Rule Based Design

Stef Joosten; Lex Wedemeijer; Gerard Michels

August 2010

Contents

Chapter 1

Inspired by Information

What inspires people to use Facebook, Apple computers or Google Earth? Why are Pixar and Dreamworks successful in animation? Which buzz draws scores of people to e-Bay and Amazon? How did Skype get where it is today? These are questions that occupy the minds of great designers of information systems. Inspiration, beauty, and fun are the common denominator of information technology today and in the years to come. Feelings, perceptions and experiences of users will increasingly drive the design of successful information systems.

Inspired design is demanding, however. It requires designers to express themselves, very much like a violinist. Before creating music that touches the hearts of your audience you must have full control over the instrument. This truth holds for any creative profession. Room for creativity comes, once your skills no longer require your attention. Designing business processes and information systems is no exception. Craftsmanship precedes art.

This book is about design. We want to design information systems for people who wish to comply to rules. Rules exist of many sorts. Some rules may be agreed upon by individuals, others imposed by law, and still others might be prescribed specific regulations that apply to your situation. From this book you will learn how to design based on those rules. We hope that this inspires you to realise information systems in which people dare to share; reassured by the knowledge that they live by the rules.

Rule based design was developed as a contribution to the profession of designing information systems and business processes. It offers hope to designers who wish to lift their work to the level of artistry. This book proposes a way to produce flawless (i.e. correct, consistent and complete) designs. It shows how you can be sure that your design fully complies with the requirements of your patrons. It promises you can save time by automating design activities and showing you how to get it right the first time. That should free some mental space to spend on inspiration...

1.1 A vision

This book is written in a firm belief that one day, information systems will be designed and built rapidly, at low cost, and 100% compliant to the rules of the organization who ordered the system. Guaranteed functionality and more predictable innovation projects are part of that professionalism. One day, informations systems will be a commodity, abundantly available at predictable (low) cost, defined quality and immediate delivery. Such professionalism will create room for inspired design.

This book was written for designers who share this desire and are willing to invest in this vision. Besides inspiration and talent, genuine artistry requires knowledge, labor, and craftsmanship. This book aims at making you successful, by providing the knowledge. Labor requires you to practice your techniques until you can do them effortlessly and correctly. Craftsmanship is what you get when doing it for real.

The next few sections share with you a vision on business rules in different perspectives: business processes, coherence, information technology, and formal methods.

1.1.1 On business processes

Business rules have a high potential for changing the way we design business processes. Since business rules can be communicated so much easier, rule based BPM carries the promise of bringing BPM closer to the business. This solves the problem of confining process modeles, imposed by workflow technology. Case management [35, 9], which is generally seen as the successor of workflow technology because it provides much more flexibility, still requires businesses to invest in process modeling.

Rule based BPM does not have these limitations. It enforces nothing but business rules that are required by the business itself. As a consequence, the design can be automated further. It has also become clear that the power of business rules, when represented in a relation algebra, goes well beyond business process management alone.

1.1.2 On coherence

Coherence among stakeholders of an organisation, both internal and external, grows on clear commitments which are kept. After all, commitments bind people. Each commitment contributes to a coherent organisation from the moment it is maintained. Maintaining a commitment requires explicit and controllable phrasing. To distinguish any agreement from an explicitly and controllably phrased commitment, we reserve the word "rule" for the latter. A rule that reflects a commitment from the business is consequently called a business rule. You will find lots of theory, leads and inspiration about business rules by reading the Business Rules Manifesto (www.businessrulesgroup.org/brmanifesto.htm) and consulting the world wide web. This book uses business rules as requirements (article 1 of the manifesto) from the business (article 9.1), of the business (article 10.1) and for the business (article 8). Besides, we will take it one step further, and enable you to specify software services using business rules.

business rule

1.1.3 On information technology

Rules apply across processes and procedures. There should be one cohesive body of rules, enforced consistently across all relevant areas of business activity (article 2.3 of the manifesto). This is precisely how information technology contributes to coherence. Computers can enforce all self-evident rules, since these do not require human interference. All other rules, which are the more interesting ones, are maintained by people. Rule violations can then be signalled by computers, prompting people to take action. The entire system of people and computers then shows coherent behavior, which is determined by the set of rules they are maintaining. The role of information technology is to support that. Because each little rule being maintained is one worry less, this vision brings about simplicity and transparency. Consistent implementation of this vision yields unpolluted databases and demonstrable compliance.

1.1.4 On formalism

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 (article 5.3 of the manifesto). This book uses a relation algebra to analyse requirements and formulate rules, a formalism equivalent to predicate logic but easier to use.

Relation algebras were invented in the nineteenth century by mathematicians, such as De Morgan, Peirce, and Schröder [8, 26, 30]. Their efforts have been studied, maintained and enhanced throughout the twentieth century, providing a well established language and well understood language for describing business rules today. Relation algebras enable professionals of today to describe and manipulate relations effectively, quickly, and without mistakes. Proficiency in this skill will allow you to make specifications, conduct conceptual analyses, and search for and invent new rules.

This book makes an effort to introduce a relation algebra slowly, taking the reader on the path to understanding one step at a time. The introduction to relation algebras is found in part 'Theory' of this book. If you are new to this topic, prepare for some effort, but rest assured that it will be worthwhile.

1.1.5 On methods

Design methods were proposed in the early seventies as an answer to the problem of size and complexity. Today, design methods recognize that several perspectives on an information system are required to get a design right. Design patterns are being advocated in order to reuse design knowledge. Designers of information systems even use the word *architecture* to underscore the complexity of their task. Progress is being made, but very slowly. If anything, design methods have exposed the full scale of the problem. Different representations of a design, depicted graphically, textually or otherwise, have allowed designers to share the full complexity of the task they embarked in. As a reaction, we have seen the advent of packaged enterprise information systems, promising out-of-the-box solutions on an enterprise scale. This promise too has proven difficult to achieve. By and large, many organisations still have difficulties designing information systems reliably and repeatably. This is precisely the problem addressed by this book.

1.2 Background

Many business rule engines that are available in the marketplace use rules of the type condition-action, or event-condition-action, both of which can be executed by a computer. Such rules are called "imperative", because they contain an action that tells a computer what to do (similar to programming languages). The Business Rules Manifesto [28] stresses the importance of *declarative* rules (as opposed to imperative rules) for the purpose of capturing the rules of the business. This book does not make business analysts write conditions and actions. We will focus on the commitments, rules and agreements of the business instead, and let conditions and actions be derived by computers. This works, presuming we have the technology to generate software for that purpose.

Business rules are as close to the business as one can get. They enlarge the scope of requirements engineering to include business goals [23]. Examples of business rules are:

- Applications for a new pension plan are decided upon within three days. (e.g. in the context of a life insurance company)
- Applications for green cards are put aside if the identity of applicants cannot be established legally. (e.g. in the context of immigration)
- Payments must be authorized by two different staff, both of whom are authorized for the full amount paid. (e.g. in the context of a medical benefits administrator)

Business rules formalize mutual agreements between stakeholders, commitments of managers, rights and obligations of employees, etc. into 'laws', intended to serve the purpose(s) of an organization. The creation and withdrawal of rules is an ongoing process, similar to a legislative process in a state.

This book introduces an approach, Ampersand, which is inspired by the belief that compliance with business rules should come automatically, once agreement about rules has been established. Design artifacts such as data models and service specifications ought to be derived from business rules, rather than drawn up by designers. This requires a formal method supplemented with tools, in which designers represent business rules in heterogeneous relation algebra [29]. The Ampersand approach, a successor of CC [10, 19], was developed for that very purpose. In conjunction with this approach, software has been developed that generates functional specifications directly from business rules. Also, there exists software that generates a protype application to enforce all business rules. For practical reasons, Ampersand has been designed such that any formal language is used exclusively by designers only. At no occasion the need arises to confront users with the formalism. Yet, business analists and software architects must learn how to use it, though. Our experience in teaching the method has shown that they can learn to use the language in a 100 hour course.

Rule based design draws on research on business rules, software engineering, relation algebra, and design methodology. Let us look at each of these topics briefly.

Research on business rules has a long tradition. Much of the research is related to active databases [7], that use the ECA pattern (event-condition-action) to describe rules. In research aimed at automating software design, such as [37], this leads to the assumption that business rules must be executable. From a business perspective, this is not necessarily true. A counterexample: the rule 'every action performed must be authorized by a superior', would become quite unworkable if every action performed would trigger an authorization request to a superior. In the Ampersand approach, actions are not specified, but derived. It is a purely declarative approach. All ECA-rules required to make computers work are derived (by a tool), enabling an error-free implementation of requirements.

In a comparison of available methodologies for representing business rules, Herbst et.al. [14] show that common methods are insufficient or at least inconvenient for a complete and systematic modeling of business rules. That study remarks that rules in all methodologies can be interpreted as ECA-rules. However, the authors do not identify the imperative nature of ECA-rules as an explanation for the shortcomings of methodologies. Methodologists generally place business rules on the data side of business models [32]. The Ampersand approach uses business rules to actually *define* the business process, making them relevant to both the data and the process side of information systems. The fact that this requires a formal method has consequences [38]. It means that system boundaries can be articulated, the functional behaviour of the system is defined, and there is proof that the system meets its specification.

Outside academia, business rules are increasingly acknowledged as an important development. Several interest groups exist:

Business Rules Forum	`http://www.businessrulesforum.com/`
Business Rules Group	`http://www.businessrulesgroup.org/brghome.htm`
European Business Rules Conference	`http://www.eurobizrules.org/`
Business Rules Community	`http://www.brcommunity.com/index.shtml`

Market researcher Gartner expects licence revenue in business rules technology to grow by 9.1% until 2010. Most rule engines that are available in the market place provide decision support by means of separating business rules from process logic. Current research on business rules takes similar directions, e.g. [2, 33, 25]. An advantage of separating business rules from process logic is that business processes (such as a mortgage application procedure) can be kept stable as rules and regulations change. By the same count, however, this separation restricts the use of business rules to applications in which the process is not affected, leaving designers with two tasks: rule modeling and process modeling. The Ampersand approach lifts this restriction, because the process is derived from business rules.

Our finding that business rules are sufficient to define business processes, corroborates the claim of the Business Rules Group [28] that rules are a first-class citizen of the requirements world (article 1.1 of the manifesto). The phrase ‘sufficient to define business processes’ implies that designers need not communicate with the business in any other way. If desired, they can avoid to discuss technical artifacts (data models, etc.) with the business. As a consequence, requirements engineering can stay much closer to the business, by using the language of the business proper.

This supports the Business Rules Group in her claims that business rules are a vital business asset, that rules are more important to the business than hardware/software platforms, and that business rules, and the ability to change them effectively, are fundamental to improving business adaptability.

Using relation algebra to describe the rules of an organization is a novel contribution of Ampersand. The formalism has been studied extensively and is well known for over a century [8, 26, 30]. Computer professionals are made familiar with it as part of their discrete mathematics courses. The information systems community is aware (e.g. [27]) that mathematical representations of business rules can be useful. Date [6] has criticized SQL for being unfaithful to relation algebra and advocates declarative business rules instead as an instrument for application development. This book implements some of Date's ideas, by using relation algebra faithfully to describe business rules and define the application both at the same time.

1.3 Practice

Various research projects and projects in business have supplied a significant amount of evidence that supports the power of business rules. These projects have been conducted in various locations. Research at the Open University of the Netherlands (OUNL) has focused on two things: developing the method, developing the course, and building a violation detector. Research at Ordina and TNO Informatie- en Communicatie Technologie has been conducted to establish the usability of ADL-rules for representing genuine business rules in genuine enterprises. Collaboration with the university of Eindhoven has produced a first design of a rule base. Experiments conducted at TNO, KPN, Bank MeesPierson, ING-Bank, Rabobank and Delta Lloyd have provided experimental corroboration of the method and insight in the limitations and practicalities involved. This section discusses some of these projects, pointing out which evidence has been acquired by each one of them.

The CC-method, the predecessor of Ampersand, was conceived in 1996, resulting in a conceptual model called WorkPAD [18]. The method used relational rules to analyze complex problems in a conceptual way. WorkPAD was used as the foundation of process architecture as conducted in a company called Anaxagoras, which was founded in 1997 and is now part of Ordina. Conceptual analyses, which frequently drew on the WorkPAD heritage, applied in practical situations resulted in the PAM method for business process management [21], which is now frequently used by process architects at Ordina and within her customer organizations. Another early result of the CC-method is an interesting study of van Beek [34], who used CC to provide formal evidence for tool integration problems; work that led to a major policy shift in IT-projects. The early work on CC has provided evidence that an important architectural tool had been found: using business rules to solve architectural issues is large projects.

For lack of funding, the CC-method has long been restricted to be used in conceptual analysis and metamodeling [20, 11], although its potential for violation detection became clear as early as 1998. Metamodeling in CC was first used in a large scale, user-oriented investigation into the state of the art of BPM tools [12], which was performed for 12 governmental departments. In 2000, a violation detector was written for the ING-Sharing project, which proved the technology to be effective and even efficient on the scale of such a large software development project. After that, the approach started to take off.

In the meantime, TNO Informatie- en Communicatie Technologie (at that time still known as KPN Research) has used CC modeling for various other purposes. For example, CC modeling has been used to create consensus between groups of people with different ideas on various topics related to information security, leading to a security architecture for residential gateways [17]. Another purpose that TNO used CC modeling for was the study of international standardizations efforts such as RBAC (Role Based Access Control) in 2003 and architecture (IEEE 1471-2000) [15] in 2004. Several inconsistencies have been found in the last (draft) RBAC standard [3]. Also, it was noticed that for conformance, the draft standard does not require to enforce protection of objects, which one would expect to be the very purpose of any RBAC system.

The analysis of the IEEE 1471-2000 recommendation has uncovered ambiguities in some of the crucial notions defined therein. Fixing these ambiguities and making the rules governing architectural descriptions explicit has resulted in a small and elegant procedure for creating (parts of) such architectural descriptions in an efficient and effective way [16]. Additional research at TNO [36] currently investigates the possibilities for creating context dependent 'cookbooks' for creating architectural descriptions in a given context, based on CC-modelling.

The efforts at TNO have provided the evidence that the CC-method works for its intended purpose, which is to accellerate discussions about complex subjects and produce concrete results from them in practical situations.

In 2002 research at the OUNL was launched to further this work in the direction of an educative tool. This resulted in the ADL language, which was first used in practice by Bank MeesPierson [5]. The researcher described rules that govern the trade in securities at MeesPierson. He found that business rules can very well be used to do perpetual audit, solving many problems where control over the business and tolerance in daily operations are in conflict. At Rabobank, in a large project for designing a credit management service center, debates over terminology were settled on the basis of metamodels built in CC. These metamodels resulted in a noticable simplification of business processes and showed how system designs built in Rational Rose should be linked to process models [4]. The entire design of the process architecture [24] was validated in CC. At the same time, CC was used to define the notion of skill based distribution. This study led to the design by Ordina of a skill based insurance back-office for Interpolis. The same CC-models that founded the design at Interpolis were reused in 2004 to design an insurance back office for Delta Lloyd. This work provided useful insights about reuse of design knowledge. It also demonstrated that a collection of business rules may be used as a design pattern [13] for the purpose of reusing design knowledge. In 2005 Ordina started a project to make knowledge reuse in the style demonstrated at Delta Lloyd into a repeatable effort. In 2006, the CC-method was refined into the Ampersand approach at the OUNL by adding the capability to generate functional specifications from rules.

1.4 Further research

This research shows that business rules have a high potential for changing the way business processes and information systems are designed. Since business rules can be communicated so much easier, rule based BPM carries the promise of bringing BPM closer to the business. The process control principle introduced in this chapter 2 takes business rules one step further: it shows that business rules can be used to control processes without using a process model. This is achieved by exploiting the implicit temporal constraints that can be derived from (static) business rules. This solves the problem of restrictively ordered activities, imposed by workflow technology. Currently, process modeling techniques force an ordering of activities upon the organization, which can sometimes be too confining. Where case management [35, 9] provides much more flexibility, it still requires process modeling and still requires a significant design effort. Rule based BPM does not have these limitations. It enforces only the rules that an organization is bound to, either by outside sources (e.g. legislation) or by creating rules internally.

To use rule based process control, designers must learn to represent business rules in relation algebra. Evidence gathered so far suggests this can be done, but more work is required to make it easy. Further research is planned to make tools that help designers formulate business rules in a graphical manner.

Potential benefits are possible in compliance and governance, as required for instance by Sarbanes-Oxley [1] and other legislature. Our findings support a tighter integration of formal methods in software engineering [38]. The increased quality of specifications can make offshoring much easier.

It has also become clear that the power of business rules, when represented in relational algebra, goes well beyond business process management alone. Rules have also been used in architectural studies, reusing knowledge in design patterns. Also, rules can be used to describe the modeling techniques and methods themselves; this is referred to as Metamodeling. The Open University of the Netherlands is currently teaching a metamodeling course and a business rule course, both of which employ the tools described in this book.

Further research is required to bring this work from principle to production. Work on a business rules repository is ongoing. The ADL-language will be extended beyond relation algebra to enhance support for process design. The prototype generator is being made suitable for use by business analysts, to enhance their ability to make good designs.

1.5 Acknowledgements

I wish to thank all of my students and all of Ordina's customers who have contributed to this work. The fact that most customers must remain anonymous does not diminish their contributions. The issues they raised in their projects have inspired Ampersand directly and indirectly. Besides, I wish to thank TNO in Groningen (the Netherlands) for being the first user of Ampersand.

1.6 Reading Guide

This introduction has provided an overview of business rules from a specific point of view: Business rules describe requirements, and therefore they can be used to design information systems and business processes. This book serves as an introduction to those who wish to make this happen in their own working environment. Chapter 2 introduces Ampersand, a method for rule based design of business processes and information systems. It argues that workers in an organization may perceive their work as "I do my job my way, as long as I stay within the rules" and it shows how information technology supports that. Chapter **??** introduces the language that is needed for the specification of business rules. Chapter **??** uses this language to specify business rules: expressions clear enough for that users to understand and work with, and at the same time unambiguous and precise enough to derive correct and buildable information systems. Chapter **??** elaborates on this, enabling the reader to develop and scrutinize rules on his own account. These skills can be practices in tools, provided with this course. The manual of this tool is presented in chapter **??**

Part I

Methodology

Chapter 2

Ampersand

An Approach to Control Business Processes

Abstract

This chapter shows how to use business rules for specifying both business processes and the software that supports them. This approach yields a consistent design, which is derived directly from business rules. This leads to software that complies with the rules of the business.

The approach, called Ampersand, is specifically suited for business processes with strong compliance requirements, such as financial processes or government processes that execute legislature. Features of Ampersand are: rules define a process, no process modeling is required, compliance with the rules is guaranteed, and rules are maintained by systematically signalling participants in the process.

Evidence that this principle works in practice is given by a case study.

2.1 Rules

Ampersand lets you design information systems to control business processes. It is based on rules. But what exactly are rules? Merriam-Webster's dictionary contains over 10 different definitions. Some of those are in agreement with this book:

- a prescribed guide for conduct or action

- an accepted procedure, custom, or habit
- a regulation or bylaw governing procedure or controlling conduct

For practical purposes, however, you will need a definition that you can use when you design information systems and business processes. We need a definition that lets you tell a rule apart from other statements. The following definition provides the means to say whether a statement is a rule or not.

DEFINITION Rule

> A *rule* is a verifiable statement that some stakeholders intend to keep true within the context of that statement.

Here is an example of a rule:

> In our club, the coats of all guests will be in the cloakroom,
> as long as they are in the club.

Let us analyse this statement with respect to the definition of rule.

scope

1. Rules have a *scope*.
   The context of this statement is the club in which this rule is valid. We call this the scope of this rule.

stakeholder

2. Rules have stakeholders.
   Anyone involved in a rule is called stakeholder. For example, guests must put their coats in the cloakroom, there may be a bell boy to take them in and hand them out again, there may be staff who sees to it that persons who carry their coats inside the club are intercepted, etcetera. Stakeholders who "live by the rules" are working to make rules true, each in his or her own way.

verifiable

3. Rules are *verifiable*. If there is anyone inside the club who carries a coat, we have a violation of this rule. That violation could signal anyone to take action (thereby restoring the truth to this rule).

For a better understanding, let us look at some counterexamples. The following statements are not rules:

- *Our club is transparent to the outside world.*
  Whether this statement is true or not is open for discussion, depending on what you think "transparent" means. For this statement to be a rule, we must be able to determine objectively whether it is true or not. Ampersand does not consider this statement to be a rule, because it is not verifiable.

- *Club members get up in the morning and go to sleep in the evening.*
  This is not a rule if none of the stakeholders really cares. If nobody is willing to maintain the truth, we have no rule.

- $E = mc^2$
  This is a law of nature, which is considered true in any scope and without the intervention for any stakeholder whatsover. Even if we would consider this to be a rule, it is not one that is very interesting or relevant to our purposes.

- *Peter Lee Jones has visited the club this morning.*
  This statement can be either true or false, so it is a verifiable statement. However, in case it is false, there is no intention to make it true and keep it true. Therefore it is not a rule.

Please note that the exact phrasing of a rule makes a difference. The statement "In our club, the coats of all guests will be in the cloakroom, as long as they are in the club." assumes a number of things, such as:

- There is a club, which we call "our club".

- Coats have owners, especially guests can be owner of a coat.

- Coats can be in the cloakroom. This also implies that there is a cloakroom.

- Guests stay in the club for a certain period of time.

If one of these assumptions is not true, the rule is meaningless.

Rephrasing might cause problems, due to the implicitness of assumptions of statements. The following examples show how seemingly meaningless rephrasings can yield unintended changes in meaning:

- In our club, guests will put their coats in the cloakroom.
  This rule does not prevent guest from taking their coat into the club. They can take the coat out right after putting it in the cloakroom.

- In our club, the coat of all guests will be in the cloakroom, as long as they are in the club.
  This rule assumes that every guest has precisely one coat. If this is not the case, then what?

- In our club, all coats must be kept in the cloakroom at all times.
  In this case, coats of members and staff are also kept in the cloakroom...

  - In our club, the bell boy will put your coat in the cloakroom.
    This rule affects anyone entering the club. It does not say what to do with your coat when the bell boy is absent. Besides, it is not specific about "you". In principle, this rule also applies to the mailman who drops by to deliver some mail...

In order to check whether a statement is a rule, please ask yourself the following questions:

1. Can you decide objectively whether it is true or false? If so, this statement is verifiable.
2. Where and in which situation does this statement make sense? This gives you the scope.
3. Can you identify who are affected by this rule? If so, these people (or groups) are your stakeholders.
4. Is there an intention to keep this statement true? If so, which stakeholder(s) take which action(s) to maintain this rule? If none of the stakeholders have such intention, your statement is not a rule.

2.2 Rules in Business

Business rules can be used to manage and control business processes. In this sense, business rules actually *define* the process. This yields compliant systems and compliant processes. This chapter explains the principle, which can be summarized as: signal violations (in real time) and act to resolve them. This defines a process engine that complies with all business rules. The consequence is that business rules are sufficient as an instrument to design compliant business processes and information systems.

Whenever and whereever people work together, they connect to one another by making agreements and commitments. These agreements and commitments constitute the rules of the business [28]. A logical consequence is that the business rules must be known and understood by all who have to live by them. From this perspective business rules are the cement that ties a group of individuals together to form a genuine organization. In practice, many rules are documented, especially in larger organizations. Life of a business analist can hardly be made easier: rules are known and discussed in the organization's own language and all stakeholders know (or are supposed to know) the rules.

The role of information technology is to help *maintain* business rules. That is: if any rule is violated, a computer can signal that and prompt people (inside and outside the organization) to resolve the issue. The Ampersand approach uses this as a principle for controlling business processes. For that purpose two kinds of rules are distinguished: rules that are maintained by people and rules that are maintained by computers.

A rule maintained by people may be violated temporarily, for the time required to fix the situation. For example, if a rule says that each benefit application requires a decision, this rule is violated from the moment an application arrives until the corresponding decision is made. This temporary violation allows a person to make a decision. For that purpose, a computer monitors all rules maintained by people and signals them to take appropriate action. Signals generated by the system represent (temporary) violations, which are communicated to people as a trigger for action.

A rule maintained by computers need never be violated. Any violation is either corrected or prevented. If for example a credit approval is checked by someone without the right authorization, this can be signalled as a violation of the rule that such work requires authorization. An appropriate reaction is to prevent the transaction (of checking the credit application) from taking place. In another example the credit approval might violate a rule saying that name, address, zip and city should be filled in. In that case, a computer might correct the violation by filling out the credit approval automatically.

Since all rules (both the ones maintained by people and the ones maintained by computers) are monitored, computers and persons together form a system that lives by the rules. This establishes compliance. Business process management (BPM) is also included, based on the assumption BPM is all about handling cases. Each case (for instance a credit approval) is governed by a set of rules. This set is called the *procedure* by which the case is handled (e.g. the credit approval procedure). Actions on that case are triggered by signals, which inform users that one of these rules is (temporarily) violated. When all rules are satisfied (i.e. no violations with respect to that case remain), the case is *closed*. This yields the controlling principle of BPM.

This principle rests solely on rules. Computer software is used to derive the actions, generating the business processs directly from the rules of the business. In comparison: workflow management derives actions from a workflow model, which models the procedure in terms of actions. Workflow models are built by modelers, who transform the rules of the business into actions and place these actions in the appropriate order to establish the desired result.

This new approach has two advantages: the work to make a workflow model can be saved and potential mistakes made by process modelers can be avoided. It sheds a different light on process models, whose role is reduced to documenting and explaining a process to human participants in the process. Process models no longer serve as a 'recipe for action', as is the case in workflow management.

The following section discusses an example, that illustrates this process control principle.

2.3 An example

Consider the handling of permit applications by a procedure based solely on rules. Each permit application is seen as a case to be handled, using the principle of rule based process control (section 2.2). First the business rules are given that define the situation. We subsequently discuss a scenario of the demonstration that is given with the generated software and the data model (also generated from the rules) on which that application is based.

The example consists of the following business rules.

1. An application for a permit is accepted only from individuals whose identity is authenticated.
2. Applications for permits are treated by authorized personnel only.
3. Every application leads to a decision.
4. An application for a particular product (the type of permit) leads to a decision about that same product.
5. Employees get assigned to particular areas. This means that an assigned employee treats request from these areas only.

First, we establish that each statement is indeed a business rule by showing that each rule is falsifiable. For that purpose, one example is given of a violation for each rule:

1. An application for a permit from an individual whose identity is unknown.
2. An application that is treated by unauthorized personnel.
3. A permit application without a decision.
4. An application for a building permit that leads to a decision about a hunting permit.
5. An employee assigned to Soho who treats a request from East End.

As for the IT consequences, notice that violation 4 can be prevented by a computer, by consistently choosing the type of the permit as the type of the corresponding decision. This causes rule 4 to be free of violations all the time. Rule 3 may be violated for some time, but in the end a decision must be made. So the work is assigned to an employee who makes that decision. Rules 1, 2, and 5 are enforced by blocking any transaction that violates the rules. Thus, a system emerges that complies with all five rules.

An application that controls this process has been built by representing the rules in the language ADL, a syntactically sugared version of relation algebra [22]. The functional specification was generated by software that translates a set of relational algebra rules into a conceptual model, a data model (see figure 2.1), and a catalog of services with their services defined formally. This specification defines a software system that maintains all rules mentioned above. The specification guarantees that rules 1, 2, 4, and 5 can never be violated and rule 3 yields a signal as long as a decision on the application is pending. A compliant implementation was obtained by building a prototype generator that produces a database application according to the given specification.

A dialog[1] between user and computer might proceed as follows:

1. an employee creates a new application for ‘Joosten’, who wants to have a ‘building permit’.
2. the system returns an error message for violating rule 1. This means that an application for a permit from an individual whose identity is unknown is not accepted.
3. the employee remembers he should have checked the identity of the applicant. He asks for identification and enters the applicant’s passport number into the system.
4. Now the employee can register the new application. As far as this employee is concerned, he is done with the application.
5. Now an employee must be allocated for making the decision. If an employee is chosen in violation of rules 2 or 5, that transaction is blocked.
6. The employee who makes the decision registers it in the system. The fact that this decision is about a building permit is copied (by the computer) from the application, without any interference from the employee.

[1] This is an actual scenario, that is used in demonstrations of information systems generated by business rules.

Notice that this system may be criticized for picking an employee ‘by hand’. This behavior is a logical consequence of *not* having the rules in place for picking employees. One could argue that the system is incomplete, because there are too few rules. Adding appropriate rules will yield a process in which employees are assigned automatically. This illustrates how a limited (even partial) set of rules can be used already to generate process control. In practice, this means that process control may be implemented incrementally.

Automated data analysis also yields a class diagram, generated from the business rules. The result is shown in figure 2.1. The generator also produces a formal specification of the software services required to maintain all rules. This service catalog is not reproduced here for the sake of brevity.

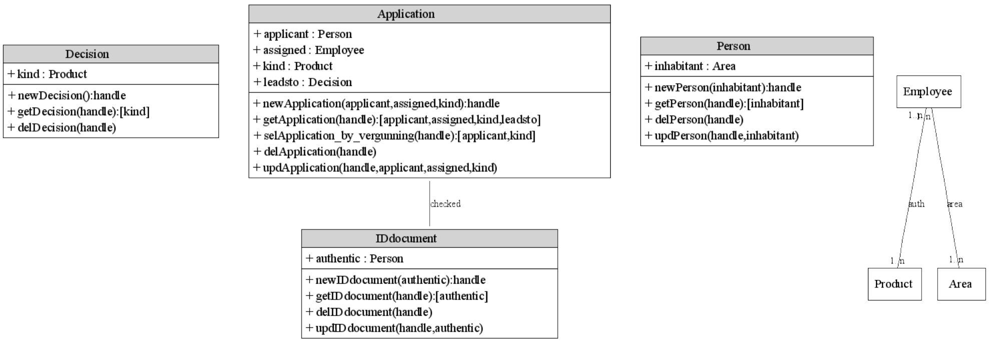

Decision
+ kind : Product
+ newDecision():handle
+ getDecision(handle):[kind]
+ delDecision(handle)
Application
+ applicant : Person
+ assigned : Employee
+ kind : Product
+ leadsto : Decision
+ newApplication(applicant,assigned,kind):handle
+ getApplication(handle):[applicant,assigned,kind,leadsto]
+ selApplication_by_vergunning(handle):[applicant,kind]
+ delApplication(handle)
+ updApplication(handle,applicant,assigned,kind)
checked
IDdocument
+ authentic : Person
+ newIDdocument(authentic):handle
+ getIDdocument(handle):[authentic]
+ delIDdocument(handle)
+ updIDdocument(handle,authentic)
Person
+ inhabitant : Area
+ newPerson(inhabitant):handle
+ getPerson(handle):[inhabitant]
+ delPerson(handle)
+ updPerson(handle,inhabitant)
Employee
1..n
n
auth
area
1..n
1..n
Product
Area

Figure 2.1: Data structure of permit applications

2.4 Control Principle

After discussing rule based design (section 2.2) and illustrating it with an example (section 2.3), let us discuss the consequences of rule based control of business processes in some more detail.

The principle of rule based BPM is that any violation of a business rule may be used to trigger actions. This principle implements Shewhart’s Plan-Do-Check-Act cycle (often attributed to Deming) [31]. Figure 2.2 illustrates the principle. Assume the existence of an electronic infrastructure that contains data collections, functional components, user interface components and whatever is necessary to support the work. An adapter observes the business by drawing information from any available source (e.g. a data warehouse, interaction with users, or interaction with information systems). The observations are fed to a detector, which checks them against business rules in a rule base. If rules are found to be violated, the detector signals a process engine. The process engine

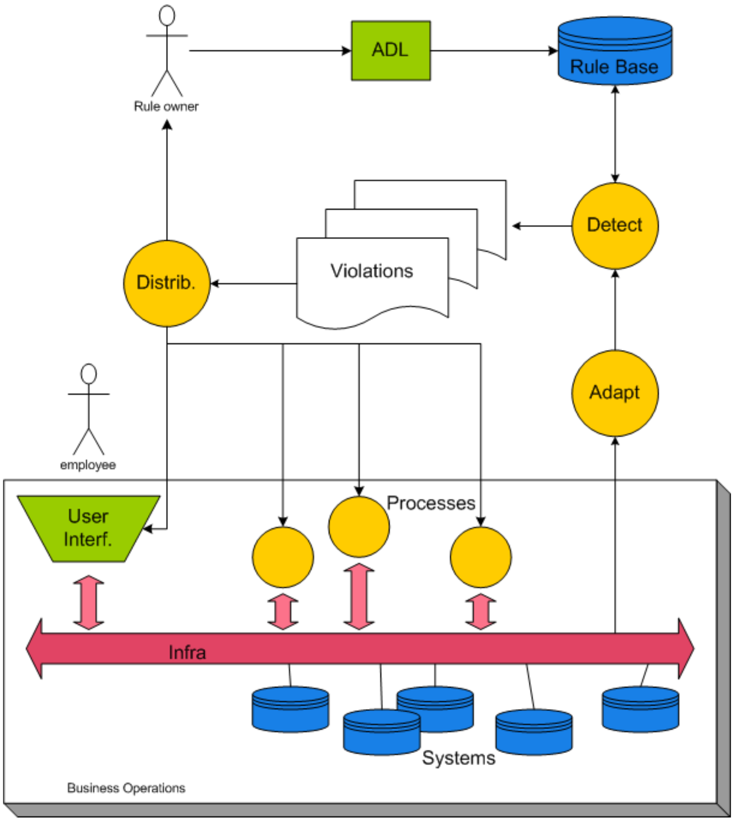


Figure 2.2: Principle of rule based process management

distributes work to people and computers, who take appropriate actions. These actions can cause new signals, causing subsequent actions, and so on until the process is completed.

The system as a whole cycles repeatedly through the phases shown in figure 2.3. The detector detects when rules are violated and signals this by analyzing events as they occur. The logic to detect violations dynamically is derived from the business rules. This results in systematic and perpetual monitoring of business rules.

Signals that are sent to specific actors (either automated or human) will trigger actions. These actions can cause other rules to produce signals by which other actors are triggered. So the actual order of events is determined dynamically.

2.5 Case Study: Order process

This section provides evidence in support of the Ampersand approach. Proof that a business process can be represented in relation algebra is

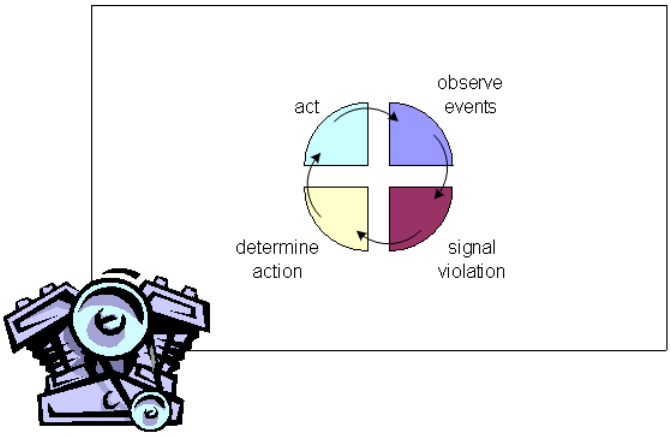


Figure 2.3: Engine cycle for a rule based process engine

given by showing the actual code in relation algebra. Proof that the corresponding application exists is available on `http://86.88.190.85/orderprocess.php`, where it can be tried out by the reader. This section supplements that by a screenshot (figure 2.4) and a dialog scenario, for those readers not in a position to inspect the application for themselves.

The case study defines an order process between providers and clients. It illustrates a multi-step process involving orders, deliveries, and invoices. First an enumeration of all business rules is given. That constitutes the order process. Then the formal representation is presented in relation algebra. Subsequently, a scenario is presented in which one order is processed from beginning to the end.

The process is defined by the following ten business rules:

1. Ultimately all orders issued to a provider must be accepted by that very provider. The provider will be signalled of orders waiting to be accepted.
2. A provider can accept only orders issued to himself.
3. No delivery is made to a client without an order accepted by the provider.
4. Ultimately, each order accepted must be delivered by the provider who has accepted that order. The provider will be signalled of orders waiting to be delivered.
5. All deliveries are made to the client who ordered the delivery.
6. A client accepts invoices for delivered orders only.
7. There must be a delivery for every invoice sent.

8. For each delivery made, the provider must send an invoice. The provider will be signalled when invoices can be sent.
9. Accept payments only for invoices sent
10. Each invoice sent to a client must be paid. Both the client and the provider will be signalled for payments due.

These rules can be used by a computer after translating them to relation algebra. These very rules are used in chapter **??** to introduce the formal notations.

