Rule Based Design

Stef Joosten; Lex Wedemeijer; Gerard Michels

December 2010

Contents

Chapter 1

Why study this book?

To help you decide whether to study this book, you might answer the following questions:

1. Do you participate, either now or in the future, in the making of business processes? (yes/no)
2. Does your participation have consequences in the organization you work for? (yes/no)
3. The business processes you make have various stakeholders. Must agreements among these people be concrete and explicit? (yes/no)
4. Are information systems being used that facilitate or enable these business processes? (yes/no)
5. Do you want to learn how to specify requirements in a precise way? (yes/no)

If you have answered all of the above questions with yes, then you are a member of the target audience of this book. Let us consider each question separately, to see what your reading effort might bring to you.

requirements elicitation

First of all, this book is written for people who are involved in the design of business processes. Architects, database administrators, functional designers, requirements engineers and consultants alike can benefit from the conceptual approach taken in this book. They will learn how to specify business processes as a composition of constraints, incrementally adding or changing rules in the course of *requirements elicitation*. All professionals who specify, design, build, or change business processes can enrich their knowledge and skills. Other stakeholders however, such as patrons, users of software applications, business partners, process owners, managers, and tool vendors, might find this book too detailed

for their purposes, because this book is about the 'how' rather than the 'why'.

Second, this book is about practice. It is meant to have consequences in the organization you work for. Please take a moment to think about your own role in the innovation process. Are you the one to make requirements explicit and concrete? If you are an auditor, it is your task to note whether or not requirements are fulfilled. If you are a scientist, it is your job to reflect on requirements. If you are an observer of business processes, you will not make requirements explicit and concrete. In all of these roles, you will scrutinize requirements rather than create them and make them explicit. However, if you are an information architect or a requirements engineer, then this is what you are hired to do. The skills described in this book will help you to succeed.

Yet, you may have already browsed through this book, seen a formula, and wondered how practical this book is. The formulas in this book make natural language precise; that is their practical value. This book explains what they mean and how you can benefit. If you learn to express yourself formally, the computer can transform your functional requirements into a full fledged functional specification. System designers can take these functional specs, add other, non-functional requirements (e.g. about cost, user interfaces, maintenance service levels etc) and create the correct information systems from them. You can also reap the benefits of a consistent specification. Once your functional requirements are free of errors, consistency of your requirements is a mathematically guaranteed property. It comes for free by using the tools that accompany this book.

Third, this book aims at making requirements explicit and concrete. For business processes that matter, explicit and concrete requirements are a necessary thing. In practice however, stakeholders may sometimes agree on vague and imprecise requirements in order to reach agreement. Being involved in the making of some business process, you may well be the person who is responsible for concrete requirements. That may require considerable social skills. This book gives you a fine instrument to formulate requirements precisely. When used wisely, you can achieve great progress in getting stakeholders to commit to concrete requirements.

We call this approach *Ampersand*. The Ampersand approach employs business rules to formulate a sound basis for subsequent information systems design and to actually *define* the processes of the business. It is named after the ampersand symbol &, which means "and". The name hints at the desire to have it all: getting the best from both business and IT, achieving results from theory and practice alike, and realising the desired results effectively and more efficiently than ever before. This book will explain the approach, and the core notions of the accompanying Ampersand tool,

Ampersand

The fourth question addresses information technology. This book is about information systems as well as business processes. Ampersand hardly makes a distinction between the two. From a business perspective, this book is about business processes that make substantial use of information technology. Typical examples are taken from administrative processes in the public and financial sector, without limiting the scope to that particular area. From a technical perspective, this book is about designing information systems by having the functional specifications generated straight from the functional business requirements. Ampersand uses tools to automate this process. Automated design techniques allow you to spend more effort on elicitation of the requirements. Business processes that do not employ any information technology are beyond the scope of this book.

Finally, this book is aimed at an audience that is willing to invest some effort to master the skill of making requirements precise. Ampersand is an approach that lets you define what you mean in a precise way. That takes effort. The more experience you get, the more patterns you will discover and reuse in different situations. For that reason, you must be prepared to make a genuine effort.

If you still answer the previous questions positively, the authors cordially invite you to read further and join us in the quest to make better use of information technology in today's organizations.

That brings us to the next question: What is this book about? If the topic is design of information systems and business processes, why do we need yet another approach? That question addresses the very vision upon which Ampersand is based. The following sections share that vision, each giving a perspective on Ampersand from a different angle.

1.1 Vision on inspired design

This book is about designing information systems that support (the execution of) business processes. The people who execute business processes, but managers and customers as well, hope for flawless system designs. A well-designed information system helps them to do their job effectively and efficiently, in full compliance with applicable rules. Rules are needed to coordinate the work among all those people. Ampersand is based on precisely those rules. It acknowledges that there are rules of different kinds. 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 *can* live by the rules.

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. Guaranteed functionality and more predictable innovation projects are part of that professionalism. One day, information systems will be a commodity, abundantly available at a predictable (low) cost, with predefined quality, and delivered almost on-the-spot. Such professionalism will create room for inspired design. It is inspiration, leading to beauty and fun, that is a 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 can be demanding. It requires designers to express themselves, very much like an artist, a violinist, painter or writer. Before creating music that touches the hearts of your audience you must have full control over the instrument, the color palette, the language. 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.

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 a more creative level, by automating much of the tedious, technical work involved. This book proposes a way to produce 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.

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, hard work, and craftsmanship. This book aims at making you successful, by providing the knowledge. It requires you to practice your techniques until you can do them effortlessly and correctly. Craftsmanship is what you learn by doing.

The next few sections explain why business rules are useful. Each section takes a different perspective: the business, processes, coherence, information technology, formalism, and methods. Together, these perspectives make up the vision, which characterizes our approach to Rule Based Design.

1.1.1 Vision on the business

Business rules are meant for communication with stakeholders in the business [28]. The idea is expressed concisely in the Business Rules Manifesto (available at www.businessrulesgroup.org/brmanifesto.htm). An example of a business rule is that all citizens over 16 years of age may

vote. This particular rule applies to Nicaragua (from 1984 onwards). It does not hold in the United States, however, which allows citizens to vote from the age of 18 (says the 26th amendment of 1971). So business rules *apply* in a particular context, during a particular span of time.

Business rules are as close to the business as one can get. They enlarge the scope of requirements engineering to include business goals [24]. Other examples of business rules are:

- Every application for a new pension plan must be decided upon within three days of receiving that application. (e.g. in the context of a life insurance company)
- An application for a green card must be put aside if the identity of the applicant cannot be established legally. (e.g. in the context of immigration)
- Every payment must be authorized by two different staff members, and any payment may be authorized only by staff that is properly authorized to the full amount to be 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.

1.1.2 On business processes

Rules have a potential for improving the way business processes are designed. Business processes are characterized by stakeholders and the agreements they make.

For example, a business process that delivers fresh flowers throughout the globe has many different stakeholders, each of which plays his own part. The sales organization accepts orders from across the globe, and forwards every order to an associated florist close to the delivery point. These florists have agreed beforehand to deliver fresh flowers according to predefined quality standards, delivery times, prices, etc. A financial broker deals with the money, and so on.

All stakeholders are committed to well defined agreements, which makes the entire process work. The result is that I can send flowers to my mother-in-law for her birthday, while I am in a location halfway across the globe. That is possible only because many stakeholders work together and fulfill their commitments.

This way of dealing with business processes focuses on agreements among stakeholders. That is precisely the point of view taken by Ampersand. Agreements are made concrete in terms of rules, which are maintained by various actors. Since a business process contains both human actors and automated actors, maintaining a rule can be done either by people or by computers.

For instance, what happens if the law lays down a rule saying that voters must be registered? Suppose voter Abe Gandalf is *not* registered? This is a clear violation of the legal rule. If that rule is maintained by a computer, Abe Gandalf's vote can not be captured as valid data as long as he is not registered. It should give Abe an error message, and perhaps explain the problem by telling about the rule. After Abe Gandalf has been registered, the computer can handle the vote. That is how a computer maintains rules. If clerk Margaret maintains that rule as an election official, she would most likely send Abe Gandalf back home. But in order to do so, she needs information about the possible violation. So when Abe Gandalf reports for voting, Margaret's information system should signal that Abe is not registered.

The example above illustrates how rules can be operated in two different ways. The first way is rules maintained by computers; they must remain satisfied at all times. The second way concerns rules maintained by people. The computer must signal violations to these people, who in turn should take some appropriate action in order to maintain the rule. Thus, the information system that supports a business process is characterized by a set of rules that people maintain.

1.1.3 On information technology

An information system contains data to represent facts in a changing world. Along with the world, data in an information system changes over time. Consequently, a rule that is satisfied at one moment may be violated some time later, as the data changes. That data should however always satisfy the business rules, whatever that data might be. In that sense, business rules constrain data. To keep data confined to these constraints is the job of an information system.

So, business rules must be formalized for two reasons. One reason is that computers must make sense of the rules. But equally important: stakeholders too must be able to obtain an unambiguous and undisputed understanding of the rules. This book describes a formalization independent of any computer language. Thus, it can be implemented in any (sufficiently powerful) computer system.

1.1.4 On cohesion

Business rules create cohesion among an otherwise unrelated set of stakeholders. Every agreement and commitment they keep is a building block to a more cohesive system. A system (of people and computers) can be highly regulated, with little room for own initiative. It can however also be regulated very sparsely, leaving much room for individual actions. In all cases, the rules serve as the 'glue' to create a coherent system that serves its purpose.

This form of cohesion leaves the maximum amount of space for individuals, which is supported by the following observations. If a rule is considered appropriate, people and computers will refrain from actions that violate the rules. If a rule is considered obsolete, it must be removed from the system in order to avoid unnecessary constraints. In this sense, a rule based process leaves people as free as possible, while maintaining a cohesive system of rules.

Rule Based Design promotes a vision in which business processes are governed by business rules. 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).

1.1.5 On formalism

Ampersand enhances business rules with a formal representation. This is necessary to ensure that commitments between stakeholders are concrete and unambiguous. A business rule in a formal representation can be checked for consistency with other rules by means of software tools. The design of an information systems follows directly and can be automated to a high extent, as promised by article 5.3 of the manifesto.

This book uses the mathematical formalism of Relation Algebra to analyse requirements and formulate rules. It applies knowledge representation techniques [6], which are firmly rooted in existing theory from the previous century and earlier [33, 9, 27, 30]. This formalism enables contemporary professionals to describe and manipulate the business concepts and the associations between them effectively, quickly, and without mistakes. Proficiency in this skill to describe and manipulate will allow you to make functional specifications, to conduct conceptual analyses, and to search for and establish new business rules.

This book makes an effort to introduce operators that help to describe the knowledge of the business. The material is introduced slowly, one step at a time, with as little formalism as the authors think possible, to be found in the 'Theory' part of this book. If you are new to this topic, prepare for some effort, but rest assured that it will be worthwhile.

1.1.6 On methods

This book proposes a method that puts functional requirements at the focal point of the design of business processes and information systems. Design artifacts such as data models, conceptual analysis, function point counts, service design, etc. must follow from these functional requirements. If business rules constrain the design space, they must be incorporated in the functional specification. If the functional specification represents the functional requirements without inconsistencies, the remainder of the design process is largely automatable.

This vision puts requirements engineers in a special position. They must elicit requirements from various audiences, helping these audiences to make their wishes concrete. This requires communicative and advisory skills. Also, a requirements engineer must interpret these requirements to select or write business rules to make requirements explicit and concrete. This requires technical insight in information systems that support business processes. It also requires the skill to ensure that business rules are correct, i.e. they represent precisely what is needed in the business. Finally, the requirements engineer must help stakeholders with solutions rather than endless questions. The stakeholders may give fragments of requirements, and the requirements engineer must make them complete and consistent.

Rule Based Design helps requirements engineers with tools that automate large parts of the design process. This helps to overcome some of the difficulties involved in designing information systems reliably and repeatably. And that is precisely the problem addressed by this book. In the following section we introduce the main flavours of business rule.

1.2 Types of business rule

Theory, leads and inspiration about business rules can be found abundantly by reading the Business Rules Manifesto, consulting the world wide web, and reading the vast amounts of literature. This book uses the word *business rule* for a rule that reflects a commitment from the business, as intended by the Business Rules Manifesto. 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).

business rule

Many business rule engines that are available in the marketplace use rules of the condition-action or event-condition-action types, both of which can be executed by a computer. A *condition-action* type of rule has the structure:

condition-action

if ⟨some condition is fulfilled⟩ **do** ⟨perform some activity⟩

event-condition-action

An *event-condition-action* rule has the structure:

when ⟨some event has happened⟩ **and if** ⟨some condition is fulfilled⟩
do ⟨perform some activity⟩

imperative rule
production rule

They are called "*imperative rule*s", because they command actions to be executed by computers. This is also why they are also called *production rule*: they "produce" actions, and the actions produce new information to be processed.

Some tools focus on the natural language aspect, allowing the user to describe business rules in (a restricted form of) natural language. This is useful to share rules in larger audiences. Other tools focus on the software, making sure that all rules can be executed directly on a computer.

constraint
invariant rule

In this book we use business rules as *constraint*s. These business rules may be called "*invariant rule*s", because an information system must keep them invariably true at all times. Invariant rules are different, because they do not necessarily prescribe a particular action. This is best explained by an example.

Consider a business rule, which states that only officers may enter the officer's mess. If GI Pete Doe tries to enter the mess, there are different ways to maintain this rule. One way is to promote Pete Doe to officer and allow him in. Another way is to send Pete to the cafeteria across the street. A third way is to bluntly refuse access. A fourth way is to move the mess out of the building and let Pete in. A fifth way is to get rid of this rule. Maybe the reader can come up with still other ways to maintain the rule...

computation rule

Yet another category of rules is called "*computation rule*s". Such rules can be used to assist in complicated computations. Tax returns are a good example to illustrate this point. The complication of tax returns is that certain rules are applicable in very specific situations only. Citizens who create an income from several businesses are confronted with many more rules than elderly people with just a single and stable pension. Business rules are popular to solve such problems, because a rule engine guides people through the complicated maze of tax rules.

The three categories of rules mentioned above are far from exhaustive. It is as if every new usage of rules, justifies a new categorization of rules. Apparently, the notion of *business rule* carries many different meanings for different people in different circumstances. Each time you encounter the phrase, you may want to identify the purpose for which they are used.

1.3 Backgrounds

Rule based design, as introduced in this book, is deeply rooted in well know theories. 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 [11, 21], 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 prototype application to enforce all business rules. For practical reasons, Ampersand has been designed such that any use of the formal language is restricted to the designers only. At no occasion the need arises to confront users with the formalism. Business analysts and software architects must learn how to use it, though.

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 [8], 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, which is a purely *declarative approach*, actions are not specified, but derived. All ECA-rules required to make computers work are derived from the rules (by a tool), enabling an implementation of requirements that is free of errors. *declarative approach*

In a comparison of available methodologies for representing business rules, Herbst et.al. [16] 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].

Ampersand uses the business rules to actually *define* the business process, making rules the common basis for 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 important areas of research and development, and market researcher Gartner expects a consistent growth of licence revenue in business rules technology. Some important interest groups can be found at:

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

Most rule engines 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, 34, 26]. An advantage of separating business rules from process logic is that the business process (such as a mortgage application procedure) will often remain stable, even if rules and regulations change. By the same count, however, this separation would restrict the use of business rules to applications in which the process is not affected, leaving designers in charge not of one, but two distinct 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.

declarative rule

Date [7] has criticized SQL for being unfaithful to relation algebra and advocates *declarative rule*s 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 applications both at the same time.

1.4 Prior experience

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 that the rules, captured by our methods, are capable of 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 into limitations and practicalities involved. This section discusses some of these projects, pointing out which evidence has been acquired by each.

The CC-method, the predecessor of Ampersand, was conceived in 1996, resulting in a conceptual model called WorkPAD [20]. 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 [23], which is now frequently used by process architects at Ordina and its customer organizations. Another early result of the CC-method is an interesting study of van Beek [35], who used CC to provide formal evidence for tool integration problems; work that led to a major paradigm 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 in large projects.

The CC-method has long been restricted to be used in conceptual analysis and metamodeling [22, 12], although its potential for violation detection was recognized 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 [13], 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 top-

ics related to information security, leading to a security architecture for residential gateways [19]. 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) [17] in 2004. Several inconsistencies have been found in the last (draft) RBAC standard [3]. Also, it was noted 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 expliciting the rules governing architectural descriptions has resulted in a small and elegant procedure for creating (parts of) such architectural descriptions in an efficient and effective way [18]. Additional research at TNO [14] looks into the possibilities of developing context-dependent 'cookbooks' for creating architectural descriptions in a given context, based on CC-modelling.

The efforts at TNO provided the evidence that the CC-method did achieve its purpose, which is to accelerate 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. A major deliverable of this work was the Ampersand software tool. This was subsequently used in software development efforts at Bank MeesPierson [5]. The tool provided engineering support in describing and analyzing rules governing the trade in securities. The engineer found that the business rules could very well be used to do continuous 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 noticeable 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 [25] 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 skills-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 [15] 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.

1.5 The power of business rules

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 that we introduce in the next chapter, 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. Although case management [36, 10] does improve 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 Ampersand tool will be extended 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.

1.6 Acknowledgements

The authors wish to thank all students of the Business Rules course for their helpful comments. A special thank you is due to all of Ordina's customers who have contributed to this work. The issues they raised in their projects have inspired Ampersand directly and indirectly. The fact that most of them must remain anonymous does not diminish their contributions.

Of course, we would like to thank our reviewers, especially Silvie Spreeuwenberg, Jaap van der Woude and others who have meticulously pointed out flaws in earlier versions. Besides, we wish to thank TNO in Groningen (the Netherlands) for being the first user of Ampersand. Thank you also to colleagues at the Open University of the Netherlands, the University of Utrecht, the University of Twente, and the Technological University of Eindhoven, who have inspired and criticized this work. The warmest thanks are for our families, who have sacrificed so much time to get this book where it is today.

1.7 Reading Guide

This introduction provides 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. The first part, Methodology, elaborates a vision on business processes control. It introduces the idea of Rule Based Design from a practical perspective, and motivates why this is a good idea. The second part, Theory, introduces a way to write business rules. You are introduced to a formal way of writing language. The third part, Practice, discusses case studies that show how Rule Based Design is used in practice.

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.

A case study completes this chapter.

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 a dozen 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.

> A *business rule* is a verifiable statement that some stakeholders intend to obey, within a certain context. *business rule*

Here is an example of a rule:

> In our club, a coat of any guest shall be in the cloakroom,
> as long as the guest is in the club.

Let us analyse this statement, to better understand what we mean by a business rule.

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

2. Rules have *stakeholders*.
   Anyone involved in a rule is called *stakeholder*. If a rule has no stakeholders, then no one is interested if the rule is obeyed or violated. Such a rule has no business merit, and we do not consider it to be a business rules. 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 people who try to smuggle their coats inside are intercepted, etcetera. Stakeholders who "live by the rules" are working to satisfy rules, each in his or her own role. *stakeholder*

3. Rules are *verifiable*. *verifiable*
   If there is a guest inside the club who holds a coat, we have a violation of this rule. We call a rule verifiable if its violations can be spotted unambiguously and objectively. That violation could be a signal to someone to take action. A floor manager might summon the offender to take his or her coat out to the cloakroom. Or, a staff member might take the coat from the guest and put it away in the cloakroom. Or even the guest himself might take some action. He might toss his coat out of the window, ensuring that it is beyond the scope of this rule. Technically, that would be an acceptable thing to do, unless there are rules in place that forbid littering the street... Whatever actions happen, the situation should be restored to where the rule is complied with.

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. Within the context of this book, this statement is not a rule because it is not verifiable.

concrete

Rules must have the property of being *concrete*.

- *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.

relevant

Rules must have the property of being *relevant*.

- $E = mc^2$
  This is a law of nature, which is considered true in any scope and without the intervention for any stakeholder whatsoever. No one will check to see if the rule is obeyed or violated, and certainly will no stakeholder put in an effort to undo violations. Even if we would consider this to be a rule, there is no need to maintain what mother Nature maintains for us. Such irrelevant rules and laws of nature are out of our scope, they are not business rules.

  Business rules must represent agreements that people care about.

- *Peter Lee Jones has visited the club this morning.*
  This statement can be either true or false, so it is a verifiable statement. And once we have established its truth, it will never change. Therefore, we call this a fact rather than rule, even though theoretically, there is no reason why facts should not be rules.

Rules usually assume a number of things tacitly. That is also the case in our example. The rule sounds: "In our club, a coat of any guest shall be in the cloakroom, as long as the guest is in the club." It assumes a number of things, such as:

- There is a club, which we call "our club". To avoid uncertainty, we had better remove the reference to "our" club, and supply the exact name of the 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. Requirements engineers, who write rules on behalf of stakeholders, must be aware of these tacit assumptions.

Also, the exact phrasing of a rule is really important. Rephrasing can cause problems, because there may be implicit assumptions underlying the statements. The following examples show how seemingly innocent rephrasings can unwillingly change the intended meaning:

- In our club, guests must put their coats in the cloakroom.
  This rule does not prevent a guest from taking his coat into the club. He or she can take the coat out, right after putting it in the cloakroom. To avoid this and similar situations, it is good practice not to prescribe actions, but to describe a state.

- In our club, the coat of each guest must 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 I decide objectively at any moment in time, whether the rule is satisfied or not? If so, this statement is verifiable. As a double check, can I think of a situation that violates the rule?

2. Where and in which situation(s) 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

define business process

Business rules can be used to manage and control business processes. In this sense, business rules actually *define* the *business 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 drives a series of events to comply with all business rules. It lets us conclude that business rules are sufficient as an instrument to design compliant business processes and information systems.

Whenever and wherever people cooperate to work together, they coordinate their work by making agreements and commitments. These agreements and commitments constitute the rules of the business. A logical consequence is that these rules must be known and understood by all who have to comply. From this perspective, business rules are the cement that ties a group of individuals together to form an organization. In practice, many rules are documented, especially in larger organizations. Life of a business analyst can hardly be made easier: rules are known and discussed in the organization's own language, and stakeholders know (or are supposed to know) the rules and abide by them.

maintain

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, a rule might say that each benefit application requires a decision. This is violated from the moment an application arrives until a corresponding decision is made. Allowing the temporary violation gives a person time 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.

procedure

closed

This principle rests solely on rules. Computer software is used to derive the actions, generating the business processes directly from the rules of the business. To compare: workflow management derives actions from a workflow model, that captures a procedure in terms of actions. Workflow models are specified by modelers, who take the rules of the business, and transform these into actions plus an appropriate but fixed order to achieve the desired result.

The new approach has two advantages: the work to draw up 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 may be accepted only from one individual whose identity is authenticated.

2. Each application for a permit must be treated only by authorized personnel.
3. Every application must lead to a decision.
4. An application for a particular type of permit may never lead to a decision about another type of permit.
5. Every employee must be assigned to one or more particular areas.
6. An employee may only handle applications from those areas to which (s)he is assigned.

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 with suspicious identity or credentials.
2. An application that is handled by an unauthorized person.
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 no area at all (perhaps an apprentice?).
6. An employee assigned to Soho who handles an application from the 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. Rule 5 may also be violated for some time, but the employee cannot handle applications for the time being. Rules 1, 2, and 6 may be enforced by preventing all transactions that might violate the rule. Thus, a system emerges that complies with all these rules.

An application to controls this process has been built on a computer. The functional specification was generated by software that translates a set of (formalized) rules into a conceptual model, a data model, and a catalogue of services with their services defined formally. This specification defines a software system that maintains all rules mentioned above. The specification guarantees that many rules can never be violated, and the remaining ones such as 3 yield 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.

An actual scenario of interleaved user and computer activities, used in demonstrations of information systems generated by business rules, proceeded 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. The employee can now record the new application. As far as this employee is concerned, he is done with the application.
5. Next, an employee must be allocated for making the required decision. If an employee is chosen in violation of rules 2 or 6, that transaction is blocked.
6. The employee who makes the decision records it in the information 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.

Notice that this system may be criticized for picking an employee 'by hand'. This behaviour 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 tools can also use the business rules to produce specification artifacts such as an UML class diagram, or formal specifications of the software services required to maintain all rules. These deliverables are not shown here for the sake of brevity.

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 2.1

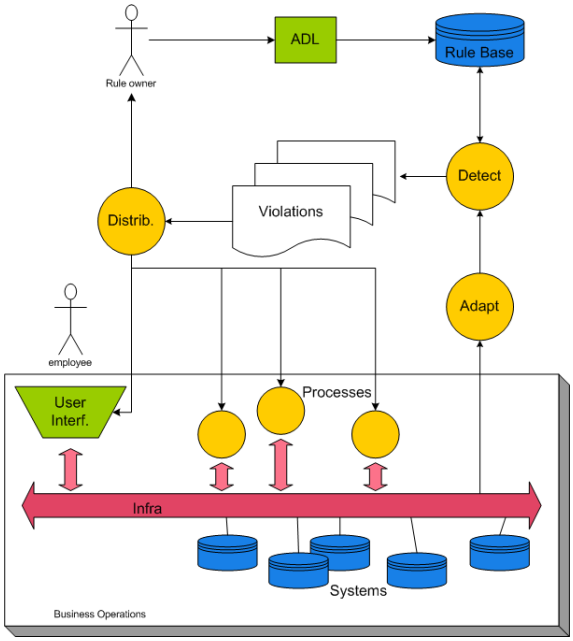

Figure 2.1: Principle of rule based process management

illustrates the principle. Assume the existence of an electronic infrastructure that contains data collections, functional components, user interface components and whatever else 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 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.

event

violation

signal

The system as a whole cycles repeatedly through the phases shown in figure 2.2. The detector detects when rules are violated and signals this by analyzing *event*s 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. Whenever a *violation* is detected, a *signal* is raised for the attention of some actor or actors Signals 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.

