output,}
\text{ @hourlyRate numeric(5, 2) output,}
\text{ @cost numeric(5, 2) output,}
\text{ @duration int output,}
\text{ @allowSave int output}
\text{AS}
\text{BEGIN}
\text{IF (NOT (@startDate is null) AND NOT (@finishDate is null) AND @finishDate < @startDate})}
\text{BEGIN}
\text{SELECT @allowSave = 0}
\text{END}
\text{IF (NOT (@startDate is null) AND NOT (@finishDate is null) AND @finishDate < @startDate})}
\text{BEGIN}
\text{PRINT ‘Finish-date cannot be before start-date.’}
\text{END}
\text{END}
\]
7.3.2.2 Transforming constraint rule 2

Constraint rule 2 uses the event Project_ContextChanged, the definition and transformation of which can be seen below.

**ProjectHours_ContextChanged** (int projectId, bool allowInsert output, bool allowUpdate output)

```sql
CREATE PROCEDURE ProjectHours_ContextChanged
    @projectId int,
    @allowInsert bit output,
    @allowUpdate bit output
AS
BEGIN
    T (CA-expr-list)
END
```

Constraint rule 2 has only one Condition-Action expression which is a singular expression. Its definition, together with its transformation according to Figure 13, Figure 15 and Figure 39 can be seen below.

\[ \exists p : Project \land (p.project_id, \text{projectId}) \land (p.finished, T) \]

\[ \text{SetArgumentValue}(allowInsert, \bot) \]

```sql
IF EXISTS (SELECT 1 FROM Project p WHERE (T (Equals (p.project_id, @projectId))) AND T (Equals (p.finished, T (T))))
BEGIN
    T (SetArgumentValue (@allowInsert, T (\bot)))
END
```

Finally, the functions and actions can replaced by their T-SQL representation, giving the following, final result.

```sql
CREATE PROCEDURE ProjectHours_ContextChanged
    @projectId int,
    @allowInsert bit output,
    @allowUpdate bit output
AS
BEGIN
    IF EXISTS (SELECT 1 FROM Project p WHERE (p.project_id = @projectId AND p.finished = 1))
    BEGIN
        select @allowInsert = 0
    END
END
```
7.3.2.3 Transforming reaction rule 2

Reaction rule 2 uses the Person_Deleted event, the definition and transformation of which can be seen below.

**Person_Deleted** (int personId, string firstName, string lastName, int age, string emailAddress)

```sql
CREATE PROCEDURE Person_Deleted
  @personId int,
  @firstName varchar(max),
  @lastName varchar(max),
  @age int,
  @emailAddress varchar(max)
AS
BEGIN
  T (CA-expr-list)
END
```

The `<CA-expr-list>` of Reaction rule 2 contains two Condition-Action expressions, namely:

```plaintext
each ph : ProjectHours [
  each pr : Project [
    →  Equals (pr.client_id, personId) ∧ Equals (pr.project_id, ph.project_id)
    ProjectHours_Delete (ph)
  ]
]
```

And

```plaintext
each pr : Project [
  →  Equals (pr.client_id, personId)
  Project_Delete (pr)
]
```

Both of these expressions are query expressions, and will be transformed according to Figure 33 and Figure 34. The first-order logic expressions and the event arguments used in the functions will be transformed according to Figure 15 and Figure 39 respectively. This gives the following result.

```sql
T (ProjectHours_Delete (ph))
FROM ProjectHours ph, Project pr
WHERE ( T (Equals (pr.client_id, @personId)) AND T (Equals (pr.project_id, ph.project_id)) )

T (Project_Delete (pr))
FROM Project pr
WHERE T (Equals (pr.client_id, @personId))
```
Finally, the functions and actions can once again be replaced by their T-SQL representations. This gives the following, final result.

```sql
CREATE PROCEDURE Person_Deleted
    @personId int,
    @firstName varchar(max),
    @lastName varchar(max),
    @age int,
    @emailAddress varchar(max)
AS
BEGIN
    DELETE ph
    FROM ProjectHours ph, Project pr
    WHERE (pr.client_id = @personId AND pr.project_id = ph.project_id)

    DELETE pr
    FROM Project pr
    WHERE pr.client_id = @personId
END
```