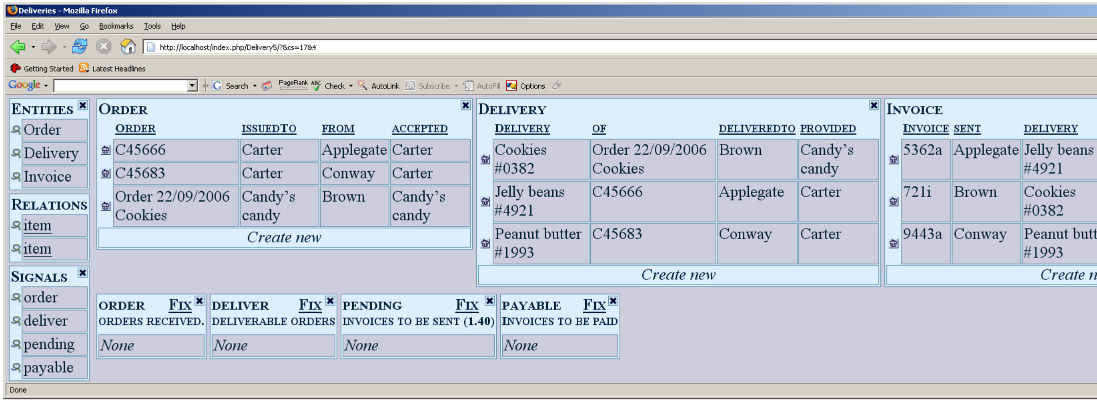
The process, which is defined by these rules is best illustrated by the following (typical) scenario:

1. The client creates a new order. This generates a signal for the provider. The signal is a logical consequence of rule 1, which is (temporarily) violated by an order that has been issued by a client, but is not accepted (yet).
2. The provider must accept the order, which generates a signal to deliver the order. The signal is a logical consequence of rule 4, which is (temporarily) violated by an order that has been accepted, but is not (yet) delivered.
3. The provider must deliver the order, generating a signal to send out an invoice. This is a logical consequence of rule 8, which is (temporarily) violated by a delivery made, without a corresponding invoice.
4. The provider sends an invoice, which creates a signal to the client that an invoice is due for payment. The signal is a logical consequence of rule 10, which is (temporarily) violated by an invoice that has not been paid (yet).
5. The client pays the invoice, and no more signals are raised. This means that the order has been processed completely and the case is closed.

The prototype generator was used to generate a service layer for rule based process control. It derives the temporal logic as shown above. Together with a generic interface, the service layer yields a fully functional application as shown in figure 2.4. The following scenario gives the flavour for readers who cannot try this for themselves. Initially all four signals (see figure 2.4) are void.

1. The client creates a new order for ‘Gates Inc.’. He does so by filling out the fields ‘addressedTo’ and ‘from’[2]. The effect of this

[2] These fields correspond with the relations used in the formalization.


Deliveries - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/index.php/Delivery5/?&cs=17&4
Getting Started Latest Headlines
Google Search PageRank Check AutoLink Subscribe AutoFill Options
Done

ENTITIES
- Order
- Delivery
- Invoice

RELATIONS
- item
- item

SIGNALS
- order
- deliver
- pending
- payable

ORDER

ORDER	ISSUEDTO	FROM	ACCEPTED
C45666	Carter	Applegate	Carter
C45683	Carter	Conway	Carter
Order 22/09/2006 Cookies	Candy's candy	Brown	Candy's candy

Create new

DELIVERY

DELIVERY	OF	DELIVEREDTO	PROVIDED
Cookies #0382	Order 22/09/2006 Cookies	Brown	Candy's candy
Jelly beans #4921	C45666	Applegate	Carter
Peanut butter #1993	C45683	Conway	Carter

Create new

INVOICE

INVOICE	SENT	DELIVERY
5362a	Applegate	Jelly beans #4921
721i	Brown	Cookies #0382
9443a	Conway	Peanut butt #1993

Create n

ORDER FIX
ORDERS RECEIVED.
None

DELIVER FIX
DELIVERABLE ORDERS
None

PENDING FIX
INVOICES TO BE SENT (1.40)
None

PAYABLE FIX
INVOICES TO BE PAID
None

Figure 2.4: Service layer interface for order process example

action is that a new order number is issued and a signal is raised for 'Gates Inc.' to accept this order.

2. Provider 'Gates Inc.' can accept the order by filling in the name 'Gates Inc.' in field 'accepted'. The effect of this action is that a signal is given to 'Gates Inc.' for delivering the order.

3. Provider 'Gates Inc.' will see a delivery record, which is partially filled in: it contains only the order number. The moment the provider tries to fill out the empty fields in this form, the computer fills out all fields in the delivery form which are uniquely determined. That is the name of the client and the name of the provider. For a user, this is natural behaviour, because the computer 'already knows' these facts. In the meantime, a signal is raised saying that an invoice can be sent.

4. When the provider tries to fill out the invoice, again the computer will fill out all know (i.e. uniquely determined) facts. This yields a partially filled invoice registration, and a signal to the client that an invoice is due for payment.

5. When the client pays the invoice, the field "paidBy" can be filled in and the case is closed. By this time, all signals are empty again.

In the actual application, different orders of events can be tried. For example, a delivery can be made before an order is accepted, or the client can be filled in long after the delivery has been made. The application will accept any order of events until it leads to violations.

2.6 Consequences

Controlling business processes directly by means of business rules has consequences for designers, who will encounter a simplified design process. From a designer's perspective, the design process is depicted in figure 2.5. The effort of design focuses on requirements engineering.

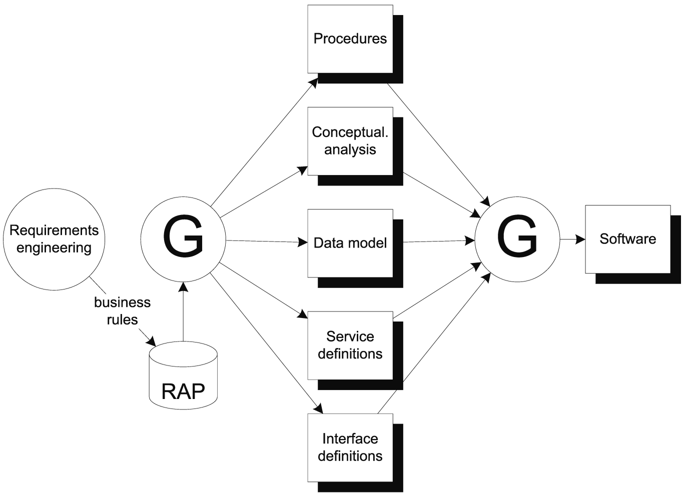


Figure 2.5: Design process for rule based process management

The main task of a designer is to collect rules to be maintained. From that point onwards, a generator (G) produces various design artifacts, such as data models, process models etc. These design artifacts can then be fed into a information system development environment (the other generator, G). That produces the actual system. Alternatively, the design can be built in the conventional way as a database application. The rule base (RAP, currently under development) will help the designer by storing, managing and checking rules, to generate specifications, analyze rule violations, and validate the design. For that purpose, the designer must formalize each business rule (in ADL). He must also describe each rule in natural language for the purpose of communication to users, patrons and other stakeholders.

From the perspective of an organization, the design process looks like figure 2.6. At the focus of attention is the dialogue between a problem owner and a designer. The former decides which requirements he wants and the latter makes sure they are captured accurately and completely. The designer helps the problem owner to make requirements explicit. Ownership of the requirements remains in the business. The designer can tell with the help of his tools whether the requirements are sufficiently complete and concrete to make a buildable specification. The designer sticks to the principle of one-requirement-one-rule, enabling him to explain the correspondence of the specification to the business.

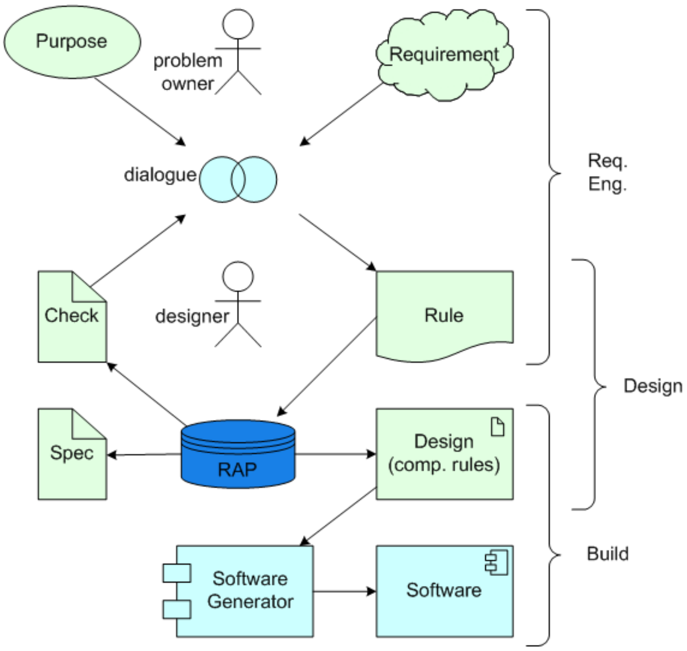


Figure 2.6: Design process for rule based process management

Part II

Theory

Chapter 3

Relational Algebra

3.1 Introduction

The Business Rule Manifesto proclaims as its central thesis or "mantra": Rules are build on facts, and facts are build on terms. But what are the terms, facts and rules that the Manifesto talks about? How must we understand them, how are they interconnected, and why can we rely on just these notions to build a rigorous framework for business rules? The answer is found in mathematics: Relational Algebra is the formal method to define, implement and work with well-formed business rules.

Relation algebras as a distinct branch in mathematics came about in the nineteenth century by the efforts of mathematicians such as De Morgan, Peirce, and Schröder [8, 26, 30]. Their results have been studied, expanded and enhanced throughout the twentieth century. This has resulted in a clear and well-understood formalism that meets our needs in describing business rules.

Relation algebras provide todays professionals with a way of thinking about rules. More important, it outlines a way of working: what to do to describe and manipulate relations effectively, quickly, and without mistakes. Proficiency in these skills will allow you to analyze business requirements, to conduct conceptual analyses, to search for and create new rules, to test the outcome, and finally to produce exact specifications.

This chapter explains the basics of Set Theory and Relational Algebra, basics that we need to identify, express, and manipulate with rules in the business environment. Examples will illustrate the theory, in order to give a better understanding of the ideas and formalisms. And do not worry, we only need a basic and almost intuitive understanding, no advanced mathematics is involved.

After reading this chapter, you are expected to be familiar with these notions because all business rules approaches, and the Ampersand approach is no exception, are built upon these fundaments. There are many excellent articles and introductions for those who want to study the the mathematical field of Relational Algebra. The "Background Material" of Jipsen, Brink, and Schmidt is recommended as a reference for relation algebras in general. We will specifically use heterogeneous relation algebras. Roger Maddux's book is an excellent resource for a further study of relation algebras.

This chapter is outlined as follows. We start with the basic notion: sets. Then we discuss concepts and relations, a.o. the Universal relation and the identity relation. The concepts and relations that are relevant in a business context, are brought together in a Conceptual Model. We then proceed to operations on relations, including negation, addition, inversion and composition and also some other operations. Next, homogeneous relations are discussed, being a special kind of relations with special properties. Then we introduce the cardinality properties of relations, considered to be simple but very important rules for relations. The chapter ends with a discussion of limitations of Relational Algebra, and it briefly introduces Proposition and Predicate Logic, as an alternative way to express the terms and facts of the business environment at a high level of detail and precision. The next chapter will be entirely devoted to rules that can be expressed in Relational Algebra.

3.2 Sets

3.2.1 Set

A set in mathematics is not much different from what it is in natural language: it is something that contains 'elements'. Synonyms for element are 'occurrence', 'instance', 'atom' or 'member' and as far as this book is concerned, these terms all have identical meaning and we will use them interchangeably. Figure B1 depicts a so-called Venn diagram of a set containing some numbers. The numbers in this example correspond to the ages of the four nieces and nephews of the author.

Figure B1. Venn diagram of a set B

By convention, we will write the name of a set with a capital: Book, Age, or Team. In mathematics, the special symbol $\in$ is used to denote that an element is a member of the set.

So in the example we have $2 \in Age$, $7 \in Age$, $5 \in Age$, and $3 \in Age$.

Notice that in general, there is no specific order or sequence among the elements of a set. In order to denote the whole contents of the set of *bracket notation*

ages, it is customary to use a *bracket notation*. Mathematicians often use curly brackets, thus: $Age = \{ 7, 3, 5, 2 \}$. Occasionally we use square brackets: $Age = [5; 3; 2; 7]$. We say “2 is an instance of Age”, or “Bridget Jones’s Diary is an element of Book”, or “Ann is a member of the Developmentteam”.

All the sets that we will be discussing in this book are finite, i.e. they contain a finite number of instances. We will not need it, but mathematics can also handle infinite sets such as the natural numbers { 1, 2, 3, 4, 5, 6, ... }. Or the set of all fractions, also known as the ‘rational numbers’; this set is somewhat harder to write down.

empty set

Mathematics also provides something called the *empty set*, denoted (o slash), which is a set even though it contains no member at all.

By definition, two sets are equal if they both contain exactly the same elements. This is why there is only one empty set, which is unique precisely because it is equal to any other empty set that might exists.

For instance, the set of ‘ages of nieces and nephews of the author’ is equal to the set of prime numbers smaller that 10. To rephrase this: a set in mathematics is not characterised by something like ‘meaning’ or ‘explanation’, but it is characterised only by what it contains.

Two sets are considered unequal if there is at least one element that is contained in one of the sets but not in the other. An element is a member of a set, or it is not. Partial membership or halfway in/halfway out, is impossible. Whenever an instance is added to a set or removed, the result will be a new set, and it is unequal to the set as it was before the addition or removal.

3.2.2 Realworld objects and stored instances

Obviously, we cannot capture an object such as a real customer, a particular car, or a given assignment in the corporate data store. We are only able to record information about that customer, the car, the assignment. Plus, we can make sure that the recorded information is truly and uniquely linked to the right customer, car or assignment. This link transgresses the boundary of the information system: it establishes a link between something in the real world and some data in the computer. For this reason, we cannot, in general, set the computer to check whether the data on record does indeed refer to the correct realworld object.

The common approach to safeguard this link between realworld objects and computer records is to use identifier or key data.

Figure B2. Linking realworld objects and stored information

Realworld objects are equipped with identifier data. That data may come naturally (e.g. date and time of birth) or it may require some extra work. A car is equipped with a licence plate, and we can record the licence plate number. A person is equipped with a Fiscal Number, Social Security Number or something like that. These identifiers are then used to link computer-held information with the realworld objects.

Once the realworld objects have proper identifying data, this data is used in computer programs to represent their corresponding objects. An identifier is a (set of) data that enables one to recognize and retrieve a designated instance amidst the set of all recorded instances, at any time. Every real world object or term in the business environment that we want to record information about is provided with such a unique key or identifier, and we must make sure that different objects and terms will always have different key identifiers. Figure B2 illustrates the idea.

Notice that identification is more than just being able to distinguish the instances. For instance: in a flock of sheep, it is possible to tell each individual animal apart, but not many people are able to recognize (identify) the same animal on two different days.

3.2.3 Entity integrity

A strict requirement for mathematical sets is that each element in the set is unique. This is sometimes called the 'discrimination requirement' because there must be a way to discriminate between elements:

- each element must differ from every other element in the set; and therefore
- an element cannot be represented twice in the set. It is meaningless to insert an element into a set if it is already present.

The requirement may appear to be obvious, but it requires some attention. For instance, a soccer team will have eleven players, but the set of their ages may contain nine members, instead of eleven. And special rules exist that must ensure that players of a team can be discriminated by their shirt numbers, i.e. the number on each shirt must differ from the numbers on every other shirt. The referee has to see to it that these rules are abided by.

The demand that each instance of a set is unique is commonly referred to as entity integrity or sometimes key integrity. Although the name originates from the database world, the demand derives from first principles in Set Theory.

A computer cannot function correctly if entity integrity does not hold: how can it deal with elements that it cannot tell apart? So entity integrity is a structural demand: this requirement cannot be violated in a well-defined Rational Algebra. So from now on, we will expect that entity integrity will hold, always and for all kinds of sets that we will be discussing.

3.2.4 Subset

One set is contained in another if every element of the former is also an element of the latter set. This situation is so common that it has its own name: subset. For instance, figure B3 illustrates the set of prime numbers smaller than 10 which is a subset of the natural numbers smaller than 10. It is also a subset of all prime numbers. In turn, both the natural numbers smaller than 10 and the prime numbers, are a subset of the rational numbers (which include fractions).

Figure B3. Venn diagram of two sets, one being a subset of the other

Mathematics uses the inclusion symbol $\subset$ to denote that a set A is contained a set B, thus $A \subset B$. Because this symbol is not easy to type we will often replace it with the symbol $\vdash$, which means exactly the same.

In general, the enveloping set B will be larger than A (i.e. it contains more elements), but it is allowed that A and B are exacty equal. If we want to draw attention to the fact that equality is permitted, we sometimes write $A \subseteq B$. But if equality is not permitted, we must write two assertions: $A \subset B$ and not $(A = B)$.

set inclusion

The following three properties of *set inclusion* are easily verified:

- *reflexive* : $A \subset A$ (every set is a subset of itself),
- *asymmetric* : if $A \subset B$ then not $B \subset A$ (unless of course $A = B$),
- *transitive* : if $A \subset B$ and $B \subset C$ then $A \subset C$ (a subset-of-a-subset is a subset)

3.2.5 Operations on sets

A number of standard operations are available that produce new sets from existing ones. Each operation unambiguously defines which elements will belong to the new set, and which ones are excluded.

intersection

- *intersection* : F ∪ G is the set that contains the elements that are contained in F as well as in G *union*

- *union* : F ∪ G is the set that contains all elements that are contained either in F or in G

difference

- *difference* : F / G is the set that contains the elements of F that are not contained in G.

Figure B4. Intersection, Union and Difference

It is an easy exercise to verify the following assertions:

$$F \cup G = (F/G) \cup (F \cap G) \cup (G/F)$$

and

$$\emptyset = (F/G) \cap (F \cap G)$$

complement

A further important operation on a set is called *complement.* The complement operation is used to discuss which elements are *not* in a relation. The operator can be applied to a set A only if A is clearly the subset of some other, larger set. For now, let us denote the larger, enveloping set by V. The new set that is created by the complement operation is the 'remainder': the set of all instances in V that are not in A. For this reason, some authors refer to this operation as negation. So, the complement of set A is the difference V/A. This can also be written as

A (A with overstrike), or

AC (the C meaning complement), or

-A (that is: minus A).

Obviously, if there is any ambiguity what the enveloping set V might be, then the complement set will also be in doubt. To illustrate the point: the complement of all students in the class is probably the set of all students not in the class, which assumes that the entire student body is being referred to. But perhaps the speaker meant the people in the classroom that are not students, such as the teacher and perhaps a visitor? If necessary, the enveloping set is denoted by the capital letter V, often with its lefthand side rendered as a double line (depending on screen and printer capabilities).

idempotent

Note that complement is a so-called *idempotent* operator: applying the operator twice to a set will reproduce the original.

3.2.6 Some laws for operations on sets

Operations on sets follow some simple laws. For instance, when determining the intersection of multiple sets, it does not make any difference in which order the intersections are calculated. Laws such as these are useful, because they allow to manipulate with entire sets instead having to consider the separate instances. To put it more formally: intersection is

- associative : $(A \cap B\ ) \cap C\ =\ A \cap\ (\ B \cap C\ )$,
- commutative : $A \cap B\ =\ B \cap A$.

Union, like intersection, is also an associative and commutative operation. And when a formula combines union and intersection, distributivity applies:

- $A \cap (B \cup C) = (A \cup B) \cap (A \cup C)$, or: the union with an intersection equals the intersection of the unions, and likewise
- $A \cup (B \cap C) = (A \cap B) \cup (A \cap C)$, or: intersection with a union equals the union of the intersections.

Finally, the so-called 'de Morgan' laws apply for complements of intersections or unions: the complement of an intersection equals the union of the complements. In formulae:

- $(A \cap B)C = AC \cup BC$,
- $(A \cup B)C = AC \cap BC$.

3.3 Concepts and relations

The Business Rules Manifesto refers to terms, facts and rules, all existing in the business environment. But, you cannot store a real something in a computer. So instead, we use surrogates: data about the terms and facts and rules in the real world. And we can store the data.

This section introduces 'concept' and 'relation' as fundamental notions in our models to stand for the realworld 'terms' and 'facts', respectively. Rules as a fundamental notion in both the real world and in our models, will be discussed in the next chapter.

3.3.1 Concept

A concept is

- a notion that is relevant to the business environment, but it is also
- a set of elements or instances that have similar meaning, similar properties, similar behavior, and similar connections with other elements or concepts.

In an organizational context, some concepts to be recognized may be for instance *Customer*, *Order*, *Account* and *Delivery*. Mister John Doe and mrs Ahmed may be customers, but we cannot store real people in an information system. So, the former definition of concept may be well applicable in the business environment, but we cannot really use it, and we will only use the latter definition: concept as a set of elements with shared semantics.

It must be stressed that a concept is fundamentally different from the notion of set in the mathematical sense. Whereas a set is characterized only by the elements it contains, a concept is also characterized by its semantics: its meaning, properties, behaviour. The implication being that two concepts may both be empty, but still are considered to be different. A *Customer* is not an *Order*, even if there are neither customers nor orders on record. This characterization is so natural that many concepts are only described by way of a definition; the user is then expected to be able to decide which realworld objects meet the definition and should be (represented as) an instance in the corresponding set.

The number of instances that an enterprise has on record for a particular concept may vary over time from zero or one, to thousands or even millions. Rarely a concept will hold only a single, unique instance that does not change over time. This we call a singleton concept. An example may be *Country*, when trying to model all the relevant regions and parts of that country, or *Company* when analyzing the hierarchic structure of the entire corporation. True singleton concepts are quite rare however, and a designer must have a sound reason to include such an exceptional concept in a model.

3.3.2 Intent and extent

Whereas sets are identified by their content only, concepts (and relations) have a definition as well as a (current) content. This is of major importance in the business environment. A proper definition is a tool that we can use to ascertain whether an arbitrary ‘thing’, to be encountered in the real world or, more restrictive, in the business environment, does or does not satisfy the definition and hence whether it should or should not be included in the current content of the set.

We define

intent

- the *intent* or intension is the definition of a concept (or relation). It describes and explains its meaning and semantics in the business environment,

extent
extension
population

- the *extent*, *extension* or *population* is the current contents, the set of all known instances in the business environment that currently meet the definition of the concept.

The intent of a concept is described by way of a sound definition that does not change overnight, regardless whether there are millions, hundreds, tens or even no instances on record for the concept. Definitions are generally stable, and although they can and do change over time, such changes are not very frequent.

Whereas intensions remain relatively unchanged over time, extension can and will change constantly. The extent is the time-varying part: at any given moment, there is a specific content, and that content can and often will be different from the content one minute before or after. New instances may be added (new customers or new orders) or may be deleted (customers depart, orders are fulfilled) and certain properties may change: a new customer is associated with no order, but after a while, hundreds of orders may be on record for that customer. Thus, the concept *Customer* may have an extension enumerating hunderds of people. Later, we will also use the word 'extension' for relations. We will say that a relation such as Customer-places-Order has an extension, for instance counting exactly 4183 tuples last friday morning.

As the extent of a concept is a set, *entity integrity* must apply.

Thus, every instance in the extension must be unique, must differ from every other instance in the same extension. A concept 'City' for instance may contain only one element called 'Mercedes', and not more. This obviously will cause problems if there are several towns called 'Mercedes', which is the case in several countries. On the other hand, it is permitted to have elements named 'Mercedes' in different concepts, say *Customer* or Make-of-Car.

In general, concept names are nouns such as '*Customer*' or 'Order', sometimes qualified to better express its meaning, e.g. 'Old-timer Car' or '*Priority Customer*'. By convention, concept names will begin with an uppercase letter. The meaning of the concept that a name refers to must be carefully explained and defined, for example:

– *Customer*: a person who or organisation that, in the past three years, has placed at least one order and has paid for it,

3.3.3 Validation

A definition tells us how to decide what ought to be in the set. The current content is what is actually in the set. You must realize that these two are fundamentally different.

validation

Indeed, there may be a discrepancy between the two: the current content of the set is not up to date. Often, this problem cannot be detected by a computer, e.g. a customer has posted a complaint but the mail has not yet been received. The activity to check whether the computer-stored information is a valid representation of the situation in the real business world is called *validation*. The check may concern the (current) contents of one or more extensions, but validation may also be used in a broader sense: whether an entire design captures all the relevant features of a business environment.

Validation in general requires a human effort, as only humans have a grasp of reality, and only in rare occasions can the computer signal validation problems.

verification

Do not confuse validation with *verification*, which is checking the stored information and signaling possible problems with the data. Such checking can be done automatically, by computers, without referring to the real world. Clearly, problems detected by verification may sometimes be caused by invalid data, but there may also be errors in the data store or in the programmed checks.

3.3.4 Naming confusion

We aim to use the notion of concept consistently throughout this book. Other authors coin other names and notions that may look rather similar in meaning and usage, but often there are subtle differences. Some important ones for us are:

– the SBVR standard (more on that later) defines term as a 'verbal designation of a general concept in a specific subject field'. The 'general concept' that this alludes to, is described as a 'unit of knowledge created by a unique combination of characteristics',

– UML programming works with 'classes', a notion that is similar but not equal to our notion of concept, and

– database modelling, where the name 'entity' is used for a notion also similar but unequal to concept.

The many different names and definitions may be a cause of confusion. Illustrative of this confusion is the SBVR statement that “a concept type is an object type that specializes the concept ‘concept’, whereas a concept is related to a concept type by being an instance of the concept type.” So according to the SBVR naming convention, a person John Doe should be called a concept, which is an instance of the concept type *Customer*. We prefer to call *Customer* the concept, and refer to the person John Doe (or rather: to the key data that represent him) as an instance thereof.

3.3.5 Cartesian Product

So far, the operations that we discussed always operated on the existing elements of then sets involved. We now do something new: we combine two existing sets -or concepts- and produce a new set with new elements, by . The Cartesian Product of two sets A and B is a set that consists of all the elements that combine one element from the first set A and one element from the second set B. We will refer to these new elements as tuples.

Figure. Set of 52 playing cards.

Some examples:

- a deck of common playing cards. This is the Cartesian Product of (the set of) four suits {clubs, diamonds, spades, hearts} and (the set of) 13 ranks {ace, king, queen, jack, ten, 9, 8, 7, 6, 5, 4, 3, 2}. Because every combination is possible, this produces a total of $4 \times 13 = 52$ playing cards (as depicted in the figure),
- a two-dimensionsal spreadsheet: every combination of column and row is possible,
- the set of all possible couples that a tennisclub may put forward for the mixed double competition, as listed in the table.

A standard representation of the contents of a a Cartesian Product is a matrix. The elements of the two sets that define the matrix are listed alongside the two axes, but keep in mind that the axes themselves are not part of the Cartesian Product. Notice how the number of elements in the Cartesian Product equals the product of the number of elements in the two defining sets; this explains (in part) why it is called a product.

The standard way to denote a Cartesian Product of a set A and a set B is: $A \times B$. Or we sometimes write $V_{[A,B]}$ or V for short, with the V’s lefthand side rendered as a double line, if possible. And to be very exact: this is not the same as $B \times A$. While the tuple (diamonds, ace)

Women	Men	Marek	John	Rob	Toine
Sophie		(Sophie, Marek)	(Sophie, John)	(Sophie, Rob)	(Sophie, T
Aisha		(Aisha, Marek)	(Aisha, John)	(Aisha, Rob)	(Aisha, To
Jenny		(Jenny, Marek)	(Jenny, John)	(Jenny, Rob)	(Jenny, To
Nellie		(Nellie, Marek)	(Nellie, John)	(Nellie, Rob)	(Nellie, To

Table 3.1: All possible couples for a mixed double competition

looks very similar to the tuple (ace, diamonds), there is a big difference as the two elements come up in inverse order. This will be important later on when we will discuss the inverse of a relation.

Some standard terminology when discussing Cartesian Products:

source
domain

- the left-hand set of the product is called *source* or *domain*,

target

- the right-hand set of the product is called *target*, or co-domain, or sometimes range or image,

tuple

- each single element of the product is called a *tuple* or pair.

3.3.6 Relation

Having defined the Cartesian Product, we are now ready to study relations. A relation is defined as

- any kind of association that is relevant in the business environment about the notions and objects in that environment, but it is also
- a subset of the Cartesian Product of two concepts, where all tuples in the subset have a similar meaning, similar properties, and similar behavior.

Just like we saw for concepts, the former notion is of less importance to us because it is oriented towards real-world objects that cannot be dealt with by a computer. We will only use the latter definition which is fundamental for all the work to follow.

By definition, a tuple in a relation refers to one instance of the source and one instance of the target, and the tuple is fully identified by exactly these two instances. Obviously, two tuples are the same if both their source- and target-instance are the same. By implication, there is no need to have special identifiers to distinguish the tuples in a relation: the identification of source- and target instance suffices.

Like with concepts, a relation must have a precise definition. The definition serves to determine which realworld objects should be represented in the extension of the relation, and which ought to be absent. The definition captures the meaning (intension) of a relation, for example:

- places : a *Customer* places or has placed an *Order*, and it has not yet been fully delivered and paid for,
- delivered : a *Delivery* has been delivered for *Order*, and it has not yet been paid for.

Notice that the definitions indicate not only the concepts that are related, but also under what conditions they are actually related: the instances may exist but the relation between them may have ceased to exist, or has not yet come into effect at the time of writing.

universal relation

Also bear in mind that in general, a relation is not the entire set of all possible combinations of elements, but only a subset, a selection of certain pairs. To emphasize the difference, the Cartesian Product of A and B is sometimes called the *universal relation* of A and B, with universal meaning that it contains all possible tuples that can be produced from (the current contents of) A and B.

If there is any chance of ambiguity, $V_{[A,B]}$ should be written in full to prevent confusion with, say, $V_{[B,C]}$ or $V_{[B,A]}$.

Furthermore, a relation is not just any subset of arbitrary pairs of elements, but it only contains those tuples that meet the definition. They having certain properties in common, a shared meaning, an interpretation that users can readily understand.

For instance, an organization may have many customers, and many orders that are currently being processed. Some useful relations may then be *Customer places Order*, or *Customer receives Order*, or *Customer cancels Order*. The number of instances (tuples) in these relations can vary from zero (there are customers, there are orders, but none of those is being placed by a customer), to millions.

declaration

A relation *declaration* consists of its name, its source and its target, together with its definition, i.e. its meaning or intent. Relation names in general are verbs, possibly qualified. We will generally write the relation names in italics and in lowercase.

signature

The *signature* of a relation is the combination of the relation name, (the name of) the source concept, and (the name of) the target concept: $r_{[A,B]}$ denotes the relation with name r, source A and target B.

We require that in a given context, every relation has a unique signature, i.e. the name, source and target are a unique combination. Often, if sources and targets are not in doubt, the relation name is enough to know about which relation we are talking.

type

Finally, the *type* of a relation is defined as (the name of) the source concept, and (the name of) the target concept.

Obviously, in order to compare the extents of two relations, they need to have the same types. If their types are different, the relations contain tuples created from elements from different source or target sets. Only if two relations have the same type, can their contents be compared.

Although the empty set is unique, we saw that empty concepts are not considered equal. Just so, a relation, even if it contains no tuples, is generally not equal to other relations with no tuples. Only if both relations are of the same type, i.e. both relations are (empty) subsets of the samen Cartesian Product, can we say that their extents are indeed equal.

n-ary relations

We defined a Cartesian Product by creating tuples out of just two sets, which we call the source and target. The literature refers to such relations as binary relations. In theory, one can define relations of three or even more sets or concepts, and such relations are called ternary or even *n-ary relations.* However, we will not study such constructs, we restrict ourselves to relations based on exactly two concepts.

Exercise: Think of a way how to define fractions, also known as the 'rational numbers', by way of a relation.

3.3.7 Notations for a relation

When discussing relations, various notations are in use. To denote a relation's signature, we write

$$r : A{\times}B$$

The customary notation to denote content of a relation is

$$r = \{(a, b)|(a, b) \in r\}$$

A more concise notation is sometimes used for a tuple: arb means that tuple $(a, b) \in r$

For a given $a \in A$, we can determine all elements $b \in B$ that are related to a. This subset, which in general may have 0, 1 or any arbitrary number of elements, is called the target of a:

$$tar(a) = \{b \in B|(a, b) \in r\}$$

Likewise, for a given $b \in B$, the subset of elements in A that relate to b is called the source of b:

$$src(b) = \{a \in A|(a, b) \in r\}$$

Notice in the above formulas that the left-hand sides do not indicate about which relation we are talking. If necessary, we can indicate the relation for which the target and source are being considered:

$$trg_{[r,]}(a), \mathrm{and} src_{[r,]}(b)$$

Pay attention, as the phrases ‘source’ and ‘target’ have double meaning. A relation r has both a source and a target concept. But each element of the source has its own target set, and each instance of the target has its source set.

3.3.8 Example relation

An example of a relation might be “the set of mixed couples selected for the upcoming tennis tournament” (this is its intent). Its signature is:

$$\text{doubles-with} : \text{Woman}\times\text{Man}$$

Using the tabular representation from table 1, we may depict this relation and also indicate its current content, i.e. the selected couples, as follows:

doubles-with

Women	Men	Marek	John	Rob	Toine	Raúl
Sophie						yes
Aisha		yes				
Jenny						
Nellie					yes	

Table 3.2: Selected couples for a mixed double competition, in matrix layout

Or we may indicate the tuples one by one: (Aisha, Marek) $\in$ *doubles-with*, and (Nellie, Toine) $\in$ *doubles-with*, and (Sophie, Raúl) $\in$ *doubles-with*. Or we may list the entire population of *doubles-with*, which presently is the set $\{$(Aisha, Marek); (Nellie, Toine); (Sophie, Raúl)$\}$. Still another representation would be the familar tabular layout:

doubles-with

A well known graphical form is the Instance Diagram or Venn diagram. This depicts the instances and their actual relations, i.e. the tuples these instances are involved in. An instance diagram provides very detailed

Woman	*Man*
Sophie	Raúl
Aisha	Marek
Nellie	Toine

Table 3.3: Selected couples for a mixed double competition, in tabular layout

insight into the current state of affairs in the business, which can be very useful sometimes.

Figure B6. A Venn Diagram of the *doubles-with* relation

The above illustrates how relations can be represented in various ways, and more variations can easily be pointed out. A telephone directory relates names to telephone numbers in a tabular layout, a dictionary does the same with words and their meanings, and a story about a bridge game may be illustrated with a diagram showing the hands of each player. The time and effort people take to communicate the contents of relations indicates the importance of suitable representations. In practice, form follows function: depending on circumstances, audience, and the sheer number of tuples, the designer picks a form that best suits the purpose.

3.3.9 Identity relation

identity relation

At this point, we need to introduce one special relation: the so-called *identity relation*, abbreviated to Id or even to $\mathbb{I}$ (for this particular relation yes, an uppercase i). This relation is defined for every set or concept, and it takes that set or concept as its source as well as its target. The contents of this relation is simple: it contains every possible tuple composed of two identical elements. Thus, in a matrix representation such as table 2 above, the identity relation will have a marking exactly on the diagonal, and nowhere else.

Identity relation

Another way to write down this identity relation is as follows:

$$Id_{[Fruit,]} = \{(a,a)|a \in Fruit\}$$

The subscript indicates the concept for which the Identity relation is taken. When the concept is not in doubt, the subscript is usually omitted.

Fruit	*Fruit*	*Apple*	*Orange*	*Pear*	*Berry*	*Grape*
Apple		x				
Orange			x			
Pear				x		
Berry					x	
Grape						x

Table 3.4: Identity relation as special selection of the Cartesian Product with source and target the same set

3.3.10 Referential Integrity

By definition, each tuple in (the extension of) a relation refers to instances of the source and target concepts. Therefore, those two instances must actually exist in the respective extents.

At first glance, this is a trivial demand. However, one must realize that extensions are constantly changing. Therefore, if an instance is deleted from (the current extension of) some concept, the consequence is that all tuples referring to that very instance ought to be deleted also, in all relations that the concept may be involved in, either as a source or as a target.

On the other hand, whenever we want to insert a tuple in a relation, we must assure that the associated source and target instances are present in their respective extensions.

referential integrity

This requirement is known in the context of database engineering as *referential integrity*. Like entity integrity, this is a fundamental property in Relational Algebras which cannot be violated in a well-defined Rational Algebra.

3.3.11 Naming confusion

Again, a word of warning as various authors may use different names for what we call a relation:

- the Business Rule Manifesto refers to relations by the noun ‘fact’,
- the SBVR uses the words ‘fact type’ and ‘fact’ where we use ‘relation’ and ‘tuple’,
- the words reference and relationship (instead of relation) are used by still others.

For some readers, the use of words like ‘source’ and ‘target’ may bring to mind the notion of hyperlinks of the World Wide Web. However, hyperlinks, strictly speaking, are not relations. Although a hyperlink does provide a link from a source (webpage) to a target (another webpage), the reverse is not true in general. For a given webpage, there is no sure way to know all webpages that have a hyperlink to it. Moreover, hyperlinks do not implement the referential integrity demand, resulting in the infamous “error 404” reports.

3.4 Models and Diagrams

3.4.1 Conceptual Model

A Conceptual Model is the exhaustive listing of all concepts and relations that are relevant in a certain (business) setting. Such a listing can be provided in textual form, which will make it dull and incomprehensible, and only a trained engineer or computer programmer will like it. The listing can also be provided in a more attractive way, such as a diagram, a graphical representation of the model, or even by way of a prototype information system for intended users to explore and play with.