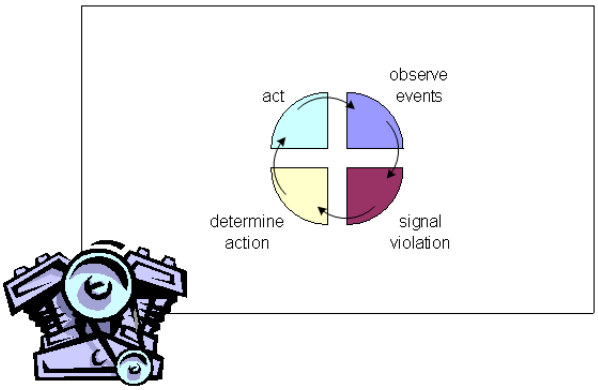

Figure 2.2: Engine cycle for a rule based process engine

2.5 Rule Management

Changing the rules is done by altering the contents of the rule base. The behaviour of the organization will change accordingly. Rules that are removed can no longer be violated, so the signals caused by that rule will cease to exist. New rules create new kinds of violations, which cause people to take new kinds of action. In this way, changing the rules has a profound impact on the processes governed by these rules.

For this reason, an organization must obviously manage their rules. Since the rules of the business change all the time, business processes change along with them. Documenting the rules is one thing, but a process needs to be in place to deal with rule changes. Even rule changes are subject to rules. The process of legislature in a country is a good example; countries have rules that govern how new laws are made. In most organizations, simpler processes than that will be in place. By defining the rule management process as the collection of rules that govern rule changes, you can use Ampersand to define that process as any other process.

2.6 Case Study: Order process

This section demonstrates the Ampersand approach by means of a case study. 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 defines the order process. Then we go straight to the demonstration, which walks through a scenario based on screenshots. That scenario sketches the processing of one order 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. In the demonstration, there will be a field named 'order' to show this signal.

2. A provider may accept only orders issued to himself.

3. A delivery may be made to a client only with an order, which is 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. In the demonstration, there will be a field named 'deliver' that shows this signal.

5. All deliveries must be made to the client who ordered the delivery. The provider will be signalled for orders that must be delivered.

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. In the demonstration, there will be a field named 'pending' to show this signal.

9. Only those payments may be accepted for which an invoice has been sent

10. Each invoice sent to a client must be paid. Both the client and the provider will be signalled for payments due. In the demonstration, there will be a field named 'payable' that shows this signal.

In chapter 3 you can read how to write these rules in the language of Ampersand. From that point onwards, a compiler can generate the software. Let us walk through the screenshots shown in figures 2.3 through 2.10.

1. Figure 2.3 shows a user interface that exposes the heart of the database that supports the order process. It contains three tables (entities: Order, Delivery, and Invoice). Only one of them, Order, is shown. The other two, Delivery and Invoice, may be opened by clicking on their names. The list of orders contains three orders from the past. Initially all four rules are satisfied, which is visible because there are no signals raised (the message shows: "This rule is OK").

2. Client Brown creates a new order for 'Gates Inc.'. He does so by clicking 'Create new' and filling out the fields 'issuedTo' and 'from'. The effect of this action is that a new order number is issued (18554), as shown in figure 2.4. Rule 'order' has raised a

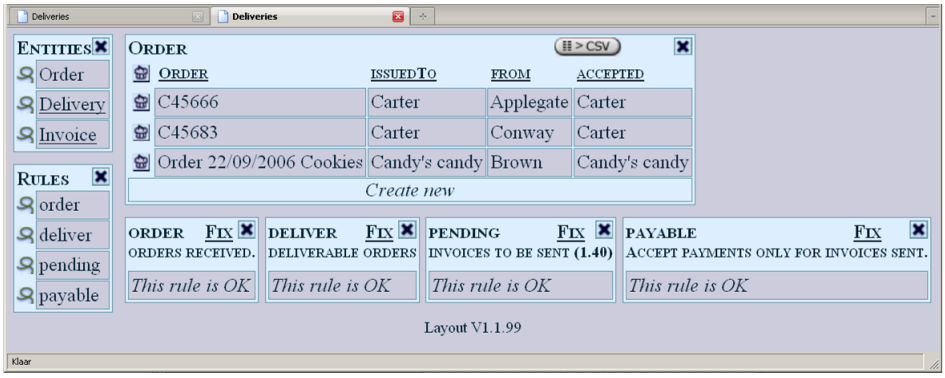

Figure 2.3: Generic user interface for the order process

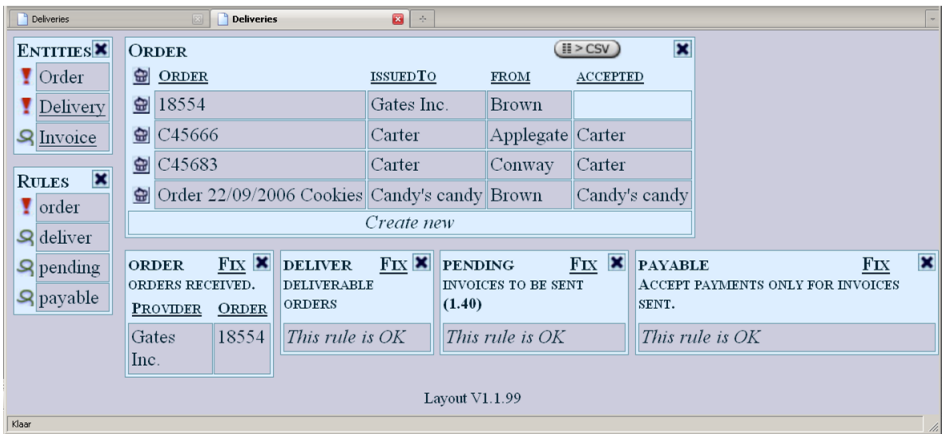

Figure 2.4: Client Brown has issued an order

signal for 'Gates Inc.', which means that there is an order waiting to be accepted. Technically, rule 1 is violated, because it is not true for 'Gates Inc.' and order number 18554.

3. When provider 'Gates Inc.' accepts the order, this shows up in the field 'accepted', where the name 'Gates Inc.' has been filled in. As a result, rule 'order' is satisfied. But now, rule 'deliver' gives a signal to Gates Inc. (figure 2.5). Gates Inc. can interpret this as a signal for delivering the order. For the sequel of this demonstration, we switch our view to the entity 'Delivery'.

4. Figure 2.6 shows an empty delivery record for order number 18554. When this record is filled, it means that Gates Inc. has delivered the order. The moment the provider tries to fill out the empty fields in this form, the computer automatically 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 (figure 2.7). For a user, this is natural behaviour, because the computer 'already

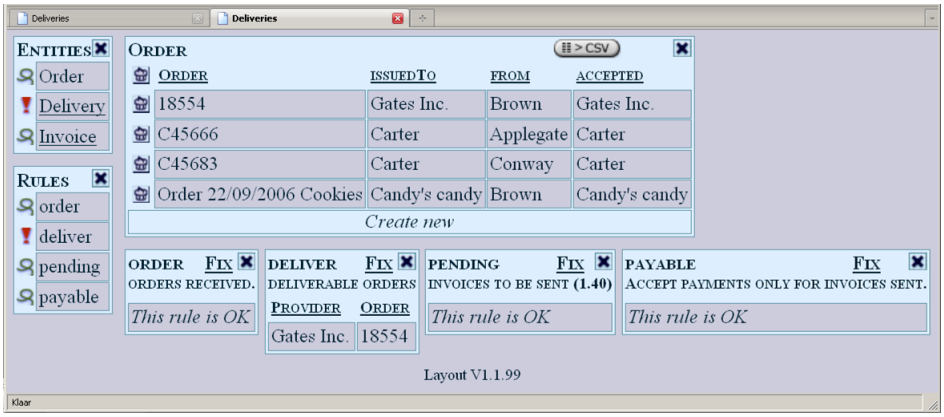

Figure 2.5: Gates Inc. has accepted the order

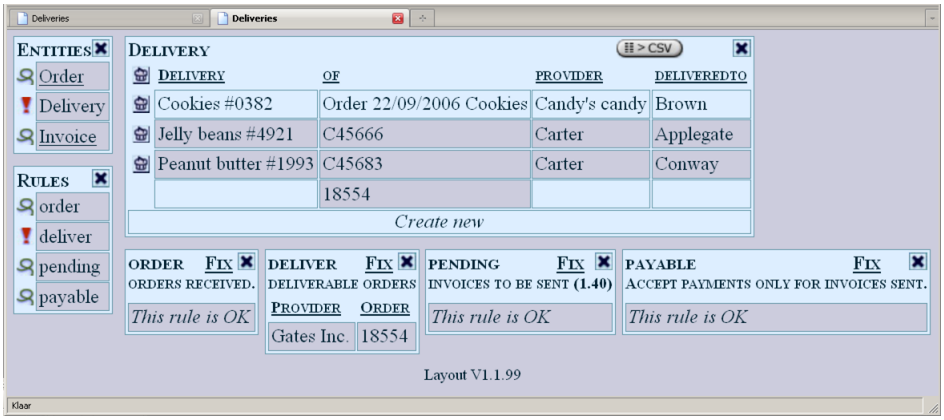

Figure 2.6: View on deliveries

knows' these facts. In the meantime, the signal 'pending' is raised saying that an invoice can be sent.

5. When we look at invoices in figure 2.8, something similar happens. An empty invoice for delivery 12145 is ready to be filled out and sent to the customer. When the provider starts to fill out the invoice, again the computer fills all known (i.e. uniquely determined) facts (figure 2.9). This yields a partially filled invoice registration, plus a signal to the client that an invoice is due for payment.

6. Figure 2.10 shows that mr. Brown has paid the invoice, because the field "paid" is filled. As all rules are satisfied, the case is closed.

From a business perspective, the following scenario has been executed.

1. The client created a new order. This generated a signal for the provider. That signal is a logical consequence of rule 1, which was

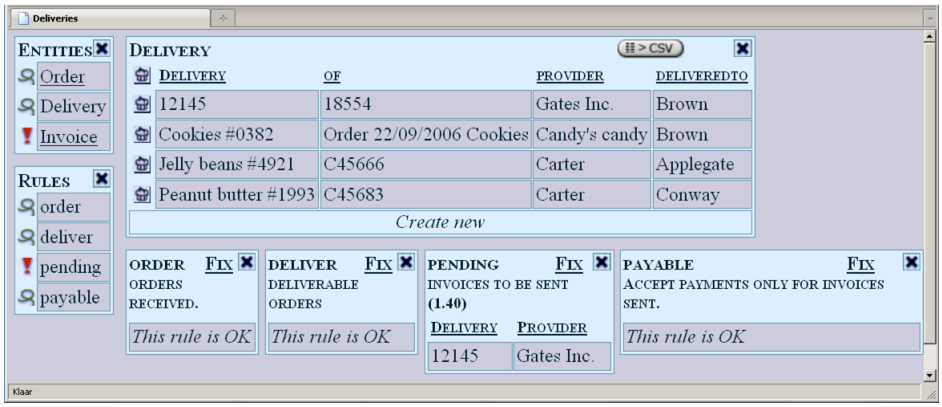

Figure 2.7: Delivery has been made

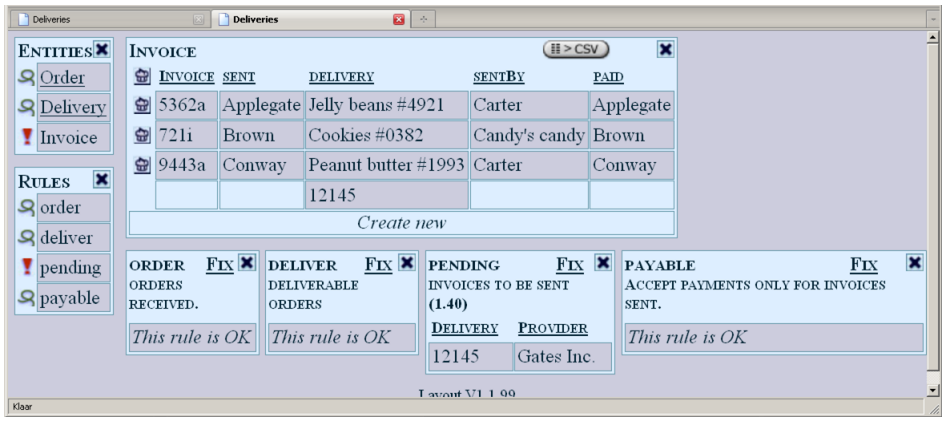

Figure 2.8: View on invoices

(temporarily) violated by an order. As long as the order remained unaccepted, there was an order issued by a client but not accepted by the provider. Accepting the order satisfied rule 1 and removed the signal from field 'order'.

2. The provider accepted the order, which generated a signal to deliver the order. That signal is a logical consequence of rule 4, which was (temporarily) violated by an order that was accepted, but not delivered. This was made visible in signal 'deliver'.

3. The provider delivered the order, generating a signal to send out an invoice. This is a logical consequence of rule 8, which was (temporarily) violated by a delivery made, without a corresponding invoice (signal 'pending').

4. The provider sent an invoice, which created 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

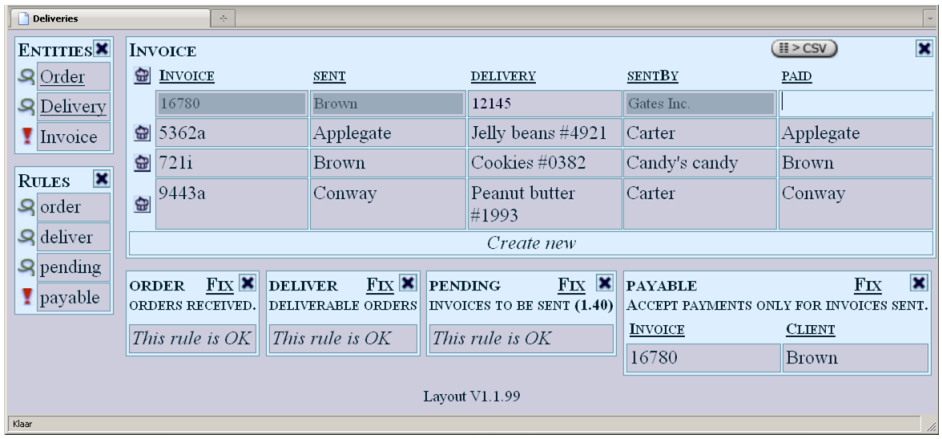

Figure 2.9: Generic user interface for the order process

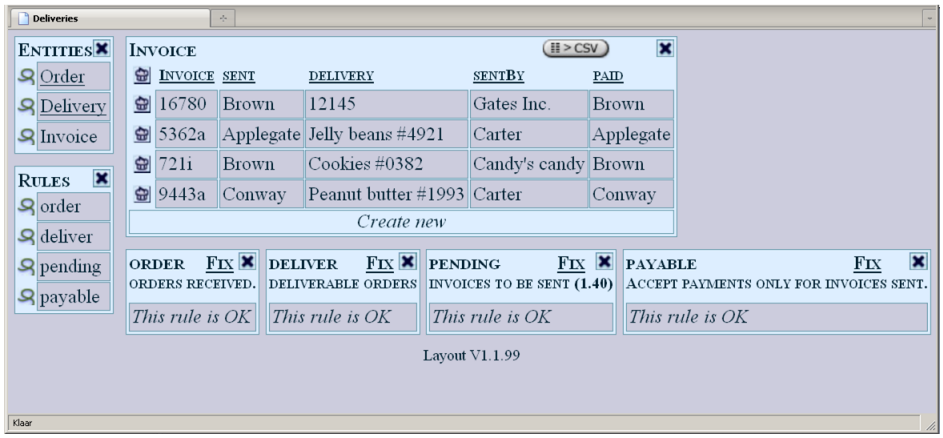

Figure 2.10: Invoice filled out

that has not been paid (yet).

5. The client paid the invoice, and no more signals were raised. This means that the order was processed completely and the case is closed.

This process demonstrates a way of controlling a business process that is typical for Ampersand. Signals being raised trigger people to take action. Each signal corresponds to a (temporary) violation of a rule. Each signal is brought to the attention of some designated actor(s) who must take action to remedy the violation.

This brings new information into the system, which is again checked for violations and the process is repeated until no violations remain. As a consequence, when all rules are satisfied, there is no more work to be done, and the case can be closed.

Thus, Ampersand provides a clear definition of a business process:

> A *business process* is an amount of work that is characterized by a set of rules. *business process*

In the actual application, different sequences of events can be tried. For example, a delivery can be made before an order is accepted, or the name of a client can be filled in long after the delivery has been made. The application will accept events in any order, except the ones that cause violations. Ampersand's concept of business process does not presuppose any event sequence, as long as the events lead to satisfaction of the rules of the business process.

2.7 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 2.11. The effort of design focuses on requirements engineering. The

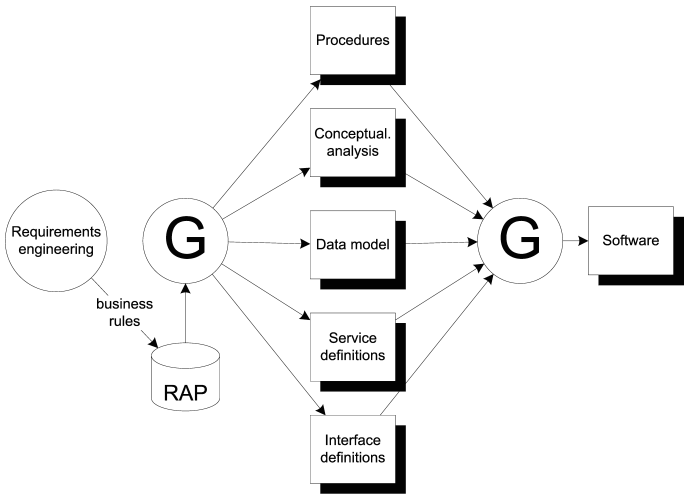

Figure 2.11: Design process for rule based process management

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 (another generator labelled 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, using a supportive tool (Ampersand). 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 2.12. The focal point of attention is the dialogue between a problem

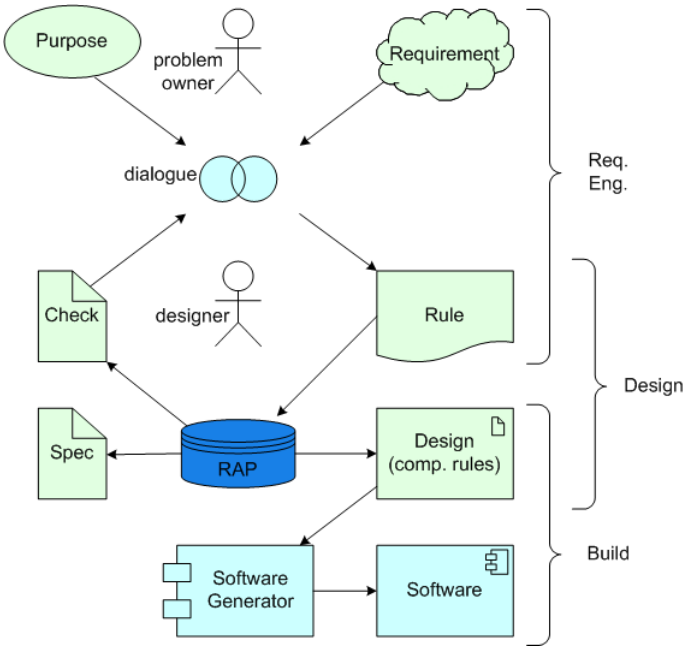

Figure 2.12: Design process for rule based process management

owner and 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.

Part II

Theory

Chapter 3

Concepts and Relations

Abstract

The Business Rules Manifesto proclaims as its central thesis or "mantra": Rules are built on facts, and facts are built on terms. But what are they, the terms, facts and rules that the Manifesto talks about? How must you understand them, how are they interconnected, and why can you rely on just these notions to build a rigorous framework for business rules? The answer is found in mathematics: Relational Algebra is a formal method to define, implement and work with well-formed business rules. Relational algebra provides todays professionals with a way of thinking and 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 analysis, to search for and create new rules, to test the outcome, and finally to produce exact specifications.

This chapter explains almost all about two basic notions that we will be needing as we identify, express, and manipulate the rules in the business environment. These two notions are: concept and relation.

We will introduce them using a language-oriented approach. Which is natural, as business workers express their requirements and ways of working in natural language. But to turn that into very exact rules that can be handled by computers, we cannot escape using a mathematical formalism, called Relation Algebra. 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 [9, 27, 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.

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 mathematical body of Relation Algebra. The 'background' material of Jipsen, Brink, and Schmidt is recommended as a reference for relation algebras in general. We will specifically use the so-called 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 from fundamentals: simple sentences. Next, we discuss some mathematical notions that the Business Rules Manifesto implicitly uses in its article 3.1 "Rules build on facts, and facts build on concepts expressed by terms". 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.

3.1 Sentences

3.1.1 Simple sentences

Business workers use natural language to talk about things in the real world. Therefore, we need a clear and unambiguous way to capture the meaning in natural language sentences. Let us consider the following simple sentences to start with:

- ABBA sang the song Waterloo,
- DoeMaar sang Smoorverliefd,
- Joosten receives building-permit number 5678,
- William Kennedy has residence-permit NL44,
- ABBA sang 'Money, money, money',
- Ersin Seyhan has residence-permit NL901,
- the Open University offers the course Business Rules,
- Fatima has passed the exam for Business Rules,
- Caroline has passed the exam for Spanish Medieval Literature.

These sentences follow the scheme: noun - verb - noun, in which the nouns refer to objects in the real world. The Business Rules Manifesto uses the word *term* to refer to the individual real-world objects such as *term*

ABBA, the building permit number 5678, the residence permit NL_44, Caroline, and the Business Rules course.

Instead of 'term', many other words are in use: software engineers call them 'instance' and mathematicians call them 'element'. In knowledge engineering and model theory, such things are called 'atom'. Throughout this book we use the word 'atom', though any synonym like 'instance', 'item', 'element', 'member' or 'object', will do in practice.

atom

An *atom* refers to an individual object in the real world, such as the student called 'Caroline'. But what if there are three different Carolines? What does it mean to say: "Caroline has passed the exam for Spanish Medieval Literature."? This sentence might be true for one Caroline, but false for the others. Clearly, to avoid ambiguous sentences, an atom must identify exactly one real-world object, no more, no less. Or rather, it suffices that the atom identifies one object within the context in which we are working: if the context is a group with only one Caroline, there will be no ambiguity. Similarly, ABBA is unique among all pop groups in the world; there ought to be only one building permit with number 5678; etcetera.

fact

Each of the simple sentences above represents a *fact*, something that is taken to be true. The example sentences follow the noun-verb-noun scheme, but the notion of simple sentence applies to just about any sentence that has two atoms related by a verb-like expression. For example, if we use the name 'Caroline' and the number '3' as atoms, the following is also a simple sentence: "In Caroline's house there are three rooms". This simple sentence contains two atoms, and if we leave out those atoms, a template remains, as in: "In ... 's house, there are ... rooms". So, a *simple sentence* in natural language relates two atoms.

simple sentence

Still, a simple sentence does not accommodate arbitrary atoms. For example, switching the atoms produces a sentence "In 3's house, there are Caroline rooms" which makes no sense at all, it is non-sense. In this particular example, the first blanks are clearly meant for a person and the second one should obviously be a number. In other sentences, other concepts may be required. For instance, in the sentence "... has passed the exam for ..." the first blanks are supposedly some student and the second blanks must be some course.

In Ampersand, an abstraction like 'student', 'number', and 'course', that you need to replace by an actual student, number, or course, is called a *concept*. Concepts should not be confused with atoms: concepts are abstract, and atoms are concrete. Atoms are the terms, the individual things in the real world that you can point out. A concept is an abstraction, it is not the individual thing but the type of thing. We can say for example that Caroline (an atom) is a student (a concept), ABBA is a pop group, and NL_44 is a residence permit.

concept

So simple sentences have a fixed template, and moreover, every term (or atom, as we prefer to call it) that you fill in on the blanks, belongs to a certain concept. The Business Rules Manifesto, in its article 3.1, hints at just this distinction between term and concept. That article can be explained as:

- a *term* expresses a business concept, it corresponds to an individual thing that is relevant in the business context,
- a *concept* is an abstract description or definition that is instantiated (expressed) by the actual realworld thing (term or object),
- a *fact* makes an assertion about the concepts, it is a simple sentence that expresses a relationship between two terms.

3.1.2 Concept

As we want to describe our business rules with computer precision, we must be rigorous in our definitions.

At the surface level, a *term* refers to one individual thing that exists in the real world. At the deep level, a *concept* stands for terms that are similar: they share the same meaning, have similar properties, similar behavior, and similar connections with other terms or concepts.

Every abstract concept comes with an *intension*: a definition that lets you determine whether an arbitrary 'thing' in the business environment is an instance of the concept. The reader of the definition is expected to understand which 'individual terms that exist in the real world' meet the definition and should be (represented as) a term in the extension of the concept. Next, a concept also has an *extension*, more often called *population* or *contents*. This is the set of all terms that the concept actually stands for, at present. It is customarily denoted using a *bracket notation*, for instance $Age = \{ 7, 3, 5, 2 \}$. Occasionally square brackets are used: $Age = [5; 3; 2; 7]$. In mathematics, the special symbol $\in$ is used to denote that an element is contained in the set, so we have: $3 \in Age$, $7 \in Age$, etc.

intension

extension

population

contents

The *inclusion* symbol $\subset$ is used to denote that one set is a subset of another, it is contained in it. For instance $\{ 3, 7 \} \subset$ Age, or { William Kennedy, Ersin Seyhan, Caroline } $\subset$ Person. Because the inclusion symbol is not easy to type, we will often replace it with the symbol $\vdash$, which means exactly the same.

inclusion

Subsets usually contain less atoms than the enveloping set, but in general, the two sets are allowed to be equal. If we want to draw attention

to the fact that equality is permitted, we sometimes write $A \sqsubseteq B$. But if equality is not permitted, we must write two assertions:

$$A \subset B \text{ and } \neg (A = B)$$

Three important properties to remember about atoms are:

- each instance of the concept is considered to be whole and indivisible, and it is why we use the name 'atom'. The element 'ABBA' is regarded as one indivisible member of the Popgroup concept, even though, from another point of view, it may be composed of four units,
- there is no specific order or sequence among the atoms of a concept,
- an atom can occur only once in the population. That is: each atom must differ from every other element within the set. A term meets the concept definition, or it does not. It is meaningless to say that it meets the definition twice.

entity integrity

The latter property is sometimes referred to as *entity integrity* by database designers.