7.4 Conclusion

Chapter 7.2 shows how the proposed grammar for well-formed reaction rules can be used to express all of the business rules stated in Appendix A, including the constraint rules and derivation rules. In these expressions, the logic of the business rule is isolated and captured in a concise expression. With this, an answer is provided for the first research question: “How can reaction rules be expressed in a platform-independent form?”.

Chapter 7.3 shows how the well-formed reaction rules can be transformed to readily executable T-SQL program code, by following the transformation steps defined in chapter 6. These transformation steps have been validated against the official syntax definitions of T-SQL to ensure that the result of the transformation is always valid according to the syntax of T-SQL. Furthermore, in chapter 6.4 the semantical correctness of the transformation is argued to make it plausible that the meaning of the business rules remain unchanged by the transformation. The transformation steps to T-SQL serve as a proof of concept, and defining transformation steps for other programming languages will allow transformations to a multitude of procedural forms. With this, an answer is provided for the second research question: “How can these reaction rules be transformed to procedural form?”.

In chapter 2.1.3, an additional requirement states that it must be possible for the expressed business rules to be coupled with a predefined set of events. This property is in fact inherent to reaction rules, as the nature of reaction rules is such that an explicit reaction to an explicit event is expressed. This is also reflected by the proposed grammar for expressing reaction rules, which contains definitions for the metatypes ‘event’ and ‘action’. This means that every well-formed reaction rule is dependent on the available, predefined instances of events and actions. In chapter 5.1 it is explained in more detail how the instances of events and actions can be compared to a bidirectional API for modelling reaction rules. To further illustrate this point, the business rules from Appendix A have been implemented using a set of events that have been inspired by those which are already used in the Thinkwise Software Factory.
8 Conclusions and future work

8.1 Conclusions

In this work, we have proposed a method to express reaction rules in a platform-independent manner, and transform the resulting expressions to a procedural form. A reaction rule is a type of business rule that is structured in an Event-Condition-Action manner. The semantics of such a rule are as follows; when an event occurs, if a condition is satisfied, then perform an action. This structure makes reaction rules very expressive and allows for constraints and derivations to be expressed as well.

The motivation for this work stems from the problem of needing to rewrite business rules whenever the platform on which the business rules are implemented and evaluated, changes. Rewriting the business rules is a matter of implementing the same business logic in a different syntax for a different platform. Since the underlying logic does not change, rewriting the business rules should not be necessary. This is not a new problem, as rule engines have been developed in the past by a variety of researchers to address similar problems. However, whether a rule engine is a good solution is very situational. It depends on factors such as the type of rules, the amount of rules, the frequency of change, and the complexity of the rules. There are also scenarios in which a rule engine may not be viable as a solution at all, or at least not in the short term. An example of such a scenario is when the business logic has been separated from the business application and the business application has built-in support for communication with a number of platforms on which this business logic may be implemented. In this scenario, it may be required to change the platform on which the business logic is evaluated, without changing the business application by adding support for communication with a rule engine. When a rule engine is not viable or not an ideal solution, code generation may be more suitable.

The proposed method describes a grammar which extends first-order logic with the concepts 'event' and 'action'. First-order logic by itself is very well suited for expressing statements about a domain, which evaluate to either true or false. By extending the definition of first-order logic, this strength is leveraged in expressing the conditions of reaction rules, which are also statements about a domain which evaluate to true or false. The grammar defines how the concepts 'event' and 'action', and the first-order logic expressions, can be combined to create well-formed reaction rules. By expressing the reaction rules in this way, the platform-independent logic is separated from the platform-dependent syntax in a concise expression.