For a Conceptual Model to be correct, we require that the signature of each relation shall be unique: for any given source and target, there is at most one relation with a certain name. The same name may be used elsewhere in the model, for relations defined on other sources and/or targets. However, it can become quite confusing for people trying to read and understand a model if different relations have identical names.

Our Conceptual Models are based on Relational Algebra theory. Many organisations use similar, but often slightly different models to capture relevant information. Depending on the underlying modeling theory, such models go by names like Relational Models, UML-models, ORM-models etc. This book will not look into differences and similarities between the various models and theories. We keep our focus on explaining and understanding business rules.

3.4.2 Conceptual Diagram

Figure B7. A Conceptual Diagram

Figure B7 is an example of a Conceptual Diagram, with dots representing the concepts, connected by arcs representing relations. This diagram employs the arrow notation, with the arrowhead pointing from the concept that is source (shaft of the arrow) to the target concept (point of the arrow). Relation names are written next to each arc, so that the unique signature of each relation (name, source and target) is easy to read from the diagram.

In this kind of diagrams, the arrowhead are sometimes placed at the base of the arc (as in the 'crow's feet' notation well known from database modelling). Other notations place it halfway, and still others place it at the end of the arc, as for instance is common in UML diagrams. Thus, various Conceptual Diagrams may look different but still may represent same Conceptual Model.

These types of diagrams are easy to understand as long as the number of concepts and relations is moderate. If the numbers go up, say more than 20 concepts, then the sheer size of the diagram makes it incomprehensible for most people, and again, only the trained specialists will like to work with such diagrams.

Remark that this particular Conceptual Diagram does not show the customers or orders that are presently on record, nor does it show what orders are related to which customer. In general, abstracting away from the contents enables you to oversee the structure of many relations at the same time.

3.4.3 Instance Diagram

It is possible to also show (some of) the contents of the concepts.Such a diagram is called an Instance diagram as shown in figure 8

Figure B8. Example of an Instance Diagram

As an example, consider a small organisation that deals with customers, orders, invoices and deliveries. Figure B8 is an example of an instance diagram that corresponds to the Conceptual Diagram in figure B7. The instance diagram provides a level of detail that is lacking in the Conceptual diagram: the same four concepts and four relations are shown, but also shows instances of those concepts and relations. Already at this small scale, the diagram becomes unreadable due to the number of items (eleven concept instances and xxx relation instances).

In general, instance diagrams are not very comprehensible due to the sheer amount of detail. Instance diagrams are mostly used only with small subsets, to illustrate a certain finesse of argument or to highlight some intricacy.

3.5 Operations

Relational operations provide us with numerous ways to produce new relations from existing ones. Learning to use operations will demonstrate to you the use and the expressive power of relations.

3.5.1 Set operations applied to relations

By definition, relations are subsets of a Cartesian Product. Therefore, common set operations may be applied to relations, but only if the relations share the same type: they are defined on the same source X and target Y. If so, then the ordinary set operators can be applied to produce a new subset of the Cartesian Product, i.e. a new relation.

Beware that if two relations are not defined on the same Cartesian Product, then the two relations are disjunct and the set operations will not produce interesting results.

- *intersection* : $r \cap s$ is the set that contains the elements that are contained in relation r as well as in s, or $r \cap s = \{(x, y)|(x, y) \in r\text{and}(x, y) \in s\}$
- *union* : $r \cup s$ is the set that contains all elements that are contained either in relation r or in s, or $r \cup s = \{(x, y)|(x, y) \in r\text{or}(x, y) \in s\}$
- *difference* : r/s is the set that contains the elements of relation r that are not contained in s, or $r/s = \{(x, y)|(x, y) \in r \not(x, y) \in s\}$

Figure B11. Intersection, Union and Difference of two relations defined on the same Cartesian Product

The complement (or negation) of a relation can also be easily calculated. This is because the enveloping set, which is required for the complement operation on arbitrary sets, comes natural for relations: the Cartesian Product serves the purpose.

- *complement* : $\overline{r}$ is the set of all tuples in the Cartesian Product that are not contained in r.

A tabular representation of the complement relation is easy to write down. In the example of teams for the mixed double, it contains a total of $4 \times 5 - 3 = 17$ tuples, representing all potential couples that are not selected for the mixed doubles competition:

complement of *doubles-with*

But, other and more interesting relational operators exist than just these. The two most important ones are the inverse operator and composition, which will be discussed here. Other ones, such as ‘dagger’ and ‘implication’, will be discussed later on.

Women	Men	Marek	John	Rob	Toine	Raúl
Sophie		yes	yes	yes	yes	
Aisha			yes	yes	yes	yes
Jenny		yes	yes	yes	yes	yes
Nellie		yes	yes	yes		yes

Table 3.5: Not the selected couples for a mixed double competition

3.5.2 Inverse operation

Earlier, we pointed out that the Cartesian Product of a set A and a set B is $A \times B$, and this is not the same as $B \times A$ (except if $A = B$, more on that later). However, a relation that contains tuples of the form (a, b) can easily be altered into a relation containing tuples of the form (b, a) simply by changing the order of the elements. Mathematically, it is a brand new tuple, as source and target differ from before.

inverse
conversion

This operation, producing a new relation from an existing one, is called 'taking the *inverse*', inversion, or sometimes *conversion*. It is denoted by writing $\smile$ (pronounced 'flip') after the relation name, thus:

$$\textit{Customer-places-Order}^{\smile}{}_{[\textit{Order},\textit{Customer}]}$$

We may pronounce this as 'flip of *Customer places Order*'. For most of us, it is more comfortable to change the relation name into something more sensible like '*Order is-being-placed-by Customer*' or '*Order originated-from Customer*'. Beware however that in general, different relation names will indicate different relations!

For a relation r declared as $r_{[A,B]}$, a more formal declaration of the inverse relation $r^{\smile}$ is:

$$r^{\smile}{}_{[A,B]}\text{with contents}\{(b,a)|(a,b) \in r\}$$

As the inverse operator merely swaps the left and right hand sides of each tuple, it is easy to write down the extension of the new relation, if given the extension of the old one. In a tabular form, the inverse operator merely swaps columns:

Flip (or inverse) of Doubles With

In the matrix form such as figure B13, the inverse operator mirrors the entire content of the matrix along the diagonal. The two axes, source and target, are swapped accordingly.

Man	*Woman*
Raúl	Sophie
Marek	Aisha
Toine	Nellie

Table 3.6: Inverse of the selected couples for a mixed double competition, in tabular layout

3.5.3 Composition operation

composition

The *composition* operator is perhaps the most important operation in Relational Algebra. It produces a new relation from two existing ones. The formal definition is as follows.

Let $r_{[A,B]}$ and $s_{[B,C]}$ be two relations, with the target of r being the same as the source of s. Then the composition of r and s, is the relation with

signature $r;s_{[A,C]}$,

and its content is

$$\{(a,c)|\text{there exists at least one}b \in B\text{such that}arb\text{and also}bsc\}$$

The composition operator is denoted by a semicolon ; between the two relation names. It is pronounced as 'composed with', in this case: r composed with s.

Figure B13. Composition of Actor *appears-in* Performance and Performance *is-of* Play

The figure illustrates how composition works: an Actor (at the left) appears in a Performance (middle), and that Performance is of a certain Play (at the right). The meaning of the composed relation is: the Actor performs in a Play for at least one Performance. Remember that you cannot compose just any two relations: if there is no middle ground, then composition has no meaning. The readers familiar with relational database operations will recognize that this composition operation corresponds to the 'natural join' operator.

Exercise

- verify that $r;s = \{(a,c)|trg_{[r,]}(a) \cap src_{[s,]}(c) unequalO - slash\}$ for any two relations r and s.

neutral with respect to

- prove that identity is *neutral with respect to* composition, i.e. if you compose any relation r with the identity relation, the outcome is exactly the relation r:

$$r; Id_{[B,]} \;=\; Id_{[A,]}; r = r$$

3.5.4 More advanced operations

Apart from composition, a number of operations on two relations can be defined. We will briefly discuss them, although in this book we will rarely use them. Moreover, mathematically speaking, it can be shown that all these advanced operations can be reduces to ordinary composition. In the following few definitions, let $r_{[A,B]}$ and $s_{[B,C]}$ be two relations, with the target set B of r being the same as the source of s.

RELATIVE ADDITION

The formal definition is as follows:

the relative addition of r and s, is the relation with

signature $r \dagger s_{[A,C]}$,

and its content is

$$\{(a, c)|\text{for all} b \in B \text{either} arb \text{or} bsc\}$$

The relative addition operator is denoted by the symbol †, pronounced as 'dagger', between the two relation names. Figure B14 is an example of this operation.

Exercise

- verify that $r \dagger s = \{(a, c)|trg_{[r,]}(\text{a}) \cup src_{[s,]}(\text{c}) = B\}$ for any two relations r and s.
- prove the following assertion: $\overline{r; s} = \overline{r} \dagger \overline{s}$, or equivalently $\overline{r \dagger s} = \overline{r}; \overline{s}$.

The latter exercise illustrates our claim that advanced operators, in this case relative addition, can be rewritten using only the composition operator

Figure B14. Relative addition of ???

To illustrate how a relative addition works, consider an example of ... to do ...

RELATIVE IMPLICATION

The formal definition is as follows: the relative implication of r and s, is the relation with signature $s_{[A,C]}$, and its content is $\{(a,c)|forallb \in B : arbimpliesbsc\}$

We denote this operator by a symbol resembling inclusion $\subset$. As an example, consider an example of students that may go on tour to some destination as offered by their school. Whether a student is eligible for a destination depends on his or her subject(s). The two relations in this example are

- Destination requires Subject
- Student passed-for Subject

and these two relations are combined to produce

- Destination is-open-to Student

Figure 17. left: relation requires, right: the inverse of relation passed-for.

The school demands that to be eligible for a tour, a student must have passed all subjects that are required for that particular tour destination:

- the study tour to Hawaii is open to student Conway, as he has passed for the required subject Surfing,
- for the trip to Rome, two subjects are required. Student Brown has completed all three subjects, so she qualifies. Student Ang is less fortunate, as she has not yet passed for Latin which is one of the required subjects.
- the Amsterdam study tour requires no subjects, it is open to all students irrespective of their achievements in class.

The school demand means that for a destination to be open to a student, it is required that for all subjects: IF Destination requires Subject THEN Student passed-for Subject. In other words, for a tuple (destination, student) to be in the relation is-open-to, the following assertion holds

given destination and student, it holds for all subjects: requires $passed\text{-}for^{\smile}$

which can also be written as:

$$is - open - to = requires[passed\text{-}for^{\smile}$$

Exercise

- verify that $r[s = \{(a,c)|trgr(a) \in srcs(c)\}$ for any two relations r and s, which explains our choice of symbol [for this operation.
- prove the following assertion: $\overline{r} \dagger s = r[s$

RELATIVE SUBSUMPTION

The formal definition is as follows: the relative subsumption of r and s, is the relation with signature $r]s_{[A,C]}$, and its content is $\{(a,c)|\text{for all}b \in B : bs c\text{implies}arb\}$

This operator is denoted by a symbol resembling inclusion in the other direction , for the same reason as above.

Exercise

- verify that $r[s = \{(a,c)|trgr(a) \subset srcs(c)\}$ for any two relations r and s.
- prove the following assertion: $r \dagger sc = r]s$

3.6 Laws for operations on relations

This section introduces a number of laws for you to work with. Laws for relations are used to rewrite formulas into other formulas without loss of meaning. For example:

1. $\mathit{for}^{\smile};\mathit{sent} \ \vdash \ \mathit{delivered\text{-}to}$ meaning: “The *Customer* will accept an *Invoice* for an Order, only if that Order was delivered to the *Customer*.”
2. $\overline{\mathit{delivered\text{-}to}};\mathit{sent}^{\smile} \ \vdash \ \overline{for^{\smile}}$ meaning: “If an Order was *not* delivered to a *Customer* and for that Order a certain *Invoice* was sent, then that certain *Invoice* can *not* be for that *Customer*.”

How do we know whether these two formulas express the same business meaning? The answer is provided by using a law, in this case the law which ensures that $q;r \vdash s$ means the same as $\overline{s};r^{\smile} \vdash \overline{q}$ for any relations q, r, and s.

Based on this law (theorem K, introduced on page 85), we can answer the question affirmatively. This involves manipulating (i.e. reasoning with) formulas, just like you would manipulate numbers if you have to compute the total on an invoice.

3.6.1 Laws for composition and for relative addition

The first two laws (3.1 and 3.2) state that composition and relative addition are associative operators. This means that it does not matter how brackets are placed in an expression with two or more of the same associative operators. As a consequence, brackets may be omitted and the expression looks less cluttered.

$$
\begin{aligned}
(p;q);r &= p;(q;r) = p;q;r \qquad (3.1)\\
(p \dagger q) \dagger r &= p \dagger (q \dagger r) = p \dagger q \dagger r \qquad (3.2)\\
r;\mathbb{I} &= r \qquad (3.3)\\
\mathbb{I};r &= r \qquad (3.4)\\
r \dagger \overline{\mathbb{I}} &= r \qquad (3.5)\\
\overline{\mathbb{I}} \dagger r &= r \qquad (3.6)
\end{aligned}
$$

The laws 3.3 and 3.4 state that the identity relation $\mathbb{I}$ may be removed if it uses in a composition: the identity relation is "neutral" with respect to composition.

diversity

Similarly, the laws 3.5 and 3.6) permit you to remove the complement of identity (sometimes referred to as *diversity*), $\overline{\mathbb{I}}$, if it appears next to relative addition.

3.6.2 Laws for the inverse operator

The following laws show how the conversion operator behaves.

$$
\begin{aligned}
r^{\smile\smile} &= r \qquad (3.7)\\
(r \cup q)^{\smile} &= r^{\smile} \cup q^{\smile} \qquad (3.8)\\
(r \cap q)^{\smile} &= r^{\smile} \cap q^{\smile} \qquad (3.9)\\
(r;s)^{\smile} &= s^{\smile};r^{\smile} \qquad (3.10)\\
(r \dagger s)^{\smile} &= s^{\smile} \dagger r^{\smile} \qquad (3.11)
\end{aligned}
$$

3.6.3 Laws for distribution in compositions

When working with compositions of relations, it often happens that an operator such as union, intersection or implication appears somewhere in the formula.

The following law shows how the union operator can be removed from compositions of relations.

$$p;(q \cup r);s = (p;q;s) \cup (p;r;s) \qquad (3.12)$$

It would be nice if the same kind of distribution would hold for the intersect operator, $\cap$, but alas not. In general, distribution does not apply for intersections. Only if certain conditions (constraints) are satisfied, does distribution apply for intersections. This will be demonstrated later in the chapter.

3.6.4 Complement

This section introduces the most important laws concerning complement. At the end, we elaborate an example in which a rule is "invented" to capture a real life situation.

The best known law concerning complement is that it is an idempotent operator: apply it twice and the original relation is reproduced.

$$\overline{\overline{r}} \quad \Leftrightarrow \quad r \tag{3.13}$$

Other important laws were formulated by the mathematician De Morgan:

$$\begin{aligned} \overline{r} \cup \overline{s} &= \overline{r \cap s} \\ \overline{r} \cap \overline{s} &= \overline{r \cup s} \end{aligned} \tag{3.14}$$

De Morgan's equivalences are frequently used because they allow the complement operator to "move around" in your formula. Similar laws concern the operators ; and †:

$$\begin{aligned} \overline{r};\overline{s} &= \overline{r \dagger s} \\ \overline{r} \dagger \overline{s} &= \overline{r;s} \end{aligned} \tag{3.15}$$

Equations 3.14 and 3.15 are all known as De Morgan's laws.

Less known, but almost as important are the following equivalences, which De Morgan called "Theorem K".

$$p;q \vdash \overline{r} \;\Leftrightarrow\; p^{\smile};r \vdash \overline{q} \;\Leftrightarrow\; r;q^{\smile} \vdash \overline{p} \tag{3.16}$$

The three parts are not identical: you can check this by looking at the tyoes (sources and targets) of the relations at the right hand sides. We use the left-and-right arrows to indicate that the three parts are equivalent: De Morgan proved that if one of the expressions is true (or false), then the other ones are true (or false) as well. Like De Morgan's laws, theorem K too is used for "moving around" the complement operator in formulas.

The use of theorem K and De Morgan's laws are illustrated in an exercise at the end of this chapter. The exercise 3.1 and its elaboration illustrate some of the thoughts you might have when analyzing real life situations.

If you are new to business rules, we suggest you try to replay this example carefully, in order to re-live the arguments and thoughts described. If you have already done this a couple of times before, you may find the argumentation too elaborate. That means you are proficient in using the laws governing relations.

3.6.5 Laws about inclusion

This section provides some laws about the inclusion operator $\vdash$ or $\subset$. The following expression is obviously true for relations r and s sharing the same types:

$(r \cap s) \subset \mathrm{r} \subset (r \cup s)$

Inclusion can be combined with the flip and complement operators, which produces the following equivalences:

$$\begin{align}
r \vdash s &\quad\Leftrightarrow\quad r^{\smile} \vdash s^{\smile} \tag{3.17}\\
r \vdash s &\quad\Leftrightarrow\quad \overline{s} \vdash \overline{r} \tag{3.18}\\
& \tag{3.19}
\end{align}$$

Notice especially how the order of r and s are reversed in the second formula.

monotonicity

Apart from flip and complement, inclusion also combines easily with other operations. This is sometimes referred to as the *monotonicity* of inclusion. Notice however that the following laws are not equivalences but work in only one direction:

$$\begin{align}
r \vdash s &\quad\Rightarrow\quad r;t \vdash s;t \tag{3.20}\\
r \vdash s &\quad\Rightarrow\quad t;r \vdash t;s \notag\\
r \vdash s &\quad\Rightarrow\quad r \dagger t \vdash s \dagger t \tag{3.21}\\
r \vdash s &\quad\Rightarrow\quad t \dagger r \vdash t \dagger s \notag\\
r \vdash s &\quad\Rightarrow\quad r \cup t \vdash s \cup t \tag{3.22}\\
r \vdash s &\quad\Rightarrow\quad r \cap t \vdash s \cap t \tag{3.23}
\end{align}$$

3.6.6 Operator precedence

To conclude this section, we give some conventions that govern precedence of operators. Just like in arithmetics, where for instance “take the square of” takes precedence over addition. These conventions save brackets and simplifies the writing of formulas with multiple operators. Table 3.7 contains the precedence rules.

expression	precedence rule	to be read as
$=$ and $\vdash$ have weakest precedence of all operators		
$p = q \cup r$	$\cup$ binds stronger than $=$	$p = (q \cup r)$
$p \vdash q \cup r$	$\cup$ binds stronger than $\vdash$	$p \vdash (q \cup r)$
$\cap$ precedes over $\cup$		
$p \cup q \cap r$	$\cap$ binds stronger than $\cup$	$p \cup (q \cap r)$
$;$ and $\dagger$ have equal precedence, stronger than $=$, $\vdash$, $\cap$, and $\cup$		
$p \cap q; r$	$;$ binds stronger than $\cap$	$p \cap (q; r)$
$p \cap q \dagger r$	$\dagger$ binds stronger than $\cap$	$p \cap (q \dagger r)$
Unary operators $^{\smile}$ and have precedence over binary operators.		
$q^{\smile}; r$	$^{\smile}$ binds stronger than any binary operator	$(q^{\smile}); r$
$q^{\smile} \dagger r$	??? $\dagger$ binds stronger than $\cap$	$(q^{\smile}) \dagger r$

Table 3.7: Precedence of operators

3.7 Homogeneous relations

So far, we discussed relations based on Cartesian Products with arbitrary sources and targets, and we discussed cardinality rules thereof. But the same set may be used for both source and target. This section is devoted to exactly such relations, and the special features that they have. We already encountered one example of a relation with identical source and target, namely the Identity relation, abbreviated $\mathbb{I}$.

homogeneous

We define a *homogeneous* relation as

– a relation for which source and targe are identical

heterogeneous

All other relations will be called *heterogeneous* relations: their source and target are not identical.

Examples of homogeneous relations are easy to think of: Person *is-related-to* Person. Or in arithmetics: Number *is-greater-than* Number. Or in chemistry: Compound *may-decompose-into* Compound. Or in software maintenance: Software-change *interferes-with* Software-change. Or in business administration: Use-case *is-subvariant-of* Use-case. In a Conceptual Diagram, the homogeneous relations are easy to spot because the relation connects a concept with itself, and there will be an arc pointing back to its origin.

Let us consider a relation $r_{[A,A]}$. By our definition, r is homogeneous as the source A and target A are the same. We can point out several characteristics thatt the relation r may, or may not have.

3.7.1 Reflexive

reflexive

A relation r is called *reflexive* if for all elements x in the set A, the tuple (x, x) is in r. The relation *is-a-close-friend-of* is an example: everybody is a close friend of themselves.

The condition can also be written as:

$$\mathbb{I} \vdash r$$

irreflexive

The complete opposite is a relation with the property that identical pairs (x,x) are forbidden. Relations with this property are called *irreflexive*. Examples of irreflexive relations are *is-parent-of* or *is-spectator-of-the-performance-by-th*

In a tabular representation, a reflexive relation will show markings in all cells on the diagonal (and probably in other places as well, but that does not concern us). To contrast, an irreflexive relation will have markings anywhere in the table except on the diagonal.

A homogeneous relation can, but need not be reflexive or irreflexive. In general, there are no specific conditions for tuples (x,x) to be or not to be in the relation. For instance, when elections are held, then a relation Person *votes-for* Person may, or may not show a number of people who voted for themselves (of course, the votes remain secret in general).

3.7.2 Symmetric

symmetric

A the relation r is called *symmetric* if for any tuple (x,y) in r, the inverse tuple (y,x) is also in r. This can also be written as:

$$r^{\smile} \vdash r$$

We leave it to the reader to verify that a relation is symmetric if and only if equality holds, i.e.

$$r^{\smile} = r$$

Examples of relations that are symmetric (or at least ought to be so) are *is-married-to*, *is-in-a-meeting-with*, and *is-in-the-same-class-as*.

asymmetric

A relation r is *asymmetric* if for any tuple (x, y) in r, the inverse tuple

(y, x) is not in r, which of course can be written as:

$$r^{\smile} \cap r = \emptyset$$

An example of a relation that is asymmetric is *awards-bonus-payment-to*. Of course, violations may occur: two managers that award bonuses to one another. Remark that an asymmetric relation is automatically irreflexive. Iindeed, we do not want a top manager to award a bonus to him- or herself.

Like with (ir)reflexivity, many homogeneous relations are neither symmetric nor asymmetric. The relation Website *has-a-hyperlink-to* Website is an example: some websites may link to one another, but there is no rule or obligation to do so.

antisymmetric

A relation may be 'almost' asymmetric, meaning that it would be asymmetric were it not for some reflexive tuples (x,x). Such relations are sometimes called *antisymmetric*, and they are characterized as

$$r^{\smile} \cap r \ \subset \ \mathbb{I}$$

3.7.3 Symmetric as well as antisymmetric

In rare situations, a homogeneous relation p may be both symmetric and antisymmetric at the same time. Some careful reasoning shows that such a relation p must satisfy the condition: $p \vdash \mathbb{I}$. We leave it to the reader to verify that both the Identity relation and the empty relation $\emptyset$ are symmetric and antisymmetric.

binary property

However, there is more to it, if we consider the elements of the source (and target!) set A. We can divide A into two subsets: one subset containing the elements x that do satisfy $(x, x) \in p$, and the subset of all other elements in A that are not in $p : (y, y) \notin p$. Apparently, the relation p divides the source set in two disjoint subsets. In other words, the relation p can be understood as a simple Yes/No property of the elements of the source A. Yes, the element x is in the subset for which (x, x) is in p. Or no, the tuple (x, x) is not in p. For this reason, a homogeneous relation that is symmetric and antisymmetric is sometimes called a *binary property* for A.

By the way: we feel there is a much better way to specify such a property on A, as will be shown in the chapter on design considerations.

3.7.4 Transitive and intransitive

If a relation is homogeneous, it is possible to define the composition with itself. In fact, it is possible to derive many new homogeneous relations. As an example, consider the relation *is-older-than*. if Albert *is-older-than* Betty, and Betty *is-older-than* Chris, then we have Albert *is-older-than* someone who *is-older-than* Chris. A relation r is called *transitive* if the composition with itself is contained in the relation itself:

transitive

$$r;r \vdash r$$

As another example of transitive relation, we can point out the "subset" relation among sets: if $A \subset B$ and $B \subset C$ then $A \subset C$

so (excuse the complicated notation!)

$$\subset;\subset\vdash\subset$$

The opposite of transitivity is shown in family-relationships as studied in genealogy. If we have Person *is-parent-of* Person, then we can create a new relation Person *is-gransparent-of* Person simply by composing the *is-parent-of* relation with itself. Repeating this, we arrive at the relation Person *is-greatgrandparent-of* Person, etcetera. This is an example of an *intransitive* relation:

intransitive

$$r;r \cap r = \emptyset$$

Again, like reflexivity and symmetry, many homogeneous relations have neither the transitive nor the intransitive properties.

3.7.5 Inclusion relation

In large sets, one can point out all kinds of subsets. For example, in the set of all customers we can pick out all customers who joined last week, or who haven't paid the last invoice, or who live in New Amsterdam, or who have not ordered a Harry Potter book. In any case, we can define a very simple relation with the subset as source, and the enveloping set as target. Any instance of the former is always an instance of the latter, no exceptions.

The definition (intention) of the first concept, which is the subset, is more limited, more restrictive than the intention of the enveloping concept. And naturally, the extension of the first concept is contained in the extension of the other concept. If this is the case, we call the source the specialisation, the target being the generalisation. An *inclusion relation* is defined as

inclusion relation

- a relation for which the source is a proper subset of the target

Examples of this kind of relation is easy to think of: Student *is-a* Person. Or in arithmetics: Prime-Number *is-a* Number. Or chemistry: Pure-Substrate *is-a* Compound. This kind of relation is quite common in Conceptual Models, and they are easy to spot because they are usually named *is-a*. But please do not make the mistake to think that these are homogeneous relations, as their sources and targets are really different!

And as a closing remark: it is quite possible that the extension of one concept is a subset ot the extension of another set. All customers in the shop happen to be male, all company cars run on diesel fuel. But this is not sufficient for generalisation-specialisation: customers need not be male by definition; company cars may run on any kind of fuel and still be a company car. The subset property is just a temporary coincidence in these instances, and the extensions of the the concepts, without changing definitions, can be quite different at some other time.

3.7.6 Structure of a set

We will not use it, but it is good to know that there is a close link between the structure of a set, and the properties of a homogeneous relation.

If for a set A we have a homogeneous relation p that is reflexive, symmetric and transitive at the same time, this means that the set A can be partitioned. It is possible to point out a number of subsets (also called partitions) A1, A2, A3 ...An. such that every element of A is contained in exactly one of the subsets. The intersection of any two subsets is empty, and the union of all subsets equals the original set A. The relation is called an *equivalence relation.* Examples of this kind of situation are not hard to find: Employee works-at-the-same-department-as Employee, or Student has-same-grade-as Student. An instance diagram of this kind of relation resembles an archipelago of islands: the entire set consists of islands, every element is in one island, and islands do not overlap.

equivalence relation

Suppose we define a relation r between bags of money, with a r b meaning that the bags a and b contain the same amount of money. It is easy to see that this relation is reflexive, symmetric and transitive, hence an equivalence relation. The partitions will be determined by “amount of money”. We could also have defined another equivalence relation for the bags of money, saying that asb if two bags have the exact same weight. This is a different relation, but it is still an equivalence relation and it enforces a different structure on the set of money bags. Both relations are worthwhile and relevant: the first corresponds to a partitioning according to money value, the second relation corresponds to a partitioning of the same set according to weight.

ordering relation

If for a set A we have a homogeneous relation p that is reflexive, asymmetric and transitive at the same time, this means that the set A has some ordering or hierarchy, and the relation is said to be an *ordering relation* or order. For example: Employee *is-subordinate-to* Employee. Other examples are organizational top-down structures, set inclusion, and ancestor relations.

partial order

An instance diagram of this kind of set resembles a tree with branches, with the elements arranged along the separate branches. Or to be precise: one or more trees. Notice that in general, the ordering is incomplete, as not every two employees will be represented in the relation, neither is subordinate to the other (they lie on separate branches). Or, there exists tuples (x,y) such that neither $(x, y) \in p$ nor $(y, x) \in p$. This is called a *partial order*.

total order

It is called a *total order*, or linear order, if any two elements can be compared: either $(x, y) \in p$ or $(y, x) \in p$. In other words, $p \in p^{\smile} = V[AxA]$. For a linear order, the instance diagram will resemble a single line, with all elements neatly arranged on that line. An example is the arithmetical *is-smaller-than-or-equal-to* relation on numbers.

3.8 Multiplicity constraints

Having defined the notion of relation (called fact-type by the Business Rule Manifesto), it must be realized that a single relation can already be subjected to control: the relation must satisfy some business rule. A business rule that governs a single relation is by its very nature rather simple and easy to understand.

Rules for a single relation are often expressed as “one-to-many”, “surjective”, “1..n”, and the like. Such rules are frequently encountered, and they have their own name: multiplicity constraints, or cardinalities. They express knowledge about how the tuples in a relation should be organised, what behaviour is expected -or prevented.

From mathematics we learn that multiplicity constraints come in exactly four flavours. It is the duty of the rule analist to establish for every relation which constraints apply. As the four flavours can be combined in any way, a total of 16 combinations is possible. Indeed, relations can be found in practice where no constraints apply (anything goes), up to relations that are under very restrictive multiplicity constraints.

3.8.1 Univalent and Total

These twee multiplicity constraints must be verified at the source set:

- a relation r is univalent if every element in the source occurs in at most one tuple of the relation, or: every element in the source is related to at most one element in the target.
- a relation r is total if every element in the source always occurs in at least one tuple of the relation, or: every element in de source is related to at least one element in the target.

These properties are easily verified in a tabular representation of the relation. If we list the relation with the source at the left, and the target across, then univalent means that there is at most one hit on every row. Total means that there is at least one hit on every row, and more than one hit is also fine.

In an instance diagram, univalence means that at most one arc emanates from every source element. Total means that at least one outgoing arc is drawn for each element.

Figure B7. Left: univalent but not total. Right: total but not univalent.

3.8.2 Injective and Surjective

These two cardinality constraints are to be verified at the target set:

- a relation r is injective if every element in the target occurs in at most one tuple of the relation, or: every element in the target is related to at most one element in the source.
- a relation r is surjective if every element in the target always occurs in at least one tuple of the relation, or: every element in the target is related to at least one element in the source.

Again, these properties are easy to verify in a tabular representation. Source at the left, target across, then an injective relation will show at most one hit in each column, whereas a surjectieve relation will show one or more hits in every column in the table.

In an instance diagram, injective means that in each target element, at most one arc arrives. Surjective means that at least one incoming arc is drawn to each element.

Figure B8. Left: injective but not surjective. Right: surjective but not injective.

3.8.3 Function

Some combinations of cardinalities appear very frequently in practical applications, and they deserve their own name. In particular:

- a relation r is a function if it is both univalent and total: every element in the source is related to exacty one element in the target set.

The word 'function' is also known in mathematics, with its notation $f(x) = y$, for instance the function 'square': $f(x) = x^2$. It is no coincidence that the same word is used: in this mathematical function, each and every number x is related to exactly one other number y. Thus, the mathematical function 'square' is both univalent and total. Would we write (x, x^2) instead of $f(x) = x^2$, then this particular relation would contain such tuples as $(1, 1)$, $(2, 4)$, $(3, 9)$ but also $(0, 0)$, $(-1, 1)$, $(-2, 2)$. Indeed, the 'square' function produces correct tuples, exactly one for each number x.

3.8.4 Functional dependency

The notion of functional dependency lies at the basis of relational database modelling. The definition is:

- a relation r is a functional dependency if it is both injective and total: every element in the source is related to an element in the target set, and every element in the target is related to at most one element in the source.

A concept in a Conceptual Model that appears in only one relation, and the relation is a functional dependency, is not very interesting. It means that the source element provides some descriptive information about its target concept. The source does not really contribute to the overall structure of the Conceptual Model, and many Conceptual Diagrams will not even show them. In Relational Database modelling, such source concepts (and relation) are called attribute. For now, we will not divulge this issue.

3.8.5 Injection, Surjection, Bijection

Functions are very frequent, and they often even have an additional cardinality, either injectivity or surjectivity. A function r (a relation that is both univalent and total) is called:

- injection, if the function is injective
- surjection, if the function is surjective
- bijection, if it is both injective and surjective.

Notice in this last case how the relation is a function, and its inverse relation is also a function. A closer inspection reveals that a bijection requires that the source and target datasets must have exactly the same number of elements, for each element in the source is related to one target element, and reversely.

Figure B9. Common combinations of multiplicities. Left to right: function, injection, surjection, bijection

Although most relations are subject to some multiplicity constraints, this does not mean that such rules apply for every concept in every relation. For example, a business may dictate that each invoice is sent to one customer. Or, for every invoice in the source concept, there is exactly one tuple in the relation is-sent-to in which that invoice appears. But there is no rule the other way around: one customer may be sent no invoice at all, or she may receive hunderds.

3.9 Laws about Multiplicities

Multiplicities are useful because a number of important laws are valid only for relations with certain multiplicities. Furthermore, it is good to know how the multiplicity properties are propagated in formulas. For instance, if r and s are both functions, then $r;s$ is a function as well. Such laws are important, for example in designing data structures in software. The laws introduced in this section will be derived from laws introduced earlier in this chapter.