A concept is characterized by its intension, a sound definition that exemplifies its business semantics, its meaning, properties, behaviour. Definitions are generally stable, and do not change overnight, regardless whether there are millions, hundreds, tens or even no instances on record for the concept. And although definitions can change over time, such changes are not very frequent.

To contrast: extensions can and will change constantly. The extension is the time-varying part. At any given moment, there is a specific content, and that content can differ from the content one minute before or after. For instance, Student is an important concept at any university, but the student population is constantly changing. A concept may even have an *empty extension* at some time, denoted $\emptyset$. But even if two concepts are both empty, we consider them to be different: a Student concept differs from the Car concept, even if there are neither students nor cars on record.

empty extension

In general, concept names are nouns such as 'Student', 'Customer' or 'Car', sometimes qualified to better express its meaning, e.g. 'Old-timer Car', 'Bachelor Student', or 'Priority Customer'. By convention, we write the name of a concept with a capital: 'Popgroup', 'Permit', or 'Course'.

3.1.3 Relation

Above, we defined 'fact' as a simple sentence that expresses a relationship between two terms. Having abstracted the individual terms to concepts, we also desire an abstract notion to replace the individual fact templates. The answer provided by mathematics is called: relations.

Let us introduce this by way of an example: some friends organizing a tennis tournament which will include a mixed-double competition. Although the group of friends has 4 women and 5 men, only three pairs want to participate in the mixed double. These are the facts:

- Aisha doubles with Marek.
- Sophie doubles with Raúl.
- Nellie doubles with Toine.

Obviously, the template for these facts is: ... *doubles with* ... with the first concept being Woman, the second concept involved is Man. We now introduce the relation named *doublesWith* which is defined as "the set of all pairs of one woman and one man wanting to participate in the upcoming tennis tournament". We can write the pairs one by one:

- ⟨'Aisha','Marek'⟩ ∈ *doublesWith*.
- ⟨'Nellie','Toine'⟩ ∈ *doublesWith*.
- ⟨'Sophie','Raúl'⟩ ∈ *doublesWith*.

Or we may list the entire population of *doublesWith* in one go:

$$doublesWith = \{ \text{(Nellie, Toine)}; \text{(Aisha, Marek)}; \text{(Sophie, Raúl)} \}.$$

We can also make a drawing of this relation, a so-called *instance diagram* or Venn diagram. This shows all facts as arcs connecting the woman and man that together make up a pair, as shown in figure 3.1. An instance diagram provides very detailed insight into the current state of affairs in the business, which can be very useful sometimes. It works fine for small examples, but this kind of diagram quickly becomes unreadable if many pairs participate.

instance diagram

We may also use a layout with two columns as in table 3.1. Facts can be represented in many ways. A telephone directory lists names with telephone numbers in a tabular layout, a dictionary does the same with words and their meanings. But a story about a poker or bridge game may be illustrated with a diagram showing the hands of each player.

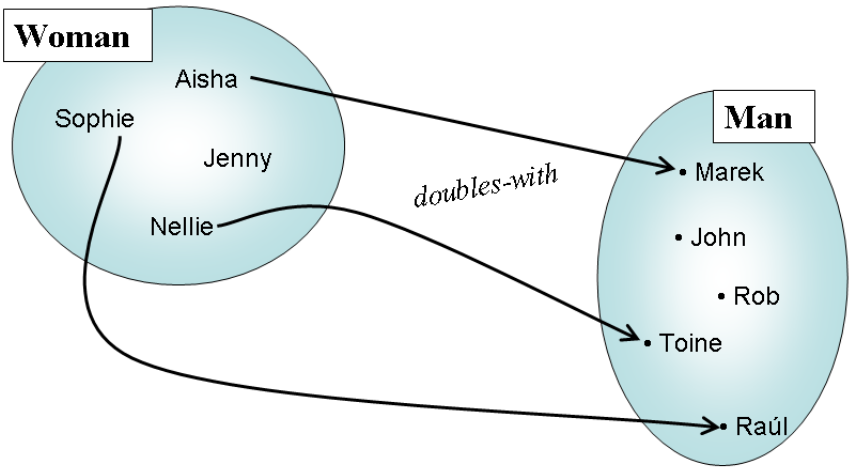

Figure 3.1: Instance diagram: which woman *doublesWith* which man

Woman	Man
Sophie	Raúl
Aisha	Marek
Nellie	Toine

Table 3.1: *doublesWith*: tabular layout of the participating pairs

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 facts, the designer chooses a representation that best suits the purpose.

3.1.4 Cartesian product

The most comprehensive representation of a relation is a two-dimensional array. It shows all instances of one concept, Woman, along one axis, and all instances of the second concept, Man, along the other axis, like a spreadsheet page as in table 3.2. It allows you to easily mark participation in the tournament.

This representation lets you consider all possible combinations of one woman and one man, a total of $4 \times 5 = 20$ possible pairs. The "set of all possible pairs" is called a *Cartesian product* in mathematics.

Cartesian product

Formally, the *Cartesian product* of two concepts A and B is (the set of) all pairs that combine one atom from the first concept A, with one atom from the second concept, B. Obviously, the number of pairs in the

doublesWith	Marek	John	Rob	Toine	Raúl
Sophie	-	-	-	-	x
Aisha	x	-	-	-	-
Jenny	-	-	-	-	-
Nellie	-	-	-	x	-

Table 3.2: Two-dimensional layout of the participating pairs

Cartesian product is equal to the number of atoms in A times the number of atoms in B; which explains (in part) why it is called a 'product'.

The Cartesian product of concepts A and B is denoted as: $A \times B$. We sometimes denote it as $\mathbb{V}_{[A,B]}$ or $\mathbb{V}$ for short, with the V's lefthand side rendered as a double line, if the printer can do that. Of course, if there is any chance of ambiguity, $\mathbb{V}_{[A,B]}$ should be written in full to prevent confusion with, say, $\mathbb{V}_{[B,C]}$ or $\mathbb{V}_{[B,A]}$.

Another good example of Cartesian product is a deck of common playing cards. There are 4 suits {spades, diamonds, clubs, hearts}, and 13 ranks. Because every combination is possible, there are 52 playing cards in the Cartesian product of $Suit \times Rank$, as shown in figure 3.2. The cards in the picture are neatly arranged, but remember that in general, there is no special ordering along the axes.

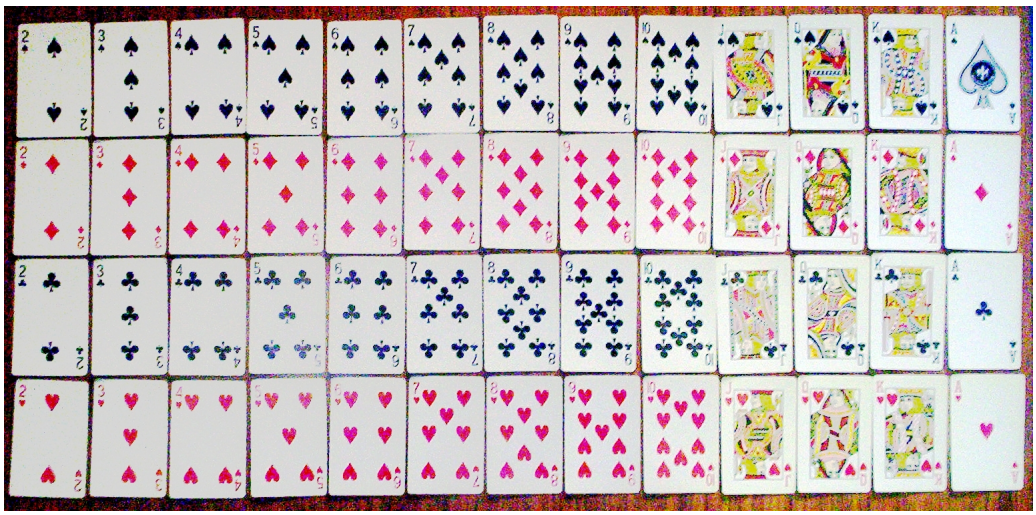

Figure 3.2: Set of 52 playing cards

To be exact: this is different from $Rank \times Suit$. While the tuple (diamonds, ace) looks very similar to the tuple (ace, diamonds), there is a big difference as the two atoms are written in inverse order. This finesse will be important later on when we discuss the inverse of a relation.

Some standard terminology when discussing Cartesian products:

source – 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*.

3.1.5 Relation in Mathematics

relation Having defined the Cartesian product, we can now explain the formal definition of a *relation* which is:

– a named subset of the Cartesian product of two concepts, where all tuples in the subset have a similar meaning, similar properties, and similar behavior.

relation name The *relation name* is usually a verb, possibly qualified, like you saw in the templates for simple sentences. We will generally write the relation name in italics and in lowercase. Whatever name you choose for a relation, make it (almost) self-explanatory. When trying to understand a design, the business stakeholders will start to look at names first. So make sure that the names are really close to the intent of the relation, its real-world meaning and pragmatics.

intension Like with concepts, a relation has an *intension*, a precise definition. The definition serves to determine which pairs should be represented in the extension of the relation, and which ought to be absent. The definition *pragmatics* or *pragmatics* captures the meaning of the relation.

You should check that *doublesWith*, with its three couples entered for the tennis competition, is indeed a relation according to our definition. Another example of a relation is the subset of community cards lying face-up on the table in a poker game, shown in figure 3.3.

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 atoms. Therefore, there is no need to have special identifiers to distinguish between the tuples in a relation: the identification of source- and target instance suffices.

To summarize so far:

– an *atom* corresponds to an individual real world thing. Each atom or *term* is captured as an instance, an element in a set, and that set is the extension of a concept,

Figure 3.3: Community cards on the table are a special selection of playing cards

- a *concept* is an abstract description or definition that is instantiated by the actual realworld things or objects. The intension is the definition of the relevant terms; the extension is the set of all terms in the business context that meet the definition,
- a *fact* is a simple sentence that expresses a truth about two terms,
- a *relation* is an abstract description of facts. Mathematically, it is defined as a subset of the Cartesian product of two concepts, and it captures the deep structure of simple sentences,
- the *intension* of the relation provides the definition or meaning of the relevant facts,
- the *extension* of the relation consists of *pairs*, and each *tuple* represents a single fact that is true in the business context.

3.1.6 Terminology and notations

Some standard terminology when discussing relations:

- the relation *declaration* is the combination of its name, its source and its target, together with its definition, i.e. its meaning or intension, *declaration*

signature

- the *signature* of a relation is the combination of relation name, (the name of) source concept, and (the name of) target concept,

type

- the *type* of a relation is defined as (the name of) the source concept, and (the name of) the target concept. In other words: a relation type indicates the full Cartesian product, the relation is its subset.

Type is important when we compare the extensions of relations. If two relations have different types, then they contain tuples with atoms taken from different source or target concepts. Only if two relations are of the same type, i.e. both relations are subsets of the same Cartesian product, can we compare their contents.

To avoid ambiguity in discussions, every relation should have 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. Sometimes one name is used for several relations. Theoretically, this is fine provided that the types are different. For stakeholders, it may be rather confusing.

To denote the signature of a relation with name r, source A and target B, we write

$r : A \times B$, or $r_{[A,B]}$ for short.

The customary notation for writing the contents of a relation is

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

A more concise notation is sometimes used for a single tuple:

$a\ r\ b$ 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 set, which in general may have 0, 1 or any arbitrary number of elements, is called the target of the element a:

$$target(a) = \{ b \in B \mid (a, b) \in r \}$$

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

$$source(b) = \{ a \in A \mid (a, b) \in r \}$$

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

$$target_{[rela,tion]}(a), \text{ and } source_{[rela,tion]}(b)$$

So now the words 'source' and 'target' have double usage. First, every relation r has a source and a target concept. And second, each atom of the source has its own target set, and each instance of the target has its source set.

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 this important difference, the Cartesian product $\mathbb{V}_{[A,B]}$ 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.

universal relation

Various mathematical *symbol*s will be needed further on. The symbol $\forall$ means 'for all', $\exists$ means 'there exists', $\wedge$ means 'and', $\vee$ means 'or', $\rightarrow$ means 'implies', and $\leftarrow$ means 'is implied by', which is the same but it goes in the other direction. These symbols are merely a mathematical shorthand notation. Just remember the meanings and pronunciations above, and you will find nothing mysterious about them.

symbol

3.1.7 Identity relation

At this point, we want to introduce to you a special relation: the so-called *identity relation*, abbreviated to Id or even to $\mathbb{I}$ (for this particular relation yes, an uppercase i).

identity relation

This relation can be defined for every concept, taking that concept for its source as well as for its target. The definition is very simple: it states that each atom (of the source) is equal to itself (now as an atom in the target). The contents of this relation is equally simple: it contains every possible tuple composed of two identical elements.

identity	Apple	Orange	Pear	Berry	Grape
Apple	x				
Orange		x			
Pear			x		
Berry				x	
Grape					x

Table 3.3: Identity relation is special selection of the Cartesian product with identical source and target

In the matrix representation of table 3.3, the identity relation has a marking exactly on the diagonal, and nowhere else. The example tells us five facts: 'fruit w is identical to fruit w', for all five fruits, ranging from apple to grape. Another way to write down this identity relation

is as follows:

$$\mathbb{I}_{Fruit} \;=\; \{(\; w, w \;) \mid w \in \mathit{Fruit} \;\}$$

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

3.1.8 Naming

We aim to use the notions of atom and fact, concept and relation 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.

For instance, UML programming works with 'classes', a notion that is similar but not equal to our notion of concept. And database modelling uses the name 'entity', or sometimes even 'type', for a notion that is also similar but not equal to our concept.

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'.

The many different names and definitions may be a cause of confusion. Illustrative of this 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 however prefer to call *Customer* the concept, and refer to the person John Doe as an instance of Customer.

Also, what we call a relation goes under different names. SBVR calls 'fact type' for what we prefer to call a 'relation'. UML coins the word 'association', and 'reference' and 'relationship' are used by still others.

For some readers, words like 'source' and 'target' may bring to mind the 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, from target back to source, is missing. 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 (see below), resulting in the infamous "error 404" reports.

3.1.9 Integrity

When you deal with concepts and relations in computers, you must always adhere to two demands.

First, for every concept, it must be so that *every atom is unique within the extension*. Remember that each atom represents a single term in the business environment. That term meets the definition of the concept, or it does not. It is a member in the population, or it is not. But never can it be in the population, twice! If students are identified by their first names, then the computer can only deal with one student named Caroline. This demand was referred to as *entity integrity*. *entity integrity*

Second, we stated that each tuple in (the extension of) a relation refers to one instance of the source and one instance of the target concepts. This requires that those two *atoms referred to must actually exist* in the current extensions of the concepts, they must be on record. This important demand is usually referred to as *referential integrity*. *referential integrity*

At first glance, these two are rather trivial demands. However, one must realize that extensions are constantly changing. Therefore, if an atom is deleted from (the current extension of) some concept, the consequence is that all tuples referring to that very atom 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 atoms are present in their respective extensions.

3.1.10 Validation

The intension (definition) of a relation tells us how to decide which facts ought to be true in the business environment. The extension is the current content, it captures all of the facts that are currently on record.

You must realize that these two are fundamentally different. For example, the current content of the relation may not be up-to-date: a fact is true, but not yet recorded. Or a tuple is recorded for a relation, but the corresponding fact is no longer true. This kinds of problems may not always be attributed to a computer issue or a business error, 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*. Such a check will concern the (current) contents of one or more extensions. Validation may also be used in a broader sense: whether an entire design captures all the relevant features of a business *validation*

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

verification

Do not confuse validation with *verification* which is only an internal check on the stored information, and reporting possible problems, especially problems with integrity. 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.

3.2 Models and diagrams

3.2.1 Conceptual Model

If you are trying to understand a business context, you will need to get a grip on the concepts and relations that are important. You start out with simple sentences, try to extract the deep structure of the sentences, and keep note of the structures you have uncovered so far.

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 system for 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 or targets. However, it can become quite confusing for people trying to read and understand a model if different relations have identical names.

3.2.2 Conceptual diagram

A good way to get a grip on the concepts and relations in your business environment is by making a diagram such as figure 3.4. A picture is often very useful to oversee the context, and to discuss important aspects with stakeholders.

The example conceptual diagram uses dots to represent concepts, connected by arcs that represent the relations. This diagram employs the arrow notation, with the arrowhead pointing from the source (shaft of

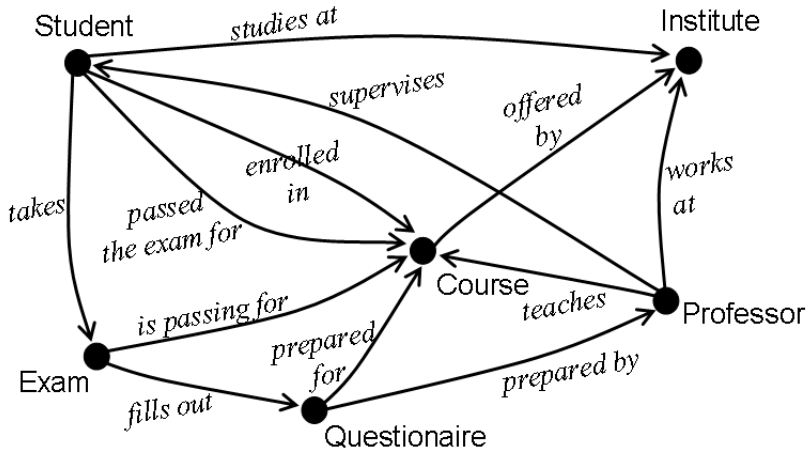

Figure 3.4: Example of a conceptual diagram

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.

Remark that the diagram does not show current students or courses that are presently available. Nor does it show which students are involved in which course. In general, abstracting away from the contents enables you to oversee the structure of many relations at once.

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

In this kind of diagrams, the arrowheads 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, or at the end of the arc, as for instance is common in UML diagrams. Still other notations omit the arrows altogether, so that source or target must be looked up in the documentation of the conceptual model.

3.2.3 Instance diagram

You can also use diagrams to show (some of) the contents of the concepts and relations. Such a diagram is called an Instance diagram, as shown in figure 3.5. The instance diagram provides a level of detail that is lacking in the Conceptual Diagram. It depicts some instances of the concepts and relations, corresponding to true facts of reality. Already

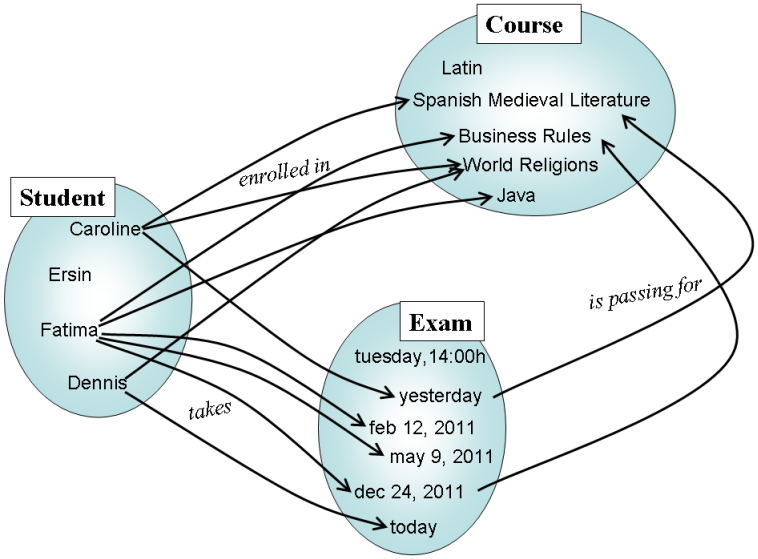

Figure 3.5: Example of an instance diagram

at this small scale, the diagram is rather cluttered. As mentioned before, instance diagrams are generally not very comprehensible due to the sheer amount of detail. They are mostly used only with small subsets, to illustrate a certain finesse of argument or to highlight some intricacy.

3.3 Operations on relations

We started out with simple sentences, facts that are true within a certain business context. Sentences can be combined in many ways to produce other sentences, that may be less simple to understand. But if you make proper combinations, then the new sentences will express something about the business that is true as well.

Relation Algebra provides numerous operations to produce new relations from existing ones. Learning to use them is vital in order to exploit the expressive power of relations: applying the operators correctly will ensure that the new sentences, your new facts, will be true as well.

We begin with two simple yet important operators: complement and inverse. Then we discuss operations that you ought to be familiar with: intersection, union, and difference. The most important operator is discussed next: composition. Other relational operators such as 'dagger' and 'implication' are discussed thereafter.

3.3.1 Complement

The *complement operator* is used to indicate the atoms that are *not* in the population of a relation. *complement operator*

Consider the mixed doubles, defined as subset of the Cartesian product of Woman × Man. The relation signature is $doublesWith_{[Woman,Man]}$. The complement is the relation that has the same type, but its contents is the 'remainder': all pairs in $\mathbb{V}$ that are *not* in the original relation. For this reason, some authors refer to this operation as *negation*. *negation*

Complement is usually denoted by an overstrike, $\overline{r}$, and you pronounce it as 'complement of r'. Because overstrike is not easy to type, it may be denoted by a minus sign in front of the relation: -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 entered for the mixed doubles competition (table 3.4).

complement of *doublesWith*	Marek	John	Rob	Toine	Raúl
Sophie	x	x	x	x	
Aisha		x	x	x	x
Jenny	x	x	x	x	x
Nellie	x	x	x		x

Table 3.4: Pairs not entered for a mixed double competition

Another way to denote a complement is by way of *set difference*, using the backslash \symbol. The set difference $F \backslash G$ is all instances that are contained in set F, at the left, but not contained in the righthand set, G. So we may denote the complement of relation $r_{[A,B]}$ as the difference $\mathbb{V}_{[A \times B]} \backslash r$. *set difference*

Obviously, if there is any ambiguity what the enveloping Cartesian product would be, then the complement would also be in doubt. To illustrate the point: the complement of all students attending class is probably those students that are enrolled for the course, but that do not attend todays class. But perhaps the speaker meant the people in the classroom that are not students, such as the teacher and perhaps a visitor?

Complement is a so-called *involutive* operator: apply the operator twice, and it will reproduce the original relation. In formula: *involutive*

$$\overline{\overline{r}} = r$$

3.3.2 Inverse

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 of course 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 a $^\smile$ symbol (pronounced 'flip') after the relation name:

$$sang^\smile{}_{[Hit\ song, Popgroup]}$$

We may pronounce this as 'flip of Popgroup *sang* Hit song'. For most of us, it is more comfortable to change the relation name into something more sensible like 'Hit song *wasSungBy* Popgroup', or 'Hit song *originatedFrom* Popgroup'. Beware however that in general, different relation names mean different relations!

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

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

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, as shown in table 3.5.

Man	Woman
Raúl	Sophie
Marek	Aisha
Toine	Nellie

Table 3.5: Inverse of the selected couples for a mixed double competition

In the two-dimensional matrix layout (see table 3.2), the inverse operator mirrors the entire content of the matrix along the diagonal. The two axes, source and target, are swapped accordingly.

involutive

Like complement, the inverse operator is also *involutive*: apply it twice, and you end up with the original relation. This may be no surprise, as the inverse operator does not change the facts, it merely rephrases them. In formula:

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

3.3.3 Set operations

Set theory in mathematics describes several operators that produce new sets from existing ones. Each operation unambiguously defines which elements will belong to the new set, and which ones are excluded.

In particular, we assume that you are familiar with the intersection, union and difference operators on sets. It ought to be an easy exercise to verify for arbitrary sets F and G that:

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

and

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

Because we defined relations as subsets of a Cartesian product, we can apply any *set operation* to two relations r and s, provided of course that they share the same types. We have: *set operation*

intersection

- *intersection* : $r \cap s$ is the relation that contains the tuples 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

- *union* : $r \cup s$ is the relation that contains all pairs that are present either in relation r or in s, or $r \cup s = \{(x, y) | (x, y) \in r \text{ or } (x, y) \in s\}$

difference

- *difference* : $r \backslash s$ is the relation that only takes tuples of relation r not present in s, or $r \backslash s = \{(x, y) | (x, y) \in r \text{ and } \neg(x, y) \in s\}$. You may check that this is equivalent to the intersection $r \cap \overline{s}$.

Remember, if relations are not defined on the same Cartesian product, they will contain tuples that cannot be compared. Such relations are disjoint, and the set operations above produce no meaningful results.