It is subsequently shown how well-formed reaction rules can be transformed to program code through a sequence of transformation steps. It is necessary to define such transformation steps for each programming language to which the expressions must be transformed. When the transformation steps have been defined and validated, the transformation of any well-formed reaction rule can be automated. In this work, the transformation steps for T-SQL, a procedural extension to the SQL standard, have been worked out and validated as a means to verify the concept. The choice for T-SQL was made because this is the most prominent programming language in which business rules are programmed in the model-driven development environment; Thinkwise Software Factory.
To validate the method proposed in this paper, a fictional business case is used. This business case contains a small domain model and a few business rules of different types and complexities in natural language. The business rules are interpreted and formalised according to the proposed grammar for well-formed reaction rules and subsequently transformed to T-SQL program code according to the defined transformation steps. A plausible interpretation of the business rules is used in the formalisation process, because it is not the focus of this work to ensure validity of this process. This point will be elaborated upon in chapter 8.2.

8.2 Future work

The focus of this work can be found in the conceptual level of reaction rules and its transition to an internal level in the form of program code. The external level of reaction rules, that is, the level where the rules will be created, has not been given attention in this work. A good, next step would therefore be to describe a mapping of an external level onto the proposed conceptual level. This way, people with no knowledge of the underlying, formal grammar can create well-formed reaction rules. Given the position of SBVR as a leading standard in the field of business rules, the conceptual level proposed in this work could be incorporated into the SBVR standard to achieve this goal. The vocabulary of SBVR and its formal foundation can be expanded with the concepts 'event' and 'action', and the Structured English mapping can be expanded accordingly.

Another conceivable step to be taken is using a rule engine for the internal level of the reaction rules. One option is to determine the possibility of describing transformations between the proposed conceptual level and the conceptual levels of existing rule engines. Another option is to work out a rule engine specifically for the proposed conceptual level.

Aside from expanding the method described in this work in breadth, there also additions possible in depth. One example of such an addition is validation at the conceptual level, to detect conflicting rules or other erroneous situations. Another example is the addition of optimisation at the conceptual level for the purpose of removing redundancy and improving the efficiency of the rules at the internal level.
9 References


9.1 Sources for the T-SQL syntax


Appendix A – Business case

This appendix describes a small domain model in which several business rules of all categories have been defined, for the purpose of establishing a fictional business case which will be used to demonstrate the methods described in this paper.

This business case describes a small project management system which contains projects and hours that can be booked on projects. To guide and constrain some of the business aspects, a few business rules have been defined. These business rules are intended to be realistic, but the business case as a whole is not intended to be complete or usable.

Entities of the domain model will be written in bold and underlined letters, while attributes of entities will only be underlined.

<table>
<thead>
<tr>
<th>Project</th>
<th>ProjectHour</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>project_id</td>
<td>project_id</td>
<td>person_id</td>
</tr>
<tr>
<td>client_id</td>
<td>person_id</td>
<td>first_name</td>
</tr>
<tr>
<td>start_date</td>
<td>date</td>
<td>last_name</td>
</tr>
<tr>
<td>finish_date</td>
<td>nr_of_hours</td>
<td>age</td>
</tr>
<tr>
<td>finished</td>
<td></td>
<td>email_address</td>
</tr>
<tr>
<td>hourly_rate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cost</td>
<td></td>
<td></td>
</tr>
<tr>
<td>duration</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Constraint rules

1. The finish_date of a Project may not be earlier in time than the start_date of that Project.
2. ProjectHours may not be booked on Projects which are marked as finished.
3. One Person may not book more than twelve nr_of_hours’ worth of ProjectHours on a single day.

Derivation rules

1. The finished status of a Project is equal to the finish_date not being empty.
2. The duration of a finished Project is equal to the start_date subtracted from the finish_date.

Reaction rules

1. When a new Project is being added, if the start_date of the Project is empty, the start_date must be set to the current date.
2. When a person is deleted, all ProjectHours of Projects for which for which this person is the client, and these Projects themselves, should be deleted.