The multiplicities of the flip of a relation will be immediately obvious, just by inspecting the definitions. So we have

$$\mathit{injective}(r) \;\Leftrightarrow\; \mathit{univalent}(r^{\smile})\mathit{surjective}(r) \;\Leftrightarrow\; \mathit{total}(r^{\smile}) \tag{3.24}$$

It also fairly obvious to check that if two relations have similar multiplicities, then their composition also has that multiplicity:

$$\begin{aligned}
\mathit{univalent}(s) \wedge \mathit{univalent}(r) &\;\Rightarrow\; \mathit{univalent}(r;s) && (3.25)\\
\mathit{surjective}(s) \wedge \mathit{surjective}(r) &\;\Rightarrow\; \mathit{surjective}(r;s) && (3.26)\\
\mathit{total}(s) \wedge \mathit{total}(r) &\;\Rightarrow\; \mathit{total}(r;s) && (3.27)\\
\mathit{injective}(s) \wedge \mathit{injective}(r) &\;\Rightarrow\; \mathit{injective}(r;s) && (3.28)
\end{aligned}$$

However, it is slightly more complicated to check which multiplicity properties are valid for the intersection $r \cap s$ or union $r \sup s$ of two relations. The results are:

$$\begin{aligned}
&\mathit{univalent}(r) \wedge \mathit{univalent}(s) \;\Rightarrow\; \mathit{univalent}(r;s) && (3.25)\\
&\mathit{surjective}(s) \wedge \mathit{surjective}(r) \;\Rightarrow\; \mathit{surjective}(r;s) && (3.26)\\
&\mathit{total}(s) \wedge \mathit{total}(r) \;\Rightarrow\; \mathit{total}(r;s) && (3.27)\\
&\mathit{injective}(s) \wedge \mathit{injective}(r) \;\Rightarrow\; \mathit{injective}(r;s) && (3.28)\\
&\mathit{univalent}(r) \vee \mathit{univalent}(s) \;\Rightarrow\; \mathit{univalent}(r \cap s) && (3.31)\\
&\mathit{injective}(r) \vee \mathit{injective}(s) \;\Rightarrow\; \mathit{injective}(r \cap s) && (3.29)\\
&\mathit{total}(r) \vee \mathit{total}(s) \;\Rightarrow\; \mathit{total}(r \cup s) && (3.39)\\
&\mathit{univalent}(r \cup s) \;\Rightarrow\; \mathit{univalent}(r) \wedge \mathit{univalent}(s) && (3.40)\\
&\mathit{injective}(r \cup s) \;\Rightarrow\; \mathit{injective}(r) \wedge \mathit{injective}(s) &&\\
&\mathit{surjective}(r \cup s) \;\Rightarrow\; \mathit{surjective}(r) \wedge \mathit{surjective}(s) && (3.30)\\
&\mathit{total}(r \cap s) \;\Rightarrow\; \mathit{total}(r) \wedge \mathit{total}(s) &&
\end{aligned}$$

The previous results inspire to do similar derivations for $r \cap s$ and $r \cup s$. Let us start with $r \cap s$, deriving the conditions under which it is univalent.

$$\begin{array}{ll}
 & (r \cap s)^{\smile}; (r \cap s)\\
= & \qquad\qquad \{\text{apply law 3.9 on } (r \cap s)^{\smile}\}\\
 & (r^{\smile} \cap s^{\smile}); (r \cap s)\\
\vdash & \qquad\qquad \{\text{distribute (law 4.6)}\}\\
 & (s^{\smile}; r) \cap (r^{\smile}; r) \cap (s^{\smile}; s) \cap (r^{\smile}; s)\\
\vdash & \qquad\qquad \{\text{assume } r \text{ is univalent}\}\\
 & (s^{\smile}; r) \cap \mathbb{I} \cap (s^{\smile}; s) \cap (r^{\smile}; s)\\
\vdash & \qquad\qquad \{\text{by inclusion wrt } \cap \text{ (law } \mathbf{??})\}\\
 & \mathbb{I}
\end{array}$$

This derivation says that $(r \cap s)^{\smile}; (r \cap s) \vdash \mathbb{I}$, meaning that $r \cap s$ is univalent, under the assumption that r is univalent. So we may conclude:

$$\mathit{univalent}(r) \;\Rightarrow\; \mathit{univalent}(r \cap s)$$

Since $r \cap s \;=\; s \cap r$, we may conclude:

$$\mathit{univalent}(r) \vee \mathit{univalent}(s) \;\Rightarrow\; \mathit{univalent}(r \cap s) \qquad (3.31)$$

Laws for Distribution in compositions Distribution of composition does not apply for intersectino, in general. But for a univalent p and injective s can we prove this law:

$$\begin{array}{lrcl r}
\text{if } s \text{ is injective} & (q \cap r); s & = & q; s \cap r; s & (3.32)\\
\text{if } p \text{ is univalent} & p; (q \cap r) & = & p; q \cap p; r & (3.33)
\end{array}$$

Another way to formulate these laws in an unconditional way is:

$$p; (q \cap r); s \ \vdash \ p; q; s \cap p; r; s \qquad (3.34)$$

¿ELDERS¡ Two other useful laws allow you to move relations across the $\vdash$ symbol. However, this law is conditional for functions (i.e. univalent and total relations).

$$\begin{array}{lrcl r}
\text{if } f \text{ is a function} & r \vdash s; f^{\smallsmile} & = & r; f \vdash s & (3.35)\\
\text{if } f \text{ is a function} & r \vdash f; s & = & f^{\smallsmile}; r \vdash s & (3.36)
\end{array}$$

Functions are also useful to replace † in your formulas by ; operators.

$$\begin{array}{lrcl r}
\text{if } f \text{ is a function} & \overline{f} \dagger r & = & f; r & (3.37)\\
\text{if } f \text{ is a function} & r \dagger \overline{f^{\smallsmile}} & = & r; f^{\smallsmile} & (3.38)
\end{array}$$

Again, by reversing the derivation, we can derive another result.

$$\begin{array}{ll}
 & \mathbb{I}\\
\vdash & \qquad\qquad\quad \{\text{assume } r \text{ is total}\}\\
 & r; r^{\smallsmile}\\
\vdash & \\
 & (r; r^{\smallsmile}) \cup (s; r^{\smallsmile}) \cup (r; s^{\smallsmile}) \cup (s; s^{\smallsmile})\\
= & \qquad\qquad\quad \{\text{distribute (law 4.3)}\}\\
 & (r \cup s); (r^{\smallsmile} \cup s^{\smallsmile})\\
= & \qquad\qquad\quad \{\text{apply law 3.8 to } r^{\smallsmile} \cup s^{\smallsmile}\}\\
 & (r \cup s); (r \cup s)^{\smallsmile}
\end{array}$$

This derivation produces the following result:

$$\mathit{total}(r) \ \Rightarrow \ \mathit{total}(r \cup s)$$

Since $r \cup s \ = \ s \cup r$ we may rephrase this as:

$$\mathit{total}(r) \vee \mathit{total}(s) \ \Rightarrow \ \mathit{total}(r \cup s) \qquad (3.39)$$

The following derivation studies the univalence of $r \cup s$:

$$\begin{array}{ll}
 & (r \cup s)^{\smile}; (r \cup s) \ \vdash \ \mathbb{I} \\
\Leftrightarrow & \qquad\qquad \{\text{apply law 3.8 on } (r \cup s)^{\smile}\} \\
 & (r^{\smile} \cup s^{\smile}); (r \cup s) \ \vdash \ \mathbb{I} \\
\Leftrightarrow & \qquad\qquad \{\text{distribute (law 4.3)}\} \\
 & (r^{\smile}; r) \cup (s^{\smile}; r) \cup (r^{\smile}; s) \cup (s^{\smile}; s) \ \vdash \ \mathbb{I} \\
\Rightarrow & \\
 & (r^{\smile}; r) \cup (s^{\smile}; s) \ \vdash \ \mathbb{I} \\
\Leftrightarrow & \qquad\qquad \{A \cup B \vdash C \ \Leftrightarrow \ A \vdash C \wedge B \vdash C\} \\
 & (r^{\smile}; r \vdash \mathbb{I}) \ \wedge \ (s^{\smile}; s \vdash \mathbb{I})
\end{array}$$

This derivation shows that if $r \cup s$ is univalent, then r and s are both univalent:

$$\mathit{univalent}(r \cup s) \ \Rightarrow \ \mathit{univalent}(r) \wedge \mathit{univalent}(s) \tag{3.40}$$

Now let us investigate $r \cap s$ for totality

$$\begin{array}{ll}
 & \mathbb{I} \\
\vdash & \qquad\qquad \{\text{Let } r \cap s \text{ be total}\} \\
 & (r \cap s); (r \cap s)^{\smile} \\
= & \qquad\qquad \{\text{apply law 3.9 on } (r \cap s)^{\smile}\} \\
 & (r \cap s); (r^{\smile} \cap s^{\smile}) \\
\vdash & \qquad\qquad \{\text{distribute (law 4.6)}\} \\
 & r; r^{\smile} \ \cap \ s; r^{\smile} \ \cap \ r; s^{\smile} \ \cap \ s; s^{\smile} \\
\vdash & \qquad\qquad \{\text{inclusion (law } \mathbf{??})\} \\
 & r; r^{\smile} \ \cap \ s; s^{\smile}
\end{array}$$

We get a result by putting the first and last lines of this derivation together, and make one more step:

$$\begin{array}{ll}
 & \mathbb{I} \ \vdash \ (r; r^{\smile} \cap s; s^{\smile}) \\
\Leftrightarrow & \qquad\qquad \{\ \} \\
 & (\mathbb{I} \ \vdash \ r; r^{\smile}) \ \wedge \ (\mathbb{I} \ \vdash \ s; s^{\smile})
\end{array}$$

The entire derivation yields:

$$\mathit{total}(r \cap s) \ \Rightarrow \ \mathit{total}(r) \wedge \mathit{total}(s) \tag{3.41}$$

There are two more derivations, we'd like to share with you

$$\begin{array}{ll}
 & \mathbb{I} \\
\vdash & \qquad\qquad \{\text{Assume } s \text{ is total}\} \\
 & s; s^{\smile} \\
\vdash & \qquad\qquad \{\text{Let } s \vdash r\} \\
 & r; r^{\smile}
\end{array}$$

This derivation yields:

$$\mathit{total}(s) \ \wedge \ s \vdash r \ \Rightarrow \ \mathit{total}(r) \tag{3.42}$$

Similarly, we derive:

$$\begin{array}{lll} & r^{\smile};r \vdash \mathbb{I} & \\ \vdash & & \{\text{Let } s \vdash r\} \\ & s^{\smile};s \vdash \mathbb{I} & \end{array}$$

This derivation yields:

$$\mathit{univalent}(r) \ \wedge \ s \vdash r \ \Rightarrow \ \mathit{univalent}(s) \tag{3.43}$$

The comprehensive list of how multiplicities combines with union and intersection.

3.10 Other approaches than Relational Algebra theory

As we define it, a Conceptual Model captures all relevant features within (a part of) an organisation by means of concepts and binary relations. In many organisations, similar but slightly different models and theories are used to capture and model the relevant data and rules in the corporate enterprise. Depending on the underlying modeling theory, such models go by names like Relational Models, UML-models, ORM-models etc.

There are subtle differences however. The Relational Model refers to entities and attributes, and not to concepts as defined in our theory. And although the Relational Model includes a notion called relation, it differs from our notion of it: the relations in relation models use foreign keys that are composed from the attributes in an entity. However: this book is not the proper place to go into the details of the similarities and differences between Set Theory and Relational Algebra, and Relational Modelling or any other theory.

3.10.1 RuleSpeak

RuleSpeak is an accepted standard to formulate terms, facts and rules about them. The basic idea of RuleSpeak is to use natural language, but to put in certain restrictions so that only clear, unambiguous statements can be formulated, and no vague, uncertain, or undecidable sentences are allowed. Ideally, the business user can describe her entire business context and way of working using RuleSpeak sentences, which would provide rule designers with a rich resource to base their designs on.

¿MEER¡

3.10.2 First-Order Logic

Proposition Logic and Predicate Logic, which we jointly call First-Order Logic, is particularly suited to learn about correct reasoning.

Proposition logic deals with propositions. Each proposition is regarded as an isolated fact, and proposition logic studies how to make straightforward inferences from those facts. Remember that what SBVR calls a fact is, in our dictionary, a tuple in a relation. Two propositions are for instance:

– *Customer* “EGMJ-001” has-paid *Invoice* “20110303-17”

– *Invoice* “20110303-17” is-sent-to *Address* “Moon Crescent 12, Mickleston”

These two propositions are treated in Propositional logic as distinct expressions, and no inference can be made from these two. It would not be correct to automatically assume that the customer is living at the said address. If we add another proposition:

– *Invoice* “20110303-17” *was-received-by Customer* “EGMJ-001”

then it is logical to infer that

IF *Customer* “EGMJ-001” has-paid *Invoice* “20110303-17” THEN *Invoice* “20110303-17” *was-received-by Customer* “EGMJ-001”

This argument is valid for only one specific customer and one specific invoice. But it is quite natural to extend it, and expect it to hold for all customers and all invoices:

IF *Customer* has-paid *Invoice* THEN *Invoice was-received-by Customer*

A more formal notation of the propositions can be given by using the universal quantifier , pronounced “for all”: (cu *Customer*) (bi *Invoice*) cu has-paid bi bi *was-received-by* cu

The right arrow is the implication symbol, pronounced “implies”. In logic, this symbol stands for conditional (if/then) reasoning: only if the lefthand side (“hypothesis” or “premisse”) is true, will the righthand side (“conclusion”) be true also. If the lefthand side is false, then you know nothing about the righthand side: it may either be true or false.

Notice how we jumped from single facts, about one individual customer and one invoice, to a claim about all customers and all invoices.

This is what a good designer does: first, she understands the details of the business, and then she infers general statements that describe it for all situations.

The above formula may look impressive, but this is only due to the abbrevations and shorthand notations that we use - the meaning of the formula is not very complex or mysterious, you can easily pronouce it as follows: for all customers, and for all invoices, it is true that: if the customer has paid the invoice, then the invoice was received by the customer Rephrasing this, we can also say: for all customers, and for all invoices, it is true that: if tuple (customer, invoice) has-paid, then tuple (invoice, customer) *was-received-by* In the formalism of Relational Algebra that this book uses, this means that the first relation is a subset of (the converse of) the second relation: has-paid was-received-by$^{\smile}$

Exercise Use the existential quantifier , pronounced "there exists" or "for at least one", to express that a counterexample (or more) can be found for the assumption that always if a customer has paid an invoice, then that invoice was received by that customer.

Predicate logic adds predicates and quantifiers to propositional logic. This enables us to reason about combinations of facts, in our dictionary: compositions (of instances) of relations with different types. For instance, if we alter the example above, look at the set of three facts

- *Invoice* "20110303-17" is-sent-to *Address* "Moon Crescent 12, Mickleston"
- *Invoice* "20110303-17" *was-received-by Customer* "EGMJ-001"
- *Customer* "EGMJ-001" lives-at *Address* "Moon Crescent 12, Mickleston"

In this case, we infer that the customer receives the invoice because it was sent to an address where the customer actually lives: IF *Invoice* "20110303-17" is-sent-to *Address* "Moon Crescent 12, Mickleston" AND *Customer* "EGMJ-001" lives-at *Address* "Moon Crescent 12, Mickleston" THEN *Invoice* "20110303-17" *was-received-by Customer* "EGMJ-001"

Notice that the actual address doesn't really matter, as long as the address for this customer is known in the corporate database. So the assumption is in fact: IF *Invoice* "20110303-17" is-sent-to some address AND *Customer* "EGMJ-001" lives-at precisely that address THEN *Invoice* "20110303-17" *was-received-by Customer* "EGMJ-001"

This argument, valid for this customer and one invoice, is again extended and expected to hold for all customers and all invoices: IF there is some address SUCH THAT *Invoice* is-sent-to that address AND *Customer* lives-at that address THEN *Invoice was-received-by Customer*

Again, be aware of the jump from single facts, about an individual customer and one invoice, to a conclusion about all customers and all invoices. A more formal notation uses first universal quantifier, to express that it holds for all customers and all invoices, and next the existential quantifier to express that at least one proper address should exist:

$$\forall cu \in \mathit{customer}, iv \in \mathit{Invoice}, ad \in \mathit{Address} : iv\ \mathit{is\text{-}sent\text{-}to}\ ad) \text{ and } (cu\ \mathit{lives\text{-}at}\ ad \text{ implie}$$

Exercise Verify that the implication above translates in Relational Algebra to the following expression that includes a composition of two relations: is-sent-to ; $\mathit{lives\text{-}at}^{\smile} \vdash \mathit{was\text{-}received\text{-}by}$

3.10.3 Translating to first-order logic

Relations and operations on relations can always be written as an expression in predicate logic. This section shows how to perform such translations.

The symbol ∀ means 'for all', ∃ means 'there exists', ∧ means 'and', ∨ means 'or', → means 'implies', and ↔ means 'is equivalent'. For example, this is what you get when you translate the definitions of reflexivity and symmetry, properties that homogeneous relations may have:

$$\begin{aligned} \text{reflexive} &\quad \forall a : a\ r\ a \tag{3.44}\\ \text{transitive} &\quad \forall a, b, c : a\ r\ b\ \wedge\ b\ r\ c\ \rightarrow\ a\ r\ c \tag{3.45}\\ \text{symmetric} &\quad \forall a, b : a\ r\ b\ \leftrightarrow\ b\ r\ a \tag{3.46}\\ \text{antisymmetric} &\quad \forall a, b : a\ r\ b\ \wedge\ b\ r\ a\ \rightarrow\ a = b \tag{3.47} \end{aligned}$$

Next, let us translate the property of univalence (**??**) into a formula of proposition logic. Recall the definition, which states that a relation r is univalent if:

$$r^{\smile}; r\ \vdash\ \mathbb{I} \tag{??}$$

Translated to predicate logic, this property is written as

$$\forall a, b, c :\ c\ r\ a\ \wedge\ c\ r\ b\ \ \rightarrow\ \ a = b \tag{3.48}$$

Equation 3.48 reads as follows: For every possible value of a, b, c: if $c\ r\ a$ and $c\ r\ b$ then $a = b$. In other words, if two right hand side elements, a and b, are both related to the same c, they must be the same element.

Translation procedure for predicate logic

For every rule r to be maintained, start your translation with $\forall a, b : \boxed{a\ r\ b}$. As long as there is a boxed expression in your translation, look it up in the translation table and substitute accordingly. Repeat until no boxes remain.

expression	translate by	
$\boxed{a\ (r \cup s)\ b}$	$\boxed{a\ r\ b} \quad \vee \quad \boxed{a\ s\ b}$	(3.49)
$\boxed{a\ (r \cap s)\ b}$	$\boxed{a\ r\ b} \quad \wedge \quad \boxed{a\ s\ b}$	(3.50)
$\boxed{a\ (r;s)\ c}$	$\exists b :\ \boxed{a\ r\ b} \quad \wedge \quad \boxed{b\ s\ c}$	(3.51)
$\boxed{a\ (\overline{r};s)\ c}$	$\exists b :\ \neg(\boxed{b\ s\ c} \quad \rightarrow \quad \boxed{a\ r\ b})$	(3.52)
$\boxed{a\ (r;\overline{s})\ c}$	$\exists b :\ \neg(\boxed{a\ r\ b} \quad \rightarrow \quad \boxed{b\ s\ c})$	(3.53)
$\boxed{a\ (\overline{r};\overline{s})\ c}$	$\exists b :\ \neg(\boxed{a\ r\ b} \quad \vee \quad \boxed{b\ s\ c})$	(3.54)
$\boxed{a\ (r \dagger s)\ c}$	$\forall b :\ \boxed{a\ r\ b} \quad \vee \quad \boxed{b\ s\ c}$	(3.55)
$\boxed{a\ (\overline{r} \dagger s)\ c}$	$\forall b :\ \boxed{a\ r\ b} \quad \rightarrow \quad \boxed{b\ s\ c}$	(3.56)
$\boxed{a\ (r \dagger \overline{s})\ c}$	$\forall b :\ \boxed{b\ s\ c} \quad \rightarrow \quad \boxed{a\ r\ b}$	(3.57)
$\boxed{a\ (\overline{r} \dagger \overline{s})\ c}$	$\forall b :\ \neg(\boxed{a\ r\ b} \quad \wedge \quad \boxed{b\ s\ c})$	(3.58)
$\boxed{a\ r^{\smile}\ b}$	$\boxed{b\ r\ a}$	(3.59)
$\boxed{a\ \overline{r}\ b}$	$\neg(\boxed{b\ r\ a})$	(3.60)
$\boxed{a\ \mathbb{I}\ b}$	$a = b$	(3.61)
$\boxed{a\ (r \vdash s)\ b}$	$\boxed{a\ r\ b} \quad \rightarrow \quad \boxed{a\ s\ b}$	(3.62)
$\boxed{a\ (r = s)\ b}$	$\boxed{a\ r\ b} \quad \leftrightarrow \quad \boxed{a\ s\ b}$	(3.63)
$\boxed{a\ r\ b}$	$a\ r\ b$, if r is a relation	(3.64)

The procedure is straightforward and can be done entirely by computers. These laws allow us to perfom any translation in steps. Consider for instance the translation from equation **??** to predicate logic:

$$\begin{array}{ll}
 & \forall a,b: \ \boxed{a\ (r^{\smile};r\ \vdash\ \mathbb{I})\ b} \\
\Leftrightarrow & \qquad\qquad \{\text{law 3.62}\} \\
 & \forall a,b: \ \boxed{a\ (r^{\smile};r)\ b} \ \rightarrow\ \boxed{a\ \mathbb{I}\ b} \\
\Leftrightarrow & \qquad\qquad \{\text{apply law 3.51 to } a\ (r^{\smile};r)\ b \text{ and law 3.61 to } a\ \mathbb{I}\ b\} \\
 & \forall a,b: \ (\exists c: \ \boxed{a\ r^{\smile}\ c}\ \wedge\ \boxed{c\ r\ b})\ \rightarrow\ a=b \\
\Leftrightarrow & \qquad\qquad \{\text{apply law 3.59 to } a\ r^{\smile}\ c\} \\
 & \forall a,b: \ (\exists c: \ \boxed{c\ r\ a}\ \wedge\ c\ r\ b)\ \rightarrow\ a=b \\
\Leftrightarrow & \qquad\qquad \{\text{predicate logic}\} \\
 & \forall a,b,c: \ c\ r\ a\ \wedge\ c\ r\ b\ \rightarrow\ a=b
\end{array}$$

The resulting expression is in fact equation 3.48. This derivation proves that the first line and the last line are equivalent:

$$r^{\smile};r\ \vdash\ \mathbb{I}\quad\Leftrightarrow\quad\forall a,b,c:\ c\ r\ a\ \wedge\ c\ r\ b\ \rightarrow\ a=b \tag{3.65}$$

This result has been obtained systematically by applying the appropriate laws where possible. In the first step of the derivation, only law 3.62 (page 72) was applicable. In the second step, laws 3.51 and 3.61 were both applicable, which we did. The third step allowed us to apply only law 3.59. Finally, we used a law from predicate logic to bring the expression in its final shape. In this translation we systematically work our way from the 'outer' structure (the entire expression) to smaller structures in every step we take, until no applicable translation law remains. This way of translating works for every expression in a relation algebra.

Now let us do another example, in which you can verify the procedure for yourself. We translate the definition of surjective step by step, explaining the line of thought as we go. This time, we omit the boxes. First, we quote the definition of surjective (equation **??**), and put it in the starting form for translation:

$$\forall a,b:\ a\ (\mathbb{I}\ \vdash\ r^{\smile};r)\ b$$

When looking at the translation laws, you can see that law 3.62 is applicable. We use this law to substitute the expression of surjectivity into the following expression.

$$\forall a,b:\ a\ \mathbb{I}\ b\ \rightarrow\ a\ (r^{\smile};r)\ b$$

Notice that we have used no interpretation; we have just "moved characters" as prescribed by equation 3.62. Also, you can establish that equation 3.62 is the only translation (out of all laws in the translation table) that applies here. So you can do this translation without even knowing what it means.

For the next step, we can see two laws that may be applied. Law 3.61 can be applied to subexpression $a\ \mathbb{I}\ b$ and $a\ (r^{\smile};r)\ b$ can be transformed using law 3.51. Let us do both transformations simultaneously, yielding the following expression.

$$\forall a, b: \; a = b \;\; \rightarrow \;\; (\exists c: \; a \; r^{\smile} \; c \;\; \wedge \;\; c \; r \; b)$$

Again, the transformation was done purely on the basis of laws. Again, we had no choice but to apply the appropriate laws. This is the case in the next step too, because we can only apply law 3.59:

$$\forall a, b: \; a = b \;\; \rightarrow \;\; (\exists c: \; c \; r \; a \;\; \wedge \;\; c \; r \; b)$$

Now we are done. The entire expression has been rewritten to predicate logic. Considering that $a = b$, this result can be simplified further in predicate logic:

$$\begin{array}{ll} & \forall b: \; b = b \;\; \rightarrow \;\; (\exists c: \; c \; r \; b \;\; \wedge \;\; c \; r \; b) \\ \Leftrightarrow & \qquad\qquad \{\text{predicate logic}\} \\ & \forall b: \; \exists c: \; c \; r \; b \;\; \wedge \;\; c \; r \; b \\ \Leftrightarrow & \qquad\qquad \{\text{simplify}\} \\ & \forall b: \; \exists c: \; c \; r \; b \end{array}$$

The last step is precisely the definition of surjective as we would expect it in predicate logic.

We get the entire derivation by putting all steps together, linking them by the logical connectives from the laws we used (in this case $\Leftrightarrow$ and $\Rightarrow$):

$$\begin{array}{ll} & \forall a, b: \; a \; (\mathbb{I} \; \vdash \; r^{\smile}; r) \; b \\ \Leftrightarrow & \qquad\qquad \{\text{law 3.62}\} \\ & \forall a, b: \; a \; \mathbb{I} \; b \;\; \rightarrow \;\; a \; (r^{\smile}; r) \; b \\ \Leftrightarrow & \qquad\qquad \{\text{law 3.61 and 3.51}\} \\ & \forall a, b: \; a = b \;\; \rightarrow \;\; (\exists c: \; a \; r^{\smile} \; c \;\; \wedge \;\; c \; r \; b) \\ \Leftrightarrow & \qquad\qquad \{\text{law 3.59}\} \\ & \forall a, b: \; a = b \;\; \rightarrow \;\; (\exists c: \; c \; r \; a \;\; \wedge \;\; c \; r \; b) \\ \Rightarrow & \qquad\qquad \{\text{Let } a = b\} \\ & \forall b: \; b = b \;\; \rightarrow \;\; (\exists c: \; c \; r \; b \;\; \wedge \;\; c \; r \; b) \\ \Leftrightarrow & \qquad\qquad \{\text{predicate logic}\} \\ & \forall b: \; \exists c: \; c \; r \; b \;\; \wedge \;\; c \; r \; b \\ \Leftrightarrow & \qquad\qquad \{\text{simplify}\} \\ & \forall b: \; \exists c: \; c \; r \; b \end{array}$$

This derivation yields as a result:

$$\mathbb{I} \; \vdash \; r^{\smile}; r \quad \Rightarrow \quad \forall b: \; \exists c: \; c \; r \; b \tag{3.66}$$

3.11 Exercises

3.11.1 Theorem K and more

This exercise is an example, in which theorem K plays a key role. The elaboration of the exercise shows thoughts and arguments, illustrating an actual attempt to formulate a business rule.

Exercise 3.1. Suppose a mentor gets a bonus if all of his students have completed their year. Propose a rule in terms of the following relations:

$$\begin{aligned} \mathit{mentor} &: \mathit{Student} \rightarrow \mathit{Mentor} && (3.67)\\ \mathit{bonus} &: \mathit{Mentor} \times \mathit{Year} && (3.68)\\ \mathit{completed} &: \mathit{Student} \times \mathit{Year} && (3.69) \end{aligned}$$

The expression s *mentor* m means that student s has m as a mentor. Expression m *bonus* y means that mentor m has been awarded a bonus for year y. Expression s *completed* y means that student s has completed year y.

Let us attempt to answer exercise 3.1. In order to find a rule that describes the situation, we start with the obvious. When a mentor m has reaped a bonus for a year in which he mentored student s, then surely s has completed year y. So:

$$\mathit{mentor}; \mathit{bonus} \vdash \mathit{completed} \qquad (3.70)$$

Although this describes the situation correctly, it is not obvious whether all violations can be caught by this rule. It says nothing about the situation in which the mentor reaped no bonus. In order to find out, let us try to reason the other way around. If a student s of mentor m did not complete year y, there surely is no bonus for the mentor! So we try this:

$$\mathit{mentor}^{\smile}; \overline{\mathit{completed}} \vdash \overline{\mathit{bonus}} \qquad (3.71)$$

According to De Morgan's theorem 'K' however, equations 3.70 and 3.71 are equivalent. So we still have not described the situation in which the mentor got no bonus.

Another attempt to reason 'the other way' might be to reverse the ⊢:

$$\mathit{completed} \vdash \mathit{mentor}; \mathit{bonus} \qquad (3.72)$$

This means: if a student has completed a year, there is a mentor who got a bonus. Obviously, this does not represent the situation correctly, so we reject this attempt.

What if we reverse the $\vdash$ in equation 3.71?

$$\overline{bonus} \vdash mentor^{\smile};\overline{completed} \tag{3.73}$$

This reads: if mentor m did not get a bonus for the class of year y, then m had a student who did not complete year y. This is obviously something we do want. The question remains whether this equation (3.73) differs from equation 3.71

In order to find out whether all violations are covered, we need some inspiration. Let us investigate one specific example: a situation where only *Kersten*, *Joosten*, and *2005* exist:

$$\begin{aligned} mentor &= \{\ \langle Joosten, Kersten\rangle\ \} \\ bonus &= \{\ \} \\ completed &= \{\ \langle Joosten, 2005\rangle\ \} \end{aligned}$$

This situation means that Joosten completed the class of 2005 and had Kersten as a mentor. Kersten, however, received no bonus. Since all students have completed the class of 2005, surely he is entitled to a bonus. This constitutes a violation of the business rule.

This situation satisfies equation 3.70, because relation *bonus* is empty. Equation 3.73 is not satisfied, however, because the left hand side, $\overline{bonus}$, contains an element and the right hand side, $mentor^{\smile};\overline{completed}$, does not. So the example proves that both rules are different, even though we want both 3.71 and 3.73 to serve as business rules. So, let us combine the two rules into one:

$$\begin{aligned} & (mentor^{\smile};\overline{completed} \vdash \overline{bonus})\ \cap\ (\overline{bonus} \vdash mentor^{\smile};\overline{completed}) \\ \Leftrightarrow\quad & \qquad \{\vdash \text{ is antisymmetric (??)}\} \\ & \overline{bonus}\ =\ mentor^{\smile};\overline{completed} \\ \Leftrightarrow\quad & \qquad \{\text{formulate positively}\} \\ & bonus\ =\ \overline{mentor^{\smile};\overline{completed}} \\ \Leftrightarrow\quad & \qquad \{\text{De Morgan}\} \\ & bonus\ =\ \overline{mentor^{\smile}}\ \dagger\ completed \end{aligned}$$

This result describes that a mentor gets a bonus if all students either have completed a year or are not mentored by him. Restated in understandable English: a mentor gets a bonus over a year if all students that are mentored by him have completed the year. The other way around, if all students of a mentor have completed the year, that mentor is entitled to a bonus over that year. This defines precisely when the mentor is entitled to a bonus and when he is not.

3.11.2 For functions, dagger and composition are closely related

The following exercise shows how for functions, the dagger and composition operators are closely related.

Exercise 3.2. Prove $\overline{f} \dagger r \;\Leftrightarrow\; f;r$ for every function f and arbitrary relation r (law 4.9)

First answer, without resorting to predicate logic:

$$\begin{array}{lll}
 & f;r & \\
\Leftrightarrow & & \{\text{introduce the empty relation } \overline{\mathbb{V}} \text{ by law } \textbf{??}\} \\
 & \overline{\mathbb{V}} \cup f;r & \\
\Leftrightarrow & & \{\text{equation 4.1}\} \\
 & \mathbb{V} \vdash f;r & \\
\Leftrightarrow & & \{\text{use law 4.8, using the condition that } f \text{ is a function}\} \\
 & f^{\smile};\mathbb{V} \;\vdash\; r & \\
\Leftrightarrow & & \{\text{theorem K (eqn. 4.11)}\} \\
 & f;\overline{r} \;\vdash\; \overline{\mathbb{V}} & \\
\Leftrightarrow & & \{\text{equation 4.1}\} \\
 & \overline{f;\overline{r}} \;\cup\; \overline{\mathbb{V}} & \\
\Leftrightarrow & & \{\text{De Morgan (eqn. 3.15) and remove } \overline{\mathbb{V}} \text{ by law } \textbf{??}\} \\
 & \overline{f} \dagger \overline{\overline{r}} & \\
\Leftrightarrow & & \{\text{remove double complement (eqn. 3.13)}\} \\
 & \overline{f} \dagger r &
\end{array}$$

Second answer, using predicate logic:

$$\begin{array}{lll}
 & x\ (\overline{f} \dagger r)\ y & \\
\Leftrightarrow & & \{\text{equation 3.56}\} \\
 & \forall z : x\ f\ z\ \rightarrow\ z\ r\ y & \\
\Leftrightarrow & & \{f \text{ is a function}\} \\
 & \forall z : z = f(x)\ \rightarrow\ z\ r\ y & \\
\Leftrightarrow & & \{\text{predicate logic}\} \\
 & f(x)\ r\ y &
\end{array}$$

Likewise, we translate $f;r$

$$\begin{array}{lll}
 & x\ (f;r)\ y & \\
\Leftrightarrow & & \{\text{equation 3.51}\} \\
 & \exists z : x\ f\ z\ \wedge\ z\ r\ y & \\
\Leftrightarrow & & \{f \text{ is a function}\} \\
 & \exists z : z = f(x)\ \wedge\ z\ r\ y & \\
\Leftrightarrow & & \{\text{predicate logic}\} \\
 & f(x)\ r\ y &
\end{array}$$

Together, these derivations prove law 4.9

These derivations both show that $f;r \;\Leftrightarrow\; \overline{f} \dagger r$, but only if f is a function. This proves law 4.9, as required.

3.11.3 Further exercises

These exercises are left for the reader to do.

Exercise 3.3. Using only the laws of relation algebra, prove $r \dagger \overline{f^{\smile}} \Leftrightarrow r; f^{\smile}$ for every function f and relation r (law 4.10)

MOVE TO NEXT CHAPTER

As a side remark, notice the difference between operations on sets, that produce new sets from existing ones, and assertions about sets that evaluate either to true or false. There are close links between operations and assertions. For instance, if we claim that "the union operation is associative", we are saying that the assertion $(A \cup B) \cup C = A \cup (B \cup C)$ is true for arbitrary sets A, B and C. In the assertion, the union operation figures four times,

Chapter 4

Rules

The real advantages of using a relation algebra become clear when you use the formalism to express business rules clearly but rigorously. This chapter defines and explains what business rules are, and it helps you to read and write rules fluently.

First, in section **??** we define the notion of rule, within the framework of Relation Algebra theory explained in the previous chapter. The next section **??** discusses rule violations: what is it, what needs to be done to detect violations, and what kinds of business rules cannot be violated at all. Next, section 4.5 provides a procedure to translate a formula into natural language, which helps you to better understand the rules to yourself and to explain them to others. Then we give a number of examples how to use laws (section 3.6) in order to manipulate with rules and relations. This enables you to transforming one way to express a rule into another expression, without a change of meaning, in order to find the best way to formulate and explain the rules. Finally, we discuss some aspects of rule design: how to go about, and what pitfalls to avoid.

4.1 Rule definition

4.1.1 Business Rule

In all simplicity: a business rule is a guide for conduct. Chapter 2 gave a more precise formulation:

DEFINITION Rule

A *rule* is a verifiable statement that some stakeholders intend to keep true within the context of that statement.

The business rule is nothing more than ”just” some regulation or principle that governs the conduct (business processes and/or human activities) within a particular area (or ”context”) of the business. The rule provides guidance, knowledge, descriptions or prescriptions regarding potential, allowed, or prohibited actions. So the key words are: governance and guidance.

Not the IT department, but the business itself is responsible for its rules: to have them laid down, implemented, enacted, reviseed, or in the end, to discontinue them. As the business sees fit. If a rule cannot be managed in this way by the business, then it should not be considered a business rule. It can still be a rule of course, for instance 1+1=2, or (-r) ∪ (-s) = -(r ∩ s), or the laws of gravity. These are not considered business rules, because there is no interest to maintain them, no one is interested. And these rules are completely rigid, there is no flexibility in them. Such rules do not provide real guidance, as they are unavoidable.

To be a true ”business” rule, the business must have jurisdiction in formulating and maintaining the particular rule.

The essence of a business rule is in the guidance it provides for the business. If we have a rule, and write it down in different ways but always keeping the same semantics and guidance, then we still have the same rule. We can write the same rule in one natural language or another (even Greek), or write it in some semi-formal language (such as RuleSpeak), in relation algebra or in any other way. Nor does it matter if the rule is implemented in some information systems, in workflow procedures, or if it remains implicit knowledge that workers stick by. No matter how the rule is phrased or captured, the rule is there and the business should adhere by it.

Business people often know their rules and talk about them in rather offhand ways. But for rule engineers, analysts and developers, such liberty is unacceptable. Precision in rule formulation is adamant, and a strict and formal language is called for.

4.1.2 Rules in Relation Algebra

assertion

An *assertion* in Relation Algebra is a formula that compares two sets, or more in particular: two relations, to produce either a ”true” or ”false” answer. Like this:

$R \vdash S$ or $R = S$ (or of course $R \vdash S$, which is just the first formula with the letters R and S interchanged).

The trick here is that both relations, R and S, may not be so simple. They are allowed to involve any number of relations and operators, and the assertion may actually be very complex to understand.

An assertion about inclusion $R \vdash S$ or equality $R = S$ may evaluate sometimes to a logical "true", and to "false" at other times, depending on the contents of the relations R and S at the time of evaluation. There is no option "undecided", though. Whether the assertion is true or not, must be decided from the instances recorded for the base relations involved in R and S. An assertion that, regardless of the information stored in the relation extensions, always turns out to be "true" (or "false", for that matter), is not very interesting, and we ignore it. For instance, if we have two relations p and q, if R is set to $\overline{p} \cup \overline{q}$ and S is set to $\overline{p \cap q}$, then we know in advance that $R = S$: this is a law of Relation Algebra. This is not a rule under business jurisdiction.

business rule

A *business rule* is then an assertion that, according to the business, ought to evaluate to true, always. For any and all content of all relations within context. It may turn out that the assertion at some point in time is violated, which means that work must be done to remedy this problem.

Observe that multiplicity constraints, introduced in the previous chapter, are rules; we will return to this shortly.

4.1.3 Expressions in Relation Algebra

expression

An *expression* in Relation Algebra produces new relations from existing ones, so an expression is not an assertion.

There are close links between operations and assertions. For instance, if we claim that "the union operation is associative", we are saying that the assertion (A ∪ B) ∪ C = A ∪ (B ∪ C) is true for arbitrary sets A, B and C. In the assertion, the union operation figures four times,

In fact, any assertion can be turned into an expression, and this is true for business rules of the type $R \vdash S$ in particular. The assertion states that every tuple in R must belong to S also, or: no tuple should exist in R that is not S. In other words: the relation R/S must be empty at all times, any tuple in R/S means that there is a violation of the business rule. So apparently, we can match the assertion $R \vdash S$ with the expression R/S. Likewise, the business rule $R = S$ will be matched to the expression $R/S \cup S/R$, called the *symmetric difference* of R and S.

symmetric difference

4.1.4 Categories of rules

Business rules come in a great variety, and not all categories allow to be expressed as assertions in Relation Algebra. The many categories that are not readily captured in Relation Algebra include:

- statistical rules: at least 80% of all flights must leave on time.
- rules of thumb, based on past experience: our employees are always between the age of 15 and 70.
- arithmetical rules and calculations: three strikes and you're out, or: delivery will cost 15% of the net value for postage and package with a maximum of $ 20.
- comparisons of physical values: blood acidity should remain under pH 7, or: date of delivery is between 7 and 15 days after date of order acceptance.

and we do not even claim that this list is exhaustive. But other categories, and indeed a large variety may be captured in Relation Algebra, although some ingenuity is sometimes called for.

Unconstrained Conceptual Model

In theory, we could envision a Conceptual Model that is stripped of all rules and mulitplicity requirements, sometimes referred to as an *Unconstrained Conceptual Model* ¿LITERATUUR REFERENTIE¡. Such a model can be populated with any and all kinds of instances and, provided that entity integrity and referential integrity are complied with, no violations would ever occur. The idea now is that we may impose subsequent business rules on this Unconstrained Conceptual Model, and resolve the violations that emerge found for each rule as it is being added. In a number of steps, the model would have all its rules in place. This approach would provide the designer with an insight of the true impact of each separate rule. We consider this approach to be rather theoretical and impracticable: it will never work out in practice. Still, it is worthwhile to consider the unconstrained Conceptual Model and think about the impact of each rule as it is being imposed upon the model. Nonetheless, by adding the simple rules of multiplicity first, before adding the more complex rules based on assertions on the model, we do use that very approach to some extend.