3.3.4 Composition

The *composition* operator is perhaps the most important operation in Relation Algebra. It produces a new relation from two existing ones. Or more down to earth: it combines two facts, two simple sentences into a single sentence that is not so simple any more. The formal definition of composition is as follows. *composition*

Let $r_{[A,B]}$ and $s_{[B,C]}$ be two relations, with the same concept being the target of r as well as the source of s. Then the composition of r and s, is the relation with type A $\times$ C. Its content is calculated as

$$\{ (a, c) \mid \text{there exists at least one } b \in B \text{ such that } a \; r \; b \text{ and } b \; s \; c\}$$

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

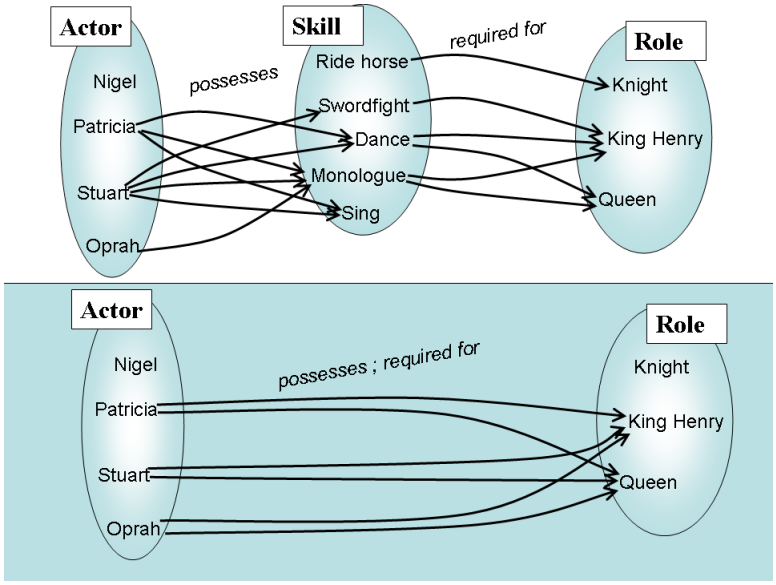

Figure 3.6: Instance diagram of a composition

The figure 3.6 illustrates how composition works. An Actor (at the left) *possesses* some Skills (middle), and Skill *required-for* a Role (at the right). The meaning of the composition relation is: the Actor *possesses at least one of the Skills required for* the Role. In natural language, you might say that the actor has some skill for the role, but the formalization is much more precise. It is a combination of two facts: the actor possesses a particular skill. And the skill is required for the role. In fact, the actor may possess several skills, or the role may require many skills, but exact information about skills is absent from the composition. For an actor to be related to a role means only that the actor has at least one required skill for the role.

Another example was seen in figure 3.5 by composing relation *takes* with *isPassingFor*. Student Caroline takes an exam, yesterday, that is passing for Spanish Medieval Literature. Also, student Fatima takes several exams, and one of them is passing for the Business Rules course. But according to the diagram, no exam was taken by Dennis that is passing for World Religions.

Obviously, you cannot compose just any two relations. If there is no middle ground, then composition has no meaning. Readers familiar with relational database theory will recognize that composition corresponds to the 'natural join' operator.

3.3.5 Advanced operators

Apart from composition, several other operations can be defined for two relations. We will briefly discuss them, although we will rarely use them. Moreover, mathematically speaking, it can be shown that all these advanced operations can be reduced to ordinary composition. In the following definitions, let $r_{[A,B]}$ and $s_{[B,C]}$ be two relations, with B being the target of r and the source of s.

The *relative addition* of r and s, is the relation with signature $(r \dagger s) : A \times C$, and its content is defined as *relative addition*

$$\{ (a, c) \mid (\forall\, b \in B) \text{ either } a\ r\ b \text{ or } b\ s\ c \text{ or both}\}$$

The relative addition operator is denoted by the symbol †, pronounced as 'dagger', between the two relation names.

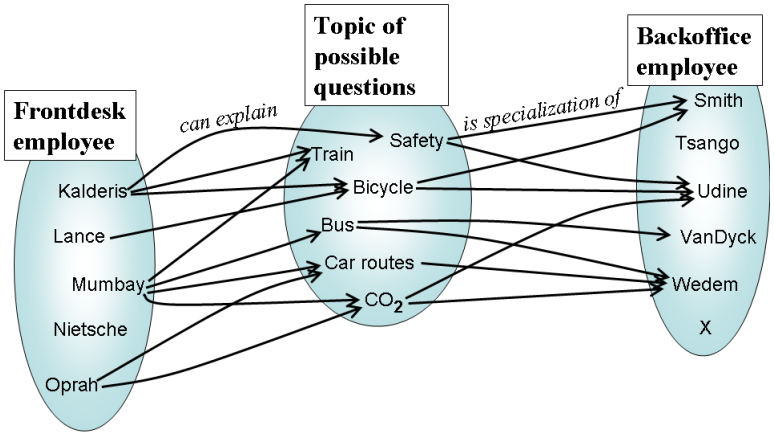

Figure 3.7: Relative addition for Frontoffice-Backoffice teams

To illustrate how relative addition works, consider a company that receives a lot of customer questions about a number of topics. Frontdesk employees are able to explain some of the topics over the telephone to customers. Or they may connect a customer through to some back office employee who specializes in certain topics. A frontdesk employee can be teamed up with a backoffice employee if together, they cover all possible topics. Figure 3.7 shows an instance diagram that contains three possible teams.

The natural language interpretation of relative addition is just that: a tuple (a, c) is in the relative addition if a and c together cover the entire middle ground of B.

The *relative implication* of r and s, is the relation with signature $(r \lfloor s) : A \times C$. By definition, its content is *relative implication*

$$\{ (a, c) \mid (\forall\, b \in B) \text{ if } a\ r\ b \text{ then } b\ s\ c \}$$

We denote this operator by a square bracket [, resembling the inclusion symbol $\subset$.

As an example, suppose the institute offers tours abroad for students. Whether a student is eligible for a tour destination, depends on whether he or she has passed for the required courses. Two relations are involved:

- Course *requiredFor* Destination
- Student *passedTheExamFor* Course

and these are combined to produce a new relation

- Student *isEligibleFor* Destination

Figure 3.8 shows a sample of the instance diagram. Caroline is eligible to go to Madrid, because she has passed for the required course Spanish Medieval Literature. For the trip to Rome, two courses are required, Latin and World Religions. Student Ang has passed for both courses, so he qualifies. Student Brown is less fortunate, as she has not yet passed for Latin which is one of the required subjects.

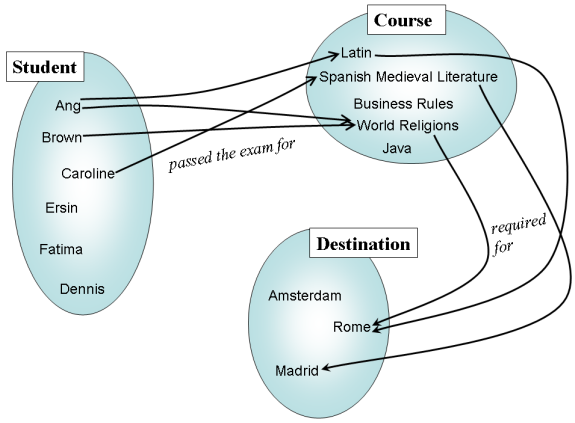

Figure 3.8: Relative addition of *passedTheExamFor* and *requiredFor*

The demand means that for a destination to be open to a student, it is required that for all courses, if Course *requiredFor* Destination, then Student *passedTheExamFor* Course. In other words, for a student to be eligible for a destination, it is required for every course that *passedTheExamFor* [*requiredFor*. Remark that the destination of Amsterdam requires no courses at all, hence all students are eligible irrespective of their achievements in exams.

relative subsumption Finally, the *relative subsumption* of relations r and s, is the relation with

signature $(r \rceil s) : A \times C$. By definition, its content is

$$\{ (a, c) \mid (\forall\, b \in B) \text{ if } b\ s\ c \text{ then } a\ r\ b \}$$

This operator is denoted by the] symbol, which resembles the inclusion in the other direction, for the same reason as above.

3.4 Laws for operations on relations

Using clauses like 'and', 'or', 'if', 'then', 'for all' and 'exists', you can combine simple sentences into rather complex phrases. Just so, relation operators enable you to write complex formulas. But often, one meaning can be phrased in several equivalent ways, and a mathematical formula can be written in several ways. It is often not easy to decide whether different expressions have the exact same meaning, and if they will be true (or false) in all same circumstances. For example:

- $for^{\smile}; sent \vdash deliveredTo$, meaning: 'The *Customer* will accept an *Invoice* for an Order, only if that Order was delivered to the *Customer*.'
- $\overline{deliveredTo}; sent^{\smile} \vdash \overline{for^{\smile}}$, 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*.'

Do these formulas express the same business meaning, or not? As a designer, you want to pick an easy way to express a certain text. To do that, you want to be able to rewrite one formula into another without loss of meaning.

This section introduces important laws from Relation Algebra that allows you to do exactly that. The laws are useful, because they allow you to manipulate with entire extensions at once, instead of having to consider all the tuples one by one. As a rule designer, you must learn to use these laws, and manipulate and reason with formulas and operators, just like you have once learned to manipulate with numbers.

3.4.1 Laws for set operators

Operators on sets follow some simple laws. For instance, when determining the intersection of multiple sets, it makes no difference in which order the intersections are calculated. To put it more formally: intersection is

$$\text{associative: } (A \cap B) \cap C = A \cap (B \cap C) = A \cap B \cap C \tag{3.1}$$

$$\text{commutative: } A \cap B = B \cap A \tag{3.2}$$

associative
commutative

Union, like intersection, is also *associative* and *commutative*, i.e. $\cap$ may be replaced by $\cup$ in the two laws above. When a formula combines union and intersection, the laws of distributivity apply:

$$\begin{array}{rcl} A \cap (B \cup C) & = & (A \cup B) \cap (A \cup C) \\ A \cup (B \cap C) & = & (A \cap B) \cup (A \cap C) \end{array} \tag{3.3}$$

The first line reads: "the union with an intersection equals the intersection of the unions", and the second one can be pronounced as "intersection with a union equals the union of the intersections".

3.4.2 Laws for composition and relative addition

Composition and relative addition are associative operators. Just like in law 3.1, it does not matter how brackets are placed in an expression with two or more of the same operators. As a consequence, brackets may be omitted and the expression looks less cluttered.

$$(p;q);r \;=\; p;(q;r) \;=\; p;q;r \tag{3.4}$$

$$(p \dagger q) \dagger r \;=\; p \dagger (q \dagger r) \;=\; p \dagger q \dagger r \tag{3.5}$$

The next law is special for composition. It states that the identity relation $\mathbb{I}$ may be removed if used in a composition: the identity relation is *neutral* with respect to composition and can be omitted.

neutral

$$r;\mathbb{I} \;=\; \mathbb{I};r \;=\; r \tag{3.6}$$

Similarly, you may remove the complement of identity $\overline{\mathbb{I}}$ (sometimes referred to as *diversity*), if it appears in a relative addition.

diversity

$$r \dagger \overline{\mathbb{I}} \;=\; \overline{\mathbb{I}} \dagger r \;=\; r \tag{3.7}$$

3.4.3 Laws for the inverse operator

The following laws show how the conversion operator behaves.

$$\text{involutive: } r^{\smile\smile} \;=\; r \tag{3.8}$$

$$\text{distributive: } (r \cup q)^{\smile} \;=\; r^{\smile} \cup q^{\smile} \tag{3.9}$$

$$\text{and also: } (r \cap q)^{\smile} \;=\; r^{\smile} \cap q^{\smile} \tag{3.10}$$

In the next two laws, the order of the relations r and s is reversed:

$$(r;s)^{\smile} \;=\; s^{\smile};r^{\smile} \tag{3.11}$$

$$(r \dagger s)^{\smile} \;=\; s^{\smile} \dagger r^{\smile} \tag{3.12}$$

3.4.4 Laws for distribution in compositions

When working with compositions of relations, it often happens that a union or intersection appears somewhere in the formula. There is one distributive law that allows to remove the union operator from compositions of relations.

$$p;(q \cup r);s \quad = \quad (p;q;s) \cup (p;r;s) \tag{3.13}$$

It would be nice if the same kind of distribution would hold for intersection, $\cap$, but alas not. Distribution does not apply for intersections in general. Only if certain conditions are met, does distribution apply for intersections. This will be demonstrated later on.

3.4.5 Laws for complement

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

$$\text{involutive: } \overline{\overline{r}} = r \tag{3.14}$$

The mathematician *De Morgan* was first to formulate these laws: *De Morgan*

$$\overline{r} \cup \overline{s} \quad = \quad \overline{r \cap s} \tag{3.15}$$
$$\overline{r} \cap \overline{s} \quad = \quad \overline{r \cup s} \tag{3.16}$$

The laws above are frequently used because they allow the complement operator to 'move around' in your formulas. There are similar laws involving ; and † operators, also named after De Morgan:

$$\overline{r};\overline{s} \quad = \quad \overline{r \dagger s} \tag{3.17}$$
$$\overline{r} \dagger \overline{s} \quad = \quad \overline{r;s} \tag{3.18}$$

Equations 3.16 and 3.18 are collectively known as De Morgan's laws. Lesser known, but almost as important are equivalences which De Morgan called "Theorem K". These will be discussed in the next chapter.

3.4.6 Laws about inclusion

Laws about set inclusion are very important, because inclusion lies at the basis of rule assertions that the next chapter is all about. This section

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

$$(r \cap s) \vdash r \vdash (r \cup s) \tag{3.19}$$

Inclusion can be combined with the flip and complement operators, which produces the following equivalences. Beware though that the order of r and s is reversed in the second formula only:

$$r \vdash s \;\Leftrightarrow\; r^{\smile} \vdash s^{\smile} \tag{3.20}$$
$$r \vdash s \;\Leftrightarrow\; \overline{s} \vdash \overline{r} \tag{3.21}$$

monotonicity

Inclusion also combines easily with operators other than flip and complement, sometimes referred to as the *monotonicity* of inclusion. But beware that the following laws are not equivalences but work in only one direction:

$$r \vdash s \;\Rightarrow\; r \cap t \vdash s \cap t \tag{3.22}$$
$$r \vdash s \;\Rightarrow\; r \cup t \vdash s \cup t \tag{3.23}$$
$$r \vdash s \;\Rightarrow\; r;t \vdash s;t \tag{3.24}$$
$$r \vdash s \;\Rightarrow\; t;r \vdash t;s$$
$$r \vdash s \;\Rightarrow\; r \dagger t \vdash s \dagger t \tag{3.25}$$
$$r \vdash s \;\Rightarrow\; t \dagger r \vdash t \dagger s$$

3.4.7 Operator precedence

To conclude this section, we give some conventions that govern precedence of operators. Just like in arithmatics, 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 3.6 contains the precedence rules.

3.5 Homogeneous relations

So far, we discussed relations based on Cartesian products with sources and targets arbitrary. But the same concept may be used for both source and target. This section is devoted to exactly such relations, and the special features that they have.

homogeneous

A *homogeneous* relation is a relation for which one concept serves both as source and as target.

expression	precedence rule	to be read as
unary operators $r^\smile$ *and* $\overline{r}$ *have highest precedence*		
$\overline{r}; q$	flip and complement always take precedence	$(\overline{r}); q$
$\overline{r} \dagger r$	(also if flip or complement appear	$(\overline{r}) \dagger q$
$r^\smile; q$	at the righthand side)	$(r^\smile); q$
$r^\smile \dagger r$		$(r^\smile) \dagger q$
; *and* $\dagger$*, having equal precedence, take precedence over* $\cap$ *and* $\cup$*,* $=$ *and* $\vdash$		
$p \cap q; r$	; and $\dagger$ bind stronger than $\cap$ and $\cup$, $=$ and $\vdash$	$p \cap (q; r)$
$p \cap q \dagger r$	(also if ; or $\dagger$ appears at the lefthand side)	$p \cap (q \dagger r)$
$p \cup q; r$		$p \cup (q; r)$
$p \cup q \dagger r$		$p \cup (q \dagger r)$
$\cap$ *precedes over* $\cup$		
$p \cup q \cap r$	$\cap$ binds stronger than $\cup$	$p \cup (q \cap r)$
$=$ *and* $\vdash$ *have weakest precedence of all*		
$p = q \cup r$	$\cup$ and $\cap$ bind stronger than $=$ or $\vdash$	$p = (q \cup r)$
$p = q \cap r$		$p = (q \cap r)$
$p \vdash q \cup r$		$p \vdash (q \cup r)$
$p \vdash q \cap r$		$p \vdash (q \cap r)$

Table 3.6: Precedence of operators

All other relations will be called *heterogeneous* relations: their source and target concepts are different. *heterogeneous*

We already encountered one example of a relation with identical source and target, namely the Identity relation, abbreviated $\mathbb{I}$. Examples of homogeneous relations are easy to think of, once you realize that they correspond to simple sentences that express facts about similar terms. Some examples are Person *isRelatedTo* Person, in arithmetics: Number *isGreaterThan* Number, and in chemistry: Compound *mayDecomposeInto* Compound. Or in software maintenance: Software-change *interferesWith* Software-change. In business administration: Department *isSubordinateTo* Department. And in process design: Use-case *isSubvariantOf* 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 its source and target are the same. We can point out several characteristics that the relation r may, or may not have.

3.5.1 Reflexive

A relation $r_{[A,A]}$ is called *reflexive* if for all elements x in the set A, *reflexive*

the tuple (x, x) is in r. The relation *isACloseFriendOf* is an example: everybody is a close friend of themselves. The condition can be stated 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*. Two irreflexive examples are *isSpectatorOfPerformanceByActor* or *isParentOf*.

In a matrix 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 may have markings anywhere, except on the diagonal.

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

3.5.2 Symmetric

symmetric

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 *isMarriedTo*, *isInAMeetingWith*, and *isInTheSameClassAs*.

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 *awardsBonusPaymentTo*. Of course, violations may occur: two managers that award bonuses to one another. Remark that an asymmetric relation is automatically irreflexive. Indeed, we do not want a manager to award a bonus to him- or herself.

Like with (ir)reflexivity, many homogeneous relations are neither symmetric nor asymmetric. The relation Website *hasHyperlinkTo* Website

is an example: some websites may link to one another, but there is no rule or obligation to do so.

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

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

3.5.3 Property relation

In some 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 with the atoms x that do satisfy $(x, x) \in p$, and the subset of all other elements in A that are not in $p : (y, y) \notin p$. 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 atoms of A. Yes, the atom x is in the subset for which (x, x) is in p. Or no, the tuple (y, y) is not in p. For this reason, a homogeneous relation that is symmetric and antisymmetric is sometimes called a *binary property* for A. *binary property*

By the way: there are other ways to specify properties for atoms of A, as will be shown in the chapter on design considerations.

3.5.4 Transitive and intransitive

If a relation is homogeneous, then we may define the composition with itself. In fact, this can be done over and over again, and you can derive many new homogeneous relations in this way. As an example, consider the relation *isCloseTo*. If Utrecht *isCloseTo* Hilversum, and Hilversum *isCloseTo* Almere, then we have Utrecht *isCloseTo* someplace that *isCloseTo* Almere. A relation r is called *transitive* if the composition *transitive* with itself is contained in the relation itself:

$$r; r \vdash r$$

Another example of a transitive relation is the 'subset' relation among sets: if $A \subset B$ and $B \subset C$ then $A \subset C$, and 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 *isParentOf* Person, then we can compose the *isParentOf* relation with itself. The composite is in fact the *isGrandparentOf* relation, and a person may not be both parent and grandparent of a child. Hence, *isParentOf* is an example of an *intransitive* relation, meaning:

intransitive

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

Repeat the composition and you arrive at Person *isGreatgrandparentOf* Person, etcetera. Genealogy now jumps to one generalized relation Person *isAncestorOf* Person, which is a transitive relation. The reader should verify that this particular relation does satisfy the inclusion $r;r \vdash r$, but equality $r;r = r$ does not hold. Again, like reflexivity and symmetry, many homogeneous relations have neither the transitive nor the intransitive property.

3.5.5 Summary of homogeneous properties

The various properties that homogeneous relations may have, can be written using mathematical symbols. If you do not readily grasp the notations, you should check back in order to practice your skills in translating formulas into natural language, and vice versa. The mathematical shorthand aims to capture the simple business facts, no more, no less. For you, there should be no mystery involved.

reflexive	$(\ \forall a\ )\ a\ r\ a$	(3.26)
irreflexive	$\neg(\exists a\ )\ a\ r\ a$	(3.27)
symmetric	$(\ \forall a,b\ )\ a\ r\ b\ \rightarrow\ b\ r\ a$	(3.28)
asymmetric	$(\ \forall a,b\ )\ a\ r\ b\ \rightarrow\ \neg(b\ r\ a)$	(3.29)
antisymmetric	$(\ \forall a,b\ )\ a\ r\ b\ \wedge\ b\ r\ a\ \rightarrow\ a=b$	(3.30)
transitive	$(\ \forall a,b\ )\ (\exists\ x)\ (a\ r\ x) \wedge (x\ r\ b)\ \rightarrow\ a\ r\ b$	(3.31)
intransitive	$(\ \forall a,b\ )\ a\ r\ b \rightarrow (\forall\ x) \neg(a\ r\ x) \vee \neg(x\ r\ b)$	(3.32)

3.5.6 Structure of a set

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) A_1, A_2, A_3 ..., up to A_n, 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.

equivalence relation The relation is called an *equivalence relation*. Examples of this kind of sit-

uation are easy to find: Employee *worksAtTheSameDepartmentAs* Employee, or Student *hasSameGradeAs* 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. This is also called a *partitioning* of the set A. *partitioning*

It is well possible to have more than one equivalence relation (and matching partitioning) for a concept. For example, cars can be partitioned according to color, or age, or mileage, weight, value or any other property.

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. The relation is said to be an *ordering relation* or order. For example: Employee *isSubordinateTo* Employee, amounting to hierarchy among workers. Or Department *isPartOf* Department, which corresponds to organizational top-down structure. Other examples are set inclusion and ancestor relations. *ordering relation*

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, in the instance diagram they are depicted 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*. *partial 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, $p \cup p^{\smile} = \mathbb{V}$. For a linear order, the instance diagram will resemble a single line, with all elements neatly arranged on that line. An example is the common *isSmallerThanOrEqualTo* relation on numbers. *total order*

3.5.7 IsA relation

To conclude this section on homogeneous relations, we finish with an ubiquitous relation that is *not* homogeneous. 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. The corresponding sentences are very simple, like 'X, the customer who joined last week *is* customer X', or 'Y, the customer who lives in New York *is* customer Y'. The common template looks like '.... *isA* ....' In all cases, we have a relation with the subset as source, and the enveloping set as target. An instance of the former *isA* instance of the latter, no exceptions.

The definition (intension) of the first concept, which is the subset, is more limited, more restrictive than the intension of the enveloping concept. And necessarily, the extension of the first concept is contained in the extension of the other concept. Whenever this is the case, we call the source the *specialisation*, the target being the *generalisation*. *specialisation* *generalisation*

> An *inclusion relation* is a relation for which the source is a proper subset of the target *inclusion relation*

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

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

3.6 Multiplicity constraints

Simple sentences express single facts. By studying the deep structure of sentences, we arrived at the notion of relation, called 'fact-type' by the Business Rules Manifesto. The Manifesto states that rules build on facts. Indeed, even a single relation can 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. Such rules are frequently encountered, and they have their own name: *multiplicity constraints*. They express knowledge about how the tuples in a relation should be organised, and what behaviour is expected or prevented.

multiplicity constraint

From mathematics we learn that multiplicity constraints come in exactly four flavours. It is the duty of the rule analyst 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 and anything goes, up to relations that are under very restrictive multiplicity constraints.

3.6.1 Univalent and Total

These two multiplicity constraints must be verified at the source concept. A relation r is said to be:

univalent

- *univalent*, if every atom in the source occurs in at most one tuple of the relation, or: every atom in the source is related to at most one atom in the target,

total

- *total*, if every atom in the source always occurs in at least one tuple of the relation, or: every atom in de source is related to at least one atom in the target.

These properties are easily verified in a matrix representation of the relation. If we write 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 (figure 3.9), 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.

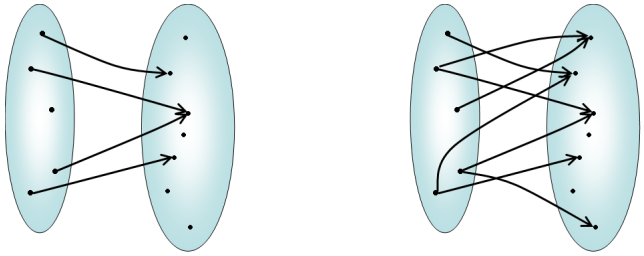

Figure 3.9: Left: univalent, not total; right: total, not univalent

3.6.2 Injective and Surjective

These two multiplicity constraints are to be verified at the target. The relation r is called

injective

- *injective*, if every atom in the target occurs in at most one tuple of the relation, or: every atom in the target is related to at most one atom in the source.

surjective

- *surjective*, if every atom in the target always occurs in at least one tuple of the relation, or: every atom in the target is related to at least one atom in the source.

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

In an instance diagram (figure 3.10), 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.

3.6.3 Function

Some combinations of multiplicity constraints 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 atom in the source is related to exactly one atom in the target concept. *function*

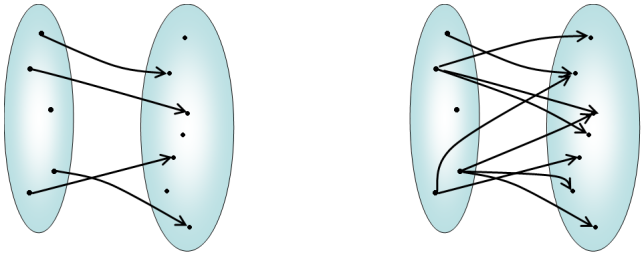

Figure 3.10: Left: injective, not surjective; right: surjective, not injective

The word 'function' is also used in mathematics, with notation $f(x) = y$. For instance the function 'square': $f(x) = x^2$. It is no coincidence that the same word is used: in a 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. This becomes clear if we write (x, x^2) instead of $f(x) = x^2$. This relation contains tuples such as $(1, 1)$, $(2, 4)$, $(3, 9)$ but also $(0, 0)$, $(-1, 1)$, $(-2, 4)$. Indeed, the 'square' function produces correct tuples in this way, exactly one tuple for each number x.

3.6.4 Injection, Surjection, Bijection

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

injection

surjection

bijection

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

Figure 3.11 depicts these properties. Notice in the rightmost example that the 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 atoms, for each atom in the source is related to one target element, and reversely.