The Conceptual Model plus the rules defined for it, constitutes a coherent set of descriptive business rules and for us, it is the set of rules. It captures the information that we want to store about the ongoing business. Within this context, within the approach we describe in this book, exactly four categories of business rules are recognized and no other rules can exist:

- concepts: Contract, Customer-order, Back-order.
- relations: Customer places Customer-order, Back-order forwarded-by Special-Delivery.
- multiplicity constraints on relations, being rules for a single relation: Customer has at most 1 Name, Special-Delivery contains at least 1 Back-order.

  - invariant rules involving more than one relation: if Customer places Customer-order, then the Customer-order requires full Payment and the Customer must make the Payment.

Of course, this categorization can be disputed.

For one, some authors classify both concepts and relations as "definitional rules". Some practicable definition is available that outlines exactly which phenomena seen in the realworld can and should be recorded as an instance of the concept or relation. Although mistakes can be made in recording the information (a back order accidently being recorded as customer order), recording the data as such can never violate a business rule in a direct way. Some indirect violation may be signalled, though: the back order, being recorded as a customer order, lacks an obligatory delivery date. Or an email address may be recorded where the house address of an employee was required. Again, this kind of mistake calls for checking of recorded data against the real-world and is by its very nature out-of-scope for the information system. The mistake does not directly cause a relational assertion to evaluate to "false".

For another, the four categories are not strictly orthogonal: it is possible to rewrite a rule and move it from one of the four categories to another. The point is that our four categories are based, not on business semantics, but on modelling semantics. We will come back to this point later in section **??**.

A further objection may be that our classification is concerned with static rules, while many business rules are dynamic in nature. A familiar example of static rule is that a person must be classified as "married" if (s)he has a spouse; and it comes with a dynamic rule that the classification "married" may never change back to "unmarried", nor may "unmarried" change directly to the "divorced" status. It is certainly possible to incorporate static and dynamic rules in one Conceptual Model, but like we said: some ingenuity may be called for.

A final word to proponents of the business process approach. Business processes are commonly described as a series of steps (activities) to produce a desired outcome for some trigger or business event. To coordinate the corerct flow of steps, we need sequencing rules: if step 1 is finished, then execute either step 2a or step 2b. Or: if steps 61 and 34 and 27 all have finished, then execute step 63. The sequence on the process is partly dictated by business requirements ("delivery is done after payment is received") and partly by design choices. The process designer may enhance process efficiency by organizing the steps in a particular way ("perform order picking is done from largest to smallest objects to ease packaging"). Efficiency rules, the latter kind, are genuine business rules too. The point however is that their origins and their life cycle are different from the original business requirements. The rule designer must be aware of such differences, even though the differences cannot be expressed in the rules as such. Because true business requirements are much less volatile in comparison to other design choices, they are sometimes named *Invariant-rules*.

Invariant-rules

4.2 Laws for rule expressions, or: equivalence of rules

The previous chapter already gave you some laws that enable you to write a formula in various ways.

4.2.1 Laws at meta-level

Another useful law is the following:

$$p \vdash r \quad = \quad \overline{p} \cup r \tag{4.1}$$

It shows how to remove an inclusion operator ($\vdash$) from any equation. Of similar use is a law to replace an equality by two subsets.

$$(p = r) \quad = \quad p \vdash r \cap r \vdash p \tag{4.2}$$

Two relations are equal means that the contents of one is contained in the other, and vice versa.

4.2.2 Distribution laws in compositions

When working with compositions of relations, it often happens that a set operator such as union, intersection or implication appears somewhere in the formula.

The following law shows how the union operator can be removed from compositions of relations.

$$p;(q \cup r);s \quad = \quad p;q;s \cup p;r;s \tag{4.3}$$

¿ELDERS¡ It would be nice if the same kind of distribution would hold for the intersect operator, $\cap$, but alas not: distribution does not apply in general for intersections. Only for a univalent p and injective s can we prove this law:

$$\begin{array}{lrcl}\text{if } s \text{ is injective} & (q \cap r); s & = & q; s \cap r; s \qquad (4.4)\\ \text{if } p \text{ is univalent} & p; (q \cap r) & = & p; q \cap p; r \qquad (4.5)\end{array}$$

Another way to formulate these laws in an unconditional way is:

$$p; (q \cap r); s \ \vdash \ p; q; s \cap p; r; s \qquad (4.6)$$

¿ELDERS¡ Two other useful laws allow you to move relations across the $\vdash$ symbol. However, this law is conditional for functions (i.e. univalent and total relations).

$$\begin{array}{lrcl}\text{if } f \text{ is a function} & r \vdash s; f^{\smile} & = & r; f \vdash s \qquad (4.7)\\ \text{if } f \text{ is a function} & r \vdash f; s & = & f^{\smile}; r \vdash s \qquad (4.8)\end{array}$$

Functions are also useful to replace † in your formulas by ; operators.

$$\begin{array}{lrcl}\text{if } f \text{ is a function} & \overline{f} \dagger r & = & f; r \qquad (4.9)\\ \text{if } f \text{ is a function} & r \dagger \overline{f^{\smile}} & = & r; f^{\smile} \qquad (4.10)\end{array}$$

4.2.3 Theorem K

Earlier, we discussed several laws colelctively known as "De Morgan laws". A less well known but equally important law about relations found by De Morgan was named "Theorem K".

$$\begin{array}{lrcl} & p; q & \vdash & \overline{r}\\ = & & & \\ & p^{\smile}; r & \vdash & \overline{q}\\ = & & & \\ & r; q^{\smile} & \vdash & \overline{p}\end{array} \qquad (4.11)$$

All three assertions are equivalent to one another. This theorem K too is also useful for "moving around" the complement operator in your formulas.

The use of theorem K and other De Morgan's laws are illustrated in an exercise at the end of this chapter. The exercise 3.1 and its elaboration illustrate some of the thoughts you might have when analyzing real life situations. If you are new to business rules, we suggest you try to replay this example carefully, in order to re-live the arguments and thoughts described.

4.3 Rule violations

behavioural rules

Business rules are assertions that ought to always evaluate to "true". Business rules that, within the context of a Conceptual Model, can be violated, are called *behavioural rules.*

structural rules

The opposite is *structural rules* or definitional rules. Previously we explained how it is impossible for a definitional rule to be violated. Definitions are captured in the concepts and relations of the Conceptual Model. Although there may be some apparent violation in the recorded data, this is due to some error or mistake. To check for mistakes and to correct them, we need to validate the recorded data with the corresponding real-world phenomena. This will usually cause the recorded data to be altered so that they reflect the true situation in the real world, only rarely would we change the real world to match the data. This transgresses the context of the Conceptual Model, thus we can say that, within the context of the Conceptual Model, violation of a definition is impossible. The business workers may challenge the structure of a Conceptual Model or the designers may change it, but as long as the model is in place, it lays down the structure and the structural rules that come with it, and there is no way to work around it.

4.3.1 Dealing with rule violations

Business rules are assertions of the form $R \vdash S$ that ought to always evaluate to "true". To determine whether the rule is violated, you just have to calculate the difference R/S which ought to be the empty set, always.

In an automated information system, the relational equation for all rules can be checked directly by computer. If violation of a rule is detected, several courses of action may be taken by the information system:

- Immediate enforcement: the transaction that attempts to create the violation is cancelled at runtime,
- Delayed enforcement: the violation is permitted, but it is signalled on a "to-do" worklist at runtime, or
- No enforcement: violations are simply ignored at runtime, and instead, periodic consistency checks are conducted that produce extensive worklists.

Which course of action is most appropriate depends both on business preferences and (in)capabilities f the implemented design. It can vary per rule: some rules are so important that immediate action is required to prevent any violation at all. Other rules may considered to be not very important, and periodic checking is enough. The author is aware of one example where violations were ignored for a number of years. Management, confronted with the backlog, decided not to follow up on the many violations, but to just accept them all as one-time exemptions, resulting in a loss of several millions of euros.

The ”immediate enforcement” behaviour is similar in appearance to a definitional rule: violation is impossible. But it works quite differently. Violating a definitional rule is impossible because the structure of the Conceptual Model is incapable to record data not compliant with the rule. Violating an operational rule that by the designers' choice has immediate enforcement is made impossible only because there is active checking for violations, which presumably is done fully automatic. Moreover, preventive action is taken before a violation materializes, probably by the information system as well. In database management systems, this mechanism is known as a rollback.

A business rule with the ”delayed enforcement” behaviour is associated with a ”to-do” worklist. However, it does not say what must be done: a violation may be remedied in many ways. For instance, the violation may be authorized as an exeption by a supervisor. Or some corrective action may be taken by an employee. Or, the information system may wait three days and, if the violation still exists, cancel the original transaction that caused it.

In fact: what to do when a business rule is violated, is guided by other business rules. As stated by the Business Rules Manifesto:

¿business rules must be separated from how they are maintained / repaired¡

This rules-approach how to resolve violations differs markedly from the business process approach. In a business process, violations are either within the scope of the process and resolving them is part of the process: an Exam paper may only be entered by a Student registrered for the Exam; if not, the Student on-the-spot registration is possible. Or the violations are deemed beyond the scope of the process and it is ignored in the process design: an Exam paper may only be entered by a Student with a valid Student-number.

4.4 Multiplicity rules

In the previous chapter we established the various multiplicity properties for a single relation. It was also stated that a multiplicity property can be regarded as a constraint, a business rule that applies to a single relation. In this section we show how all four multiplicity properties can be written as assertions in relation algebra: they are indeed business rules according to our book.

4.4.1 Multiplicity properties as relational expressions

Consider a relation with signature r :: A x B. We have

- r univalent is $r^{\smile}$;r ⊢ Id [B]
- r injective is r;$r^{\smile}$ ⊢ Id [A]
- r total is Id [A] ⊢ r;$r^{\smile}$
- r surjective is Id [B] ⊢ $r^{\smile}$;r

TABLE:

To illustrate the convenience of these formulas, we claim that

$$\begin{aligned} \mathit{injective}(r) &\Leftrightarrow \mathit{univalent}(r^{\smile}) && (4.12)\\ \mathit{surjective}(r) &\Leftrightarrow \mathit{total}(r^{\smile}) && (4.13) \end{aligned}$$

Using the formulas, it is remarkedly easy to prove that the inverse of an injective relation is univalent. Let r be injective, then according to the table above we have:

$$\begin{array}{ll} r;r^{\smile} \ \vdash \ \mathbb{I} & \\ = & \{r = r^{\smile\smile} \text{ by law 3.7}\} \\ r^{\smile\smile};r^{\smile} \ \vdash \ \mathbb{I} & \end{array}$$

In a few lines, this derivation shows that $r^{\smile}$ is univalent (definition **??**). Applying the conversion again, it is also evident that the inverse of a univalent relation is injective. We leave it to the reader to show in a similar fashion that the inverse of a total relation is surjective, and vice versa.

The consequence for you as a designer is that when you outline a Conceptual Model, you have a choice when and where to put the multiplicity properties. You can simply incorporate the property in the relation declaration, or you can write it as a distinct business rule, completely separated from the declaration.

Exercise 4.1. Check that indeed the formula for *total* is correct, by proving that $\mathbb{I} \ \vdash \ r;r^{\smile}$ translates to $\forall a : \exists b : a \ r \ b$

Exercise 4.2. Do the same for *injective*, i.e. prove that $r;r^{\smile} \ \vdash \ \mathbb{I}$ translates to $\forall a, b, c : a \ r \ c \wedge b \ r \ c \ \Rightarrow \ a = b$

4.5 Rules in natural language

Learning to understand a relational formula on paper is essential for a rule designer. This section is a brief introduction how to translate a relational formula into natural language. You should know what the relational symbols in the formulas mean, but more importantly, you must learn to read the natural language it stands for. For example, expression $exam;\overline{class}$ can stand for the sentence: ”an exam was taken by some student, and that student did not attend class”. Being able to translate a formula is importnat when you explain your formulas to business workers or collegues, when you verify whether a rule is a correct representation of a business requirement, or simply whenever you check your own work.

We illustrate how the property of univalence (**??**) may be translated into natural language. This demonstration is rather extensive, with the purpose that you clearly understand each step. In the end, is is expected that you do do the translations without intervening steps.

4.5.1 Example translation to natural language

As a first example, let us begin with the relational formula for the univalence of a relation r :: A x B, and translate that. The first part is a rather mechanical ”fill in the dots” exercise:

$r^\smile;r \ \vdash\ \mathbb{I}$

means *(translate the implication, by literally copying text for implication from table 8.3)*

For every $x \in B$ and $y \in B$, if $x\ (r^\smile;r)\ y$ then $x\ \mathbb{I}\ y$.

means *(translate $x\ (r^\smile;r)\ y$, by the translation for composition.)*

For every x and y, if there exists z such that $x\ r^\smile\ z$ and $z\ r\ y$, then x equals y.

means *(translate $x\ r^\smile\ z$, by the translation of conversion.)*

For every $x \in B$ and $y \in B$, if there exists $z \in A$ such that $z\ r\ x$ and $z\ r\ y$, then x equals y.

Of course, all relations in a rule also have a definition, a meaning in natural language. Although that meaning is not relevant for mathematical manipulation of the rule, it is very relevant for its translation in to natural language. Suppose for now that our univalent relation r means that certain persons (instances of set A) can be reached at some telephone numbers (instances of the set B). Then the translation can now be interpreted to become:

For every $x \in B$ and $y \in B$, if there exists $z \in A$ such that person z has telephone number x and that same person z has telephone number y, then x equals y.

means *(reformulate)*
For every x, y, and z, if person z has telephone number x and has telephone number y then x and y are the same number.

means *(reformulate)*
Every person can be reached at at most one telephone number.

Notice that the last few steps are of a different nature than the first series of steps. In the last few steps, we actually molded the language for human use. That requires human understanding of the sentence. In practice, some of the steps will often be quite mechanical, but in the end, you have to use your wits to come up with a simple, natural interpretation.

4.5.2 Translation procedure

A simple procedure for translating relation algebra expressions to natural language comprises four steps; steps that we followed in the example above:

1. translate each operator according to the translation table; (do not forget to write ‘for every x, y’ when translating $\vdash$) and repeat this until all operators have been translated.
2. translate each relation according to its popular meaning;
3. attempt simplifications for the verbose text;
4. put into decent english.

Table 8.3 contains the necessary translation laws, using the operators from chapter **??**. The four-step procedure is intended to help you translate any relation to its natural language meaning. By following it, you can obtain a proper translation for any of your rules, and constitute the relationship between the formalism and what is ’out there’ in the real world. Which, by its very nature, is non-automatable.

Let us elaborate another example. Suppose an insurance company distributes work on insurance policies among employees in the form of work packages. Consider the following three relations:

$$\begin{array}{lll} \mathit{policyNr} & : \mathit{Workpackage} & \rightarrow \mathit{Policy} \\ \mathit{product} & : \mathit{Workpackage} & \rightarrow \mathit{Product} \\ \mathit{type} & : \mathit{Policy} & \rightarrow \mathit{Product} \end{array}$$

The interpretation of each relation is important for translating it to natural language, so we must know the of each relation within the context of the insurance company. Here it is:

The expression *w policyNr n* means that work pack
icy number *n*,and for instance, work package “`asse`
about policy number “`NL084728839`”.
The expression *w product p* means that work pac
procedure for product *p*. For example, the work pack
`3314`” follows the procedure for product “`car insu`
The expression *n type p* means that the policy numb
product of type *p*. An example is policy number “`N`
is a product of type “`car insurance`”.

Finally, assume that the following rule holds:

$$policyNr \ \vdash \ product ; type^{\smile}$$

A systematic translation (table 4.5.2) is obtained if we follow the procedure meticulously.

Exercising through a fair number of translations carefully is the best way to gain experience and become proficient at reading formulas. At first, you will often refer back to the translation table but soon, you will get the hang of it and you can do without the translation table. You will find that it will help you to find and correct errors in formulas -errors made by others. Or even by yourself.

A final hint: use the conventions that govern the precedence of relational operators given in the previous chapter. This may save you some brackets and thus make formulas (and translations) look better.

4.5.3 Flexibility: changing the rules

flexible wrt rules, stable wrt Conceptual Model

4.6 Summary

Invariant rules can be well defined and can be checked for compliance with the business requirements.

We claim that the use of Ampersand will deliver high-quality rule-based designs.

4.6.1 Comparison to Event-Condition-Action approach

4.6.2 Comparison to Database Triggers and Stored Procedures approach

4.6.3 Comparison to Petrinet Paradigm

4.6.4 Limitations of rules in Relation Algebra

Relation Algebra is a powerful tool to use with business rules that are invariant in nature, and that focus on existence of things, on relations having correct extensions. The business rules serve to determine correctness in context: the rules say how to combine extensions from several relations and find violations.

However, you cannot do arithmetics in Relation Algebra. It is impossible to add up the cost for various items and calculate the bill total. Nor can we make comparisons: one car is more expensive than another, the number of students in a class must always be between 4 and 140. Similar problems arise when a business requires that a password always contains at least 8 tokens, including at least 1 uppercase letter and 2 non-alphabetic symbols.

Spatial or temporal knowledge is also not part of Relation Algebra. Although not impossible, it is very hard to achieve good reasoning with adjacent geographical areas, distances, or time frames.

These drawbacks are inherent to the theory of Relation Algebra.

4.7 Exercises for this chapter

name	notation	translation in natural language
existence	$c \in C$	there is a C, which is represented by c (sct. **??**). Also: c is an instance of C.
fact	$x\ r\ y$	there is a pair between x and y in relation r. Also: the pair $\langle x, y\rangle$ is an element of r, $(\langle x, y\rangle \in r)$.
declaration	$r : A{\times}B$	There is a relation r with source A and target B. (eqn. **??**)
implication	$r \vdash s$	For every x and y, if $x\ r\ y$ then $x\ s\ y$ (sct. **??**). Alternatively: $x\ r\ y$ implies $x\ s\ y$. Alternatively: r is included in s or s includes r.
equality	$r = s$	For every x and y, $x\ r\ y$ is equal to $x\ s\ y$
cartesian product	$A \times B$	all pairs $\langle x, y\rangle$ with $x \in A$ and $y \in B$
complement	$\overline{r}$	all pairs not in r. Also: all pairs $\langle a, b\rangle$ for which $x\ r\ y$ is not true. (eqn. **??**)
conversion	$r^{\smile}$	all pairs $\langle y, x\rangle$ for which $x\ r\ y$. (eqn. **??**)
union	$r \cup s$	$x(r \cup s)y$ means that $x\ r\ y$ or $x\ s\ y$. (eqn. **??**)
intersection	$r \cap s$	$x(r \cap s)y$ means that $x\ r\ y$ and $x\ s\ y$. (eqn. **??**)
composition	$r; s$	$x(r; s)y$ means that there is a z such that $x\ r\ z$ and $z\ s\ y$. (eqn. **??**)
relative addition	$r \dagger s$	$x(r \dagger s)y$ means that for all z, $x\ r\ z$ or $z\ s\ y$ is true. (eqn. **??**)
	$\overline{r} \dagger s$	$x(\overline{r} \dagger s)y$ means that for all z, $x\ r\ z$ implies $z\ s\ y$. (eqn. **??**)
	$r \dagger \overline{s}$	$x(r \dagger \overline{s})y$ means that for all z, $z\ s\ y$ implies $x\ r\ z$. (eqn. **??**)
identity	$\mathbb{I}$	equals. Also: $a\ \mathbb{I}\ b$ means $a = b$. (eqn. **??**)

Table 4.1: Translating rules to natural language

$\mathit{policyNr} \ \vdash \ \mathit{product}; \mathit{type}^{\smile}.$

Translate $\vdash$ to natural language (implication, table 8.3)

> For each work package w and every policy n:
> If w *policyNr* n then w $(\mathit{product}; \mathit{type}^{\smile})$ n.

Translate ; to natural language (composition, table 8.3)

> For each work package w and every policy n:
> If w *policyNr* n then there exists a product type t such that w *product* t and t $(\mathit{type}^{\smile})$ n.

Translate $^{\smile}$ to natural language (conversion, table 8.3)

> For each work package w and every policy n:
> If w *policyNr* n then there exists a product type t such that w *product* t and n *type* t.

Translate each relation to natural language

> For each work package w and every policy n:
> If w is about policy number n then there exists a product type t such that w follows the rules of product type t and n is of type t.

since *type* is a function, there is precisely one product type, which is the type of policy n.

> For each work package w and every policy n:
> If w is about policy number n then w follows the rules of the product type of n.

In english, we leave out the "For each work ..." part and use an article instead.

> If a work package w is about policy number n then w follows the rules of the product type of n.

Work on the english phrasing and make sure your audience understands.

> Any work package must follow the rules of the product. These rules are defined through the policy number of that work package, because each policy has a unique product type.

Chapter 5

Design Considerations

The previous chapter outlined how to specify business rules. But that is not all: as a designer, you are expected to specify the right business rules. So what must you pay attention to? How do you create a high-quality design? How do you establish a large Conceptual Model and accompanying Business Rules in such a way that you meet the quality demands?

There are many design considerations and rules of thumb that you can and should know about. This chapter outlines some of them. In practice, you will discover their merits.

But beware: there is no golden bullet. No matter how many books and examples of good design you look at, there is always more to it. The best, and perhaps the only way to learn good design is by doing it. Practice makes perfect.

- kies je voor specialisatie of generalisatie in je model;
- check per cycle of je regels compleet zijn; knip te-lange regels in semantisch zinvolle brokken;
- how to spot redundancies and how to deal with them

5.0.1 Capturing a binary property

Earlier, we saw how a homogeneous relation $p_{[A,A]}$ that is symmetric and asymmetric captures a binary property of the elements of the source concept A. The homogeneous relation may be a fine way to model the property, but it is worthwhile to consider another way to capture it in your model.

For this, let us consider a new set with just two elements: H = { “yes” , “no” }. Now, create a new relation $p^*_{[A,H]}$ in the following way: for all a, tuple (a, “yes”) is in p^* if and only if (a, a) is in p. for all a, tuple (a, “no”) is in p^* if and only if (a, a) is not in p.

Actually, p* is a function, and it is easy to see how the relation p can be replaced by p* without loss of meaning. The point here is that p* is much easier to understand. Moreover, the function p* can be extended, for instance to cover a situation with a third option, e.g. where the yes or no hasn’t been decided yet. We only need to extend the target set H to become { “yes”, “no”, “undecided” }. That is all: from now on, p* can also contain tuples which are undecided. Now try to alter the relation p in such a way that the third option can be recorded!

Populaties zijn de kern voor het begrijpen van een bedrijfsregel. Pas als ik snap wat een regel doet op de populatie, kan ik de regel zelf snappen. En omgekeerd: zolang ik niet snap wat een regel doet op een populatie, kan ik de regel niet snappen.

5.1 Checking your input

5.1.1 Checking your concepts

– is every concept formulated as in singular, not in plural?

– classification: does the definition allow us to clearly distinguishe the instances of this concept from things that are n=ot instances of this concept ?

– discrimination: does the definition allow us to clearly distinguish one instance of this concept from other instances ?

– identification: does the definition allow us to recognize the same instance of the concept at any future time ?

– unmix your concepts : bestelling isnot levering, verzoek isnot respons, vraag isnot answer, naamvan isnot het ding zelf

(gemiste) generalisaties, cardinaliteiten, TYPE-auto verward met AUTO TEMPORAL problemen aparte sectie aan wijden

5.1.2 Overall checks

– does the work “flow” ?

– Euler cyclomatic number OK ?

- cycle lengths acceptable? length 3 is fine (and easy), length 6 is high, anything higher must be regarded with suspicion (but can be perfectly right)
- for simple rules, did you consider formulating as a simple rule or as a self-composition (like r; r)
- no redundancy ? In particular regarding simple rules ? For instance: $r \vdash \overline{s}$ as well as $s \vdash r$ (both express the exact same fact; which can also be stated in still another form $r \cap s = \emptyset$)
- no derived concepts ? "a list is produced that contains all facts about..."
- rules are as-simple-as-possible (but no simpler)
- NO meta-switch !!

Pitfalls ... IF the required data is filled in THEN the form is accepted OR ... IF the form is accepted THEN at least the required data is filled in

5.2 Writing a rule

Once a Conceptual Model and its applicable business rules are agreed upon, you will find that the rules will be reused in many different situations. So, you may expect that in the future, you will need to examine, understand and adjust an existing set of rules much more often than you will need to write a new one.

Still, you need to begin with conceiving new rules. This section outlines a procedure how to devise a single rule. The procedure consists of four steps: determine concepts, determine relations, invent a rule, and validate. We start off with an example of this procedure that goes through all four steps. Next, we explain the details of this design procedure. Of course, you need to exercise in order to become familiar with the procedure. Practice does the rest, turning you into a proficient rule designer.

Primary rules concern the services that are produced for customers, whereas secondary rules govern the way how to perform production and procedures correctly. Authorizations, read and write permissions, prioritizations, sequencing of steps etc are secondary. In our opinion, it is smart to aim for a strict separation between these two. Do not create Conceptual Model that mixes primary rules and secondary rules.

5.2.1 Rule design example

Let us look at an example from the immigration legislation in the Netherlands.

> Article 14 sub 2 of the Dutch immigration law (Vreemdelingenwet) states:
>
> A residence permit for limited duration is issued under conditions that are related to the purpose of stay.

In order to turn this text into a rule, we must formalize it. And we must validate it: does our formalization represent the text correctly? We proceed by doing four consecutive steps:

1. Which concepts are involved in this rule?
2. Which relations are involved in this rule?
3. Conceive a business rule.
4. Validate the rule: does it capture the original text of the article?

First, the text mentions three concepts, namely (notice that we use singular, not pluras names):

- Residence permit for limited duration, abbreviated RPLD
- Condition
- Purpose of stay

Next, we look for relations among these concepts. The article states that conditions apply for the residence permits, so we conceive a relation *issued_under* with source *RLPD* and target *Condition*.

If, for example, residence permit *NL_45* was issued because the permit requestor was employed somewhere, then the relation *issued_under* contains a tuple such as (”*has_labour_contract*”, “*NL_45*”).

Next, a relation between conditions and a purpose of stay is mentioned, so we have *related_to* from *Condition* to *PurposeOfStay*. A pair in this relation might be for instance the tuple (”*has_labour_contract*”, “*employment*”), or (”*is_over_21*”, “*employment*”).

Finally, residence permits are related to purposes of stay. As the phrase “the purpose of stay” is in singular, we gather that each residence permit is related to exactly one purpose of stay, so the relation has from RPLD to PurposeOfStay is injective.

In the next step, we "invent" a business rule by trying to rephrase the law in terms of our relations. Taking the point of view of a single residence permit for limited duration, we know that it should mention all applicable conditions. And that some purpose of stay exists. That is to say: if there exists a purpose of stay p, and that purpose of stay requires the set of conditions $\{c1, c2, c3\}$, then the residence permit is issued under all those conditions. Or we could say it like this: for every permit *rlpd* and every condition c, if there exists (at least) one purpose of stay, such that *rlpd* has the purpose p and c is related to p, then the permit is issued under that condition. Restating this as an assertion in Relation Algebra:

$$\mathit{related\_to} ; \mathit{has}^{\smile} \;\vdash\; \mathit{issued\_under} \tag{5.1}$$

Finally, we must check whether this rule captures the intention of the article. We translate it back to natural language using the procedure explained in section **??**. If a condition c is related to purpose of stay p, and p is the purpose of stay of r, then c is required for residence permit r, the permit is issued under that condition. Which matches with the original phrasing "A residence permit for limited duration is issued under conditions that are related to the purpose of stay." This last step is validation: it requires your own judgment both as a speaker of natural language and as expert with business knowledge.

Now let us revisit the four steps in some more detail.

5.2.2 Design considerations for concepts

In the first step you determine which concepts we need to include in our Conceptual Model. Remember that concepts represent the kind of things encountered in the real world; for each concept, its instances must correspond to actual, individual things that exist in the business environment, that can be created, altered, destroyed.

(1) WHAT CONCEPTS

So in the example, we can visualize a "residence permit for limited duration" as an real piece of paper. Also, "purpose of stay" is a concept. This concept however is not something physical or tangible, but still, over a hundred different purposes of stay are mentioned in an annex to the immigration law ("Vreemdelingenbesluit 2000"). And you can easily imagine new ones being made or obsolete ones being removed. Likewise, the concept "condition" is abstract. Here too, you can imagine how legislators may introduce new conditions or discard obsolete ones. All concepts share the property that individual instances can be pointed out, and they can be created, altered, or deleted.

The core question that a designer needs to answer when considering a concept is: how can a single instance of my new concept be identified? This question actually has two parts. The first part is: how to recognize an instance of your concept "in the wild". Here we have a residence permit, but that other thing is not a residence permit (but a building permit, or a parking ticket, or an application for a residence permit). Once you have restricted yourselves to things of one kind, the second part is: how can you identify and discriminate one individual element from aming its peers. Here we have one condition, there is another condition and that is yet another condition.

A sound definition that clearly explains both parts is absolutely necessary. It must specify how to know that some real world thing is an instance of the concept. And how to distinguish and identify one instance from the other, not only today but also tomorrow.

So now, if you wonder whether you would designate "joy" as a concept, try to answer the question if you want to create new joys or discard obsolete ones.

(2) GRANULARITY OF CONCEPTS

A further design consideration concerns the level of detail, an issue that is sometimes called granularity or specificity. You might argue that "residence permit for limited duration" is a concept, but the next man might argue that "residence permit" is the true concept, and you are only looking at one special kind of residence permits. For instance, there may be residence permits for unilimited duration. Others might argue that there are permits of different kinds, and residence permits are just one form of permit. Still another might come and say: "No, these things are all documents".

These arguments are correct, and the rule of thumb is: choose your concepts as specific as possible, depending on the business context that applies for your design. In the context of Dutch immigration law, "residence permit for limited duration" is just fine. However if other types of (residence) permit would also be within scope, then a broader perspective is called for.

There are at least two acceptable ways to resolve the issue. One solution is to employ one general concept, and then distinguish between the various subconcept by means of an is-a property (we come to that later). Another approach is to just use distinct concepts, which is probably better only if the concepts are not related in any way.

For example: birds and monkeys are both animals, but they are certainly distinct: no bird will ever become a monkey, no monkey will ever be a bird. So apparently, two different concepts may be used, “bird” and “monkey”. However, from the point of view of the zoo keeper, both are merely a kind of animal that require feeding, caretaking, veterinary care etc. So in the context of zoo keeping, birds and monkeys have similar relations with other concepts, and it is probably better to employ the general concept of “animal” and add specialization.

As another example, consider the concepts “employee” and “customer”. These concepts may even overlap: some instances of customer may be employee. To allow a particular employee to act as a customer in another situation, then a more general concept is needed, e.g. “person”. In that case, you must use relations to represent that a person is an employee or a customer.

Our example of residence permits can go either way. However, a closer look at the regulations shows that residence permits for limited duration and those for unlimited duration exclude each other once and for all. A residence permit cannot be transformed from limited to unlimited or otherwise; a new one has to be applied for in such cases. This consideration suggests -but does not enforce- to use two different concepts.

separate rules from your concepts / keep constraints apart from your concept (and relation) definitions

In chapter 3, we gave an example definition of a concept named Customer, as follows:

– Customer: a person who or organisation that, in the past three years, has placed at least one order and has paid for it,

The above definition may look smart, but it is not very sound. This is because the definition is not focused on the essence only, but it adds extra demands. Those demands in fact specify some business rules about customers, instead of helping us to understand and define the notion of customer! It is often a good idea to look for alternative definitions in a dictionary, for example:

– Customer: one who uses or buys any of our products or services, or

– Customer: any person or organization who views, experiences, or uses the services of another person or organization.

Notice how these definitions lack any mention of quantifications or qualifications, as they try to capture the core meaning of the concept.

5.2.3 Symmetric as well as antisymmetric

In rare situations, a homogeneous relation p may be both symmetric and antisymmetric at the same time. Some careful reasoning shows that such a relation p must satisfy the condition: $p \vdash \mathbb{I}$. We leave it to the reader to verify that both the Identity relation and the empty relation $\emptyset$ are symmetric and antisymmetric.

However, there is more to it, if we consider the elements of the source (and target!) set *A*. We can divide *A* into two subsets: one subset containing the elements x that satisfy (x,x) p, and the subset of all other elements in *A* that are not in p: (y,y) p. So, this relation p divides the source set in two disjoint subsets. In other words, the relation p can be understood as a simple Yes/No property of the elements of the source *A*. Yes, the element x is in the subset for which (x,x) is in p. Or no, the tuple (x,x) is not in p. By the way: we feel there is a much better way to specify a Yes/No property on *A*, as will be shown in the chapter on design considerations. For this, think of a new set with just two elements: H = { “yes” , “no” }. Next, we create a new relation p* from *A* to H in the obvious way: for all x, tuple (x, “yes”) is in p* if and only if (x,x) is in p.

Actually, p* is a function (more on that later), and it is easy to see how the relation p can be replaced by p* without loss of meaning. The point here is that p* is much easier to understand. Moreover, the function p* can be extended, for instance to cover a situation where the yes or no hasn’t been decided yet. We only need to extend the target set H to become { “yes”, “no”, “undecided” }, and that’s about it, p* can now also contain tuples which are undecided. A similar adjustment for a relation p that is homogeneous, symmetric and antisymmetric relation is not so easy to make.

5.2.4 Design considerations for relations

In the second step you determine which relations are involved. The example lead us to determine three relations. In rule based design, all meaning is expressed in terms of rules. Every rule is an expression in which relations are the most elementary building block. Therefore, all meaning is “carried” by relations. Even properties of a concept are represented as a relation, as shown in section **??** on page **??**.

Never hesitate to introduce a new relation, but do so only for the purpose of using it in a rule. Also, choose the most basic meaning in natural language you can think of.

For instance:

c *condRP* r means: Condition c .

This is clearly different from : Condition c is satisfied in residence permit r. If a condition is applicable to a residence permit, this does not necessarily mean that the condition is satisfied. Two different meanings require two different relations. For this reason, it is advisable to keep relations as basic as possible. An example of a non-basic natural language meaning is: Condition c must remain satisfied for residence permit r to stay valid. A reason to reject this, is that this sentence itself can be represented as a rule and is therefore not basic enough. Keep in mind that the sentence must be true for every pair $\langle c, r\rangle$ in the relation.

Changing the semantics of a relation means that all rules need to be revalidated in which the relation is involved.

5.2.5 Design considerations to conceive a business rule

Inventing a formula is the essential part in formalizing rules. Once we have figured out which concepts and which relations are relevant for this rule, formalization comes down to finding the right operators and connect these relations into the right formula. It helps to represent the relations in a conceptual model, as was done in figure **??**.

In the example of article 14 sub 2, we reasoned our way into a formula. Let us follow the steps more closely.

First, we figured that this article represents a requirement. Not every allocation of conditions to a residence permit is acceptable. If some specific condition is required for a particular purpose of stay, this implies something for all residence permits with that purpose of stay. The word "implies" in this sentence points us toward the operator $\vdash$.

Consider on the type of rule, i.e. what users will expect from this rule and how it is maintained: "structural" (no violatiion possible) or "operational". For the latter, choose between "immediate" enforcement or signalled (or ignored)

Suppose for now that the rule is considered to be an operational rule.

Then, we needed to think about the left hand side and the right hand side of the rule. On the right hand side, we want to make an expression that says which conditions are applicable to which residence permits. This makes *issued_under* the obvious (and only possible) choice for the right hand side. On the left hand side, we therefore need another expression that links conditions to residence permits. Obviously, we need the relation *condPurp* for representing the fact that we need a condtion linked to a purpose.

The rule *condPurp* $\vdash$ *issued_under* would not do however, because it is incorrect. Besides, we need to express the purpose of the residence permit. For this we need to use relation *purpose*. In order to make it match the types, we use $purpose^{\smile}$. So that is how we invented rule **??**:

$$condPurp; purpose^{\smile} \ \vdash \ condRP$$

5.2.6 Validation

The question whether this formula represents the original text needs to be addressed in order to detect any possible mistakes made in the previous reasoning. We used the translation to natural language, and were satisfied with the result.

In practice, you will often validate a rule by trial and error. If for example, a residence permit is found that by relation “has” is related to the purpose of stay “au pair”, and that permit does not mention the condition that the foreigner must be single, while by the relation *issued_under* it is known that that condition is imposed for the purpose of stay “au pair”, then this ought to be signalled as a violation of the rule.

A more sophisticated approach to trial-and-error is by automatically generating test cases. At present, software exists that will automatically create content for all of your relations, and then calculate whether the rules are violated by the content. Such software gives you another way of validating your rules.

Thirdly, you might try to falsify a rule by inventing a counterexample. Every time you discard a rule with a counterexample, you have learnt and therefore increased your understanding of the situation. So, falsification is at least as valuable as corroboration.

Another way to falsify or corroborate is to prove equivalence of (part of) a rule to something else, of which you already know certain properties. The following paragraphs will show you how to do such things.