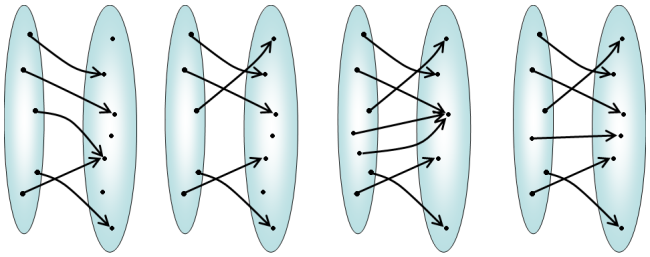

Figure 3.11: Common combinations of multiplicities. Left to right: function, injection, surjection, and bijection.

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 *isSentTo* in which that invoice appears. But not the other way around: one customer may be sent no invoice at all, or she may receive hunderds.

Multiplicity constraints are very important in conceptual models, and we will discuss some more features of multiplicities in the next chapter that is all about rules.

Chapter 4

Rules

Abstract

Why would you want to write rules of your own in a precise manner? For rules that are used to specify accurately what 'the business' wants, your skill to write rules precisely is a most desirable asset. It sets you apart from all those analysts who specify 'roughly' what the business needs. For rules that are used to define requirements of information systems, your skill is appreciated to produce unambiguous specifications. You will receive your greatest compliment, however, when stakeholders from the business acknowledge that you have understood them. This skill does three things for you: It allows you to think and work precisely on your own. It lets you share requirements with 'systems people' with mathematical precision. And it lets you share business requirements in the (natural) language of the business. That is what you can achieve by practicing the techniques in this chapter.

This chapter defines the notion of rules, discusses the properties within the framework of Relation Algebra theory, and helps you to learn reading and writing rules fluently.

First, we discuss rules in general, and we outline how rules can be formulated in Relation Algebra. We provide a number of laws that you must learn in order to write the same rule in different ways. Next, we discuss rule violations: what are they, what needs to be done to detect them, and what kinds of business rules cannot be violated at all. The section that follows outlines a procedure to translate a rule assertion to natural language. That will help you to understand rules for yourself, and also to explain them to others. A section is included with examples showing how to use laws and how to manipulate with rules and relations. This illustrates how to transform one rule assertion into another, without a change of intent or meaning. The goal is to find a best way to formulate and explain the rules to business stakeholders.

4.1 Rule definition

4.1.1 Business Rule

A business rule is a prescribed guide for conduct and/or action, and it always removes some degree of freedom in the everyday work practice. In chapter 2 (page 20) we gave our definition of *business rule* as:

> a verifiable statement that some stakeholders intend to obey within a given context.

A business rule is some regulation or principle that governs the conduct within a particular area (or "context") of the business. The rule provides guidance, knowledge, descriptions or prescriptions regarding potential, allowed, or prohibited actions, business processes and/or human activities. Keywords are: governance and guidance.

The essence of a business rule is in the guidance it provides for doing 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 rewrite the same rule in one natural language or another (even Greek), or write it in some controlled language (such as RuleSpeak), in Relation Algebra or in any other way. Nor does it matter if the rule is implemented in an information system, in workflow procedures, or if it remains a spoken instruction for workers to follow. No matter how or where the rule is implemented, the rule is there and the business should abide by it.

The business, and not the IT support department, is responsible for the business rules. To formulate them, to have them implemented, to enforce them and take appropriate action if violated, to revise or even delete 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 $\overline{p} \cup \overline{q} = \overline{p \cap q}$, or the law of gravity. The point is that these are not *business* rules: nobody in the business is interested in maintaining them. Such rules do not provide any intelligent guidance for the business.

To be a business rule, there must be *business jurisdiction* in formulating and maintaining that particular rule. And at least one stakeholder in the business must have the intent to have the rule obeyed, at all times.

business jurisdiction

4.1.2 Categories of rules

Business rules come in a great variety. To list but a few categories described in the literature:

- Abstract guidance: our company respects customer privacy.
- 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.

We do not claim that this list is correct and exhaustive. For example, you might say that some of them are not rules, because they do not express obligations or necessities, words like 'must' or 'cannot' are lacking to restrict freedom of actions. And others would extend the list to include 'advice'. An advice confirms or reminds that some degree of freedom does exist or is allowed, and therefore is just the opposite of a rule that removes some degree of freedom.

It may be disappointing for you to learn that in fact, many categories cannot easily be described in Relation Algebra. Luckily however, many rules that guide everyday business conduct can be well captured in expressions and assertions of Relation Algebra, although some ingenuity is sometimes called for.

4.1.3 Definitions are rules, too

Concepts and relations may be referred to as *structural rules*. Intuitively, concepts and relations provide structure, but how can they be perceived as rules?

Each time a relation or concept is introduced, you can interpret that as a rule dictating its existence. The declaration of a new relation, e.g. Student *takes* Exam, can be interpreted as a rule, which says: "In our context, there must be a relation between students and exams, to be referred to by the name *takes*". It also says that, in this context, if anyone at all *takes* an exam, it must be a student. And if a student is involved in this relation *takes*, then it is an exam that is taken, and not another kind of thing. Similarly, the mere introduction of a concept, e.g. *Customer*, means that the stakeholders agree that: "In our context, there exists a concept *Customer*". By implication, they also agree that there can be atoms, such as 'Kim', 'Paul', and 'Marek', that are instances of *Customer*. So in a formal sense, there is nothing wrong in perceiving concepts and relations as 'rules'.

structure

Nevertheless, business stakeholders might not always recognise them as valuable business rules. They might find it so self evident! So why call anything so utterly obvious a rule? To spare business stakeholders discussions about the obvious, it is useful to distinguish *structure* from *constraints*. Concepts and relations belong to the structure, rules belong to the *constraints*. Structure (i.e. concepts and relations) allows us to distinguish simple sentences (such as "In Caroline's house, there are 3 rooms.") from non-sentences (such as "In 3's house, there are Caroline rooms."). The structure formalizes language. It defines which types of sentences make sense in the business context. Constraints tell us which of those sentences are true. They allow us to distinguish facts (i.e. true sentences) from non-facts.

A final word about true facts, false facts, and non-facts. We are concerned only with actual facts of the business, and not with hypothetical ones. The latter kind causes all kinds of philosophical problems, as the example conceived by Russell may illustrate: "the king of France is bald". Clearly, this fact is false. But the opposite sentence, "the king of France is not bald", is equally false. What is the problem here? If we would draw up a list of all bald persons, and another list of people known to be not bald, then the king of France would not turn up in either list. In other words, an atom "the king of France" does not refer to any real person, it is not within our context. Therefore "the king of France" is not really an atom, and sentences about it cannot be dealt with.

4.1.4 Unconstrained Conceptual Model

In theory, we could envision a conceptual model with definitions of concepts and relations only, without any rules or multiplicity requirements. This is sometimes referred to as an *unconstrained conceptual model*. It 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. In terms of the SBVR, the unconstrained model constitutes a *business vocabulary*, and it captures the *structural business rule*s, but no more.

unconstrained conceptual model

business vocabulary

structural business rule

The idea behind the unconstrained conceptual model is that you can now impose the *operational business rule*s one by one. Each time, you must resolve the violations that emerge as the rule is being added. After a number of steps, the model would have all its rules in place with no remaining violations. This approach would provide the designer with an insight of the true impact of each individual rule.

operational business rule

4.1.5 A rule and its enforcement are separate concerns

Does rule management mean that all rules are enforced by a computer?

The answer to that question is a definite No! There are infinitely many ways in which a business may enforce its rules. Some examples are:

- Guideline: advocate the rule to workers and teach it in training courses, but do not enforce.
- Comply or explain: always allow rule override, but require that an explanation is provided.
- Post-authorize: if a rule is violated, it may be approved after the fact, but you may be subject to sanctions or other consequences if you break the rules beyond your authorization.
- Pre-authorize: only allow exceptions for workers with prior approval.
- Sanction: if you violate the rule, you will always incur a penalty.
- Deferred enforcement: allow violations only temporarily, in order to wait for workers with the required knowledge and skills to fix the violations.
- Enforce immediately: never allow a violation, and systematically prevent any attempt to break the rule.

The above levels of enforcement apply to the business workers: they must know how to deal with possible violations of the rule, and know the consequences. The level may be made to vary over time (start with mild tutelage but be increasingly severe), among departments (the front office may use deferred-enforcement but immediate enforcement is used at the back office), and among employees (no penalties for novices).

How information systems enforce rules is an entirely different matter. In the Ampersand philosophy, a computer supports workers, but should never patronize them. All rules that are enforced by the business are off limits for the computer. Only if the business deliberately decides to have a rule maintained fully automatic, may a computer do so. This happens for example when the business perceives a rule as almost a "law of nature". In those cases the computer may bluntly enforce the rule. For example, the rule that a financial transaction should never lose money on the way, can be regarded as a law of nature from a business perspective. Naturally, computers must maintain such rules under all conceivable circumstances at all times.

For rules that are maintained (enforced) by the business, a computer may signal violations, but go no further. So after detecting a violation, a computer may react in one of two fundamentally different ways:

immediate

- *immediate enforcement*: if a transaction attempts to add, change or delete data in such a way that it would cause a rule to be violated, then the entire transaction will be rejected. The fact that is recorded is presumably not true, because it breaks the rule. If a user asks why this transaction is rejected, the computer can refer to the rule that was violated as the reason for rejecting the transaction.

deferred

- *deferred enforcement*: if a transaction attempts to add, change or delete data in such a way that it would cause a rule to be violated, then this rule may permit the transaction, but every violation is immediately signalled to some stakeholder. Since users are involved in the enforcement, the computer must allow the rule to be violated as long as it takes to restore truth.

4.1.6 Static versus dynamic rules

Another classification is concerned with static versus dynamic rules. A familiar example of static rule is that a person must be classified as 'married' if (s)he has a spouse. Next, a dynamic rule says that a classification 'married' may never change to 'unmarried'. Nor may an 'unmarried' person have her status changed directly into 'divorced'.

To enforce such dynamic rules, one must be able to compare things before and after a change. Thus, the Conceptual Model has to carry some notion of time, or a sequence of statuses. This is very well possible, and in our opinion, the distinction between static and dynamic features is a gradual one, not absolute. Although both static and dynamic rules can be captured in one Conceptual Model, we do not presume that it is easily done. A lot of ingenuity may be called for to come up with a design that is consistent as well as understandable.

4.2 Expressions and assertions

4.2.1 Business rules and rule assertions

Business people often know their rules and talk about them in rather offhand ways. The co-workers understand each other, even if they use ambivalent expressions for the rules. But for rule engineers, analysts and developers, such liberty is unacceptable. Precision in rule formulation is a necessity, and a strict and formal language is called for. Whereas business rules are communicated in natural language, we resort to Relation Algebra in order to achieve unambiguity in asserting the rules.

But first, we must make a clear distinction between expressions and assertions.

expression

An *expression* in Relation Algebra produces a new relation from existing ones. It constitutes a formal definition (description) of the new relation. The previous chapter provided various operators for this: set operators such as union $\cup$, complement $\overline{r}$, and difference $\backslash$. And relational operators like composition, relative addition, and inverse (flip). The good news is that these operations produce exact and unambiguous definitions of the new relation. The bad news is that it is often hard to explain the definition and meaning of the new relation in natural language (we will come to that later).

assertion

An *assertion* is a formula that compares two sets, or more in particular: two relations. The comparison will produce either a 'true' or 'false' answer. The two common forms of assertion are the inclusion, $r \vdash s$, and the equality $r = s$.

The assertions above look simple, but in practice, both r and s may be complicated expressions, involving any number of relations and operators so much so that the assertion becomes hard to explain to the business stakeholders.

An assertion may evaluate to a 'true' at some times, and to 'false' at other times. Which, depends on the current extensions of the relations at the time of the comparison. It has to be determined by inspecting the recorded instances for the relations involved in both sides of the assertion.

Some assertions will always turn out to be true, regardless of the information stored in the relation extensions. For instance, we know that $\overline{r} \cup \overline{s} = \overline{r \cap s}$, regardless of the extensions of r and s. This assertion is a law of Relation Algebra, it is not a rule under business jurisdiction.

rule assertion

A *rule assertion* is an assertion, a formula in Relation Algebra, that, according to some stakeholder in the business, ought to evaluate to true, always and for any and all content of the relations involved in the assertion. If it turns out that the assertion at some point in time is violated, then the stakeholder has work to do in order to fix the problem. But violations of the rule do not signify that the rule itself is invalid and useless.

4.2.2 Matching an assertion with an expression

Assertions and expressions are not the same. But the two are closely linked. In fact, any assertion can be matched to an expression. The match is easy to explain: we just need to consider violations of the assertion.

An assertion such as $r \vdash s$ states that every tuple in r must belong to s also. No tuple should be present in r that is absent from s. That is to say: the assertion comes down to $r \backslash s = \emptyset$, or $r \cap \overline{s} = \emptyset$, both formulas capturing exactly the same meaning, the same business rule.

Now, if one (or more) tuples exist in $r \backslash s$, then the business rule is violated, and the assertions above are not true. Therefore, we can match the assertion $r \vdash s$ with the expression $r \backslash s$ (which can also be written as: $r \cap \overline{s}$). Important to realize: the expression is a relation, but the assertion is a logical 'true' or 'false'.

The implication of the match is that the latter expressions ought to be empty, always. If any tuple is present in the set $r \backslash s$, then the business rule is violated.

In an instance diagram for these relations, the lefthand expression r, should always end up as a subset of s, the righthand expression. And when we interpret this in controlled language, we will have a rule that says: 'if' a tuple is contained in the relation r, 'then' that tuple must be contained in relation s.

Likewise, the assertion $r = s$ can be matched to the union of set differences $(r \backslash s) \cup (s \backslash r)$, which, by the way, can be rewritten in several ways: as $(r \cap \overline{s}) \cup (s \cap \overline{r})$, as $(r \cup s) \cap (\overline{r} \cup \overline{s})$, as $(r \cup s) \cap \overline{(r \cap s)}$, or as $(r \cup s) \setminus (r \cap s)$.

4.3 Laws about rule expressions

Relation Algebra comes with many laws: rules about expressions that are always true, regardless of the current extensions of the expressions. Such laws let us reason about expressions, they enable us to rewrite expressions in different ways without loss of meaning. The previous chapter already introduced some of those laws of Relation Algebra. This section lists some more laws for you to know and work with. We cannot be exhaustive though. In practice, if you are faced with a particular rule expression, there may always be some special law that lets you rewrite and simplify it.

4.3.1 Rule expressions for multiplicity constraints

The multiplicity constraints were introduced in the previous chapter by making statements about individual elements and tuples in the relation, for instance:

- Relation $r : A \times B$ is *univalent* if every element in the source occurs in at most one tuple of the relation r, or: every element a in the source A is related to at most one element b in the target B.

In fact, multiplicity properties are expressable as rule assertions. For this, we look at various compositions of the relation r with its own flip. We claim that:

- $r^{\smile}; r \vdash \mathbb{I}_B \Leftrightarrow r$ is univalent,
- $\mathbb{I}_B \vdash r^{\smile}; r \Leftrightarrow r$ is surjective,

and

- $\mathbb{I}_A \vdash r; r^\smile \;\Leftrightarrow\; r$ is total,
- $r; r^\smile \vdash \mathbb{I}_A \;\Leftrightarrow\; r$ is injective.

We leave it to the reader to prove these four equivalences. It may help to draw instance diagrams in order to get a good understanding of the equivalences. For clarity, we indicated the source and target of the identity relations on A and B.

For the sake of completeness, we also stipulate:

- $\mathbb{I}_A \vdash r; r^\smile \;\wedge\; r^\smile; r \vdash \mathbb{I}_B \;\Leftrightarrow\; r$ is a function,
- $\mathbb{I}_A = r; r^\smile \;\wedge\; r^\smile; r = \mathbb{I}_B \;\Leftrightarrow\; r$ is a bijection.

4.3.2 Laws involving multiplicities

The above equivalences come in handy on many occasions. For instance, multiplicities of the flip of a relation will be immediately obvious, just by inspecting the assertions above:

- r is injective $\;\Leftrightarrow\; r^\smile$ is univalent, and
- r is surjective $\;\Leftrightarrow\; r^\smile$ is total.

Proving these equivalences is not hard. For instance, r is injective is equivalent to $r; r^\smile \vdash \mathbb{I}_A$. Now flip is an involutive operator, therefore $r = r^{\smile\smile}$. Thus we may rewrite the expression as $r^{\smile\smile}; r^\smile \vdash \mathbb{I}_A$. And this formula tells us that $r^\smile$ is univalent. We leave it to the reader to check that the reverse is also valid, i.e. to prove that r is univalent implies that $r^\smile$ is injective. And to prove the above equivalence of surjective and total for r and $r^\smile$.

It also fairly easy to check that if two relations have the same multiplicities, then their composition also has that multiplicity: if both r and s are univalent (or total, or ...), then $r; s$ is univalent (or total, or ...) as well.

If two relations r and s are defined on the same Cartesian Product, then we can inspect the multiplicities of their intersection $r \cap s$ or union $r \cup s$ of two relations. For instance, if r is univalent, then $r \cap s$ will also be univalent. The following line of reasoning proves this:

1. We know $(r \cap s) \vdash r$ and obviously, $(r \cap s)^\smile \vdash r^\smile$ as well.
2. It follows that $(r \cap s)^\smile; (r \cap s) \vdash r^\smile; r$.
3. We also know r is univalent, thus $r^\smile; r \vdash \mathbb{I}$.
4. Therefore $(r \cap s)^\smile; (r \cap s) \vdash \mathbb{I}$, or in other words: $r \cap s$ is univalent.

As the intersection operator is commutative (you may interchange the order of the two relations), it is clear that univalence of either r or s is sufficient for $r \cap s$ to be univalent. Regrettably, this does not hold the other way around: $r \cap s$ may be univalent without either r or s being univalent.

Similar issues arise for the union of two relations, $r \cup s$. The above line of reasoning shows that if the union is univalent, then r is univalent automatically, as $r = r \cap (r \cup s)$ and the latter relation is univalent. But not the other way around: both r and s may be univalent without $r \cup s$ being so. The laws for multiplicity constraints, and equivalences more in general, are not trivial, and a good rule designer should always check her rule rewritings to avoid mistakes.

4.3.3 Laws for rewriting compositions

When working with compositions of relations, it often happens that a set operator such as union, intersection or implication appears somewhere in the expression. The following distribution law shows how the union operator can be moved out of a composition:

$$p;(q \cup r);s \quad = \quad (p;q;s) \cup (p;r;s) \tag{4.1}$$

It would have been nice if distribution would also hold for the intersection operator, $\cap$. Alas, distribution does not apply for intersections in general, it can only be proven that:

$$p;(q \cap r);s \quad \vdash \quad (p;q;s) \cap (p;r;s) \tag{4.2}$$

But equality does hold under special circumstances:

$$\text{if } p \text{ is univalent then} \qquad p;(q \cap r) \;=\; (p;q) \cap (p;r) \tag{4.3}$$

$$\text{if } s \text{ is injective then} \qquad (q \cap r);s \;=\; (q;s) \cap (r;s) \tag{4.4}$$

For functions (i.e. univalent and total relations), the following laws may come in handy, for instance to move a relation across an inclusion $\vdash$, or to replace the relative addition operator $\dagger$ by ordinary composition. If f is a function, then we have:

$$r \vdash (s;f^{\smile}) \quad \Leftrightarrow \quad (r;f) \vdash s \tag{4.5}$$

$$r \vdash f;s \quad \Leftrightarrow \quad (f^{\smile};r) \vdash s \tag{4.6}$$

$$\overline{f} \dagger r \quad = \quad f;r \tag{4.7}$$

$$r \dagger \overline{f^{\smile}} \quad = \quad r;f^{\smile} \tag{4.8}$$

4.3.4 Theorem K

The previous chapter announces a law by De Morgan that was named "Theorem K". This theorem concerns an inclusion rule for relations involved in a composition. The assertion may be written in different ways.

$$p;q \vdash \overline{r} \;\Leftrightarrow\; p^{\smile};r \vdash \overline{q} \;\Leftrightarrow\; r;q^{\smile} \vdash \overline{p} \tag{4.9}$$

The three assertions are not identical: you can check this by inspecting the types of the relations involved. We use the left-and-right arrows to indicate that the three assertions are equivalent. De Morgan proved that if one of them is true (or false), then the other ones are true (or false) as well.

Theorem K is very convenient for "moving around" the complement operator in formulas. Like other laws, theorem K is handy if you want to rewrite an expression, for instance when you need to replace a complement operator in one of your formulas.

You should become familiar with theorem K, De Morgan's laws and others, and we strongly suggest that you go through a number of exercises. Only by doing exercises and understanding corrections and elaborations, will you get the hang of it and become proficient in using the laws governing relations.

4.3.5 Sound rule

Of course, you should try to formulate good rule assertions. But what makes a rule 'good'? A main advantage of stating rules in Relation Algebra is that the assertions are precise: they can only mean one thing, in the Conceptual Model. The rule assertion must be (re)phrased using the words, terms and examples that are meaningful within the business context. If that explanation can be understood by some stakeholder to mean something else than intended, you should remedy either the explanation, or improve the rule.

But there are other considerations for quality, and here are a few guidelines that you should adhere to when you write down your rule assertions.

declarative

- *declarative*: a rule assertion declares what should be kept true. Not, how this should be enforced, when, or by whom.

atomic

- *atomic*: a business rule catches one aspect only, not more. The rule assertion ought to be indivisible; if you can break it up by rewriting it as several assertions without loosing information, then do so. The separate rules can be validated and managed (changed) independently.

business oriented

- *business oriented*: the rule explanation must be in a language that the business stakeholders can understand and accept.

The last item in this list is particularly important. It says that in the end, it is not up to you to decide whether a rule is good enough. It is the business stakeholders who are responsible for their own rules. You are merely the expert helping them to catch the rules and write them down. The guidelines above are concerned with only one rule at a time. In the next chapter, we will discuss guidelines to assess the quality of entire sets of rules.

4.4 Violation of a rule

Rules are assertions of the form $r \vdash s$ that ought to evaluate to 'true', always. If you want to use business rules to guide your way of doing business, you must be very precise not only about your rules, but also about how to handle violations. Business rules can and should be well-defined, so that they can be checked for compliance with your business requirements. We claim that the use of Relation Algebra in designing for business rules will deliver high-quality rule-based designs.

Because we specify an exact formula, the computer can determine whether the rule is violated. This is done by inspecting the expression that we matched with the assertion, i.e. difference $r \backslash s$, and determine its extension. The contents of this difference ought to be the empty set, always.

violation

A *violation* of a rule is a single tuple in the relation $r \backslash s$.

Beware that one violation of the rule assertion need not automatically be one infraction of the business rule in the real world. You may not make the assumption that a business requirement or business rule is always captured in a rule assertion in a perfect one-on-one correspondence. For instance, a rule "a bridge players' hand is 13 cards" will be violated twice if a fourteenth card is dealt to one player, as another player has got only 12 cards.

Also, there may be no infraction in the real world but an error in the current recording of data, or the violation may even be a problem of the design. It is the stakeholders responsibility to remedy the violation, but in some cases, it is the rule assertion that must be repaired to prevent false violations.

To clearly understand a rule, you must clearly understand its violations: what does the single violation mean in the business environment, how did it happen, what should you do to repair the violation and make the rule 'true' again, both in the data store and in the real world. If you do not understand the violations, then you do not truly understand your rule.

4.4.1 Enforcement strategies

The "immediate enforcement" strategy dictates that any transaction that would create a violation, is prohibited. This is similar in appearance to a definitional rule: violation is prevented at all times. But it works quite differently. Violating a structural rule is impossible because you cannot record a fact that is not compliant with the structure of the Conceptual Model. Violating of a behavioural rule that by the designers' choice has immediate enforcement is made impossible only because there is active checking for violations, presumably fully automatic. And preventive action is taken before the violation materializes, presumably fully automatic as well. In database management systems, this mechanism is known as a transaction-rollback.

A business rule with the "deferred enforcement" strategy is associated with a list of all the tuples that are in violation of the rule. This list constitutes a "to-do" worklist for the stakeholders engaged in keeping this rule true. 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 exception 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, undo the original transaction that caused it.

4.5 Rules in natural language

Learning to understand and explain a Relation-algebra expression is essential for a rule engineer. The rule engineer coordinates with business workers to learn their functional requirements and capture them in formalized rule assertions. But the other direction is equally important, and should never be postponed to some later phase. The engineer must translate the formal assertions back into the every language of the work place, in order to establish agreement or detect discrepancies between the business requirements and the formulas.

This section provides an introduction on 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, the expression $takesExam;\overline{attended}$ nicely translates to the sentence: "there is some student who takes an exam, and that student has not attended class". Notice that even though the expression does not mention students, exams and classes, the sentence may refer to them because the relations take those concepts for their sources and targets.

The ability to translate and interpret formulas is important when you have to explain rule expressions 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, you should be able to do such translations without intervening steps.

4.5.1 Example translation to natural language

As a first example, let us begin with a relation r: $A \times B$ that is univalent. Above, we presented the relational assertion for univalence. Let us translate that formula into natural language. The first part is a rather mechanical "fill in the dots" exercise:

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

means *(name the source and target elements)*
Consider every source element $x \in B$ and target element $y \in B$.

means *(translate the inclusion: copy text from table 4.5.3)*
For every $x \in B$ and $y \in B$, if $x\ (r^{\smile}; r)\ y$ then $x\ \mathbb{I}\ y$.

means *(translate the identity relation $\mathbb{I}$)*
For every $x \in B$ and $y \in B$, if $x\ (r^{\smile}; r)\ y$ then x equals y.

means *(translate the composition $r^{\smile}; r$)*
For every x and y, if there exists $z \in A$ such that $x\ r^{\smile}\ z$ and $z\ r\ y$, then x equals y.

means *(translate inverse $r^{\smile}$)*
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.

This is as far as mechanical manipulation goes. But we can achieve more, because any concept and relation in an expression also has a definition, a meaning in natural language. The meaning is irrelevant for the mechanical manipulations above, but it is very relevant for the translation into natural language.

Suppose for now that our univalent relation r means that persons (instances of set A) can be reached at telephone numbers (instances of the set B). The translation of univalence of the relation r can now go on, as follows:

For every telephone number $x \in B$ and telephone number $y \in B$, if there exists a person $z \in A$ 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 one telephone number at the most.

The last few steps are different from the first series of steps. In the last few steps, we reformulated the abstract text and moulded it into a language fit for human use. It requires your understanding of the sentence. Although the translation starts out quite mechanical, 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 a number of steps. Most steps were demonstrated in the example above:

- Start out by naming the source and target elements of the composite expression(s), i.e. say 'for every source element x and target element y'.
- Translate each operator according to the translation table (see table 4.5.3), working outward-in and minding the precedences of operators.
- Repeat this until no composite relations remain.
- For each relation, check its multiplicity constraints to see if you can use it to simplify the text so far.

- Replace each relation by its definition or popular meaning.
- Reformulate the controlled-language text by making further simplifications, but be careful to keep the meaning correct and unambiguous.
- Rephrase into understandable business language.

This simple procedure is intended to help you translate any relation to its natural language meaning. By following it, you will obtain a proper translation for any rule. It helps you to explain the meaning of your rule assertions about what is 'out there' in the real world to the user community. But, by its very nature, explaining and understanding is a human activity, and can never be fully automated.

4.5.3 Translation table

Table 4.5.3 contains the necessary translation phrases for the operators introduced in the previous chapter. Also remember that $a \in A$ means that the term 'a' is an atom of concept A, and $a\ r\ b$ means that the tuple (a, b) is in relation $r_{[A,B]}$, that is: atom 'a' from A is related by relation r to the atom 'b' of B.

Name:	*Operator:*	*Translates to controlled language:*
identity	$\mathbb{I}$	equals. $x\ \mathbb{I}\ y$ means $x = y$.
inclusion	$r \vdash s$, or $r \subset s$	if $x\ r\ y$ then $x\ s\ y$. Or: $x\ r\ y$ implies $x\ s\ y$.
equality	$r = s$	if $x\ r\ y$ then $x\ s\ y$ and also if $x\ s\ y$ then $x\ r\ y$.
complement	$\overline{r}$, or -r	not $x\ r\ y$.
inverse	$r^{\smile}$	$y\ r\ x$.
union	$r \cup s$	either $x\ r\ y$ or $x\ s\ y$ (or both).
intersection	$r \cap s$	$x\ r\ y$ as well as $x\ s\ y$.
composition	$r; s$	there exists at least one atom z such that $x\ r\ z$ and $z\ s\ y$.
relative addition	$r \dagger s$, or $\overline{\overline{r}; \overline{s}}$	for all possible atoms z, either $x\ r\ z$ or $z\ s\ y$ (or both). Or: there exists no atom z such that z is unrelated to both x and y.
relative implication	$r \lceil s$, or $\overline{r} \dagger s$	if atom x is related to an atom z, then z is related to y. Or: for all possible atoms z, if $x\ r\ z$ then $z\ s\ y$. Or: any z that x relates to, does relate to y.
relative subsumption	$r \rfloor s$, or $r \dagger \overline{s}$	if atom y is being related to by the atom z, then x is relates to z. Or: for all possible atoms z, if $z\ s\ y$ then $x\ r\ z$. Or: any z that relates to y, is related to by x.

Table 4.1: Translations of operators to controlled language

4.5.4 Advanced translation example

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 (in fact, functions):

$$\begin{array}{lll} \mathit{concerns} & : \mathit{Workpackage} \to \mathit{Policy} \\ \mathit{followsProcedure} & : \mathit{Workpackage} \to \mathit{Product} \\ \mathit{belongsTo} & : \mathit{Policy} \to \mathit{ProductType} \end{array}$$

And assume that the following abstract assertion holds:

$$\mathit{concerns} \vdash \mathit{followsProcedure}; \mathit{belongsTo}^{\smile}$$

What does this rule mean in natural language? The mechanical translation yields an explanation:

> For every workpackage w and every policy numbered n, if instance w is related by *concerns* to instance n, then there exists a productType p such that instance w is related by $followsProcedure$ to instance p and instance n is related by $belongsTo$ to instance p.

Next, the actual meaning of each relation is needed to achieve a good translation in natural language. We need to understand the meaning of each relation in the context of the insurance company. The following interpretations are provided:

- w *concerns* n means that workpackage w concerns the policy number n. For instance, workpackage 'assessClaim3314' concerns policy number 'NL084728839',
- w $followsProcedure$ p means that workpackage w follows the procedure for productType p. For example, the workpackage 'assessClaim3314' follows the procedure for productType 'carInsurance',
- n $belongsTo$ p means that the policy numbered n belongs to the productType p. An example is policy number 'NL084728839' that belongs to the productType 'carInsurance'.

If we continue the translation procedure, we obtain the following interpretation:

> For every workpackage w and every policy numbered n, if instance w concerns a policy numbered n, then there exists a productType p such that the policy numbered n is of productType p and the work package w follows the procedures for the productType p.

This we can reformulate as:

> For every workpackage and every policy, the work package follows the procedure for the productType that the policy belongs to.

To summarize the steps of the procedure that we followed:

1. Translate the asserted implication $\vdash$ or equality $=$ into natural language.

2. Translate the relational operators ; and others, to natural language.
3. Translate other operators ($\breve{}$, $\bar{}$, $\cap$, etc).
4. Replace all formal relations by their natural language pragmatics.
5. Simplify the text by virtue of multiplicity constraints. Try to translate univalence, totality, injective etc into intelligible language.
6. Work on the english phrasing, to improve understandability for the intended audience.

Exercising through a fair number of translations carefully is the best way to gain experience and become proficient at reading formulas. You will find that it will help you to find and correct errors in formulas. The errors made by others, or even those made by yourself. At first, you will often refer back to the translation table 4.5.3 but soon, you will get the hang of it and can do without the translation table. And you will learn to translate intuitively, without explicitly taking all the steps of this procedure.

4.6 Sample rules and formalizations

Rules build on facts, but the rules may not always be so easy to find. As a designer, you are expected to assist business stakeholders in formulating their rules. Understandably, users will rarely express their rules as faultless Relation Algebra statements. It will take some prodding, and experience, to turn the business lingo into clear, *declarative rule* assertions. Let us consider a simple example.

declarative rule

4.6.1 Customers, Items, and Bills

A group of artists (painters, potters) sells their work to customers. Each piece being sold is a unique item, so we don't need to distinguish between the thing or the type-of-thing (item as-delivered to the customer versus item as-advertised in the catalogue). Figure 4.1 depicts the Conceptual Model. For now, we ignore

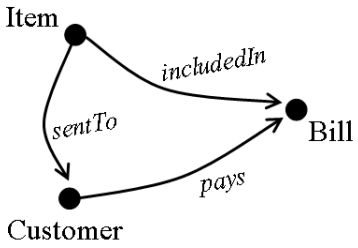

Figure 4.1: Diagram of the conceptual model with three relations

the obligation to provide proper definitions for the concepts and relations. An instance diagram (with some improbable tuples) is shown in figure 4.2.

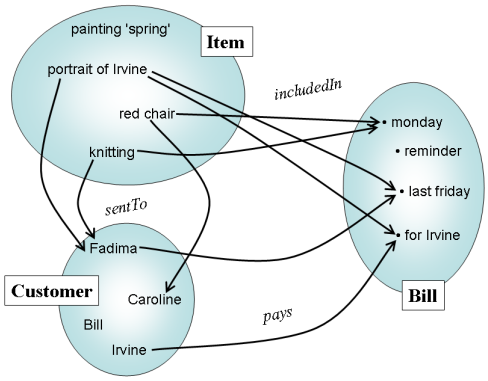

Figure 4.2: Instance Diagram for the relations

4.6.2 One relation

If we consider only one relation, we are looking at multiplicity constraints. The artists may for instance claim that 'every item has its customer, eventually'. This rule of thumb is probably not rigorously enforced, but we can still formulate it as a rule assertion, as follows:

> Natural language: every item has its customer.
>
> Controlled language: Every Item is *sentTo* some Customer.
>
> Rule assertion: $sentTo$ is total; or: $\mathbb{I}_{Item} \vdash sentTo \ ; \ sentTo^{\smile}$.

In the rule assertion, you will notice that the lefthand side ($\mathbb{I}_{Item}$) is merely the list of all items, now written in the form of a relation. The righthand side $sentTo \ ; \ sentTo^{\smile}$ is a listing of all items that are related, via *sentTo*, to some customer. Other multiplicity constraints are easy to think of, e.g.: an item may be sent to at most one customer. Or: a bill includes at least one item. We leave it as an exercise to decide for all three relations which ones of the four multiplicity constraints should hold.

Regarding relations, recall that structural rules are rules too. One of the structural rules embedded in this structure is that one customer pays one bill at most once. Referential integrity makes it impossible to capture a second payment of the same bill by the same customer in this model. If the stakeholders require the ability to capture duplicate payments, then the model has to be adjusted. One solution may be to distinguish between bills and payments; this is an example of *reification* which we will discuss later on.

Another structural rule implicit in the model as-is, is that it does not allow for items to be sent to galleries or exhibitions. if an item is sent anywhere, then it is to a customer by definition.

4.6.3 Two relations

Two relations may come together to make up business rules that are simple to understand. For example, we want every sent item to be billed:

> Natural language: every sent item must be billed.
>
> Controlled language: *if* an Item is *sentTo* some Customer, *then* that Item must be *includedIn* some Bill.
>
> Rule assertion: $sentTo\,;sentTo^{\smile} \vdash\ includedIn\ ;\ includedIn^{\smile}$.

In the rule assertion, the lefthand side is again the list of all items that are related, via *sentTo*, to some customer. The righthand side is similar, but relates to bills, not customers. The inclusion guarantees that each sent item must be included in some bill. A violation occurs if an item is sent that has not been included in a bill; this is probably best repaired by writing a bill for that item. Remark that the rule does not ensure that the bill is presented to the correct customer, nor that the bill will be paid by anyone; we will come to that shortly.

First, consider another simple rule involving two relations, that is the opposite of the previous one. :

> Natural language: every billed item must be sent.
>
> Controlled language: *if* an Item is *includedIn* some Bill, *then* that Item must be *sentTo* some Customer.
>
> Rule assertion: $includedIn\ ;\ includedIn^{\smile} \vdash\ sentTo\,;sentTo^{\smile}$.

Or consider a composite relation, e.g. $includedIn\ ;\ pays^{\smile}$. You can formulate a multiplicity constraint for this, just like for any other relation. Assume, in natural language, that every item shall be paid. The Conceptual Model imposes the structural rule that whatever is being paid is, by definition, a bill. The model cannot deal with payments for any other type of thing. So you need to rephrase the requirement and refer to bills:

> Natural language: every item is paid.
>
> Controlled language: every Item is *includedIn* a paid Bill
>
> Rule assertion: $(includedIn;pays^{\smile})$ is total;
>
> or: $\mathbb{I}_{Item} \vdash (includedIn;pays^{\smile});(includedIn;pays^{\smile})^{\smile}$.

A violation of this rule means that the item is not included in any bill, or the bill is not paid.

4.6.4 Three relations

We can write down more business rules variants if we consider all three relations. Of course, some of these assertions may be in conflict with another one, or they may not reflect how business is conducted. The purpose of these examples is

to understand how various rule assertions can be formulated. You must learn to translate and explain them to the stakeholders, so that they can decide on the ones that reflect their actual way of doing business.

Business rule variant 1:

> Natural language: a customer must pay if some item on the bill was sent to her.
>
> Controlled language: *if* an Item is *sentTo* a Customer, *then* that Item must be *includedIn* some Bill and that Customer should *pay* it.
>
> Rule assertion: $sentTo \vdash includedIn \ ; \ pays^{\smile}$.

Business rule variant 2:

> Natural language: a customer will pay if some item on the bill was sent to her.
>
> Controlled language: *if* a Customer *pays* a Bill, *then* at least one Item must have been *sentTo* that Customer and *includedIn* the Bill.
>
> Rule assertion: $pays \vdash sentTo^{\smile} \ ; \ includedIn$.

Notice the fine distinction here. Variant 2 may be rephrased as: *if* the customer pays the bill *then* at least one item is sent. But if we rephrase variant 1, it will switch those if- and then-clauses around: *if* at least one item is sent *then* the customer pays the bill. Or to put it more bluntly: every sent item must be paid for. This finesse of natural language clearly shows the need for precision and a structured, controlled-language approach!

Business rule variant 3:

> Natural language: paid items were sent.
>
> Controlled language: *if* a Customer *pays* some Bill and a certain Item is *includedIn* that Bill, *then* that Item was *sentTo* that Customer.
>
> Rule assertion: $pays \ ; includedIn^{\smile} \vdash sentTo^{\smile}$.

This variant has an effect opposite to variant 1. Whereas variant 1 says that sent items must be paid, this says that paid items must be sent. A customer who pays for an unsent item, causes a violation of this rule. This can be remedied by sending the item belatedly; or perhaps the money is returned.

Business rule variant 4:

> Natural language: a customer must pay if all the items on the bill were sent to her.
>
> Controlled language: *if* every Item *includedIn* the Bill is *sentTo* the Customer, *then* the Customer *pays* the Bill, or also: *if* every Item is either *sentTo* the Customer or *not includedIn* the Bill, *then* the Customer *pays* the Bill.
>
> Rule assertion: $sentTo^{\smile} \dagger \overline{includedIn} \vdash pays$.

Variants 5 and 6 are negations of variant 4:

> Natural language: a customer will not pay if some item on the bill was not sent to her.
>
> Controlled language: *if* there is some Item that is *not sentTo* the Customer while it is *includedIn* the Bill, *then* the Customer will *not pay* the Bill.
>
> Rule assertion: $\overline{sentTo}^{\smile}$; $includedIn \vdash \overline{pays}$.

Business rule variant 6:

> Natural language: if a customer does not pay then some item on the bill was not sent to her.
>
> Controlled language: *if* the Customer does *not pay* the Bill, *then* there is some Item that is *not sentTo* the Customer while it is *includedIn* the Bill.
>
> Rule assertion: $\overline{pays} \vdash \overline{sentTo}^{\smile}$; $includedIn$.

One of the rules or variants above may exactly cover the business requirements, or perhaps some rules should be combined. For instance, variants 5 and 6 together state that the customer shall not pay if and only if not every item was sent to her. But if we rephrase that, and put in another negation, then we have yet another likely rule variant: the customer shall pay a bill if and only if every item included in the bill was sent to her.

However: certainly not all rules and variants will apply to the business environment simultaneously. It is up to the designer to consult with stakeholders and find out exactly which rules do apply. Those rules may be similar to any one of the examples above, but they may also be very specific for the business situation at hand.

4.7 Other approaches

Relation Algebra is a powerful tool to use with business rules. It helps you to deliver designs that correspond to stakeholder responsibilities: rules must be kept true at all times.

4.7.1 Limitations in Relation Algebra

Relation Algebra focuses on the existence of things, on concepts and relations having correct extensions. The rule assertions then specify how to combine (extensions of) relations to detect any violations.

However, Relation Algebra has its drawbacks. For one: you cannot do arithmetics in Relation Algebra. It does not provide a formalism 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 hard to achieve good reasoning with adjacent geographical areas, distances, or time frames.

These drawbacks are inherent to the theory of Relation Algebra as it lacks arithmetical, spatial and temporal features. But there is also the problem that many business rules are not exact to the last decimal. And rules are often a bit vague on their consequences for the business. For instance, consider a statistical rule that says "at least 85% of all trains should leave on time". It leaves open what a 'train' (departure) is, it is unclear which departure will be the first to violate the rule, and finally: what to do if the rule is violated?

4.7.2 Conceptual modelling

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

Relational Model

There are subtle differences however. For one, the *Relational Model* refers to entities, attributes, whereas we only have concepts and relations 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 key*s that are composed from the attributes in an entity.

foreign key

4.7.3 Proposition and Predicate Logic

Proposition Logic and Predicate Logic, jointly called First-Order Logic, is particularly suited to learn about correct reasoning with facts. It relies heavily on "if/then" *conditional reasoning*: *if* the lefthand side of a sentence ("hypothesis" or "premisse") is true, *then* 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. But if you know that the righthand side is false, then of course the lefthand side must be false also.
Notice how this sentence is in fact an example of conditional reasoning.

conditional reasoning

Predicate logic adds the use of predicates and quantifiers to propositional logic, which allows to reason about combinations of facts. We introduced operators such as composition and relative addition for the purpose, which enables us to reason not only with individual facts, but with the entire contents of relations, i.e. abstractions of facts.

Relations and operations on relations can always be rewritten as expressions in predicate logic. In fact, you do this when you translate a rule assertion or expression into controlled language.

4.7.4 SBVR and RuleSpeak

The abbreviation *SBVR* stands for Semantics for Business Vocabulary and Rules. SBVR is a standard accepted by the Object Management Group, and it is available at http://www.omg.org/spec/SBVR/. *SBVR*

It comes with its own variant of controlled natural language to capture business rules, called RuleSpeak (see http://www.rulespeak.com). This is also an accepted standard to formulate terms and facts, plus the 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 undecideable sentences are allowed. RuleSpeak can be classified as a well-structured, controlled language: it does not allow arbitrary utterances, but only sentences that follow strict formatting rules. 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.

4.7.5 And more

The above list of approaches is far from exhaustive. However, this book is not the proper place to go into details of the similarities and differences between Relation Algebra, Set Theory, variants of description logic, Relational Modelling, SQL constraints and embedded database procedures, RuleSpeak, or any other theory or de-facto standards.

In our approach, Ampersand, the emphasis is on business rules and their formalization. We employ Relation Algebra as the theory of choice to focus on the existence of things, on concepts and relations having correct extensions. The rule assertions then specify how to combine (extensions of) relations to detect any violations. Relation Algebra is a powerful tool to use with business rules, helping you to deliver designs that correspond to stakeholder responsibilities: Yet it is not a simple tool, nor can it take away the real complexity of the business. You need to learn how to use it to achieve the best results. The next chapter will provide useful insights how to use the theory in order to create simple, yet expressive model for business rules.

Chapter 5

Design Considerations

Abstract

The previous chapter outlined how to specify a single business rule. But that is just the beginning. As a designer, you are expected to specify business processes by means of rules. So what must you pay attention to? How do you create a high-quality design? There are many design considerations, principles and rules of thumb that will help you to achieve useful results. This chapter outlines some of them. Their merits have been established in practice, so you may rest assured that they work.

A cookbook-recipe for good design does not exist. Neither does a standard way of working. Nevertheless, there are many principles, are many ideas that will guide and support your work. A good way to learn designing business processes is by doing: Practice makes perfect. The core idea of Ampersand is that a designer:

1. A designer first defines a context with:
   - a purpose (why does this context exist?)
   - named stakeholders (who participates and why?)
   - scope (which topics are relevant and which are not?)
2. A designer creates agreement among the stakeholders about language, as little as possible and as much as required to formulate requirements in that context.
3. Together with stakeholders, the designer elicits concrete rules that define the process(es) of this particular context.
4. Stakeholders "from the business" are addressed by the designer in natural language and their own jargon only.

The designer will formalize the "agreed language" by means of relation declarations, so (s)he will know which statements of stakeholders make sense (in

the agreed context) and which do not. Rules are formalized too, so the designer will know that all rules are consistent, concrete and complete. By doing so, (s)he will be able to tell apart the statements that are true (in the agreed context) and those that are not. With this knowledge, a designer can coach the stakeholders through the requirements elicitation process in their own language. That is: without bothering other stakeholders with formulas, design models, or "design speak" of any kind. Thus, Ampersand can be used in a conventional requirements engineering practice.

The remainder of this chapter focusses on the skill to create rules. In Ampersand, the natural language representation of a rule and the formal representation go hand in hand. It is the responsibility of a designer to make sure these match. To do so, designers get help from the notation and from tools.

5.1 Design example

To get a feel how to go about rule design, let us look at an example. 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.

This is a business rule in the context of an Immigration Service. Employees of that service are stakeholders. It is their job to keep this rule satisfied, so this rule is obiously one of the building blocks of the primary process of the Immigration Service. But the rule is formulated as a legal text in natural language. We want to express it in relation algebra, to be sure that it is formulated concretely enough. This necessary to decide whether this rule is satisfied or not in every conceivable situation. It also makes it possible for computers to work with. For every rule expressed correctly in relation algebra, the designer can decide whether concrete sentences uttered by stakeholders are true or false.

So we want to go about carefully: does our formalization represent the text correctly? We proceed as follows.

5.1.1 Look for concepts and relations

We start by looking for concepts. It is easy to spot three concepts in the text, namely:

- Residence permit for limited duration (abbreviated to RPLD)
- Condition
- Purpose-Of-Stay

Concepts are typically used in the singular form, rather than in plural. For example, Condition is used in singular, and not in plural as in the original text. This is because we need to capture and identify each condition separately. Also

notice that within this context, Stay is not considered to be a concept. This is because the text of the law does not mention anything about the actual stay.

So which relations can we define among these three concepts? The text of the article states that residence permits are issued under certain conditions. Apparently, this type of sentence in natural language is meaningful to the stakeholders, which justifies a relation declaration. So we conceive a relation *issuedUnder* with source RPLD and target Condition. If, for example, residence permit *nl45* was issued because the requestor of that permit has a contract to work for some employer, then the relation *issuedUnder* contains a tuple such as `⟨'nl45','hasLabourContract'⟩`.

The law says that there may be conditions related to the purpose of stay. This too is natural language over which stakeholders will agree. So we conceive a relation *relatedTo* from *Condition* to *PurposeOfStay*. A pair in this relation might be a tuple such as `⟨'hasLabourContract','employment'⟩`, or `⟨'isOver21','hasLabourContract'⟩`. And finally, residence permits and purposes of stay are related, a relation *with* exists between RPLD and Purpose-Of-Stay.

So apparently we have three relations:

- $issuedUnder : RPLD \times Condition$
- $relatedTo : Condition \times PurposeOfStay$
- $with : RPLD \times PurposeOfStay$

By writing down these three concepts and three relations, we have defined a conceptual model shown in figure 5.1. This model represents a small piece of the common language of the stakeholders in the context of immigration. Agreement among these stakeholders is likely, because this is the language that comes straight from the law.

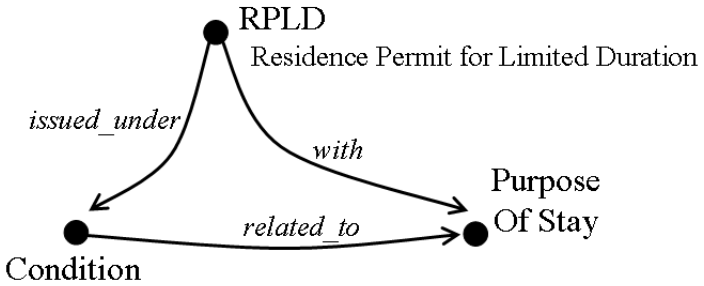

Figure 5.1: Conceptual model for the design example

The following section discusses how business rules are conceived.

5.1.2 Elicit business rules

The text of the law shows two interesting business rules. First, we understand that each residence permit is related to exactly one purpose of stay. There is at least one (total) and at most one (univalent) purpose of stay for each residence permit. So the relation *with* is must be univalent and total. This is a multiplicity constraint that we can insert into the model.

The wording of the law is not completely clear what is meant by "is issued under conditions that are related". What does it mean exactly? This is an issue to be clarified in conversation with relevant stakeholders. In the actual case, they came forward with other regulations, saying that each single residence permit must mention all the applicable conditions related to the purpose of stay. This can be rephrased as: if a purpose of stay p is recorded for a permit, and one or more conditions, say $\{cond-1, cond-2, cond-3\}$ are related to that purpose of stay, then the residence permit is issued under all those related conditions.

So if we consider one permit *nl45*, we know that whenever the permit *nl45* came with purpose p, and that purpose p is related to some condition *condN*, then the permit is issued under that condition. Restating this as an assertion in relation algebra:

$$with; relatedTo^{\smile} \vdash issuedUnder$$

which is now the second rule in our model for the Article 14 sub 2.

5.1.3 Validate the results

Having elicited a conceptual model and rules from the text, the question is: did we do it right? Have we captured all the relevant rules? We spotted two rules in the text, but have we captured them correctly? For this, we need to check whether the rules, written in relation algebra, match with what the people at the Immigration Service are trying to do.

Checking whether our formulas represent the original text, and detecting any possible mistakes in the line of reasoning, is called *validation*. Not the IT designers, but business stakeholders have to certify that all the business rules are captured correctly. *validation*

There are several methods for validation, to be discussed later in this chapter. For now, we claim that our small model with its two rules is adequate.

5.1.4 Practical implications

If stakeholders must keep rules satisfied, you must (as a requirements engineer) be aware of the implications your rules have in practice. Not only must a rule be described accurately, but we must establish which events may happen that violate a rule and which reactions restore that rule to satisfaction. It also involves making choices, because some reactions can be performed by a computer and others must be done by a person of flesh an blood. To put your model to work, we can either incorporate the rules in information systems,

or we can instruct business workers how to enforce the rules. In most cases however, we will do both: some of the rules can be enforced by automated systems, while other rules need to be maintained by people who can apply discretion and make intelligent decisions about rule violations.

Even though the example described just one or two rules, the stakeholders want to keep them satisfied under various events, e.g.

- when a new residence permit for limited duration is issued, or an existing one prolongued,
- when one particular purpose of stay is altered to become two different purposes, and residence permits for limited duration must refer either to one or the other, but not to both purposes,
- when a new condition is introduced that is related to a purposes of stay, so that some residence permits for limited duration may have been issued that do not yet mention the new condition,
- when an existing condition is dropped from the list of applicable conditions.

5.1.5 Design steps

Now let us revisit what we have done so far. Given the business rule as a natural language text, we have performed three steps in sequence:

1. Model design: which concepts and relations are involved?
2. Rule design: how to capture the rules from the business, as formal assertions in the model?
3. Validation: do people in the business confirm that the model and assertions do capture their requirements correctly?