More sophisticated ways exist. If, for example, you have tools that analyze your rules and derive certain implications, you might find that some particular implication of your set of rules is unacceptable within the business context you work with, and so you might want to alter or even reject some rule in your set. But it is beyond the scope of this book how to actually do this.

Nevertheless it is relevant to point out that you can never be 100% sure. Your claim about the validity of a rule is based upon a certain amount of evidence. You will have to provide that evidence and decide in each situation whether it is sufficient.

¿let op inclusie NU versus ALTIJD¡

¿INVARIANT RULES VERSUS PROCEDURAL / SEQUENCING RULES¡

5.3 Rule design in accordance with Ampersand

5.3.1 Capture all requirements

5.3.2 Establish a Conceptual Model

5.3.3 Phrase the requirements as Invariant Business Rule

5.3.4 Validate the design

- check all requirements
- check back with knowledgeable business representatives
- output comparison with existing information systems that cover the same context
- run automated test cases

5.4 The Deliverable

A Rule Based Design that meets the business requirements and has splendid design quality.

5.4.1 Ingredients of the deliverable

5.4.2 Invariant Rules at work: beyond Case-Management

5.4.3 Clear-cut documentation for subsequent Software developerment

5.5 Improving your design

refactor a model hierarchy

prune superfluous property-relations ¿EXPLAIN THEM FIRST in the previous chapter¡

Rule rewriting (?)

When to put a rule as Multiplicity Constraint or as Compound Rule

Altering the Conceptual Model will impact your rules Migrate a rule from structure (definitional; cannot be violated) to operational level (can be violated)

5.5.1 Cycle chasing

¿CYCLE¡ intro interregel, intraregel is niet zo relevant mits het idee van src en trg helder is

The number of cycles is: E = no. of Coherent-Blocks - no. of Concepts + no. of Relations

Separate paragraph on is-a relations (inserting one specialization with its is-a relation does not affect the overall count.

For every cycle, rules may be considered. There may be 0, 1 or more rules. 0 Means that the cycle is entirely unconstrained.

Byline: Database toxic cycle based on key inheritance.

5.5.2 Cycle improvement

5.5.3 Enhance invariance, reduce dependency on workflow

5.5.4 Quality considerations

- invariant
- independent of implementation
- syntactically sound
- simpte to understand (potential for self-explanatory rules, relations and concepts)
- complete (all requirements are expressed, in as few as possible locations)
- correct
- consistent = free of contradictions = populatable
- valid

When you outline a Conceptual Model, you have a choice how to capture multiplicity properties of your relations. You can simply incorporate the property into the relation declaration, or you can write it as a distinct business rule, completely separated from the declaration. The difference between the two options depends on your tools, how compliance checking and enforcement is implemented.

Sometimes an adjustment to the Conceptual Model can be pointed out that would improve generality of the model. The old model is then a restriction of the new one. For instance: xxx XXXXXXXXXX This goes to show that concepts and relations can indeed be considered as rules, but only from a very theoretic point of view.

Part III

Practice

Chapter 6

Document Management

> Document management can be characterized in some tens of rules. Because of this limited number, all rules can be presented in a single chapter of this book. This is good news from a designer's perspective, for who it becomes quite doable to incorporate document management in the architecture of an information system. It is a matter of picking the appropriate rules and adapting things to the situation if necessary.
>
> This chapter as intended as a reference model in your own projects. It can be useful when you select document management tools, for comparing standard built-in functionality with the functionality described in your requirements. It can also be of use when you design your own document management. The rules in this chapter can then serve as functional requirements, which you can choose to pick or leave alone as you think fit.

6.1 Introduction

Document management systems exist in abundance, and they all have their own unique properties. Yet in a specific way all of these systems are similar. They all store documents, documents can be collected in folders, and they all regulate access to documents. The things these systems have in common may be considered as the 'laws of document management systems'.

This chapter introduces a design pattern called *DocPAD*, which stands for Document management Pattern for Architecture and Design. This chapter serves as an example of defining a theme (document management) in terms of laws that are characteristic of that theme.

In case your own project has aspects of document management in it, you might use this chapter in designs of your own. Since many projects to date involve digital documents, this chapter might come in handy if you are involved in one. And if you require different rules, you can make laws of your own as explained in chapters **??** and **??**.

DocPAD addresses document management: storage, disclosure, and management of a vast multitude of documents of very different nature, with various owners and many different access restrictions. Every worker who sits at a desk and uses documents is a stakeholder. Some have an interest in getting paper out of the office, others have an interest in retrieving documents from huge archives, and yet others may have an interest in the authenticity of a stored document. These, and many more concerns are addressed by the person(s) responsible for implementing a document management system in his or her organization. To store and manipulate documents electronically requires the embedding of document imaging tools in the electronic data infrastructure of an organization. This yields various architectural questions, such as:

- How to cope with documents stored in different formats?
- Can access to documents be arranged properly?
- How does the architecture ensure durability and persistence of stored documents?
- How do we arrange electronic documents in folders?
- etcetera

These and many more design questions lead to different solutions in different situations. However, DocPAD captures some common ground shared by many solutions. It reflects a collective experience of several process architects, which has "crystallized" into the laws presented in this chapter. Experience with document management in various projects has also contributed to the soundness and relevance of these laws. The rules in this chapter can serve as functional requirements of a DocPAD compliant information system.

6.2 Document_Management

This section introduces the concepts Document, Folder, Application, Format, Medium, and Device type. Then we proceed to define the fundamentals of documents that reside into folders.

document

1. A *document* is an object meant to carry information across place

and/or time. For instance: contracts, newspapers, letters, e-mails, a taped conversation, or even a telephone conversation. A document can be stored and retrieved as a whole in a document store. Information in a document is possibly unstructured.

folder

2. A *folder* is a data object, in which documents are kept together

application

3. An *application* is a software component that can be invoked by a user in order to get a specific automated service, such as the applications that are invoked in the 'Start' menu on MS-Windows desktops.

format

4. A *format* is a way to store the information on a document, such that it can be reproduced for viewing or manipulation by users. Examples: 'MS Word 2000 vs. 9.0', 'PS-Adobe-2.0', or 'neato version 1.7.6'.

device type

5. A *device type* is a type of hardware, needed to process documents, such as 'ACME Laserjet printer 4000 TN'

medium

6. A *medium* is the physical means on which a document is stored, such as paper, PC-floppy disk, magnetic tape, or microfilm.

Figure 6.1 shows a conceptual diagram of this theme.

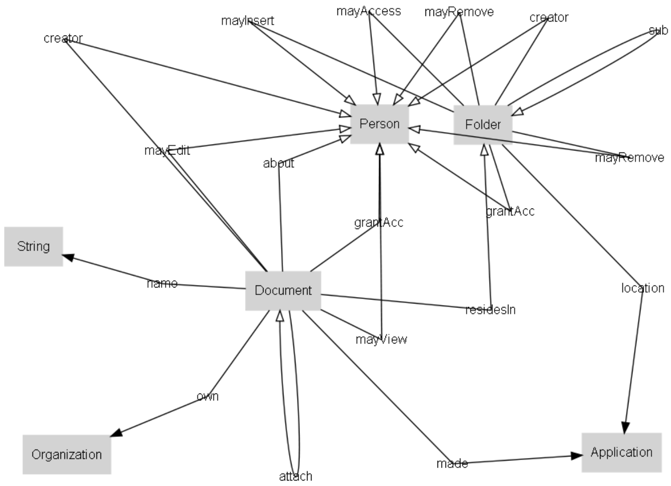


Figure 6.1: Conceptual diagram of Document_Management

Unique name Documents have a name by which they can be referred to. Each document has name exactly one string.

Each document resides in at least one folder. The formalization (equation 6.3) requires the following two relations.

$$\begin{aligned} \mathit{name} &: \mathit{Document} \to \mathit{String} && (6.1)\\ \mathit{residesIn} &: \mathit{Document} \times \mathit{Folder} && (6.2) \end{aligned}$$

Within each folder, documents have a unique name.

$$\mathit{name}; \mathit{name}^{\smile} \cap \mathit{residesIn}; \mathit{residesIn}^{\smile} \ \vdash \ \mathbb{I} \qquad (6.3)$$

Document hierarchy Folders are organized in a hierarchical fashion. So, folders can have subfolders. In order to formalize this, we introduce relation sub (6.4):

$$\mathit{sub} \ : \ \mathit{Folder} \times \mathit{Folder} \qquad (6.4)$$

The expression $f\ sub\ f'$ means that folder f is a subfolder of folder f'.

If a document resides in a subfolder, it implicitly resides in the enveloping folder(s). As a consequence, a document can reside in any number of folders simultaneously. We use definition 6.2 (residesIn) to formalize this.

$$\mathit{residesIn}; \mathit{sub}^{*} \ \vdash \ \mathit{residesIn} \qquad (6.5)$$

Anonymous documents The person(s) who created the document must be traceable. So, the sentence: “Document d was created by person p” is meaningful (i.e. it is either true or false) for any document d and person p. These sentences are represented by the relation *creator* (6.6):

$$\mathit{creator} \ : \ \mathit{Document} \times \mathit{Person} \qquad (6.6)$$

Documents that are not traceable to their creator are signalled. A signal is produced when:

$$\mathbb{I}_{[\mathit{Document}]} \cap \overline{(\mathit{creator}; \mathit{creator}^{\smile})} \qquad (6.7)$$

6.3 Access

Which applications have access to which documents? This question is relevant to ensure that users may gain access to the right documents in all circumstances. This section introduces several relations and rules that help to organize this. Figure 6.2 shows a conceptual diagram of this theme.

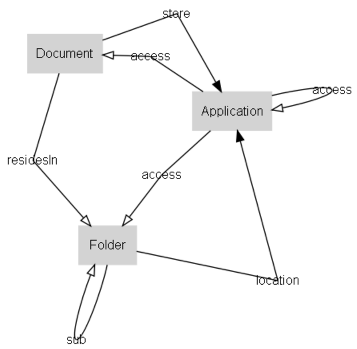


Figure 6.2: Conceptual diagram of Access

Self access to stored documents Documents are kept (maintained, stored) in applications, which can therefore be called document stores. Each document is kept in exactly one application. In order to secure the contents of documents, access to a document can be granted per application. So, the sentence: "Application a has access to document d" is meaningful (i.e. it is either true or false) for any application a and document d. The formalization (equation 6.10) requires the following two relations.

$$\begin{aligned} store &: Document \rightarrow Application \qquad (6.8) \\ access &: Application \times Document \qquad (6.9) \end{aligned}$$

An application has access to all documents it stores.

$$store \vdash access^{\smile} \qquad (6.10)$$

Access hierarchy In order to secure the location of a document, access to folders can be granted per application. So, the sentence: "Application a has access to folder f" is meaningful (i.e. it is either true or false) for any application a and folder f. In order to formalize this, we introduce relation access (6.11):

$$access : Application \times Folder \qquad (6.11)$$

If an application has access to a folder, the application implicitly has access to its sub-folders. We use definition 6.4 (sub) to formalize this.

$$access; sub \vdash access \qquad (6.12)$$

Access per folder If an application has access to a document, it must have access to the application in which that document is stored. We use definitions 6.11, 6.9, and 6.2.

$$access; residesIn^{\smile} \vdash access \qquad (6.13)$$

Location access From a user perspective, a folder is located in (accessible by) some application, even though it might physically be distributed over various locations. Each folder is stored in exactly one application. If one application (say 'a') can access documents inside another application (say 'b'), these applications are related. We say 'a access b'. So, the sentence: "Application a has access to documents kept in application a' " is meaningful (i.e. it is either true or false) for any application a and application a'. The formalization (equation 6.16) requires the following two relations.

$$\begin{aligned} location &: Folder \rightarrow Application && (6.14) \\ access &: Application \times Application && (6.15) \end{aligned}$$

If an application has access to a folder, it must also have access to the application in which that folder is stored. We use definition 6.11 (access).

$$access; location \vdash access \qquad (6.16)$$

Store access If an application has access to a document, access is required to the store in which that document resides. We use definitions 6.15, 6.9, and 6.8.

$$access; store \vdash access \qquad (6.17)$$

6.4 Permissions

In document management systems, documents can be accessed only when permission is granted. The granting of a permission is regarded as an event. Permissions themselves may be valid during a certain period of time. This section introduces a number of relations and rules that help to organize this. Figure 6.3 shows a conceptual diagram of this theme.

Rule8 In order to provide access permissions on a detailed level, permissions can be granted for individual objects. Each permission is valid for at least one document. Events are meant to trigger useful work. So we may assume that every event affects a particular object that is referred to by an identifier. Each event works on the object identified by exactly one document.

The sentence: "Permission p has been granted by person p' " is meaningful (i.e. it is either true or false) for any permission p and person p'. A permission is always granted to individuals, who are called 'permissee'. Without a permissee, the permission

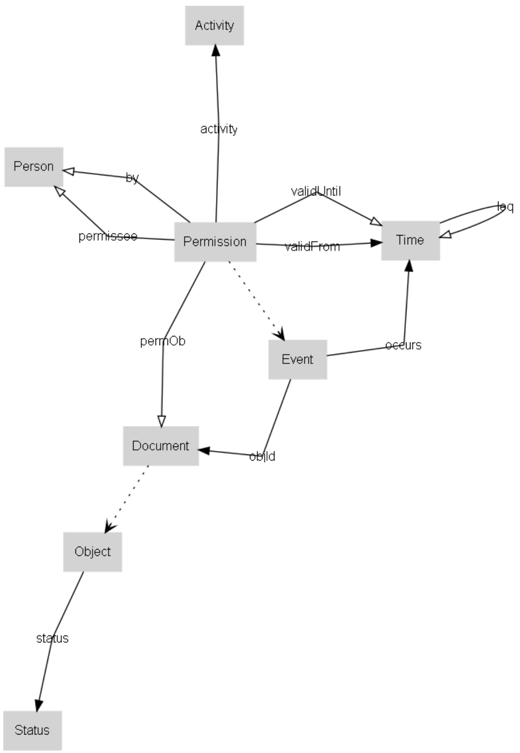


Figure 6.3: Conceptual diagram of Permissions

itself ceases to exist. Each permission is valid for at least one person. The formalization (equation 6.22) requires the following four relations.

$$\begin{aligned} permOb &: Permission \times Document && (6.18)\\ objId &: Event \rightarrow Document && (6.19)\\ by &: Permission \times Person && (6.20)\\ permissee &: Permission \times Person && (6.21) \end{aligned}$$

Objects may require permission to change. More precisely, for an event to occur on any object that requires permission, that permission must be granted to the person on whose behalf the event takes place.

$$permOb; objId^{\smile}; by \ \vdash \ permissee \qquad (6.22)$$

valid permission A permission has a period of validity. A permission is valid from a certain point in time. Each permission is valid from exactly one time. The smaller-than relation on time. So, the sentence: "Time t is smaller than time t' " is meaningful (i.e. it is either true or false) for any time t and time t'. Events occur at a certain point in time. Actually, it does not exist until it occurs. That is why every event has a moment when it occured. Each event occurs exactly one time. The formalization (equation 6.26) requires the following three relations.

$$\begin{aligned} \mathit{validFrom} &: \mathit{Permission} \to \mathit{Time} && (6.23)\\ \mathit{leq} &: \mathit{Time} \times \mathit{Time} && (6.24)\\ \mathit{occurs} &: \mathit{Event} \to \mathit{Time} && (6.25) \end{aligned}$$

A permission is valid from the moment specified in 'validFrom'. For that we use definitions 6.18 and 6.19.

$$\mathit{permOb}; \mathit{objId}^{\smile} \ \vdash\ \mathit{validFrom}; \mathit{leq}; \mathit{occurs}^{\smile} \qquad (6.26)$$

expiring events A permission may expire. A permission is valid until a certain point in time. Each permission is valid from zero or one time. Different situations may have different authorization rights. In some case, the distinction between access and no access might be sufficient. Other cases might require a read/write/create/delete distinction to grant more refined access rights to users. The authorization type has been made variable to suit any conceivable access right classification. Each permission grants exactly one activity rights. The formalization (equation 6.29) requires the following two relations.

$$\begin{aligned} \mathit{validUntil} &: \mathit{Permission} \times \mathit{Time} && (6.27)\\ \mathit{activity} &: \mathit{Permission} \to \mathit{Activity} && (6.28) \end{aligned}$$

Any event must have occurred before the appropriate permission has expired. For that we use definitions 6.18, 6.19, 6.24, 6.25, 6.20, and 6.21.

$$\mathit{validUntil}^{\smile}; (\mathit{permOb}; \mathit{objId}^{\smile} \cap \mathit{permissee}; \mathit{by}^{\smile} \cap \mathit{activity}; \mathit{activity}^{\smile}); \mathit{occurs} \ \vdash\ \mathit{leq} \qquad (6.29)$$

6.5 Versions

As long as documents are under construction, a document management system can keep track of the different versions of that document. Even though different versions of a document may exist, properties of that document must be maintained across all these versions. This section introduces several relations and rules to help organize this. Figure 6.4 shows a conceptual diagram of this theme.

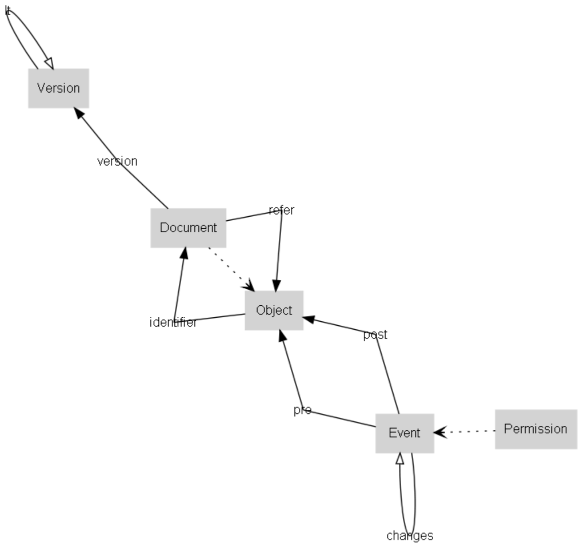


Figure 6.4: Conceptual diagram of Versions

objectkey An object is recognised by an identifier. Each object is identified by exactly one document. Over time, during the construction of a document, there may be several successive versions of the same document. The version of each document is exactly one version. The formalization (equation 6.32) requires the following two relations.

$$\begin{aligned} \mathit{identifier} &: \mathit{Object} \rightarrow \mathit{Document} && (6.30) \\ \mathit{version} &: \mathit{Document} \rightarrow \mathit{Version} && (6.31) \end{aligned}$$

When the identifier and version number is known, any object is uniquely determined. Note the difference with 'object', which identifies the most recent version of an object by the identifier only.

$$\mathit{identifier}; \mathit{identifier}^{\smile} \cap \mathit{version}; \mathit{version}^{\smile} \vdash \mathbb{I} \qquad (6.32)$$

Rule12 A document refers to an object. Even if the object changes during its life (e.g. a document), the identifier remains the name under which the most recent version is reachable. Each document refers to exactly one object. This relation represents the less-than relation on versions. Lt is an ordering relation on versions. The formalization (equation 6.35) requires the following two relations.

$$\begin{aligned} \mathit{refer} &: \mathit{Document} \rightarrow \mathit{Object} && (6.33) \\ \mathit{lt} &: \mathit{Version} \times \mathit{Version} && (6.34) \end{aligned}$$

The relation refer points to the most recent version of an object. For that we use definitions 6.30 and 6.31.

$$\mathit{version}^{\smile}; \mathit{identifier}; \mathit{refer}; \mathit{version} \ \vdash \ \mathit{lt} \cup \mathbb{I}_{[\mathit{Version}]} \tag{6.35}$$

Rule13 When an event occurs, this results in a changed object. That object can be seen as an output to the operation that is being perfomed. Changes is a property of events. When an event occurs, it may change an object. That object can be seen as an input to the operation that is being perfomed. Each event has changed exactly one object. When an event occurs, this results in a changed object. That object can be seen as an output to the operation that is being perfomed. Each event has produced exactly one object. The formalization (equation 6.39) requires the following three relations.

$$\begin{align} \mathit{changes} &: \mathit{Event} \times \mathit{Event} \tag{6.36}\\ \mathit{pre} &: \mathit{Event} \to \mathit{Object} \tag{6.37}\\ \mathit{post} &: \mathit{Event} \to \mathit{Object} \tag{6.38} \end{align}$$

An event is said to change an object if the pre-object is not equal to the post-object, but both do share the same identifier. We use definition 6.30 (identifier).

$$\mathit{changes} \ = \ \mathit{pre}; \overline{\mathbb{I}}; \mathit{post}^{\smile} \cap \mathit{pre}; \mathit{identifier}; \mathit{identifier}^{\smile}; \mathit{post}^{\smile} \tag{6.39}$$

Rule14 The occurrence of an event increases the version number of objects. We use definitions 6.36, 6.37, 6.38, 6.31, and 6.34.

$$\mathit{version}^{\smile}; \mathit{pre}^{\smile}; \mathit{changes}; \mathit{post}; \mathit{version} \ \vdash \ \mathit{lt} \tag{6.40}$$

6.6 Media

A document management system must keep track of formats of documents, media these documents are stored on, and applications and equipment that are able to manipulate these documents. This section introduces the relations needed to keep track of this. Figure 6.5 shows a conceptual diagram of this theme.

The right equipment Applications may require specific equipment. A document scanning application, printing applications, etc. are examples of applications that work with specific equipment. So, the sentence: “Application a works with device type d” is meaningful (i.e. it is either true or false) for any application a and device type d. Different devices require different media. Printers require paper, a word processor requires files on a computer, and

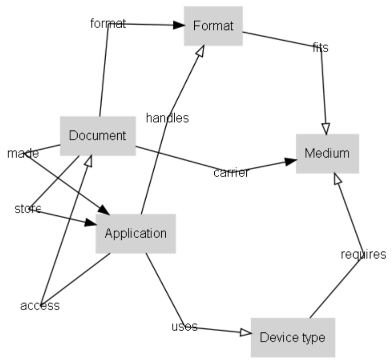


Figure 6.5: Conceptual diagram of Media

a call center application requires a telephone, to name just a few examples. So, the sentence: "Device type d requires medium m" is meaningful (i.e. it is either true or false) for any device type d and medium m. Documents can reside on different media. In order to have information technology work with different media, we must know on which medium each document resides. Each document is carried on exactly one medium. The formalization (equation 6.44) requires the following three relations.

$$\begin{aligned} \mathit{uses} &: \mathit{Application} \times \mathit{Device\ type} && (6.41)\\ \mathit{requires} &: \mathit{Device\ type} \times \mathit{Medium} && (6.42)\\ \mathit{carrier} &: \mathit{Document} \rightarrow \mathit{Medium} && (6.43) \end{aligned}$$

If an application can access a document, this implies that the equipment is available for the carrier of that document. We use definition 6.9 (access).

$$\mathit{access} \vdash \mathit{uses}; \mathit{requires}; \mathit{carrier}^{\smile} \qquad (6.44)$$

Handle self-made documents If one wants to work with a document in its original form, information about the creating application is required. Each document was created using exactly one application. For applications to work with (i.e. read, print, scan, etc.) documents, the format of a document must be known. Each document has exactly one format. Applications may pose restrictions to the format of documents they handle. For instance, MS-Word can cope with various MS-Word formats, but not with PDF. So, the sentence: "Application a handles format f" is meaningful (i.e. it is either true or false) for any application a and format f. The formalization (equation 6.48) requires the following three relations.

$$\begin{aligned} \mathit{made} &: \mathit{Document} \rightarrow \mathit{Application} && (6.45)\\ \mathit{format} &: \mathit{Document} \rightarrow \mathit{Format} && (6.46)\\ \mathit{handles} &: \mathit{Application} \times \mathit{Format} && (6.47) \end{aligned}$$

If a document was written in (created by) an application, that application can handle that document.

$$\mathit{made}^{\smile};\mathit{format}\ \vdash\ \mathit{handles} \tag{6.48}$$

Store handles format Since different documents can have a different formats, a document store must be able to handle the formats of all documents stored. That may be achieved for many formats in one application by doing little more than storing alone, as happens in a file store. More intelligent applications might do more (e.g. print or show documents), which poses limitations on the formats a document can be in. We use definitions 6.46, 6.47, and 6.8.

$$\mathit{store}^{\smile};\mathit{format}\ \vdash\ \mathit{handles} \tag{6.49}$$

Format matches carrier A document format is suitable for specific media only. For instance, PDF is only suitable for digital media, whereas the SUN-tabloid format is suitable for paper. Each format is available on at least one medium. In order to formalize this, we introduce relation fits (6.50):

$$\mathit{fits}\quad:\quad \mathit{Format}\times \mathit{Medium} \tag{6.50}$$

Every document has a carrier. A letter might be carried on paper, on a disk, or on Internet. An internet document can be held in HTML, but to have HTML as a format for a contract might be forbidden. That is why. For that we use definitions 6.46 and 6.43 to formalize this.

$$\mathit{format}^{\smile};\mathit{carrier}\ \vdash\ \mathit{fits} \tag{6.51}$$

handlesTot Every application handles at least one format We use definition 6.47 (handles). A signal is produced when:

$$\mathbb{I}_{[\mathit{Application}]}\cap\overline{(\mathit{handles};\mathit{handles}^{\smile})} \tag{6.52}$$

So far in this chapter, we have introduced requirements and formalized them. That gives a basis for document management. Using that basis in practice may require additional requirements, which can be added at will. However, from the requirements introduced so far, a fair amount of implementation consequences can be derived. That is the topic of the following sections.

6.7 Data structure

The requirements from the previous sections have been translated into the class diagram in figure 6.6. There has resulted in five data sets, 26 associations, and two generalisations. Scalars, i.e. classes without attributes, have been drawn as small rectangles without content. For an implementation by means of a relational database, this means that five tables have to be built, one for every data set, and 26 binary tables are required to accommodate all associations.

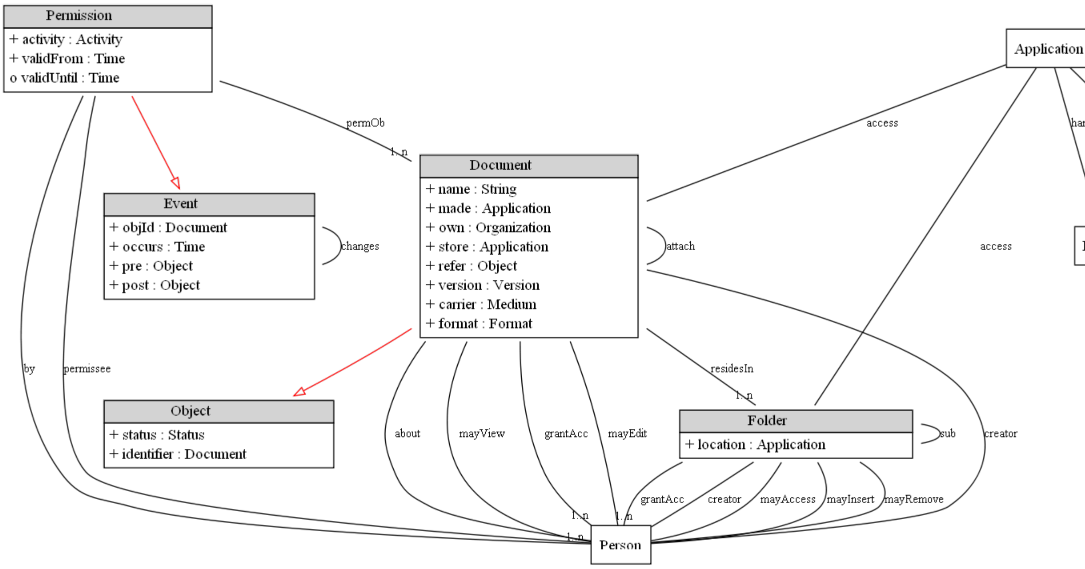


Figure 6.6: Class Diagram of DocPAD

DocPAD has the following associations and multiplicity constraints.

relation	total	surjective
$residesIn_{[Document,Folder]}$	$\surd$	
$sub_{[Folder]}$		
$about_{[Document,Person]}$		
$mayView_{[Document,Person]}$		
$grantAcc_{[Document,Person]}$	$\surd$	
$mayEdit_{[Document,Person]}$		
$creator_{[Document,Person]}$		
$mayAccess_{[Folder,Person]}$		
$grantAcc_{[Folder,Person]}$	$\surd$	
$mayInsert_{[Folder,Person]}$		
$mayRemove_{[Folder,Person]}$		
$creator_{[Folder,Person]}$		
$attach_{[Document]}$		
$access_{[Application,Document]}$		
$access_{[Application,Folder]}$		
$access_{[Application]}$		
$permOb_{[Permission,Document]}$	$\surd$	
$permissee_{[Permission,Person]}$	$\surd$	
$leq_{[Time]}$		
$by_{[Permission,Person]}$		
$lt_{[Version]}$		
$fits_{[Format,Medium]}$	$\surd$	
$uses_{[Application,Devicetype]}$		
$handles_{[Application,Format]}$		
$requires_{[Devicetype,Medium]}$		

Additionally, the homogeneous relations come with the following properties:

relation	Rfx	Trn	Sym	Asy	Prop
$sub_{[Folder]}$				$\surd$	
$attach_{[Document]}$					
$access_{[Application]}$					
$leq_{[Time]}$					
$lt_{[Version]}$		$\surd$		$\surd$	
$changes_{[Event]}$			$\surd$	$\surd$	$\surd$

6.7.1 Document

The attributes in Document have the following multiplicity constraints.

attribute	type	compulsory	unique
I	Document	√	√
name	String	√	
made	Application	√	
own	Organization	√	
store	Application	√	
refer	Object	√	
version	Version	√	
carrier	Medium	√	
format	Format	√	

6.7.2 Event

The attributes in Event have the following multiplicity constraints.

attribute	type	compulsory	unique
I	Event	√	√
objId	Document	√	
occurs	Time	√	
pre	Object	√	
post	Object	√	

6.7.3 Permission

The attributes in Permission have the following multiplicity constraints.

attribute	type	compulsory	unique
I	Permission	√	√
activity	Activity	√	
validFrom	Time	√	
validUntil	Time		

6.7.4 Object

The attributes in Object have the following multiplicity constraints.

attribute	type	compulsory	unique
I	Object	√	√
status	Status	√	
identifier	Document	√	

6.7.5 Folder

The attributes in Folder have the following multiplicity constraints.

attribute	type	compulsory	unique
I	Folder	$\surd$	$\surd$
location	Application	$\surd$	

6.8 Conclusion

This chapter has introduced a partial specification of a document management system. The specification consists only of concepts, relations and rules. So any system that satisfies these rules is compliant to the specifications in this chapter. It may consequently be called DocPAD-compliant.

For practical purposes, the DocPAD rules may be complemented by other rules of more specific nature. In that way you can build specifications that suit more specific purposes. The rules may also be used as part of larger system specifications.

Chapter 7

INDiGO

7.1 Introduction to this chapter

In 2007, the Dutch immigration authority, IND, published a tender for its information provisioning. The IND is responsible for executing the laws and regulations related to immigration. This involves residence permits, asylum, naturalisation and appeals in court. The size of this bid and a tight tendering schedule forced contenders to combine forces. Only three consortia qualified for participating in the competition. Competitors were required to define their solutions exhaustively in their offers, forcing the entire design stage to take place within the timeframe of the bid.

One of the consortia, headed by a tandem of the companies Ordina and Accenture, went for a no-risk design approach. They designed an integral solution, named INDiGO[1], consisting of commercial-off-the-shelf (COTS) software components only. Yet the solution had to respect every rule in the Dutch immigration law. In order to meet this quality constraint within the tight schedule of the bid, the consortium decided to compose the solution architecture in a formal manner. The contract was awarded at the end of 2007. When the project started early 2008, the consortium had delivered a start architecture document that contained the integral solution, with detailed descriptions of all key ideas involved in the design. In the summer of 2010, the system went live.

The IND tender marks the first occasion in which Ampersand was used on the critical path in a large IT project. For this reason, INDiGO makes an interesting case study for those who want to use Ampersand. This chapter discusses some of the key ideas of INDiGO and illustrates them with the actual specifications. The first section provides an overview of the guiding principles that govern the design of INDiGO. The second

[1] This name was chosen by IND after a naming contest among IND personnel

section introduces some of the key ideas, two of which are elaborated in a subsequent section. At the end of this chapter we reflect on the use of formal methods in the design of large information systems and business processes.

This chapter illustrates a way of working that is typical for rule based design. Each idea that is key to a design is defined as a set of rules. This creates room to discuss different key ideas with different groups of people. The composition of all key ideas yields the solution architecture. In INDiGO, key ideas have been discussed during the tendering phase. Dialogue sessions were held in which customer (IND) and the consortia discussed the details of the offer in great detail. As a consequence, an comprehensive architecture was available at the start of the project. The contribution of Rule Based Design was to maintain correctness and consistency of this architecture under the extreme time pressure that is common to tenders.

7.2 Guiding principles

This section summarizes the actual requirements of the IND in terms of guiding principles. The IND wanted to become flexible, customer oriented, effective, and efficient. Throughout the design, these four principles have been used to motivate design choices.

Flexibility in the eyes of IND means that changes in legislation and regulation can be absorbed instantly, without any changes in the software. Coming from a situation in which custom built database applications dominated, this was understandable. IND was used to very long turnaround times indeed, when software changes are needed as a result of new regulations.

Customer orientation means that IND wants to give her clients a red carpet treatment. This involves careful handling of client data, informing clients correctly and in time, complying with the requirements by law, and compliance to all explicit and implicit requirements. Coming from a situation with considerable backlogs and several public incidents, customer focus was a clear driver of many of IND's requirements.

Effectiveness means to IND that everything single action taken every day on any location is compliant with current regulations, and traceble to the original motivation and legal evidence. IND wanted to eliminate even the possibility to make legal mistakes.

Efficiency means to IND that no unneccesary actions are taken and that processes are well controlled, well organized, and irredundant. The IND came from a situation in which unnecessary manual actions were required, double work existed, and employees sometimes felt like red tape workers. In the minds of these employees, efficiency has a very concrete meaning.

The general principles of flexibility, customer orientation, effectiveness and efficiency are solution independent principles, prescribed by IND. During the tendering process, these principles have been augmented by six others: quality, integration, manageability, consistency, compliance, and security.

Quality means that all properties of interest have been made objectively quantifyable, in ways that enables management control during the transition phase. Since the entire transition is planned in a period of four years, there is a genuine need for making that transition manageable.

Integration means that the user gets the impression of a single application, which results in the perception of simplicity. Integration is achieved by means of a service oriented design.

Manageability means that every singular management incident involves a single adaption in the system, without any software change. This requires that all data is stored in a single place, and all dependencies are implemented in a generative fashion.

Consistency means that all data is consistent with the design rules and remains consistent. Consistency has been built in up front by making all design rules explicit in an Ampersand analysis. The primary instrument for data consistency is the knowledge model.

Compliance means that INDiGO will respect all current legislation and regulations in a durable way. For that reason, the relevant regulations are translated in a knowledge model.

Security means that the design complies with all requirements from VIR-BI, which is the applicable security standard in government. INDiGO can handle all information up to the level “departmentally confidential”. Information that is more confidential than that must be kept outside the system.

7.3 Key ideas

The solution for IND combines a number of ideas into a solution. These key ideas concern permit applications, legal decisions, data validation, traceability, determination of title, security, identity management, infrastructure, (among others). This section discusses them to provide some insight in the solution and into the overall design choices that were made.

follow the rules An organization in the public sector must follow the rules. These rules originate from many different sources, such as immigration regulations, IND policy, security constraints, applicable laws and legislation, instructions from the department of Justice, etcetera. INDiGO supports this by means of a knowledge model that contains these rules, at least inasmuch they bear consequences for the IT. The main task of INDiGO is to help the IND follow the rules in transparent and traceable ways. The challenge was to design INDiGO such that frequent changes in these rules can be absorbed practically without delay and without any operational disturbance.

standard software The IND insisted on standard software with proven track records. This requirement was fulfilled by an architecture that consists of commercial-off-the-shelf (COTS) software components, whithout tailormade components. The experience with two 'playgrounds' in the laboratories of the consortium produced the final selection that was used in the offer to the IND. The solution contains a case management component, a document management engine, a knowledge engine, a (fuzzy) matching engine, a portal environment, a business intelligence component, and a service bus to put these components together.

separation of know and flow One of the most explicit requirements was to separate knowledge about immigration (the "know") from the management of the primary process (the "flow). This separation was achieved by analyzing the conceptual structure of the IND's process management and the conceptual structure of the knowledge engine, and removing all possible contradictions. After proving this conceptually correct, it posed no problem to create working prototypes of this feature. Oracle-Siebel was selected as case management engine to support the primary process, whereas the knowledge engine was realised by Be Informed.

dynamic changes in the structure of data In a dynamically evolving society, it is impossible to predict all future changes. Still, the IND must follow developments in society closely and without delay. This paradox affects the very data model of the solution, which may change on a day-to-day basis. So the architecture must ensure that any conceivable property can be stored when the need arises. This problem has been solved by putting a knowledge model in place of data model, restricted to immigration knowledge. Since the knowledge model governs the data logic at the solution level (not inside the components), special measures were taken for different components. For example, Oracle-Siebel employs a special table accommodate structural changes without changing its built-in data model.

decision making Each decision made by the IND about a particular alien follows a particular procedure. The architecture has been simplified by choosing the decision as the scope of each procedure. If a number of decisions are made in a single case, this means that as many procedures are conducted. In practice, only simple cases consist of a single procedure.

traceable decisions A decision structure has been designed to enable the IND to trace every decision to its legal origins, even after many years. The core of this structure is a link between facts that are established by the IND and the documents to prove it. These facts are contained in a decision tree to document the reasoning that supports the decision. References to applicable laws and regulations are automatically inserted in that tree wherever possible. When the decision becomes definitive, the entire decision structure is secured in records management. The complexity of this structure lies in the collaboration between architectural components. The decision tree is composed by the knowledge engine, facts and data are stored in the case management engine, and documents are maintained in the document manager. So the entire structure requires a closely knit collaboration between these components on a service bus level. The advantage is that IND personnel can reproduce the legal reasoning behind any past decision together with all the evidence at the push of a button.

Logical Data Model INDiGO employs an enterprise service bus (ESB) to overcome differences in the internal data models of the various COTS-components. A logical data model is central in that discussion. During the tendering phase, The conceptual model of Ampersand and the data model that was derived from it have played this role during the tender phase[2]. This model was used to maintain the semantic consistency of all components on the level of the service bus. It maintains business requirements that affect multiple components. In order to maintain this "semantic integrity", the requirements were analyzed using Ampersand.

dynamic action plan INDiGO features a process engine, Oracle-Siebel, and a knowledge engine, Be Informed. Together, these components collaborate to produce flexibility that is hard to achieve in any other way. All cases that are processed by the IND are controlled by a process engine, that monitors the process. The actions to be taken, the order in which they are perfomed, and the staff involved may vary from case to case. The knowledge engine is used to determine an action plan for each case, using specific knowledge of that case and knowledge about the applicable laws and regulations. This knowledge system computes precisely the right actions for each specific case. It is as though every case gets a process

[2] at an earlier stage, this model was called Common Message Model, following the conventions of service orientation in the UML

model of its own, rather than the one-size-fits-all process model of earlier systems.

knowledge governs data In order to follow developments, IND introduces new characteristics on a daily basis. Needless to say that these changes can only be followed instantaneously if they can be absorbed without affecting the data model of any component. By representing this knowledge in the knowledge model of Be Informed, these changes can be carried out by changing the knowledge model. This results (almost) instantaneously in the correct changes in the primary process.

Each one of these key ideas has been analyzed, each yielding a set of business rules.

Two of the key ideas will be elaborated. Decision making is shown in section 7.4, defining the administration of decisions about residence permits. A number of rules that govern the logical data model are discussed in section 7.5. When studying these examples, it is worth to notice that rules

1. are valid throughout the entire scope (of INDiGO);
2. can be considered independently from other rules;
3. are few in number.

7.4 Decision making

This section elaborates a particular theme from the IND case: the decision making process. After explaining some of the background, we provide the formalization of rules that govern the process of deciding about the IND's decision to grant residence permits (green cards).

application
decision

In 2007, the IND observed that immigration procedures have a generic structure. Only when immigration specific regulations are involved, procedures start to vary. The reason for this can be found in the law. A generic procedure is described in the Awb ("Algemene wet bestuursrecht", the bill that governs administrative law in the Netherlands), and immigration specific issues are described in the VW ("Vreemdelingenwet", the immigration law) and related regulations. Article 1:3 Awb introduces the ideas of *application* and *decision*, which ly at the heart of the IND's activities. An application is a request by a stakeholder to take a decision. The IND's primary business process includes all the work required to make decisions about particular aliens. Examples are an application for a visa in a consulate or embassy (a procedure called

MVVDIP), a prolongation for a residential permit (VVRVRL), or an asylum procedure (ASIEL).

alien

An *alien* is anyone who does not possess the Dutch nationality and is not entitled by law to treatment as a Dutch national (VW, Art 1.m). In the IND's daily practice, aliens are persons who wish access and the right to stay in the Netherlands, or those who are staying legally in the Netherlands and wish to obtain the Dutch nationality. Hence, aliens are stakeholders to which the IND provides services.

procedure

Any given decision is based on one particular article in the law. That article characterizes the decision. For that reason, each type of decision requires different actvities to be performed. These activities are prescribed by the law. Thus, INDiGO treats each decision as an instance of one particular procedure. A *procedure* is a set of activities performed to produce a decision on the application of an alien.

A decision may be followed by other decisions (an objection or an appeal, etc.) that are related to the same case. So, every case will get a decision in first instance. Other decisions may be added to the case until the decision is accepted or all routes of appeal are exhausted. Every case is about a single residence application, enforcement or other event for which a procedure can be conducted. Every case also has a single purpose for staying, which cannot change during the lifetime of the case. If the purpose for staying changes, applicants have to resubmit a new application and redo the procedure. Figure 7.1 is an example of a married couple, with each spouse going through various procedures for entering the country and staying there. This example shows the procedures in the upper bar. The two lower bars show the situation each of the spouses is in at every moment in time. The example illustrates dependencies that may exist between different cases.

The conceptual analysis that follows focuses on the rules that govern decisions, which are to be maintained by INDiGO. A conceptual model is shown in figure 7.2. A selection of rules has been made to show the essence of the specification and to communicate the flavour of the specification at hand. The formalization is presented one rule at a time, where new relations are introduced when needed.

Rule: decision by INDiGO A rule engine constructs a decision tree for every decision that is being made. The description of the rule engine's decision is looked up from the decision type. In order to formalize this, we introduce function 7.1:

$$\begin{aligned} \mathit{reDescr} &: \mathit{Procedure} \rightarrow \mathit{Description} && (7.1)\\ \mathit{reVal} &: \mathit{Procedure} \rightarrow \mathit{DecisionValue} && (7.2)\\ \mathit{decisionName} &: \mathit{DecisionValue} \rightarrow \mathit{Description} && (7.3) \end{aligned}$$

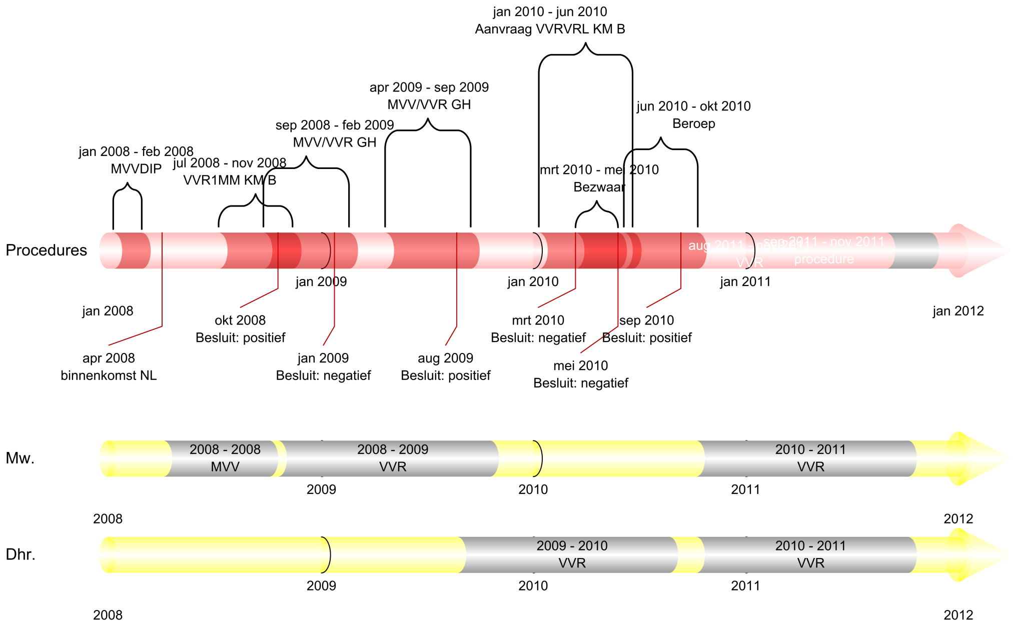


Figure 7.1: Example: related cases going through different procedures.

This rule maintains:

$$reDescr \;=\; reVal; decisionName \tag{7.4}$$

Ihe formalization brings the process back to a simple lookup of standard texts. In many specifications, not only the IND's, such simplification commonly occurs as a side effect of the formalization process. It is an example of the simplifying effect of the mental effort that precedes a good specification

Rule: IND's decision Similarly, the description of the IND's decision is looked up from the decision value. In order to formalize this, we introduce function 7.5:

$$decisionDescr \;:\; Procedure \to Description \tag{7.5}$$

This rule maintains:

$$decisionDescr \;=\; decisionVal; decisionName \tag{7.6}$$

Definitions **??** and 7.3, introduced in the previous rule, are reused for this purpose. This lookup is formalizes as follows:

$$decisionDescr \;=\; decision; decisionName \tag{7.7}$$

This rule is similar to rule **??**, illustrating the point that such lookups are common.

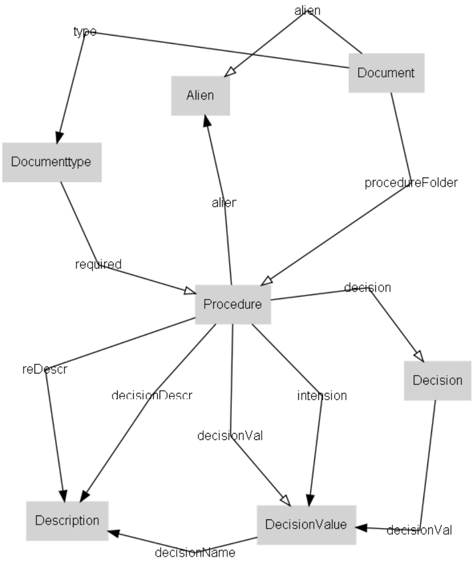


Figure 7.2: Conceptual analysis of Decision making

document

Rule: deficiencies The IND starts by checking whether an application for residence is complete. Required documents that are missing must be signalled, so that an employee can request additional information. Since that check can be automated, we formulate a rule that checks for missing documents. Which documents are needed can be derived from the type of application. A *document* is anything that can be stored in a folder. A document is called *formal* if it represents a right granted on the basis of a decision. It is also the physical product that is issued to someone as a result of a decision.

The formalization (equation 7.13) requires the following five relations.

$$\begin{aligned} required &: Documenttype \times Procedure && (7.8)\\ type &: Document \rightarrow Documenttype && (7.9)\\ procedureFolder &: Document \times Procedure && (7.10)\\ alien &: Document \times Alien && (7.11)\\ alien &: Procedure \rightarrow Alien && (7.12) \end{aligned}$$

The rule that every application shall be complete can be formalized as follows:

$$required \vdash type^{\smile}; (procedureFolder \cap alien; alien^{\smile}) \qquad (7.13)$$

This rule says that if a document type is required for a particular procedure, there must be a document about the alien in the procedure's folder. To signal any missing document, the system can compute:

$$\mathit{required} \cap \overline{(\mathit{type}^{\smile}; (\mathit{procedureFolder} \cap \mathit{alien}; \mathit{alien}^{\smile}))} \tag{7.14}$$

Rule: discrepancy1 Not a computer, but an employee of the IND decides on any given application. So an intended decision may turn out different from the answer given by the rule engine. This is formalized in equation 7.19, which requires the following three relations.

$$\begin{aligned} \mathit{decision} &: \mathit{Procedure} \times \mathit{Decision} && (7.15)\\ \mathit{decisionVal} &: \mathit{Decision} \rightarrow \mathit{DecisionValue} && (7.16)\\ \mathit{intension} &: \mathit{Procedure} \rightarrow \mathit{DecisionValue} && (7.17) \end{aligned}$$

The rule that says the two must be equal is:

$$\mathit{decision}; \mathit{decisionVal} \;=\; \mathit{intension} \tag{7.18}$$

This rule may be violated, but any discrepancy between the IND's decision and the rule engine's computation are made visible for accountability purposes. The deviant decisions can be computed by:

$$(\overline{(\mathit{decision}; \mathit{decisionVal})} \cup \overline{\mathit{intension}}) \cap (\mathit{intension} \cup \mathit{decision}; \mathit{decisionVal}) \tag{7.19}$$

Rule: discrepancy2 Similar to the previous rule, the intended decision may deviate from the final decision. We use definitions 7.17 and **??**. The rule that says the two must be equal reads:

$$\mathit{intension} \;=\; \mathit{decisionVal} \tag{7.20}$$

This rule reports the following signals:

$$(\overline{\mathit{intension}} \cup \overline{\mathit{decisionVal}}) \cap (\mathit{decisionVal} \cup \mathit{intension}) \tag{7.21}$$

So far, the formalization defines (a selection of) the rules that govern the decision process.

Exercise 7.1. [title=derive the signals, label=ex:discrepancy2] Prove that the signals obtained by 7.21 are the violations of rule 7.20

In order to prove that the signals obtained by 7.21 violate rule 7.20, we compute those signals. These signals are the complement of rule 7.20, i.e. the violations of that rule.

$$\begin{array}{ll}
 & \overline{\mathit{intension} \;=\; \mathit{decisionVal}} \\
= & \qquad\qquad \{\text{by law 4.2}\} \\
 & \overline{\mathit{intension} \vdash \mathit{decisionVal} \cap \mathit{decisionVal} \vdash \mathit{intension}} \\
= & \qquad\qquad \{\text{De Morgan: law 3.14}\} \\
 & \overline{\mathit{intension} \vdash \mathit{decisionVal}} \cup \overline{\mathit{decisionVal} \vdash \mathit{intension}} \\
= & \qquad\qquad \{\text{law 4.1}\} \\
 & \overline{\overline{\mathit{intension}} \cup \mathit{decisionVal}} \cup \overline{\overline{\mathit{decisionVal}} \cup \mathit{intension}} \\
= & \qquad\qquad \{\text{De Morgan: law 3.14}\} \\
 & (\mathit{intension} \cap \overline{\mathit{decisionVal}}) \;\cup\; (\mathit{decisionVal} \cap \overline{\mathit{intension}}) \\
= & \qquad\qquad \{\text{distributivity (page \textbf{??}) and absorption}\} \\
 & (\overline{\mathit{intension}} \cup \overline{\mathit{decisionVal}}) \;\cap\; (\mathit{decisionVal} \cup \mathit{intension})
\end{array}$$

7.5 Logical data model

This section discusses the key principles that govern the logical data model. It tells how applications for residence permits, documents that are in the alien's file, procedures to handle applications and decisions are interrelated.

application

Aliens who wish to enter the Netherlands to stay must apply for a residence permit. An application for residence is an application as intended by the administrative law. It is defined in AWB 1.3 sub 3: An *application* is a request of a stakeholder to take a decision.

Figure 7.3 shows a conceptual analysis of this theme. This analysis contains the concepts discussed in this section.

initiate procedures Every application has been submitted on behalf of precisely one alien. Conceptually it is possible to combine applications of different persons into one, for instance for families or groups that stay together. For each individual, a separate procedure is conducted however. For each application, a procedure is conducted that leads to a decision. The formalization (equation 7.24) requires the following two relations.

$$\mathit{behalf} \;:\; \mathit{Application} \times \mathit{Alien} \tag{7.22}$$

$$\mathit{application} \;:\; \mathit{Procedure} \to \mathit{Application} \tag{7.23}$$

Besides, we use definition 7.12 (alien). This rule maintains:

$$\mathit{behalf} \;\vdash\; \mathit{application}^{\smile}; \mathit{alien} \tag{7.24}$$

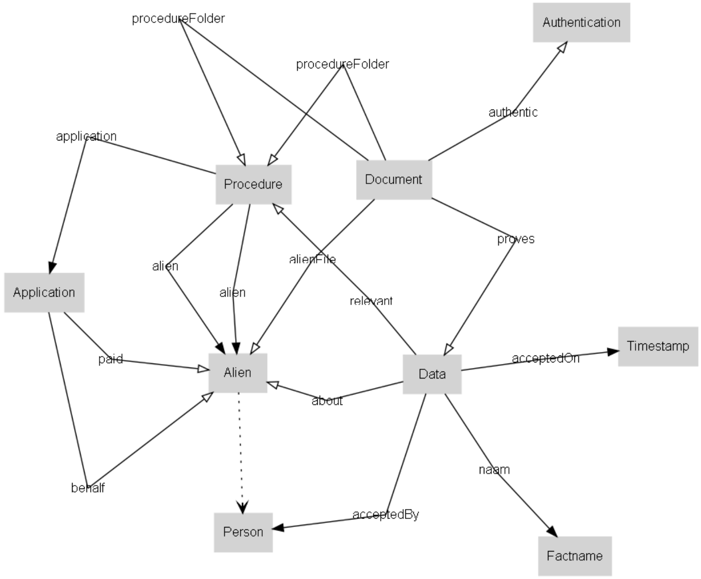


Figure 7.3: Conceptual analysis of Applications

alien file Documents that are in the folder of a particular procedure, are implicitly resident in the alien's folder too. We use definitions 7.12, 7.10, and **??**. This rule maintains:

$$\mathit{procedureFolder}; \mathit{alien} \ \vdash \ \mathit{alienFile} \tag{7.25}$$

relevant facts All data about a person that has been accepted as a fact is relevant for every decision taken about that person. To formalize this (in equation 7.28) the following two relations are introduced:

$$\mathit{about} \ : \ \mathit{Data} \times \mathit{Alien} \tag{7.26}$$

$$\mathit{relevant} \ : \ \mathit{Data} \times \mathit{Procedure} \tag{7.27}$$

Besides, we use definition 7.12 (alien). This rule maintains:

$$\mathit{about}; \mathit{alien}^{\smile} \ \vdash \ \mathit{relevant} \tag{7.28}$$

procedure file All evidence that is relevant for establishing a fact is relevant for the procedure in which that fact is used. In order to formalize this, we introduce relation 7.29:

$$\mathit{proves} \ : \ \mathit{Document} \times \mathit{Data} \tag{7.29}$$

Besides, we use definitions 7.27 and 7.10 to formalize requirement **??** (page **??**): This rule maintains:

$$\mathit{proves}; \mathit{relevant} \ \vdash \ \mathit{procedureFolder} \tag{7.30}$$

unpaid All applications incur legal fees, which have to be paid by or on behalf of the applicant. This rule signals all applications with fees due. In order to formalize this, we introduce relation 7.31:

$$\mathit{paid} \;:\; \mathit{Application} \times \mathit{Alien} \tag{7.31}$$

Besides, we use definitions 7.12 and 7.23 to formalize requirement **??** (page **??**): This rule signals:

$$\mathit{application}^{\smile};\mathit{alien} \;\vdash\; \mathit{paid} \tag{7.32}$$

This rule reports the following signals:

$$\mathit{application}^{\smile};\mathit{alien} \cap \overline{\mathit{paid}} \tag{7.33}$$

authentication All documents in a folder must eventually be authenticated. All documents that have not yet been authenticated are signalled. In order to formalize this, we introduce relation 7.34:

$$\mathit{authentic} \;:\; \mathit{Document} \times \mathit{Authentication} \tag{7.34}$$

Besides, we use definition 7.10 (procedureFolder) to formalize requirement **??** (page **??**). This rule signals:

$$\mathit{procedureFolder};\mathit{procedureFolder}^{\smile} \;\vdash\; \mathit{authentic};\mathit{authentic}^{\smile} \tag{7.35}$$

This rule reports the following signals:

$$\mathit{procedureFolder};\mathit{procedureFolder}^{\smile} \cap \overline{(\mathit{authentic};\mathit{authentic}^{\smile})} \tag{7.36}$$

This concludes our discussion of rules from two themes, decision making and the application procedure. In the actual INDiGO project, slightly over 100 rules were used to get sufficient evidence that all key ideas could be implemented by means of the COTS-components chosen for the IND. For the purpose of this chapter, it is unneccessary to discuss them all. The discussion presented here gives the full flavour of these rules.

The next section presents the data analysis that was generated for this project. It has been generated entirely by the ADL compiler. Therefore we can claim with mathematical certainty that this data model can accommodate all data needed to maintain the rules from the previous sections.

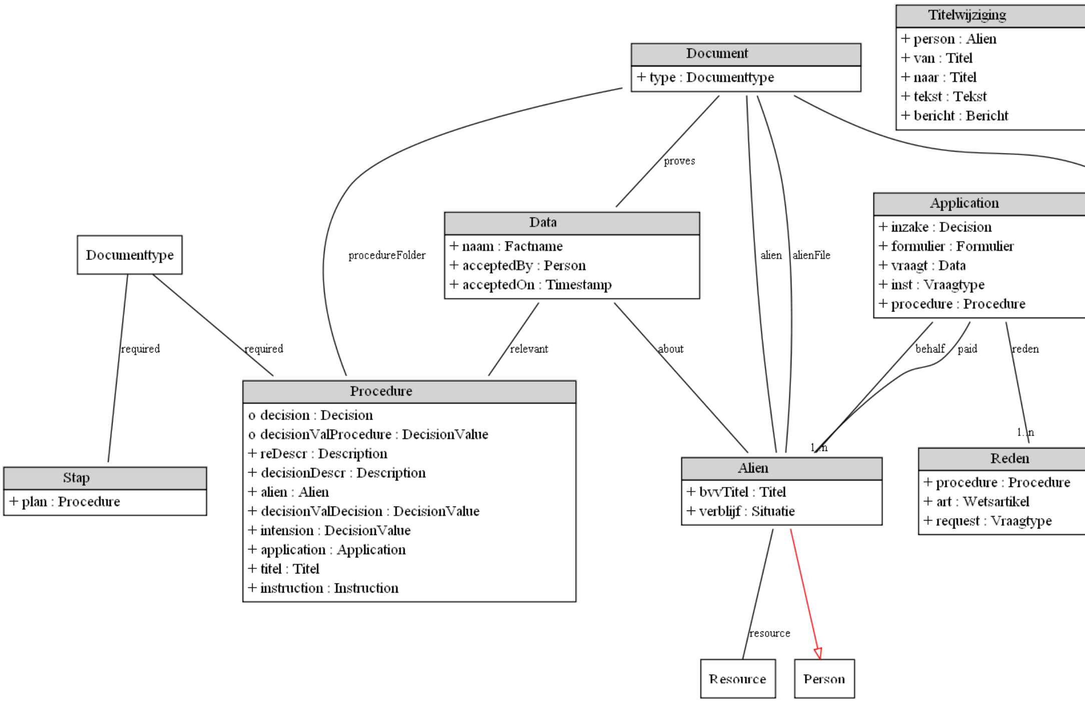


Figure 7.4: Class Diagram of IND

7.6 Data Analysis

The requirements, which are listed in chapter **??**, have been translated into the class diagram in figure 7.4. There are 21 data sets, 23 associations, one generalisation, and no aggregations. The ADL-script has a total of 33 concepts. From this script, a data model was derived in which all rules from the ADL-script can be represented. This data model has been generated by the ADL compiler. In practice, designers can extend this model with other attributes and other classes. Restricing the model by removing attributes is not allowed, because that would jeopardize compliance to the rules.

The relevance of this data model lies in a comprehensive analysis over the various COTS components in INDiGO. For example, data about aliens are stored in the Oracle-Siebel component, decision trees are built in the Be Informed component, and documents are stored in the document management component. So any rule involving documents related to decisions about aliens affects these three components. The rules involved in the analysis typically affect several components. They describe the semantic consistence between components, which is most visible on the service bus level.

From the data analysis, constraints have been derived (by the ADL compiler) that affect the implementation in software. These constraints are tabulated for the convenience of programmers who configure the data stores. Table 7.1 shows (a selection of) INDiGO's associations

relation	total	surjective
$required_{[Documenttype,Procedure]}$	$\surd$	
$procedureFolder_{[Document,Procedure]}$		
$alien_{[Document,Alien]}$		
$required_{[Documenttype,Stap]}$		
$alienFile_{[Document,Alien]}$		
$behalf_{[Application,Alien]}$		
$paid_{[Application,Alien]}$		
$proves_{[Document,Data]}$		
$about_{[Data,Alien]}$		
$authentic_{[Document,Authentication]}$		
$relevant_{[Data,Procedure]}$		
$resource_{[Alien,Resource]}$		

Table 7.1: Associations and their multiplicity constraints

and table 7.1 shows the required attributes of the data set 'Procedures' with its multiplicity constraints. This data set generates the following

attribute	type	compulsory	unique
I	Procedure	$\surd$	$\surd$
decision	Decision		$\surd$
decisionValProcedure	DecisionValue		
reDescr	Description	$\surd$	
decisionDescr	Description	$\surd$	
alien	Alien	$\surd$	
decisionValDecision	DecisionValue	$\surd$	
intension	DecisionValue	$\surd$	
application	Application	$\surd$	
titel	Titel	$\surd$	
instruction	Instruction	$\surd$	

Table 7.2: Associations and their multiplicity constraints

signals 7.4 and 7.4 on page 134.

–

$$decision; decisionVal \ = \ intension$$

–

$$intension \ = \ decisionVal$$

In a similar fashion, the ADL compiler generates design information for the other data sets. These are not shown here for brevity's sake.

7.7 Way of Working

The INDiGO project marked the first occasion in which Ampersand has contributed to a large IT project in practice. This experience has not only shown the value of a formal approach, but has also taught some valuable lessons.

ADL-specifications and the formulas that occur in the function specification are meant for requirements engineers only. Ampersand eliminates the need to share any formalism with user communities, officials, and members of the demand organisation. Designing large information systems is much like the design of a bridge: the formulas and computations required to make a good design (and prevent the bridge from falling down) need only be shared with professionals who must understand these things.

Formalization saves time. The INDiGO architecture has proved robust, and was not changed a lot during the project. This is a direct result of its consistency, which eliminates the need to change the architecture. The large amount of concrete details, that were already designed during the tendering phase, is also a consequence of doing things right the first time.

Correctness of an ADL specification means several things. Firstly, the type checker ensures absence of type errors. This means that the specification is representable on a database system, so the system can actually be built. In the specification we encountered two types of rule: rules that must be maintained and rules whose violations are signalled. In the former case, correctness means that the software must ensure the truth of a rule at any moment in time. The other rules may be violated, but their violations are signalled so that people may intervene. Thus, correctness means that the system of people, supported by computers, maintains all rules throughout time. It is measurable, because the outcome of each rule is either true or false at any given moment. It is achievable, because each requirement can be implemented in software. The requirements are relevant, because they have been discussed and scrutinized by the IND. But the most important benefit of correctness is what is not there: there is no rework due to mistakes and there are no corrective actions. This saves a lot of frustration. That is why we need to bother about correctness.

Since every requirement is universally valid, they are implmented uniformly throughout the organisation. This kind of uniformity saves effort and creates a sense of simplicity. Even though the art of writing requirements in ADL is not easy, the resulting system is as simple as can be considering the requirements.

Chapter 8

ADL

This chapter defines the structure, the syntax and the semantics of ADL. As abbreviation, ADL stands for A Description Language. This specification can be used to define a repository in which to store rules in their contexts.

This chapter defines ADL in a formal way. Any implementation that satisfies the properties defined in this chapter is a valid implementation of ADL.

This chapter is built up as follows. The first section specifies the terminology of concepts, relations, rules, patterns and contexts informally. It is meant as an introduction to the language used to describe ADL. Section 8.2 specifies the terminology of concepts, relations, rules, valuations, and patterns formally, in a way that a rule repository can be built. Finally, section 8.3 specifies a family of languages that can be used to fill the repository. Any language and any repository that satisfies this specification is a proper implementation of ADL.

8.1 Terminology

This section starts with informal definitions of the words used in ADL. After that, all words are introduced formally, with explanations on the way.

The following words are used

Atom	An object that cannot be divided in smaller parts.
Concept	A collection of things of one kind. A concept corresponds to a set of atoms.
Context	A set of rules with power of law (within that context).

Declaration	A statement saying that a relation exists by mentioning the name, source- and target concepts of that relation. (synonym: Relation definition, or Definition)
Identifier	A string that identifies a concept (if it starts with a capital letter) or a relation (if it starts with a lower case letter).
Pair	A combination of two atoms, one to the left and one to the right.
Pattern	A set of rules inside an architecture that represents a generic solution to a design problem. A pattern has a name and can be used within a context. Patterns can be inserted into or removed from an architecture.
Relation	A set R of pairs, where each pair p contains two atoms, s and t. Pair p represents a fact $\mathit{fact}_R(p)$. Atom s is an instance of a concept $\mathit{source}(R)$ and t is an instance of a concept $\mathit{target}(R)$.
RelationRef	part of a rule that denotes one relation in any context where this rule applies. Usually, the name alone is sufficient to identify the relation uniquely. In some cases it is necessary to write the full signature (i.e. the name together with source- and target concepts) to prevent ambiguity.
Rule	A statement in relation algebra that restricts the possible interpretations of relations used in the rule.
Set	A collection of atoms.
Valuation	An assignment of pairs to every relation reference in a relation. Every particular valuation makes a rule true or false within its context.

8.1.1 Concepts

Concepts are treated as sets. For example, you may think of the concept `Person`, as a set with elements `Abraham`, `Bob`, and `Charlotte`. All known properties of sets are valid for concepts as well. For instance, if a concept C has element `Bob` and $C \subseteq C'$, then `Bob` is an element of C' also. The statement `Bob` $\in C$ is pronounced ”Bob is an element of C” and it means that there exists a C called Bob. Elements of concepts are called *atom*, indicating that these elements are the smallest things to consider. The collection of all conceivable atoms is sometimes called ‘universe of discourse’ in the literature. We talk about the *type* of an atom as in `Abraham`, `Bob`, and `Charlotte` have type `Person`. The set of atoms that corresponds to a concept is called the *population* of that concept.

atom

type

population

Since relations are (by definition) sets, a relation has elements too. For example, the pair (`Bush`, `USA`) might be an element of a relation

source
target
type

`president`. Two pairs are equal if their left atoms are equal and their right atoms are equal. All pairs in a relation have the same types. That is: the left atoms of all pairs in a relation have the same type, and so do the right atoms. The concept to the left is called *source* and the concept to the right is the *target*. Within a *context*, every relation is defined uniquely by its name and the source and target concepts. We write $r_{[A,B]}$ to denote the relation with name r, source A and target B, The *type* of $r_{[A,B]}$ is defined as $[A \times B]$. We write $[A \times B] \sqcap [C \times D]$ to denote the most specific of two types $[A \times B]$ and $[C \times D]$. The notation $[A \times B] \sqcup [C \times D]$ denotes the most generic of both types.

8.1.2 Rules

ADL uses the idea of patterns to collect a number of rules that describe one particular theme[1]. So rules are defined within a *pattern*, which is therefore a collection of rules. When that pattern is used in a context, all rules defined in that pattern apply within the context. Within the context itself, extra rules may be defined for the purpose of glueing patterns together[2]. So, all rules that apply in a context are the ones defined in patterns used by the context plus the rules defined within that context.

The syntax of rules is given in Backus Naur Form (BNF):

```
Rule ::= Expression "|-" Expression "." .
Rule ::= RelationRef "=" Expression "." .
```

You can read the first line as: a rule can be an expression followed by the `|-` symbol, followed by another expression and terminated by a dot (full stop symbol). An example of a rule is `elem;subset |- elem`, in which `elem;subset` is an expression and `elem` is an expression too. Similarly, you read the second line as: a rule can be a RelationRef followed by the `=` symbol, followed by an expression and terminated by a dot.

expression

The syntax of expressions is given by:

```
Expression ::= RelationRef .
Expression ::= Expression "~" .
Expression ::= "-" Expression .
Expression ::= Expression ";" Expression .
Expression ::= Expression "/\" Expression .
Expression ::= Expression "\/" Expression .
Expression ::= "(" Expression ")" .
```

[1] Patterns are discussed in section 8.1.5

[2] Glueing of patterns is discussed in ??

The unary operators `~` and `-` bind stronger than any other operator, so the expression `r;s~` is different from `(r;s)~`. Of all three infix operators, `;` binds the strongest, followed by `/\` and finally `\/`. This means that `r/\s;p` is read `r/\(s;p)` and `r/\s\/p` is read `(r/\s)\/p`.

A relation reference (`RelationRef`) is part of a rule that denotes one relation in any context where this rule applies. For instance, the rule `elem;subset |- elem` contains relation references `elem` and `subset`. The source- and target concepts together are called the *type*. ADL requires that the type of every relation reference is determined, in order to bind that reference to one specific relation. As ADL can often deduce the type from the context, the name alone is often sufficient to identify a relation uniquely. To prevent ambiguities, you may write a relation reference in full, i.e. the name together with source- and target concepts as (for example) `elem[Atom*Set]`.

type

Identifiers of a relation always start with a lower case letter (`mIdent`), but names that identify concepts always start with a capital letter (`cIdent`). ADL has one special relation: `I`, represents the identity relation. It is predefined in ADL for every concept. The syntax of relation references is given by:

```
RelationRef ::= mIdent .
RelationRef ::= mIdent "[" cIdent "*" cIdent "]" .
RelationRef ::= mIdent "[" cIdent "]" .
RelationRef ::= "I" .
RelationRef ::= "I" "[" cIdent "]" .
```

If you specify the type, as in `president[Name*Country]`, you know for sure that this is the type used by ADL. If you specify only one concept, as in `friend[Name]`, ADL interprets this as `friend[Name*Name]`. If you do not specify any concepts, ADL will try to find out which type you mean.

Rules, expressions, and relation references have types. The type of `r` is denoted by $\mathcal{T}\texttt{r}$. Types are defined as follows:

$$\begin{array}{ll}
\mathcal{T}\texttt{m[A*B]} & = [\texttt{A} \times \texttt{B}] \\
\mathcal{T}\texttt{I[A]} & = [\texttt{A} \times \texttt{A}] \\
\mathcal{T}\texttt{r[A*B];s[B*C]} & = [\texttt{A} \times \texttt{C}] \\
\mathcal{T}\texttt{r[A*B]\textasciitilde} & = [\texttt{B} \times \texttt{A}] \\
\mathcal{T}\texttt{r/\textbackslash s} & = \mathcal{T}\texttt{r} \sqcap \mathcal{T}\texttt{s} \\
\mathcal{T}\texttt{r\textbackslash/s} & = \mathcal{T}\texttt{r} \sqcap \mathcal{T}\texttt{s} \\
\mathcal{T}\texttt{r|-s} & = \mathcal{T}\texttt{r} \sqcap \mathcal{T}\texttt{s} \\
\mathcal{T}\texttt{m=r} & = \mathcal{T}\texttt{m} \sqcap \mathcal{T}\texttt{r} \\
\mathcal{T}\texttt{-r} & = \mathcal{T}\texttt{r}
\end{array}$$

current context

You always work in one particular context, called the *current context*. Every relation reference is bound to precisely one relation in your current context. Notice that the same relation reference may be bound to different relations in different contexts, because one rule (which is defined in a pattern) applies in all contexts that use this rule.

If you work in a context (e.g. the context of Marlays bank) you may define a new context (e.g. Mortgages) as an extension of an existing context. This means that all rules that apply in the context 'Marlays bank' apply in the context 'Mortgages' as well. The rules that apply in the generic context ('Marlays bank') are a subset of the rules that apply in the specific context ('Mortgages').

8.1.3 Relations

Any relation is a set of pairs. For example:

```
president = { ("Bush","USA"),
              ("Clinton","USA"),
              ("Sarkozy","France") }
```

declaration

This defines the contents of a relation. The following statement is called a *declaration*. It introduces a new relation and states its type:

```
president :: Name * Country
```

source
target

The concept on the left hand side is called the *source* and the right hand side concept is called *target*.

The left atom of every pair in a relation r is an element of the source of r. Similarly every right atom in r is an element of the target concept. Within any context, the name, source and target determine a relation uniquely. A tuple in a relation matches the type of that relation. That is: the left atom of a tuple is atom of the source of the relation in which that tuple resides. Idem for the target.

Any pair in relation r is also in relations of which r is a subrelation. The reason is that a relation is a set of pairs, so subsets are subrelations.

8.1.4 Valuations

Any rule is a statement that can be applied to many different atoms. For example, the rule:

```
president |- citizen .
```

This rule says specifies that the relation `president` is included in the relation `citizen`. So it says that you can be a president only if you are a citizen. We can apply this rule to different atoms, to see whether it is true or not. For instance, let

```
president = {("Bush","USA"), ("Clinton","USA"),
             ("Chirac","France")}
citizen   = {("Bush","USA"), ("Clinton","USA"),
             ("Chirac","France"), ("Joosten","Netherlands")}
```

In this example, the rule is obviously true. For every value we choose, we get the expected true or false statement: e.g.

> `"Bush" president "USA"` implies
> that `"Bush" citizen "USA"`.

is clearly true. On the other hand:

> `"Joosten" president "USA"` implies
> that `"Chirac" citizen "France"`.

is clearly false.

valuation

In order to discuss semantics of rules precisely, we introduce the notion of *valuation*. A valuation is one particular assignment of atoms to a rule that makes the rule true or false. For that purpuse, a valuation needs to allocate a value (i.e. a pair) to each relation reference in the rule. In the example we have two relation references (`president` and `citizen`), so a valuation gives values to each one. In the first example the following valuation was used:

```
president -> ("Bush","USA"); citizen -> ("Bush","USA")
```

The second example used a different valuation:

```
president->("Joosten","USA"); citizen->("Chirac","France")
```

For every valuation of rule r that contains a pair l, this pair is an element of a relation in each context in which r applies.

8.1.5 Patterns

Every relation used in a rule is defined in the same pattern as that rule.