If done correctly, you have established an implementable model. You will often find that these steps need to be iterated several times before the stakeholders are satisfied with your design. This does not signify that you are a bad designer. Rather, it is a token that the stakeholders come to a better understanding of the actual business requirements. It is their responsibility to formulate requirements.

The essential art of the designer is in understanding the requirements and formalizing the rules. That is: turn business language into exact formulas. You will need to learn how to figure out which concepts and relations are relevant, and how to incorporate them into your conceptual model. Also, you need to determine the right way to connect relations by operators, obtaining the assertions that represent the rules stakeholders care about.

5.2 Considering the Conceptual Model

First you determine which concepts and relations you need to include in the conceptual model. Remember that concepts and relations represent the kind

of things (terms) encountered in the real world and the facts that you want to convey about them. For each concept, its instances must correspond to actual, individual things that exist in the business environment, that can be added, altered, or removed from the context that we consider. For each relation, its tuples must correspond to the facts that are known about the concepts.

5.2.1 What are the concepts

In the example, we can visualize a 'Residence permit for limited duration' as a real piece of paper. This is clearly a concept. Also, 'Purpose of stay' is a concept. This concept however is not something physical or tangible. Luckily, there is an annex to the immigration law ("Vreemdelingenbesluit 2000") which provides an exhaustive list of all purposes of stay. At present, over a hundred different purposes are formally recognized, and you can easily imagine new ones being added to the list, or obsolete ones being removed. Likewise, the concept 'Condition' is also intangible, abstract. Again, you can imagine how legislators may introduce new conditions or discard obsolete ones.

These concepts share the property that individual instances can be pointed out, and they can be created, altered, or deleted. Indeed, the core question about a possible concept that a designer needs to answer is: what is the single instance?

This question actually has two parts. The first part is: how to recognize an instance of the concept "in the wild". Here we may see a residence permit, but there, that other thing, that is not a residence permit, but a building permit, or a parking permit, or an application for a residence permit. Try this for more abstract concepts like 'discussion', 'disagreement', 'idea', or even 'joy'. These are concepts, clearly, but without a proper definition, unambiguous and verifiable, you had better not include them in a Conception Model.

The second part is: how to identify and discriminate one individual instance from among its peers. For instance, one condition may be '*hasLabourContract*', and another is '*isUnder21*'. But how about a condition '*is under 14 with labour contract*' , for a child movie star? Is this a separate condition, or is it partially covered by other ones? Where does one condition end, where does the next one start? Again, try this identify-and-discriminate process for more abstract concepts. And if you wonder whether you would introduce 'joy' as a concept in your design, try to answer the question if you want to create new joys or discard obsolete ones.

5.2.2 Sound definition

The definition of a concept is used by the business stakeholders to decide when to insert a new instance or when to delete it if it no longer meets the definition. A sound definition of a concept must clearly address both aspects:

classification

- *classification*: how to know that some real world thing is an instance of the concept.

discrimination

- *discrimination*: how to distinguish and identify one instance from the other, not only today but also tomorrow.

Look back at the example in chapter 3, where we defined the concept *Customer* as:

- A person who or organisation that, in the past three years, has placed at least one order and has paid for it.

This definition meets the requirements of classification and discrimination. Still, it is not very sound. This is because the definition is not focused on the essence only, but adds extra demands. Those demands 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 definitions in a dictionary, for example a Customer is:

- one who uses or buys any of our products or services, or
- any person or organization who views, experiences, or uses the services of another person or organization.

essential meaning

This is better: a sound definition aims to capture the core meaning of the concept. A definition can mention some further quantification or qualification, but this is rare in practice. The basic, *essential meaning* of a concept (or relation!) often works best. Complicated additions are often a sign that something is not clear yet.

So as a rule of thumb: capture the essence in your definitions, and capture the criteria into rules.

The provisional definition of Customer could well be replaced by either of the dictionary definitions. Plus, we can add a new rule such as 'each customer must place and pay at least one order in the span of three years'. Perhaps you should consult with the stakeholders to ask if they want this new rule, and how they are going to maintain it.

5.2.3 Concept granularity

A further design consideration concerns the level of detail, also referred to as granularity, specificity, or (level of) generalisation.

You might argue that 'residence permit for limited duration' is a concept. Yet someone else might argue that 'residence permit' is the real concept, and you are only looking at one special kind of residence permits. For instance, there may be residence permits for unlimited duration. Others might argue that there are permits of different kinds, and residence permits are just one form of permit. Still others might say: "No, these things are all documents".

For another 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 details later.

All of these arguments are correct, and the rule of thumb is: choose your concepts as detailed, as specific and fine-grained as possible, while keeping the business context in mind.

Frequently you have to deal with both levels, coarse-grained and fine-grained. Animals, and birds and monkeys as well. Documents, but also permits for limited and for unlimited duration. Sometimes you can ignore the general concept altogether, and only model the distinct subconcepts. This solution is probably best if, within your business context, the concepts are essentially different, and any similarities are irrelevant. But more often you will find that you do need the general concept, but how may you include the detailed ones in your model?

There are several ways to do this:

- use a binary-property relation to identify special instances, or
- use a Type concept to hold classifying information for the instances of the general concept, or
- distinguish between the more detailed concepts by means of an *isA* subtype relation.

To discuss these options, consider figure 5.2 showing requests, with only numbers 3, 4 and 5 being accepted:

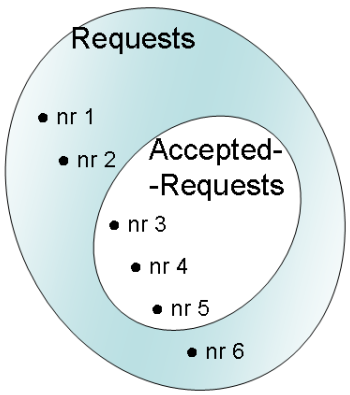

Figure 5.2: Instance diagram showing a concept with special instances

5.2.4 Binary-property relation or type concept

In chapter 3, we outlined how a binary property of instances may be captured by way of a homogeneous relation that is both symmetric and asymmetric:

- *isAccepted* : Request × Request

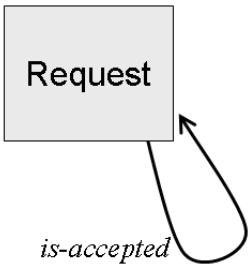

Figure 5.3: Binary property captured as homogeneous relation

This way of modelling is perfectly acceptable. And it is easy to understand, as the 'accepted' property is explicitly shown in the Conceptual Diagram.

Another way to capture the property in the conceptual model is by way of a type concept. For this, we introduce a new concept State with just two instances: State = { 'accepted' , 'rejected' }. And we define a new relation *hasState* : Request × State as follows:

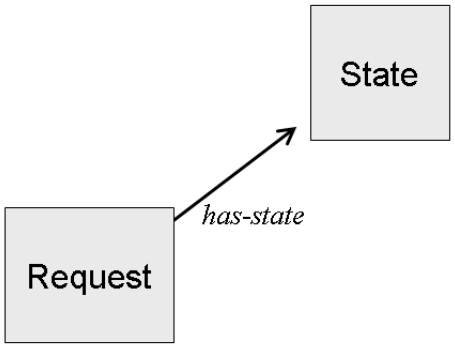

Figure 5.4: Binary property captured by a type concept

- for each request q, if (q, q) ∈ *isAccepted*, then put the tuple (q, 'accepted') in *hasState*
- for all requests q, if not (q, q) ∈ *isAccepted*, then put the tuple (q, 'rejected') in *hasState*

You should check that *hasState* is a function, and it is easy to see how the relation *isAccepted* can be replaced by *hasState* without loss of meaning. Notice that, *hasState* being univalent, each request is assigned at most one state: a request cannot be accepted and rejected at the same time.

This way of modeling is less explicit, because the reader must know that the *hasState* relation means either 'accepted' or 'rejected'. But the big advantage is that *hasState* is much more flexible than *isAccepted*.

We can easily extend it, for instance to cover a situation where acceptance or rejection hasn't been decided yet. We only need to extend the target set to become State = {'accepted', 'rejected', 'undecided'}. That is all: from now on, *hasState* can also contain tuples which are undecided. Likewise, we can add a 'partial accept', 'temporary reject' etc. Now try to alter the relation *isAccepted* in such a way that an extra option can be recorded! You will need to add one extra relation for each new instance in State. After several additions, you will have a lot of homogeneous relations, all alike.

This way of modelling, using a State or Type concept, is applicable in many situations. For instance, to model health status of the zoo animals, type of residence permit, eye color of customers, brand of car, etc. It is the customary way to model descriptive information for (the instances of) a concept. As a rule of thumb: whenever you encounter a 'list of things, all alike', you should probably capture them by way of a concept.

But remember, in the end, the business stakeholders decide on what they think is the best way to capture their requirements in the model.

5.2.5 Generalisation/specialisation

Granularity problems can also be dealt with by modelling the general concept, and the specialisations as well. Because each instance of the specialisation concept will automatically be an instance of the generalisation concept too, an *isA* relation comes quite naturally. It takes the specialisation as its source, the generalisation as its target concept. You should verify that the *isA* relation is always an injection. The disadvantage of generalisation/specialisation is that

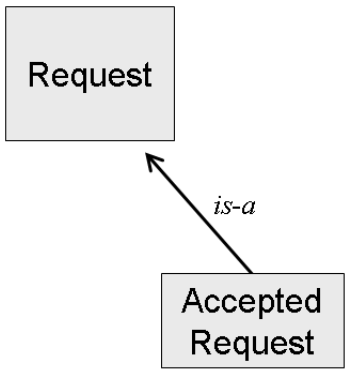

Figure 5.5: Binary property captured by way of *isA* subtype

the model contains more concepts (and relations), and therefore the Conceptual Diagram appears, at first glance, to be bigger and more complicated. However, specialisations are easy to explain.

A real advantage is that you can handle specialisations that partially overlap. Consider for example a shop where shop assistants serve customers. Normally, we would model two separate concepts, Assistant and Customer. But the two concepts may overlap: some instances of customer may be shop assistants, a shop assistant may act as a customer. To prevent fraud, we don't want

assistants to serve themselves. To this end, we first need to introduce the general concept of Person. We add two relations, Assistant *isA* Person, and Customer *isA* Person. The *isA* relations are injective functions by default.

Only then can we formulate the rule that a shop assistant and the customer being served, are not one and the same person. In formula:

$$isA_{[Assistant,Person]}{}^{\smile}; serve_{[Assistant,Customer]}; isA_{[Customer,Person]}$$

$$\neq \mathbb{I}_{Person}$$

As we have two distinct *isA* relations, the complete signatures were written for clarity. In practice, this is rarely necessary, as the signatures can often be inferred from the formula.

5.2.6 The thing, or the type of thing

A common mistake in model design is to overlook the distinction between things, and the type of things. A few examples may bring this problem home to you:

- A company sells chairs listed in a catalogue, and customers can enter their orders for chairs directly into the company mainframe. But what is 'a chair'? Is it the thing depicted in the catalogue? Or is it the individual items that are delivered to the customer to fulfill an order?
- "If you drive a car, you must possess a driver's licence for that car". As a natural-language expression, this is clear. But it is incorrect as an expression in relation algebra. Just check your driver's licence: there is no mention of a car.
- A client makes a reservation for a hotel room. Upon arrival, a room is allocated for the duration of the stay according to the reservation. Now what is the 'Hotel Room' concept?

As a designer, you should challenge your stakeholders to be clear about their concepts!

5.2.7 Redundancy

redundancy

It sometimes happens that some feature of the real world is modelled in more than one place. This is called *redundancy*. Redundancy can be recognized by the problems that it causes in your populations: you find yourself adding the same data into several concepts, or the same tuples into multiple relations. Or, one thing in the real world changes and you have to adjust data in more than one concept and relation. Soon, you will formulate some rule that demands the extensions of two concepts or relations to be equal. But perhaps it is better to simply delete one of them, as there will be no loss of information. With the exception, perhaps, of the *isA* specialisation of a concept.

Beware of derived concepts: a concept that is fully determined by other ones. For instance, a "complaints report" being defined as "the list of all current

complaints". The correct concept here is "complaint", and not "report". You should capture the derivation as a business rule, and then you may decide either to keep the derived concept, or to eliminate it from your model without loss of information. Be careful, as it may be worthwhile to retain the derived concept anyway. As an example: the total amount of a telephone bill is calculated from the telephone calls. The amount due must be on record, even after all call data have been deleted.

A nasty redundancy problem can arise if you happen to model the type of a thing in two different ways. At one place in your model you use Vehicle, with specialisations such as Lorry, Car or Bicycle. And you use a Type-Of-Vehicle concept somewhere else. This ambiguity can also occur with relations: *isChildOf*, *isSpouseOf* and *isNephewOf* at some places, but *typeOfFamilyRelationship* elsewhere.

The problem is in the abstraction level of the data. At one place, you capture the information at the level of the conceptual model: there is a Lorry concept in your diagram. But you also have a 'lorry' instance in your populations. Obviously, if you want to write a business rule about lorries, then you are in trouble. The problem should be repaired, but as it involves semantics and definitions, we cannot offer a simple solution that always works.

5.2.8 What are the relations

First of all: you must provide a sound definition for each of your relations. Business workers will need to know what tuples should be included in the relations or which ones must be deleted. They will look to the definition for guidance. Also, use one relation for one definition. Two different meanings require two different relations Be suspicious of any ".. or .." or ".. and .." clauses in your definition, as it may indicate that you are looking at a union or intersection of relations, instead of one single relation.

Although we have discussed concepts first, relations and concepts are conceived in no particular order. A concept without relations is useless, and so is a relation without concepts. Your concepts and relations exist for a purpose: To address stakeholders in their own language and to delimit the language in which stakeholder's concerns are expressed. Therefore, never hesitate to introduce a new relation or concept, but do so only for a good reason, that is: for use in a rule.

5.2.9 Concept or relation

Sometimes you find yourself wondering whether an aspect of the real world 'is' a relation or a concept. In fact: it is your decision. If you have a lot of information about it, then it is probably best to turn it into a concept and create relations to handle the information. If you need to handle very little information, then probably a relation will do.

In your model, you can always replace a relation with a concept. This is called *reification* or *objectification*. The idea is that the tuples in the old relation are upgraded to become instances of a new object (see figure 5.6). As every tuple

reification

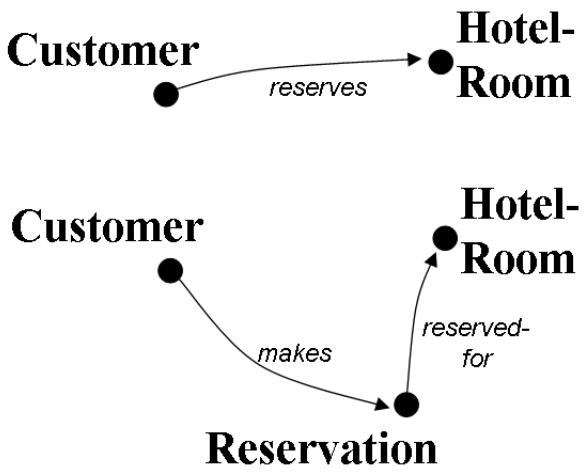

Figure 5.6: Reify a relation to become a concept

of the relation was linked to exactly one instance in the source concept, we can keep that link intact if we create a new relation from old source concept to our new concept. The univalence and total multiplicities are inherited by the new relation; and you can check that the new relation is injective and surjective automatically. Furthermore, a relation must be created with the old target concept, and its multiplicities must be determined.

It is possible to demote a concept into a relation, but this is much harder. For one, the candidate concept should be involved in no more than two relations. For another, those two relations must satisfy strict multiplicity demands before you can downgrade the concept and the two relations into a single relation.

Reification is easy, in a model. However, once a model is put into operation, reification is very hard to achieve as the impact on data population is extensive.

5.2.10 Dealing with time

In everyday speach, people talk about the current state of affairs. Peter might say, for example that he lives at Hopkins House. This fact may be true today, but tomorrow he might have moved elsewhere. When we say that a fact is true or a rule is satisfied, we always mean here (i.e. in the intended context) and now. Tomorrow there may be a different situation, if the underlying facts have changed.

A few examples may illustrate the point.

- A Student receives exactly one Grade for a Course. However, the student may fail to get a passing grade at the first or second attempt. You can model multiple relations (*hasGradeAtFirstTry*, *hasGradeAtSecondTry* etc). A more general solution would be to extend your model with an extra concept Number-of-Attempt and relate it with the concept Grade.

- A Customer has one Address. But the address may vary over time. So instead of one address, the customer may have many addresses, with only one being valid at any particular moment. You may capture this in your model by adding a new concept Valid-Since, and changing the Address concept into Address-Valid-Since. By the way: would you instead have named the new concept Sequential-Number, then the modelling solution looks very similar to the previous one.
- An Employee is allocated to exactly one Manager. This can be modeled by an univalent and total relation *hasManager*. However, the allocation may change over time. This can be modeled by reifying the relation to a new concept *hasManagerSince*. This is linked by three (not two!) relations: to the employee(s), to the managing person, and a relation to the time since when this fact, the particular *hasManagerSince* instance, is true. A solution involving a Sequential-Number concept could also works fine, except it might become rather confusing from the point of view of managers.

A common feature in the examples above is the rapid increase in the number of concepts. The conceptual model grows in size and complexity in accordance with the amount of time-dependent details that stakeholders require. As an exercise, try to construct a conceptual model that extends the *isMarriedTo* relation to correctly capture the full matrimonial history.

There is much more to say about time in conceptual models, but this book is not the place to go into details of temporal modelling.

5.3 Considering the business rules

Concepts and relations make up the conceptual model. But rules determine how users will use the model.

Once a conceptual model and its applicable business rules are agreed upon, you will find that the same rules are used in many different locations and business 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 rule.

Therefore, it is important to understand how rules work. Where in your model do rules come in, what rules should or should not be combined in a model, how can you make sure that you have all the applicable rules for a business context.

5.3.1 Distinct purposes

Earlier, we classified rules according to what they look like in a rule-based design: structural vs behavioural rules, and multiplicity rules as special behavioural rules.

Rules can also be classified according to their use and purpose in the business environment. A *primary rule* concerns the business services to deliver goods and information to customers. Stakeholders interested in maintaining these

primary rule

rules are the customers, and the workers engaged in delivering the customer services.

secondary rule

A *secondary rule* governs the way how to perform business activities and procedures correctly and efficiently. Workflow (i.e. the sequencing of business activities), authorizations, read and write permissions, priority rules etc fall into this category. Stakeholders interested in maintaining these rules are managers, auditors, IT support staff etc. To contrast: customers will not care whether these rules are maintained, they just want their demands answered!

tertiary rule

Some authors also distinguish *tertiary rule*s governing the way how IT people should go about their job of designing, implementing and maintaining information systems that support primary and secondary rules. We will not go into details, but there are many interesting issues involved in maintenance in general, and maintenance of rule based designs in particular. For example, rules may change. Then there may be rules that control the way how to deal with data violations that emerge as the rules are changed. Or rules that govern how incompatibilities between old and new rules should be dealt with.

In our opinion, a designer should aim for a strict separation between rules that serve distinct purposes. Try not to create a conceptual model that mixes primary rules and secondary rules (and tertiary ones). You will find that the respective stakeholders' views of the real world are quite incomparable, or even incompatible.

5.3.2 Rules dictating the business process

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 correct flow of steps, we need sequencing rules: if step 1 is finished, then execute step 2. Or: if steps 61 and 34 and 27 all have finished, then execute step 63.

The process sequence is partly dictated by business requirements ("delivery must take place only after payment is received") and partly by design choices. The process designer may enhance process efficiency by organizing the steps in a particular way, for example, she may follow an efficiency rule like "order picking is done from largest to smallest objects to ease packaging".

Design choices and efficiency rules are genuine business rules too. However, design and implementation choices should not be confused with original business requirements. The rule designer must be aware of such differences, as business requirements are much less volatile than design choices. Or to put it another way: a business process is described by a mix of primary, secondary, and even tertiary rules. Be aware of their different natures.

5.3.3 Exceptions to a rule are expressed by new rules

Exceptions to a rule, if you find one, are easy to capture in a model, once you come to realize that 'exception' means 'special situation'. So you just need to add specialisations to some of your concepts, and perhaps relations as well.

Next, you need to exclude the special instances from your old rule. And you need to write the new rule to deal with the exceptions.

A brief remark on theory. We expressed the need to classify and discriminate your concepts clearly. The creed about exceptions and new rules highlights the need to be just as meticulous for rules: you need to clearly demarcate each rule, to know where one rule "ends" and another one "begins". From this perspective, you, as a rule designer, should treat rules as the concepts in your own "real world".

5.3.4 Conflicting rules

Most business contexts are too big to be understood in its entirety. As a result, rules may be formulated by different stakeholders that are conflicting, meaning that you always end up with violations. Beware that you should *not* pick a solution on your own, but the business people should always have the last say.

By definition, two rules are *contradictory* if every attempt to populate all the concepts and relations used in the rules, will produce violations, no matter what you try. The only way to avoid violations is to leave some of the concepts or relations empty. Clearly, such contradictions should be detected by the designer and brought to the attention of the stakeholders. They must adjust their rules, or perhaps keep their rules intact but allow some exceptions. *contradictory*

A more general definition looks towards an entire set of rules. The set is called *consistent* if, for all concepts and relations, you can create populations (each counting more than one instance or tuple) in accordance with the definitions, and do so without any rule violation. This is why you must always for your conceptual model provide trial populations that violate none of the rules. Although this is not a conclusive proof of consistency, it does demonstrate that the set of rules is probably consistent. *consistent*

5.3.5 Enforcing the rules

Enforcement of rules concerns the way how a business puts the rules into practice: what measures are carried out to keep the rules true. In chapter 4, we argued how a rule and its enforcement are separate concerns. For structural rules, the issue of enforcement does not arise. Only behavioural rules can be violated, and active enforcement is needed. Basically, three strategies are available: immediate, deferred, or postponed enforcement.

Which enforcement strategy is most appropriate, depends both on business preferences and (in)capabilities of the implemented design. It can vary per rule: some rules require immediate action to prevent any violation at all. Other rules may be violated temporarily, where people treat a violation as a signal to take action. If thos signals are ignored, serious consequences may follow. The authors are aware of one example where such signals 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 exceptions. This resulted in a loss of several millions of euros.

Whenever you think about a rule, you also need to think about how the business people want to have it enforced. This is not a trivial matter: a wrong design choice may require major rework later on. Imagine the rework if you decide to capture a requirement as a structural rule, but stakeholders expected deferred enforcement so that they can look at violations before taking action. Or if you opt for deferred enforcement so that users can remedy violations at their ease, but users want immediate rejection because they do not want to waste time over potential violations.

5.3.6 Accumulation of violations

The implication of deferred- or postponed-enforcement is that some violating data is temporarily permitted, so that workers have enough time to understand the violation and resolve it at their ease. There is a risk however, as erroneous data is now present in the data store, and other transactions may take place that build upon the wrong data. If corrective action is postponed for too long, it becomes very difficult to correct the wrong data and undo all the wrong actions.

Some tools provide solutions to this, e.g. by locking all the data involved in a violation for subsequent transactions. However, this solution can quickly bring the entire datastore to a halt. A best possible approach that combines the ease of deferred enforcement with prevention of accumulating violations, is not available yet.

5.3.7 Rule enforcement in Ampersand

Enforcement is tool dependent. The actual behaviour that users will experience depends on the particulars of the tool at hand, i.e. how the compliance checking and enforcement is implemented, more than on the algebraic formulas of your rule. This means that when implementing a model, you need to know how your tools will implement the various enforcement options.

Ampersand tools support both immediate enforcement and deferred enforcement of rules. The immediate-enforcement strategy is default for:

- multiplicity constraints of relations, if the constraints are associated with the relation declaration, and
- properties of homogeneous relations, if the properties are associated with the relation declaration, and

maintain

- rules that are declared with the keyword *maintain*.

All other rules are implemented with the deferred-enforcement strategy. Ampersand uses the rule assertion to determine violations, that is: it calculates the contents of the left-hand and right-hand side expressions and then checks the difference between them. It creates reports of all violations that are signalled as deferred-enforcement. The lists are made available to the stakeholders who are committed to keeping the rules true at all times, so that they can make intelligent decisions on how to solve the reported violations.

By default, multiplicity constraints are implemented as immediate-enforcement. Sometimes this may be too restrictive, and luckily there is a work-around. Every multiplicity constraint can in fact be rephrased as a composite assertion, not associated with the declaration. How to do this was explained in chapter 4. So in fact, the designer does have a choice how to implement multiplicities.

5.4 Cycle chasing

Suppose your conceptual model is finished: you are satisfied that it contains all the relevant concepts and relations. But how do you check whether you have all the relevant rules? This section outlines a way to use the model for a search of rules.