A relation is bound to its definition, which is written inside a pattern used in the relation's context.

Any relation in a context is also known in more generic contexts. The reason is that a relation is a set of pairs, so subsets are subrelations.

A pattern used by a context is implicitly used by more specific contexts.

If one relation is a subrelation of another one (the super-relation), it means that they have compatible types and the subrelation is in the same or a more specific context than the super-relation.

In the following sections, formal specifications are elaborated that define ADL.

8.2 Formal Specifications

This section defines concepts, relations, rules, valuations, and patterns once again, but now formally.

8.2.1 Concepts

A concept is a collection of things of one kind.

The symbol $\sqsubseteq$ is called the 'isa' relation[3] when applied to concepts. For instance `Judge` $\sqsubseteq$ `Person` means that the concept `Judge` is more specific than the concept `Person`. The set of judges (corresponding to `Judge`) is included in the set of persons (which corresponds to `Person`). We write $A \sqcap B$ to denote the most specific of two concepts A and B. The notation $A \sqcup B$ denotes the most generic of both concepts.

The relation $\sqsubseteq$ (isa) expresses that one concept may be more specific than another.

$$\sqsubseteq :: \ \mathit{Concept} \times \mathit{Concept} \tag{8.1}$$

For instance: `Teacher` $\sqsubseteq$ `Person` means that `Teacher` is a more specific concept than `Person`. Just like $\subseteq$, $\sqsubseteq$ is reflexive, transitive, and antisymmetric. The relation $\sqsubseteq$ applies to concepts, whereas $\subseteq$ applies to sets.

[3] This relation is familiar to object-oriented programmers and artificial intelligence professionals.

The correspondence between concepts and sets is obtained by the function *pop*, which associates a set to every concept.

$$pop :: Concept \rightarrow Set \tag{8.2}$$

The relation $\sqsubseteq$ must satisfy the following property, for all concepts c and c':

$$c \sqsubseteq c' \;\Rightarrow\; pop(c) \subseteq pop(c') \tag{8.3}$$

This property says for instance that `Teacher` $\sqsubseteq$ `Person` implies that the set of teachers (corresponding to `Teacher`) is included in the set of persons (which corresponds to `Person`). That is: every atom in the set of teachers is also a member of the set of persons.

For any atom a that is element of a set that corresponds to a concept c, we say that a has type c. We use the following relation.

$$type :: Atom \times Concept \tag{8.4}$$

This relation is total, since every atom has a type. A type is a concept in whose population the atom occurs. It is defined by:

$$\forall a :: Atom; c :: Concept : \quad a \; type \; c \;\Leftrightarrow\; a \;\in\; pop(c) \tag{8.5}$$

pair
left atom
right atom

A combination of two atoms that occurs in a relation is called a *pair*. One atom is called the *left atom* and the other one is the *right atom*. The functions *left* and *right* yield the respective atoms.

$$left, right :: Pair \rightarrow Atom \tag{8.6}$$

Each pair is fully determined by its left and right atoms.

$$\forall l, l' :: Pair : l = l' \;\Leftrightarrow\; right(l) = right(l') \;\wedge\; left(l) = left(l') \tag{8.7}$$

source
target

When two atoms are linked in a relation, a pair implicitly links two concepts. The left concept is called the *source* and the right concept *target*. Functions *src* and *trg* map a pair to either concept.

$$src, trg :: Pair \rightarrow Concept \tag{8.8}$$

These functions are defined by:

$$\forall l :: Pair : \; src(l) = type(left(l)) \;\wedge\; trg(l) = type(right(l)) \tag{8.9}$$

Figure 8.1 illustrates the structure of the previously defined rules. This diagram shows an arrow for every function and a line for every relation. The tiny arrow halfway a line determines the order of arguments in each relation.

The following fragment represents the specification in terms of ADL:

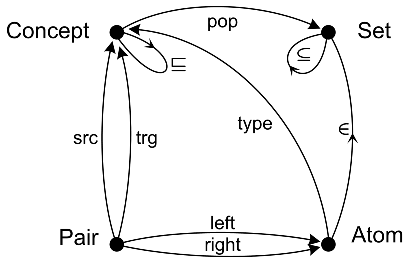


Figure 8.1: Structure of Concepts

```
PATTERN Concepts
 elem   :: Atom * Set.
 subset :: Set * Set [RFX,TRN,ASY].
 isa    :: Concept * Concept [RFX,TRN,ASY].
 pop    :: Concept * Set [UNI,TOT].
 isa    |- pop ; subset ; pop~.
 type   :: Atom * Concept [TOT].
 type    = elem ; pop~.
 left   :: Pair * Atom [UNI,TOT].
 right  :: Pair * Atom [UNI,TOT].
 I[Pair] = right ; right~ /\ left ; left~.
 src    :: Pair*Concept [UNI,TOT].
 trg    :: Pair*Concept [UNI,TOT].
 left   |- src ; pop ; elem~.
 right  |- trg ; pop ; elem~.
ENDPATTERN
```

8.2.2 Rules

Rules are the most prominent feature of ADL. They are defined in patterns and a pattern is used by any number of contexts. Which rule is defined in which pattern is represented by the function *definedIn*.

$$\mathit{definedIn} :: \mathit{Rule} \rightarrow \mathit{Pattern} \tag{8.10}$$

Which context uses which pattern is given by the relation *uses*:

$$\mathit{uses} :: \mathit{Context} \times \mathit{Pattern} \tag{8.11}$$

When a pattern is used in a context, all rules of that pattern apply within that context. So, we introduce a third relation, *appliesIn*, to express which rules apply in which context:

$$\mathit{appliesIn} :: \mathit{Rule} \times \mathit{Context} \tag{8.12}$$

These relations satisfy equation 8.13, which states that for each rule r and every context c:

$$r\ \mathit{appliesIn}\ c\ \Leftrightarrow\ c\ \mathit{uses}\ \mathit{definedIn}(r) \tag{8.13}$$

Within the context itself, extra rules may be defined for the purpose of glueing patterns together. So all rules that apply in a context are the ones defined in patterns used by the context plus the rules defined within that context.

New contexts can be defined as extensions of existing contexts. This is represented by the relation *specializes*.

$$\mathit{specializes} :: \mathit{Context} \times \mathit{Context} \tag{8.14}$$

This relation is reflexive, transitive, and antisymmetric. If you work in a context (e.g. the context of Marlays bank) you may define a new context (e.g. Mortgages) as an extension of an existing context. This means that all rules that apply in the context `Marlays` apply in the context `Mortgages` as well. The rules that apply in the generic context (`Marlays`) are a subset of the rules that apply in the specific context (`Mortgages`). This property is defined for every rule r and all contexts c and c' by equation 8.15.

$$r\ \mathit{appliesIn}\ c' \ \wedge\ c\ \mathit{specializes}\ c' \ \Rightarrow\ r\ \mathit{appliesIn}\ c \tag{8.15}$$

Relations are defined in a context. This information is available by means of the function *in*:

$$\mathit{in} :: \mathit{Relation} \rightarrow \mathit{Context} \tag{8.16}$$

If, for example (‘Relation 1’, ‘RAP’) occurs in relation *in*, this means that relation ‘Relation 1’ is available in context ‘RAP’.

relation reference

A *relation reference* is part of a rule that denotes a relation in the context of this rule. The relation *in* tells us which relation references are used in which rule.

$$\mathit{in} :: \mathit{RelationRef} \times \mathit{Rule} \tag{8.17}$$

This relation is surjective and total, since every relation reference is used in at least one rule and every rule contains at least one relation reference. The type of a relation reference is given by the function *sign*:

$$\mathit{sign} :: \mathit{RelationRef} \rightarrow \mathit{Definition} \tag{8.18}$$

Likewise, the type of a relation reference is given by another function *sign*:

$$\mathit{sign} :: \mathit{Relation} \rightarrow \mathit{Declaration} \tag{8.19}$$

This allows us to express in property 8.20 that every relation reference in a rule is bound to a relation in the applicable context. For every rule r, context c, and relation reference m

$$\begin{array}{l} m\ \mathit{in}\ r\ \wedge\ r\ \mathit{appliesIn}\ c\ \Rightarrow \\ \ \exists r' :: \mathit{Relation} :\ \mathit{sign}(m) = \mathit{sign}(r')\ \wedge\ \mathit{in}(r') = c \end{array} \tag{8.20}$$

current

You always work in one particular context, called the *current* context. Every relation reference is bound to precisely one relation in the current context[4].

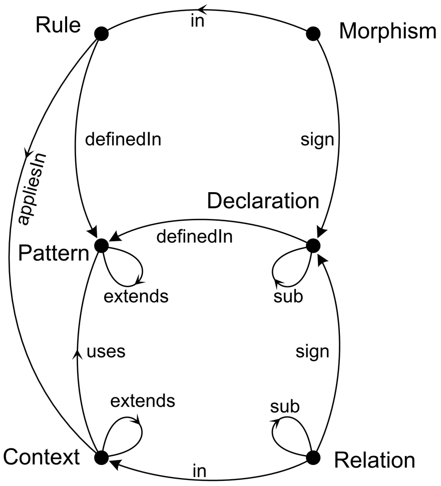


Figure 8.2: Structure of Rules

The following fragment represents the specification in terms of ADL:

```
PATTERN Rules
 definedIn :: Rule -> Pattern.
 uses      :: Context * Pattern.
 appliesIn :: Rule * Context.
 appliesIn  = definedIn;uses~.
 specializes   :: Context * Context [RFX,TRN,ASY].
 appliesIn ; specializes~ |- appliesIn.
 sign      :: RelationRef -> Declaration.
 in        :: RelationRef * Rule [SUR,TOT].
 sign      :: Relation -> Declaration.
 in        :: Relation -> Context.
 in ; appliesIn |- sign ; sign~ ; in.
ENDPATTERN
```

8.2.3 Relations

Within any context, the name, source, and target determine a relation uniquely. All relations r and s satisfy property 8.21.

$$r = s \;\Leftrightarrow\; sign(r) = sign(s) \wedge in(r) = in(s) \tag{8.21}$$

subrelation

The definition of *in* and *sign* are given in section 8.16 (pg. 150). We introduce the notion of *subrelation* to express that any pair in a sub-

[4] TBD: insert proof here

relation of r is also in r itself. Since a relation is a set of pairs, this corresponds to the subset relation. For this purpose we introduce the relation *sub*.

$$sub :: Relation \times Relation \tag{8.22}$$

This relation is reflexive, transitive, and antisymmetric. It satisfies property 8.23 for every pair l and all relations r and s:

$$l \in r \wedge r\ sub\ s \ \Rightarrow\ l \in s \tag{8.23}$$

Note that this property has the following consequence. For all relations s and r:

$$s\ sub\ r \ \Rightarrow\ s \vdash r \tag{8.24}$$

type

Since a relation is a set, we use the notation $l \in r$ to express that a pair l is an element of relation r. Every relation has a *type*, being the source- and target concepts of that relation. The following functions are used to retrieve the individual components of a declaration:

$$\begin{aligned} source, target &:: Declaration \rightarrow Concept && (8.25)\\ name &:: Declaration \rightarrow Identifier && (8.26)\end{aligned}$$

A name, source and target identify a relation uniquely, which is expressed for all declarations s and s' by equation 8.27:

$$\begin{aligned} s = s' \ \Leftrightarrow\ & name(s){=}name(s') \quad \wedge \\ & source(s){=}source(s') \quad \wedge \\ & target(s){=}target(s') \end{aligned} \tag{8.27}$$

Every tuple in a relation must match the declaration of that relation. That is: the type of the left atom of a tuple must be compatible with the source of the relation in which that tuple resides. Idem for the target. This rule states that you cannot put just anything in a relation, but the types of atoms must respect the type of the relation. For every pair l and relation r property 8.28 must hold.

$$l \in r \ \Rightarrow\ src(l) = source(sign(r)) \ \wedge\ trg(l) = target(sign(r)) \tag{8.28}$$

Relations *src* and *trg* are discussed in section 8.8 (pg. 148).

One declaration may be more generic than another, which is defined in the relation

$$sub :: Declaration \times Declaration \tag{8.29}$$

Like before, this relation is reflexive, transitive, and antisymmetric.

If one relation s is a subrelation of another relation r, both must have compatible types and s must be in the same or a more specific context than r. This requirement is given by equation 8.30

$$s\ sub\ r \ \Leftrightarrow\ sign(s)\ sub\ sign(r) \wedge in(s)\ specializes\ in(r) \qquad (8.30)$$

The relations *sign*, *in*, and *specializes* are discussed in section 8.14.

The structure of these rules is given in figure 8.3.

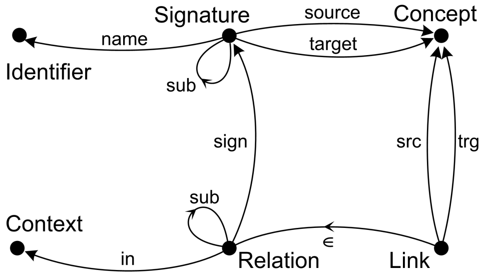


Figure 8.3: Structure of Relations

The following fragment represents the specification in terms of ADL:

```
PATTERN Relations
 source      :: Declaration -> Concept.
 target      :: Declaration -> Concept.
 in          :: Pair * Relation.
 in ; sign   |- src ; source~ /\ trg ; target~.
 I[Relation]  = sign;sign~ /\ in;in~.
 sub         :: Relation * Relation [RFX,TRN,ASY].
 in ; sub    |- in.
 name        :: Declaration -> Identifier.
 I[Declaration] = name; name~ /\
                  source; source~ /\
                  target; target~.
 sub         :: Declaration * Declaration [RFX,TRN,ASY].
 sub          = sign;sub;sign~ /\ in;specializes;in~.
ENDPATTERN
```

8.2.4 Valuations

valuation

In order to discuss semantics of rules precisely, we introduce the notion of *valuation*. A valuation is one particular assignment of atoms to a rule that makes the rule true or false (see also section 8.1.4). For this purpuse, a valuation needs to allocate a value (i.e. a pair) to each relation reference in the rule.

$$val :: RelationRef \times Pair \tag{8.31}$$

All pairs in a valuation are registered in the relation *in*

$$in :: Pair \times Valuation \tag{8.32}$$

The following definition relates rules to valuations.

$$val :: Rule \times Valuation \tag{8.33}$$

For each rule r, pair l and relation reference m, these relations must satisfy property 8.34.

$$m\ in\ r \wedge m\ val\ l\ \Rightarrow\ \exists v :: Valuation\ : r\ val\ v\ \wedge\ l\ in\ v \tag{8.34}$$

The definition of $in_{[RelationRef,Rule]}$ is given in section 8.17 (pg. 150).

For every valuation of rule r that contains a pair l, and every valuation v, l is an element of a relation in each context c in which r applies.

$$\begin{array}{l} l\ in\ v \wedge r\ val\ v \wedge r\ appliesIn\ c\ \Rightarrow \\ \ \exists r' :: Relation : l \in\ r' \wedge in(r')\ =\ c \end{array} \tag{8.35}$$

The definition of *appliesIn* is given in section 8.12 (pg. 149) and *in* is discussed in section 8.16 (pg. 150).

The structure of these definitions is illustrated in figure 8.4.

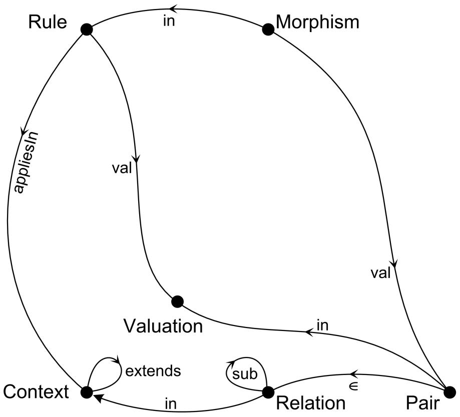


Figure 8.4: Structure of Valuations

The following fragment represents the specification in terms of ADL:

```
PATTERN Valuations
 val               :: Rule * Valuation.
 val               :: RelationRef * Pair.
 in                :: Pair * Valuation.
 in~;val           |- val;in~.
 in;val~;appliesIn |- in;in.
ENDPATTERN
```

Valuations are used to define the semantics of ADL. We introduce the notation $\mathcal{C} \models a\ r\ b$ to denote that the value $a\ r\ b$ of a relation r is true in context $\mathcal{C}$.

$$\begin{array}{llll}
\mathcal{C} \models a\ \mathtt{m}\ b & \text{iff}\ \ \mathtt{m}\ \mathit{val}\ (a,b) & & \wedge \\
 & \quad \exists r :: \mathit{Rule} : & \mathtt{m}\ \mathit{in}\ r\ \wedge\ r\ \mathit{appliesIn}\ \mathcal{C} & \wedge \\
 & \quad \exists r :: \mathit{Relation} : & (a,b)\ \mathit{in}\ r\ \wedge\ \mathit{in}(r) = \mathcal{C} & \\
\mathcal{C} \models a\ (\mathtt{r;s})\ b & \multicolumn{3}{l}{\text{iff there exists } c \text{ such that } \mathcal{C} \models a\ r\ c\ \wedge\ \mathcal{C} \models c\ s\ b} \\
\mathcal{C} \models a\ (\mathtt{r!s})\ b & \multicolumn{3}{l}{\text{iff for all } c \text{ such that } \mathcal{C} \models a\ r\ c\ \vee\ \mathcal{C} \models c\ s\ b} \\
\mathcal{C} \models a\ (\mathtt{r/\backslash s})\ b & \multicolumn{3}{l}{\text{iff } \mathcal{C} \models a\ \mathtt{r}\ b\ \wedge\ \mathcal{C} \models a\ \mathtt{s}\ b} \\
\mathcal{C} \models a\ (\mathtt{r\backslash/s})\ b & \multicolumn{3}{l}{\text{iff } \mathcal{C} \models a\ \mathtt{r}\ b\ \vee\ \mathcal{C} \models a\ \mathtt{s}\ b} \\
\mathcal{C} \models a\ (\mathtt{r\tilde{}})\ b & \multicolumn{3}{l}{\text{iff } \mathcal{C} \models b\ \mathtt{r}\ a} \\
\mathcal{C} \models a\ (\mathtt{-r})\ b & \multicolumn{3}{l}{\text{iff } \mathcal{C} \not\models b\ \mathtt{r}\ a} \\
\mathcal{C} \models a\ \mathtt{I}\ b & \multicolumn{3}{l}{\text{iff } \mathcal{C} \models a = b}
\end{array}$$

8.2.5 Patterns

A pattern is a collection of rules plus the declarations of all relation references used in these rules. The structure of the following definitions is illustrated in figure 8.2 on page 151. The function *definedIn* for rules is discussed on page 149. Besides that one, we need the following function to convey which declarations are defined in which patterns:

$$\mathit{definedIn} :: \mathit{Declaration} \rightarrow \mathit{Pattern} \tag{8.36}$$

Every relation reference m used in a rule r is defined in the same pattern as that rule and every declaration defined in that pattern is used in one of its rules.

$$m\ \mathit{in}\ r\ \Leftrightarrow\ \mathit{definedIn}(\mathit{sign}(m))\ =\ \mathit{definedIn}(r) \tag{8.37}$$

The relations *in* and *sign* are discussed in section 8.17 (pg. 150).

A relation r is bound to a declaration, which is defined in a pattern used in the relation's context.

$$\mathit{in}(r)\ \mathit{uses}\ \mathit{definedIn}(\mathit{sign}(r)) \tag{8.38}$$

The relations *in*, *sign*, and *uses* in this property are discussed in section 8.16.

Any relation r in a context c is also known in more generic contexts than c. The reason is that a relation is a set of pairs, so subsets are subrelations.

$$in(r)\ specializes\ c \ \Rightarrow\ in(r)\ =\ c \tag{8.39}$$

The definition of *specializes* is given in section 8.14 (pg. 150).

A pattern p used by a context c is implicitly used by more specific contexts c':

$$c'\ specializes\ c\ \wedge\ c\ uses\ p\ \ \Rightarrow\qquad c'\ uses\ p \tag{8.40}$$

The following fragment represents the specification in terms of ADL:

```
PATTERN Patterns
 definedIn      :: Declaration -> Pattern.
 in             |- sign ; definedIn ; definedIn~.
 in~;sign       |- uses ; definedIn~
 in ; specializes   |- in
 specializes ; uses |- uses
ENDPATTERN
```

The structure of the complete specification is illustrated in figure 8.5.

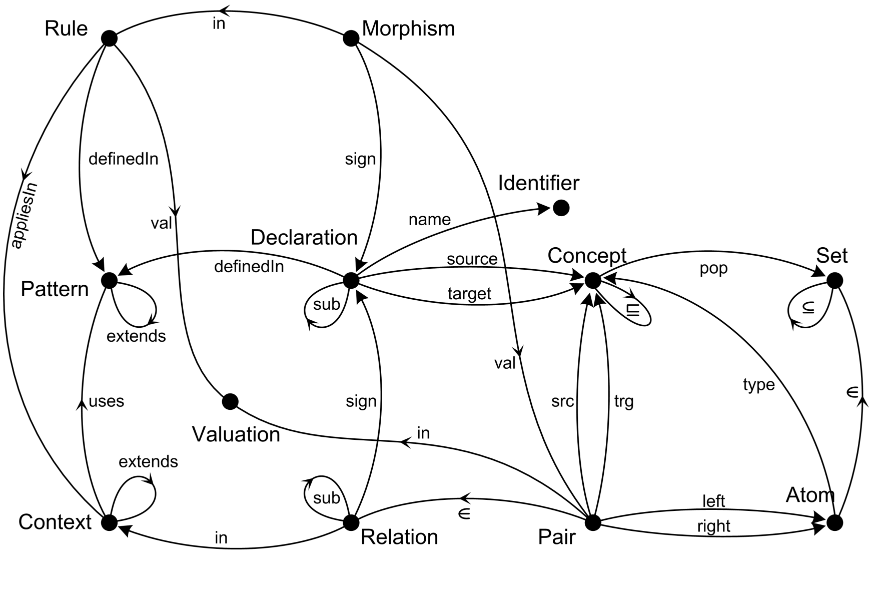


Figure 8.5: Entire Structure of ADL

8.3 Language

ADL is a relation algebra [22], which is used to express business rules. Every business rule is a formal representation of a business requirement.

Using definitions from [22], this section introduces a family of languages for expressing business rules suitable for use in the Ampersand approach. The language ADL satisfies the requirements for such a language and is therefore a suitable language for Ampersand. The presentation in this paper assumes that the reader is familiar with relation algebras.

Rules are represented in an algebra $\langle \mathcal{R}, \cup, \overline{\ }, ;, ^{\smile}, \mathbb{I} \rangle$, in which $\mathcal{R}$ is a set of relations, $\cup$ and ; are binary operators of type $\mathcal{R} \times \mathcal{R} \to \mathcal{R}$, $\overline{\ }$ and $^{\smile}$ are unary operators of type $\mathcal{R} \to \mathcal{R}$ and $\mathbb{I}$ represents an identity relation, and satisfying the following axioms for all relations $r, s, t \in \mathcal{R}$:

$$
\begin{aligned}
r \cup s &= s \cup r && (8.41)\\
(r \cup s) \cup t &= r \cup (s \cup t) && (8.42)\\
\overline{\overline{x} \cup \overline{y}} \cup \overline{\overline{x} \cup y} &= x && (8.43)\\
(r; s); t &= r; (s; t) && (8.44)\\
(r \cup s); t &= r; t \ \cup \ s; t && (8.45)\\
r; \mathbb{I} &= r && (8.46)\\
r^{\smile\smile} &= r && (8.47)\\
(r \cup s)^{\smile} &= r^{\smile} \cup s^{\smile} && (8.48)\\
(r; s)^{\smile} &= s^{\smile}; r^{\smile} && (8.49)\\
r^{\smile}; \overline{r; y} \cup \overline{s} &= \overline{s} && (8.50)
\end{aligned}
$$

To enrich the expressive power for users, other operators are used as well. These are the binary operators $\cap$, $\dagger$, $\vdash$, and $\equiv$, all of which are of type $\mathcal{R} \times \mathcal{R} \to \mathcal{R}$. They are defined by:

$$
\begin{aligned}
r \cap s &= \overline{\overline{s} \cup \overline{r}} && (8.51)\\
r \dagger s &= \overline{\overline{s}; \overline{r}} && (8.52)\\
r \vdash s &= \overline{r} \cup s && (8.53)\\
r \equiv s &= (r \vdash s) \cap (s \vdash r) && (8.54)
\end{aligned}
$$

An algebra of concepts $\langle \mathcal{C}, \sqsubseteq, \top, \bot \rangle$ is used to represent relations, in which $\mathcal{C}$ is a set of Concepts, $\sqsubseteq$: $\mathcal{C} \times \mathcal{C}$, and $\top \in \mathcal{C}$ and $\bot \in \mathcal{C}$, satisfying the following axioms for all classes $A, B, C \in \mathcal{C}$:

$$\begin{array}{rcll}
 & & A \sqsubseteq \top & (8.55) \\
 & & \bot \sqsubseteq A & (8.56) \\
 & & A \sqsubseteq A & (8.57) \\
A \sqsubseteq B \wedge B \sqsubseteq A & \Rightarrow & A = B & (8.58) \\
A \sqsubseteq B \wedge B \sqsubseteq C & \Rightarrow & A \sqsubseteq C & (8.59)
\end{array}$$

Concept $\bot$ is called 'anything' and $\top$ is called 'nothing'. Predicate $\sqsubseteq$ is called *isa*. It lets the user specify things such as 'an affidavit is a document'.

A binary relation $r : A{\times}B$ is a subset of the cartesian product $A \times B$, in which A and B are Concepts. In English: a relation $r : A{\times}B$ is a set of pairs $\langle a, b\rangle$ with $a \in A$ and $b \in B$. We call r the *name* of the relation, A the *source* of the relation, and B the *target* of the relation. A relation that is a function (i.e. a univalent and total relation) is denoted as $r : A{\to}B$

For the purpose of defining types, operator $\sqcap : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ and predicate $\diamond : \mathcal{C} \times \mathcal{C}$ are defined:

$$\begin{array}{rcll}
A \diamond B & \Leftrightarrow & A \sqsubseteq B \vee B \sqsubseteq A & (8.60) \\
A \sqsubseteq B & \Rightarrow & A \sqcap B \ = \ B & (8.61) \\
B \sqsubseteq A & \Rightarrow & A \sqcap B \ = \ A & (8.62) \\
\neg(A \diamond B) & \Rightarrow & A \sqcap B \ = \ \top & (8.63)
\end{array}$$

Predicate $\diamond$ defines whether two types are compatible and operator $\sqcap$ (the least upper bound) gives the most specific of two compatible types. Note that the conditions at the left of equations 8.61 through 8.63 are complete, so $\sqcap$ is defined in all cases. Also, in case of overlapping conditions $A \sqsubseteq B \wedge B \sqsubseteq A$, axiom 8.59 says that $A = B$. causing $A \sqcap B$ to be unique. ADL is restricted to concepts that are not $\bot$ nor $\top$, but the two are needed to signal type errors.

Any language with operators that satisfy axioms 8.41 through 8.59 is a suitable language for Ampersand. Table 8.2 gives the names and types of all operators. Not all expressions that can be written are type correct. An expression must satisfy the condition mentioned in table 8.2, otherwise it is undefined. A compiler for a language to be used for this purpose must therefore contain a type checker to ensure that all expressions satisfy the condtions mentioned in table 8.2.

The complete list of notations with meanings attached to them is given in table 8.3. In some cases, alternative meanings are also given. If x, y, and z are used in an unbound manner, universal quantification (i.e. "for all x, y, and z") is assumed.

Let $r : A{\times}B$ and $s : P{\times}Q$

name	notation	condition	type
relation	r		$[A, B]$
implication	$r \vdash s$	$A \diamond P \wedge B \diamond Q$	$[A \sqcap P, B \sqcap Q]$
equality	$r \equiv s$	$A \diamond P \wedge B \diamond Q$	$[A \sqcap P, B \sqcap Q]$
complement	$\overline{r}$		$[A, B]$
conversion	$r^{\smile}$		$[A, B]$
union	$r \cup s$	$A \diamond P \wedge B \diamond Q$	$[A \sqcap P, B \sqcap Q]$
intersection	$r \cap s$	$A \diamond P \wedge B \diamond Q$	$[A \sqcap P, B \sqcap Q]$
composition	$r; s$	$B \diamond C$	$[A, C]$
relative addition	$r \dagger s$	$B \diamond C$	$[A, C]$
identity	$\mathbb{I}_A$		$[A, A]$

Table 8.2: Definitions for typing

	notation	meaning
fact	$x\ r\ y$	there is a pair between x and y in relation r. Also: the pair $\langle x, y\rangle$ is an element of r, ($\langle x, y\rangle \in r$).
declaration	$r : A{\times}B$	There is a relation r with type $[A, B]$.
declaration	$f : A{\to}B$	There is a function f with type $[A, B]$. (A function is a relation)
implication	$r \vdash s$	if $x\ r\ y$ then $x\ s\ y$. Alternatively: $x\ r\ y$ implies $x\ s\ y$. Alternatively: r is included in s or s includes r.
equality	$r \equiv s$	$x\ r\ y$ is equal to $x\ s\ y$
complement	$\overline{r}$	all pairs not in r. Also: all pairs $\langle a, b\rangle$ for which $x\ r\ y$ is not true.
conversion	$r^{\smile}$	all pairs $\langle y, x\rangle$ for which $x\ r\ y$.
union	$r \cup s$	$x(r \cup s)y$ means that $x\ r\ y$ or $x\ s\ y$.
intersection	$r \cap s$	$x(r \cap s)y$ means that $x\ r\ y$ and $x\ s\ y$.
composition	$r; s$	$x(r; s)y$ means that there is a z such that $x\ r\ z$ and $z\ s\ y$.
r. addition	$r \dagger s$	$x(r \dagger s)y$ means that for all z, $x\ r\ z$ or $z\ s\ y$.
	$\overline{r} \dagger s$	$x(r \dagger s)y$ means that for all z, $x\ r\ z$ implies $z\ s\ y$.
	$r \dagger \overline{s}$	$x(r \dagger s)y$ means that for all z, $z\ s\ y$ implies $x\ r\ z$.
identity	$\mathbb{I}$	equals. Also: $a\ \mathbb{I}\ b$ means $a = b$.

Table 8.3: Semantics

Bibliography

[1] Sarbanes-Oxley Act of 2002.

[2] S. Ali, B. Soh, and T. Torabi. A novel approach toward integration of rules into business processes using an agent-oriented framework. *IEEE Transactions on Industrial Informatics*, 2(3):145– 154, August 2006.

[3] ANSI/INCITS 359: Information Technology. *Role Based Access Control Document Number: ANSI/INCITS 359-2004*. InterNational Committee for Information Technology Standards (formerly NCITS), February 2004.

[4] Eric Baardman and Stef Joosten. Procesgericht systeemontwerp. *Informatie*, 47(1):50–55, January 2005.

[5] Ron Barends. Activeren van de administratieve organisatie. Research report, Bank MeesPierson and Open University of the Netherlands, November 26, 2003.

[6] Chris J. Date. *What not how: the business rules approach to application development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[7] Umeshwar Dayal, Alejandro P. Buchmann, and Dennis R. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented databasesystem. In *Lecture notes in computer science on Advances in object-oriented database systems*, pages 129–143, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[8] Augustus De Morgan. On the syllogism: Iv, and on the logic of relations. *Transactions of the Cambridge Philosophical Society*, 10(read in 1860, reprinted in 1966):331–358, 1883.

[9] R. Deen. Het nut van case-handling - flexibel afhandelen. *Workflow magazine*, 6(4):10–12, 2000.

[10] Remco M. Dijkman, Luis Ferreira Pires, and Stef M.M. Joosten. Calculating with concepts: a technique for the development of business process support. In A. Evans, editor, *Proceedings of the UML*

2001 Workshop on Practical UML - Basesd Rigorous Development Methods Countering or Integrating the eXstremists, volume 7 of *Lecture Notes in Informatics*, Karlsruhe, 2001. FIZ.

[11] Remco M. Dijkman and Stef M.M. Joosten. An algorithm to derive use case diagrams from business process models. In *Proceedings IASTED-SEA 2002*, 2002.

[12] W. van Dommelen and S. Joosten. Vergelijkend onderzoek hulpmiddelen beheersing bedrijfsprocessen. Technical report, Anaxagoras and EDP Audit Pool, 1999.

[13] M. Fowler. *Analysis Patterns - Reusable Object Models.* Addison-Wesley, Menlo Park, 1997.

[14] Holger Herbst, Gerhard Knolmayer, Thomas Myrach, and Markus Schlesinger. The specification of business rules: A comparison of selected methodologies. In *Proceedings of the IFIP WG8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle*, pages 29–46, New York, NY, USA, 1994. Elsevier Science Inc.

[15] IEEE: Architecture Working Group of the Software Engineering Committee. *Standard 1471-2000: Recommended Practice for Architectural Description of Software Intensive Systems.* IEEE Standards Department, 2000.

[16] Rieks Joosten and Berco Beute. Requirements for personal network security architecture specifications. Technical Report Freeband/PNP2008/D2.4, TNO, The Netherlands, 2005.

[17] Rieks Joosten, Jan-Wiepke Knobbe, Peter Lenoir, Henk Schaafsma, and Geert Kleinhuis. Specifications for the RGE security architecture. Technical Report Deliverable D5.2 Project TSIT 1021, TNO Telecom and Philips Research, The Netherlands, August 2003.

[18] S. Joosten. Workpad - a conceptual framework for workflow process analysis and design. Unpublished, 1996.

[19] Stef M.M. Joosten. Rekenen met taal. *Informatie*, 42:26–32, juni 2000.

[20] Stef M.M. Joosten and Sandeep R. Purao. A rigorous approach for mapping workflows to object-oriented is models. *Journal of Database Management*, 13:1–19, October-December 2002.

[21] S.M.M. Joosten et.al. *Praktijkboek voor Procesarchitecten.* Kon. van Gorcum, Assen, 1st edition, September 2002.

[22] R.D. Maddux. *Relation Algebras*, volume 150 of *Studies in Logic and the Foundations of Mathematics.* Elsevier Science, 2006.

[23] Tony Morgan. *Business Rules and Information Systems: Aligning IT with Business Goals.* Addison Wesley, 2002.

[24] Ordina and Rabobank. Procesarchitectuur van het servicecentrum financieren. Technical report, Ordina and Rabobank, October 2003. presented at NK-architectuur 2004, www.cibit.nl.

[25] Bart Orriëns and Jian Yang. A rule driven approach for developing adaptive service oriented business collaboration. In *2006 IEEE International Conference on Services Computing (SCC 2006), 18-22 September 2006, Chicago, Illinois, USA*, pages 182–189. IEEE Computer Society, 2006.

[26] Charles Sanders Peirce. Note b: the logic of relatives. In C.S. Peirce, editor, *Studies in Logic by Members of the Johns Hopkins University (Boston).* Little, Brown & Co., 1883.

[27] Sudha Ram and Vijay Khatri. A comprehensive framework for modeling set-based business rules during conceptual database design. *Inf. Syst.*, 30(2):89–118, 2005.

[28] Ronald G. Ross. *Principles of the Business Rules Approach.* Addison-Wesley, Reading, Massachussetts, 1 edition, February 2003.

[29] Gunther Schmidt, Claudia Hattensperger, and Michael Winter. *Heterogeneous Relation Algebra*, chapter 3, pages 39–53. Advances in Computing Science. Springer-Verlag, 1997.

[30] Friedrich Wilhelm Karl Ernst Schröder. Algebra und logik der relative. In *Vorlesungen über die Algebra der Logik (exacte Logik).* Chelsea, first published in Leipzig, 1895.

[31] Walter A. Shewhart. *Statistical Method From the Viewpoint of Quality Control.* Dover Publications, New York, 1988 (originally published in 1939).

[32] J. F. Sowa and J. A. Zachman. Extending and formalizing the framework for information systems architecture. *IBM Systems Journal*, 31(3):590–616, 1992.

[33] Irma Valatkaite and Olegas Vasilecas. On business rules automation: The br-centric is development framework. In Johann Eder, Hele-Mai Haav, Ahto Kalja, and Jaan Penjam, editors, *Advances in Databases and Information Systems, 9th East European Conference, ADBIS 2005, Tallinn, Estonia, September 12-15, 2005, Proceedings*, volume 3631 of *Lecture Notes in Computer Science*, pages 349–364. Springer, 2005.

[34] Joost van Beek. Generation workflow - how staffware workflow models can be generated from protos business models. Master's thesis, University of Twente, 2000.

[35] R. van der Tol. Workflow-systemen zijn niet flexibel genoeg. *AutomatiseringGids*, (11):21, 2000.

[36] Erik van Essen, Berco Beute, and Rieks Joosten. Service domain design. Technical Report Freeband/PNP2008/D3.2, TNO, The Netherlands, 2005.

[37] Wan M. N. Wan-Kadir and Pericles Loucopoulos. Relating evolving business rules to software design. *Journal of Systems Architecture*, 50(7):367–382, 2004.

[38] Jeannette M. Wing. A symbiotic relationship between formal methods and security. In *CSDA '98: Proceedings of the Conference on Computer Security, Dependability, and Assurance*, pages 26–38, Washington, DC, USA, 1998. IEEE Computer Society.

Index