5.4.1 Cycles in the diagram

A *cycle* in a Conceptual diagram is a closed loop, that connects a concept to itself via a chain of relations and intermediary concepts. Figure 5.7 depicts a cycle made up of five relations. Two different pathways connect the concepts

cycle

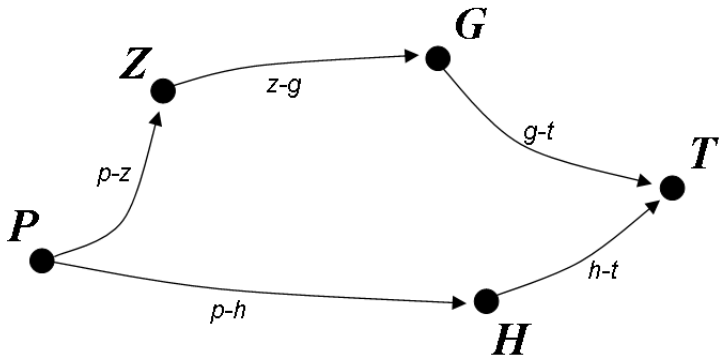

Figure 5.7: Cycle in the Conceptual Model

P en T. One passes by way of relations *p-h* and *h-t*, the other goes by way the chain *p-z*, *z-g* and *g-t*. Would we create two relations as follows:

$$R = (p - h); (h - t)$$

and

$$S = (p - z); (z - g); (g - t)$$

then the following assertion contains no errors:

$$R \vdash S$$

The assertion may be free of syntax errors, but is it meaningful? What is its semantics, does it represent an actual business rule or not? We might also have

conceived a totally different rule, navigating not from P to T but, for instance from Z to H:

$$R = ((z - g) \dagger \overline{(g - t)}\ ); (h - t)^{\smile}$$

and

$$S = (p - z)^{\smile}; (p - h)$$

Or we might use the identity relation on G for one relation, and go the entire circle for the other relation, like this:

$$R = Id_{[G,]}$$

and

$$S = (g - t); (h - t)^{\smile}\ ; ((p - h)^{\smile} \dagger (p - z)); (z - g)$$

Evidently, the number of assertions that we can conceive for this, or any cycle, is huge.

Even the simplest possible cycle, made up of only one homogeneous relation, can be constrained in a lot of ways. The relation may, or may not be, (ir)reflexive, (a)symmetric, and (in)transitive, thus giving rise to at least 8 potential business rules; and you can probably think up some more. Which ones capture an actual business rule, is for the designer to find out.

Summarizing, there are two points worth remembering. The first is that any rule involving composite relations corresponds to a cycle in the conceptual diagram. The second point turns this around: you can use cycles to help you discover rules. In any cycle in a conceptual diagram, you can conceive a number of assertions that might well be a business rule. The latter insight is the basis for the design activity of *cycle chasing*.

cycle chasing

5.4.2 Cycle chasing

Whenever you see a cycle in the Conceptual Diagram, you should be aware that a rule may be involved. The rule would control consistency of information in the cycle: going down one pathway from source to target, is somehow correlated to going down the other way. It is the designers responsibility to examine each and every cycle, and elicit the potential business rule (or rules). Rules that sometimes the business users never thought of.

It must be emphasized that in cycle chasing, you should examine all cycles. And you may expect to find rules to control all cycles. It is conceivable, but rare in practice, to find a cycle without any corresponding business rule, the implication being that anything goes, there is absolutely no coherence in the relations involved.

The correspondence between rules and cycles is not one-to-one. A multiplicity constraint will control a single relation, without so much as a cycle in the diagram. Really complex rules may control several cycles at once. Or some cycles may be controlled by more than one rule. Although the correspondence between cycles and rules is not exact, you should always check: is every cycle controlled by rules? Is there a difference between the cycle count and the number of rules, and if so, why?

5.4.3 Counting cycles

How many cycles are there in a complex Conceptual diagram? This is determined by the *cyclomatic number*, and you calculate it as: *cyclomatic number*

$$1 - \textit{number of concepts} + \textit{number of relations} \tag{5.1}$$

The simplest possible Conceptual Diagram has just one concept: the number of cycles is 0. The extra unit of 1 prevents that we would calculate a negative number for this. Now, if you add one new concept, and use one relation to connect it with the remainder of the model, then the number of cycles does not increase. But you do create an extra cycle when you add a relation connecting two concepts already present.

In a small Conceptual diagram, the cycles are easy to count: they correspond to the "loops" or "eyes" in the diagram. However, for more complex diagrams, with lots of crossing lines, it is much harder to count the loops. The formula makes life easier.

Beware that the rules controlling the loops may be more intricate than what the visible eyes may suggest. Figure 5.8, with 6 concepts and 9 relations illustrates this. The cyclomatic number is $1 - 6 + 9 = 4$, and indeed there are four "eyes", spanned by the relations a-b-c, c-e-f, d-e-h and f-g-i (ignoring inverses, for now).

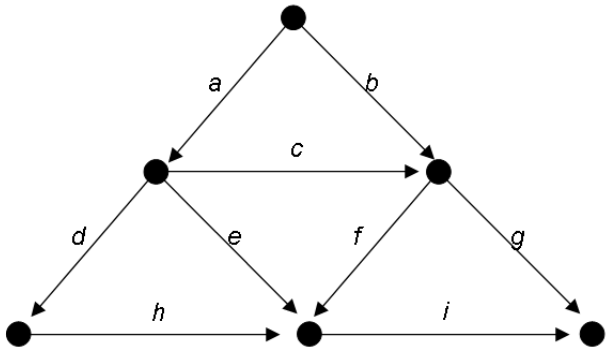

Figure 5.8: Relations to be constrained by rules

But controlling rules may for instance traverse the cycles spanned by a-d-h-i-g-b, by a-d-h-f-b, by a-e-i-g-b, and c-d-h-f. And these four rules may even be mashed together into a single rule by using the $\cap$ operator (logical and). Notice however that every relation is included in, and hence controlled by, at least one rule. So there is a correspondence between the business rules and the cyclomatic number. You should use this as a rule of thumb, as a design guideline, when checking your set of rules for completeness.

5.4.4 Cycle length

Try to formulate concise rules that are clear and easy to understand. A rule assertion that is too long and complicated, is probably too hard to understand

and may contain errors without your noticing it. And you can almost be sure that you will not be able to explain the rule to your business stakeholders.

Most rules control cycles that involve a small number of relations, up to 5 or 6 at most. Anything higher must be regarded with suspicion. A rule involving many relations may be perfectly right, but more likely it is not well understood. You can try to simplify such a rule by looking at its cycle (try to draw it as a loop), and simplify the rule using a shortcut across the loop. Perhaps you need a new relation for this shortcut, but as said before: never hesitate to introduce a new relation. Simplification and clarification of rules is a very good reason to do so.

If you add a new relation, then the cyclomatic number goes up by one. Thus, while trying to simplify one old rule, you may end up with two simpler ones. Together, they should cover the meaning of the old rule.

5.4.5 Redundant rules

What if a designer comes up with two rules such as $r \vdash \overline{s}$ and $s \vdash \overline{r}$?

These two assertions clearly express the same business rule. The rule can also be written in yet another form: $r \cap s = \emptyset$. Obviously, there is some redundancy here. In this example we can easily check that the rules are equivalent because we can invoke a law of relation algebra about set negation (see Chapter 3). But it is not always so easy.

less strict behaviour

redundant rule

The way out is to look at the behaviour of rules, i.e. at the violations they produce. We say that an assertion R imposes *less strict behaviour* than assertion S if any violation of R always corresponds to a violation of S. In other words: R is more permissive, S causes more violations than R. When one assertion imposes more strict behaviour than the other, then we say that the latter, weaker one is a *redundant rule*. Notice that we do *not* require that sources and targets of the assertions are identical.

The definition above concerns only two rules. If we consider a set of many rules, then a more complex definition of rule redundancy is required, because several rules may together impose the more strict behaviour. However, we cannot explore this extensive subject within the context of this book.

If a rule is redundant, then it is safe to omit it from your conceptual model, because any violations will be caught by the stricter rule (or set of rules). But do consult your stakeholders about it, as they may still desire violations of the weaker rule to be reported separately.

5.4.6 Check for redundant rules

If two rules traverse and control the same cycle, then this may indicate a redundancy. But not always: rules that traverse and control the same cycle may also augment one another. This is so if neither one is more strict than the other. For instance, think of a homogeneous relation: it may well be transitive and symmetric at the same time. You may consider combining the two into a single rule, which calls for some rewriting of your rule expressions. But always

keep in mind that it is the business stakeholders, not you who have the final say on what the rules are.

If two rules traverse different cycles, then you can safely assume they are not redundant. This remains true, even if the two cycles partly overlap (some shared relations and concepts) and even if the rules are written using expressions that have the same sources and targets! In this case, the rules augment one another. Again, you might consider combining them into one rule which will then control more than one cycle.

A final remark on theory. We defined redundancy of two rules as "imposes less strict behaviour than". It is an interesting exercise to check that this constitutes a homogeneous relation among rules. As the relation is reflexive, asymmetric and transitive, it defines a partial order on the set of rules.

5.4.7 Guidelines for work

Business rules are guidelines for work. Indeed, a sound set of rules may be read by users as if it outlines their workflow. The motivating example in chapter 2 demonstrated this core idea. It showed how business rules may drive a work process. An initial customer action will start off a violation of one rule. The action to repair that violation triggers violations of subsequent rules which must be remedied. After a number of steps, no violations persist and the process is completed.

If you are able to present the rules to your stakeholders in such a natural way that it outlines a business workflow process, then do so. You should be able to demonstrate how the work "flows". Determine which event causes an initial rule violation, and then trace the successive user actions to remedy the violations and make the rules "true" again. In fact, this constitutes a *use case* and it provides you with an excellent way to validate your rule design with the users.

use case

Business rules in general ought to be *declarative*, that is to say: they should not specify any strict sequencing of behaviour. Nor should they specify how the rule should be enforced. The rules signal violations, but do not dictate what to do to make them true.

declarative

For instance, a violation of the rule "a residence permit for limited duration is issued under conditions that are related to the purpose of stay" can be remedied in several ways: by adding a new condition to the permit, or by revoking the permit. Or you can even remove the rule itself, and no longer require the condition and purpose of stay to be related. In general, the more concepts and relations that are involved in a rule, the more options a user can choose from to act and remedy the rule violations.

Indeed, an inclusion rule expressed as a relation algebra assertion can be inspected to see which relations it involves. To remedy a violation, it suffices to manipulate those relations so that one or more tuples disappear from one side in the assertion, or that tuples materialize on the other side. The most desirable or appropriate actions however cannot be decided from the rule alone.

A business-rules approach focuses on rules and violations, which differs markedly from the business process approach. Rules that do impose specific sequences

imperative

of actions are called *imperative*. This kind of rules is abundant in workflow models, where they dictate the how, when, and by whom activities must be undertaken. The imperative rules generally ignore the *why* of action, which is: to ensure compliance with agreements, to remedy the violations of business rules. 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. It remains unclear what to do with an exam from a student who does not provide a valid number.

Whereas workflow is a prescriptive way to organize work, declarative business rules merely outline which facts should be kept true and why. The good thing about declarative business rules is that they are guidelines for work, but do not enforce one specific choice of action upon the user. This gives users maximal flexibility to manage their workload, provided that they keep to the rules. It does not constrain the way of work how it should be done, in what sequence, when, or by whom.

5.5 Validation

What makes a good set of rules? This is a core concern in any design effort, no matter if you are engaged in information systems design, in home decoration, in dance choreography or any other creative effort. A set of rules is good if the stakeholders agree that this is the set they must satisfy (in the given context). So you can do your best, but in the end it is up to the user audience to decide on the quality. For this reason, validation of rules is needed to make sure you get a sign-off by stakeholders.

5.5.1 Before you start

Even before you start validating, as a designer you should perform a number of preliminary checks. Of course, each separate rule should be up to standards; as was explained in the previous chapter. But this is not enough. You should also perform some checking on the combined set of rules. If any of these checks are failed, then the design is not really finished yet and validating it has little value.

- Complete: are all of the user requirements covered?
- Correct: are all of the user requirements expressed correctly in the formulas and definitions? And do all of the formulas and definitions express a requirement correctly?
- Consistent: are the rules not in conflict with one another?
- Syntactically sound: are all formulas and definitions expressed in the correct syntax of relation algebra?

If all of the above is ok, then the design might be finished, and you can start your validation activities.

5.5.2 By trial and error

A first step in validating your design is by examining it by way of populations. By populating all concepts and relations, you can attempt to produce violations for the various rules. If the violations match exactly with the user expectations, then probably you have a valid model.

In practice, you will often use this method. In fact, populations are key to understanding your models and rules. You only understand a rule if you know what violations it will produce in a population. If you, or your business stakeholders do not understand the violations, you do not understand the rule.

A more sophisticated approach to trial-and-error is by having test cases generated and run automatically. At present, software exists that will automatically create populations for all of your relations (this is called *seeding*). It will then proceed to determine whether the rules are violated by the content. Such software gives you a fast, but not 100% reliable way of validating your rules.

seeding

Still another approach to trial-and-error is by searching for a counterexample. You invent fictitious but meaningful data, in such a way that your facts are correct, but your rule turns up a violation. Thus, the rule is wrong and should be replaced. Every time you falsify a rule with a counterexample, you have learnt something about it, and therefore increased your understanding of the situation. So, falsification is a valuable, albeit timeconsuming approach to validation.

5.5.3 By translating back

The rules in your design are expressed in a formal notation, meaning they are unambiguous and can be rigorously applied. However, rule expressions are not always easy to comprehend, not even for experienced designers. Hence, you should provide each rule with a *translation* in natural language. We outlined a translation procedure for that very purpose in chapter 4. Remember that the final step of that procedure requires your own good judgment, both as a speaker of natural language and as expert with extensive knowledge of the business at hand. And do not forget to check that your translation is meaningful and useful in the business context.

translation

5.5.4 By corroboration

Corroboration means that others confirm what you are saying. In other words, you want to confer with knowledgeable business representatives, and get them to agree with your rule assertions and explanations. And remember that you want the business intent of your set of rules to be approved as a whole, and not merely the translation that you have provided for a separate rule.

Corroboration may also require that you prepare use cases: demonstrations that show how in a populated conceptual model, an initiating event will produce a sequence of violations to be dealt with (automatically or by users) until there are no more violations and the initiating event is dealt with.

5.5.5 By comparison

Validation can also be done by comparing with something else. The idea is to prove that (a part of) your rule(s) is equivalent to something else, preferably something that is known to be true and correct. For instance, you can try to populate your design and compare the ensuing violations with the output of an existing information system that covers the same business context.

A more advanced way to do this is by reasoning with the rules and find new implications. Intelligent tools exist that can analyze rules and infer new implications and sentences from them. Then you might discover that some particular implication of your set of rules is unacceptable within the context of your business. Reasoning back, you might discover which rule is at the basis of this faulty implication, and then you can change or discard that rule. However, details of this advanced method are beyond the scope of this book.

5.5.6 Stakeholders responsibility

Above, we described how to start design validation. But in the end, validation is the responsibility of the business stakeholders, not yours: they must certify the quality of the design they asked for. Users should validate that the design that is presented to them is

- understandable: they should check that they understand all documentation, explanations and use-cases offered for their benefit,
- complete: they should check that all of their requirements are expressed in the design,
- valid: they should ascertain that the design is a good representation of the business context.

Nevertheless: you can never be 100% sure. Even if stakeholders fully agree, even if you are totally confident, the claims about the validity of a design are based upon only a certain amount of evidence.

5.6 Rule Based Design according to Ampersand

Ampersand

The way of working to deliver a Rule Based Design as described here, is called *Ampersand*. Ampersand guides you to produce a design that meets the business requirements, and that has sound design quality. Quality, of course, depending on expertise and experience of the designer, but perhaps even more so on the capability of business stakeholders to express their requirements, to agree upon the rules that must be kept true within their business context, and to validate their own ways of dealing with rules and violations.

5.6.1 Ingredients of the deliverable

Ingredients of a Rule Based Design according to Ampersand are:

- a list of all requirements as put forward by the users. This calls for *requirements elicitation*, an activity that we do not cover in this book. It may take place before, or concurrent with, the initial stages of design,
- a conceptual model that captures all the terms and facts referred to in the requirements, as covered in chapter 3,
- a complete and correct set of rules, asserted in relation algebra, that comprises all business rules listed by the users in natural language, as covered in chapter 4,
- trial populations for all concepts and relations,
- validation evidence to bear witness that the concepts, relations and rules are in full agreement to the requirements,
- technical specifications for use in subsequent IT development activities. This activity was not explained, as the specifications are produced automatically by Ampersand tools.

requirements elicitation

5.6.2 Checking a Rule Based Design

A Rule Based Design is comprised of a conceptual model plus a set of rules. Basic checks on the model include:

- name: each concept or relation has a meaningful and unambiguous name,
- definition: does the definition of each concept and relation cover a single meaning only? Is there no ambiguity, does each concept and relation capture only one real-world feature?
- classification: does the definition enable the users to clearly distinguish an instance (of the concept) or tuple (of the relation) from whatever is not an instance or tuple?
- discrimination: does the definition allow the users to clearly distinguish one instance/tuple from any other instance of the same concept or relation?
- identification: does the definition allow to recognize the same instance/tuple at any future time?

A conceptual model should also be free of duplicate, redundant, and derived concepts. That is, unless the designer has some sound arguments to keep them in the model, and has provided a full set of rules to control their contents.

Basic checks for rules defined on the conceptual model are:

- each relation has its multiplicity constraints stated? Did you consider writing the constraints as distinct business rules, separate from the declaration?
- every relation is controlled by some appropriate rule?
- all concepts and relations can be populated in such a way that no violations emerge?

- each rule assertion is rephrased in natural language, as simple as possible, so that stakeholders can grasp their meaning and corroborate that the rules are correct?
- every rule violation is understandable and can be remedied by appropriate user actions?

Furthermore, rules in a conceptual model should:

- be consistent, i.e. no contradictions, as evidenced by being populated,
- outline a logical flow of work, evidenced by use cases that show how violations are dealt with by the rules,
- be independent of implementation, i.e. there are no hidden assumptions about rule enforcement, or sequence of execution.

Rule Based Design is about creating a conceptual model together with a set of associated business rules, in such a way that high quality standards are met. Of course, the final check is with the stakeholders. It is up to the designer to deliver a model and a set of rules to capture their business requirements. It is the responsibility of the business to validate the model and the business rules, and to put the design to work.

Part III

Practice

Chapter 6

Document Management

Abstract

To illustrate the use of Ampersand, this chapter discusses document management. This theme has been chosen for its relevance to practice, as companies and IT projects have to cope with large numbers of documents in various versions. The focus of this 'written demonstration' is on incremental design. That means: by adding one rule at a time, introducing new relations as required, a rule-based design can grow in its coverage of the business, while the functional specification derived from it, remains fully consistent. Incremental design methods bring a distinctive advantage to large IT projects, because it enables such projects to decompose the problem area into manageable themes. The project can then proceed in small steps by modeling and merging each theme successively.

6.1 Introduction

Ampersand has been developed to support engineers in using business rules as a basis for the design of information systems and business processes in practical situations. It aims at a better analysis of requirements, thereby providing more control over large IT projects, resulting in higher success rates for such projects.

DocPAD

This chapter introduces a design pattern called *DocPAD*, which stands for "Documentmanagement Pattern for Architecture and Design". The subpatterns described here contain rules that apply in many different circumstances, and you may use this chapter as a reference model in your own projects. Beware however, if you wish to do so, you still need to validate the patterns and rules in your own business context. There is no universal truth, no golden bullet or perfect design. And every business context comes with its own requirements, there is always something more for you to attend to. The work in this chapter demonstrates some characteristics of Ampersand that help to control large IT projects:

composition of patterns Let different stakeholders discuss their own requirements, collect the results from each group of stakeholders, and assemble the design by merging the partial results into a unified rule-based design.

incremental design Add one or a few rules at the same time and validate the entire design with the new rule in it.

rules with an impact across different patterns A rule that affect multiple themes can be handled in the same way as rules restricted to a single theme only.

keep the amount of rules small Make each rule correspond to one business requirement, and keep the total number of rules to a bare minimum.

reuse Requirements, as captured by rules, can easily be reused from previous projects.

Composition is demonstrated by presenting a different theme in separate sections of this chapter. Each theme can be considered the exclusive domain of other stakeholders. The themes discussed in this chapter are: overall definition of document organization, access to documents, permissions for access, document versions, and temporary validity of permissions For document organization, imagine that documentalists and end users are involved to decide upon the organization of documents by way of folders and software applications. For access to documents, security officers may step in to define the requirements for access and permissions. For document versioning, think about the team leaders and librarians who want to impose their requirements about version management. In practice, there may be more discussions going on, but for the purpose of this chapter, we will discuss only these. Each of the following sections will discuss a single theme, in isolation of the other themes. Section 6.8 discusses their composition, and it shows the effect of having all themes merged into one grand rule-based design. Just for a first impression, figure 6.1 shows the conceptual diagram of all our patterns, combined.

Incremental design is demonstrated by discussing each rule separately, without any forward references. Rules contributing to document management are introduced per section. Each rule is first motivated rule from a business perspective. Then we introduce the new *business vocabulary*, i.e. we define the concepts and relations that we need. And last, we formalize the rule. For example, a discussion of permissions, what data objects may be used, is separated from the discussion about the temporary validity of the permissions

business vocabulary

Being able to handle rules one-by-one, combined with the absence of forward references, is a necessary condition for incremental design. It means that stakeholders can propose a new rule, an architect can formalize it and make sure it complies with the rules obtained so far, and add it to the body of rules. Each time a rule is added, the entire specification can be validated.

Ideally, the number of rules matches the number of functional requirements. For most information systems and business processes in practice, this number runs typically between 50 and 500. The document management pattern, discussed in this chapter, has about 20 rules. By keeping the number of requirements small, the overall design and the functional specification can be kept small too. This has a favourable impact on the resulting information system(s), that tend to grow increasingly complex as the number of functional requirements

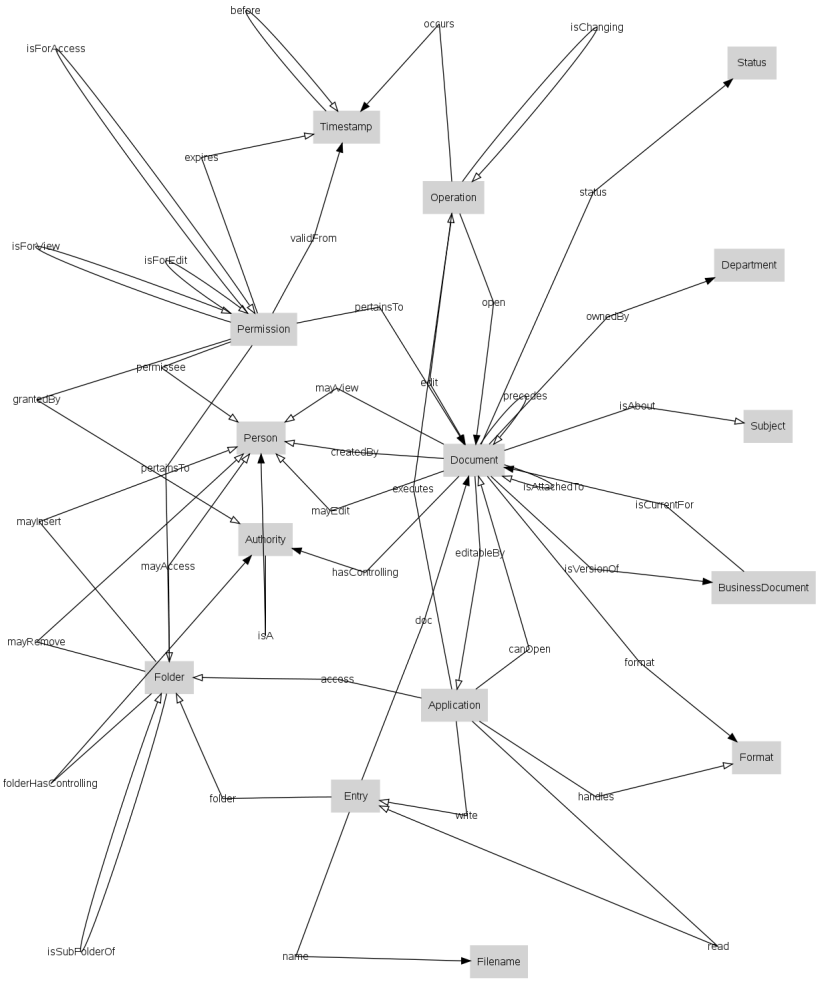

Figure 6.1: Conceptual diagram of all patterns combined

goes up. Architects who are responsible for formalizing functional requirements will be inclined to scrutinize every wish from 'the business' and minimize the number of rules in the end. The strong match between business requirements and rules is a feature of Ampersand that helps to curb the complexity of large IT projects.

Reuse may be demonstrated if you pick up some rules from this chapter in a project of your own. The patterns and rules presented here can be useful for instance for selecting 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 systems, the rules in this chapter can then aid as functional requirements, which you can choose to pick up or leave alone, as you think fit.

6.2 Document Management

This section introduces the basic concepts document, folder and puts them in context. The next sections will extend this basic pattern to define such aspects as the applications to handle documents, access permissions, and document versioning.

- A *document* is what carries (a fixed set of) information across place and/or time. The information can range from highly structured set of data (e.g. an XML document) to unstructured such as a newspaper copy, an e-mail, a taped conversation, even a telephone conversation, or a legally binding contract with numerous attachments. *document*
- A *folder* is where documents are kept under an appropriate filename. Technically, a folder may be distributed over various locations, or duplicated for safety reasons, but for our purposes, we consider folders to be atomic, i.e. indivisible. *folder*
- A *person* is understood to be someone under business jurisdiction, either an employee or a contracted worker. *person*

Figure 6.2 shows the conceptual diagram of our basic pattern.

Documents are owned by departments, and documents are about certain topics. Of course, the subject of a document may be some employee or department of the organization itself, be for now, we ignore this peculiarity. As we consider digital documents, we know that, by virtue of computer storage technology, all documents reside in digital folders. However, one can imagine a document residing in multiple folders under multiple file names. We address this issue by using reification (see chapter 5) which yields a new concept, to be called "entry". Clearly, each document corresponds to one entry at least: a document without entries, is inexistent.

Designers should notice how the unique identification of (atoms of) the "entry" concept is founded on a combination of relations. Still, the pattern does not provide an intelligible naming scheme for the documents as such. Also notice that four business rules are proposed, thus exceeding the number of cycles in the diagram.

Unique filename per folder Folder entries can have their own filenames, but still refer to the same document. By virtue of entity integrity (see chapter 3), every entry is unique. But even more: we know that filenames per folder must be unique. The formalization in 6.3 uses two relations:

$$name \;:\; Entry \to Filename \tag{6.1}$$
$$folder \;:\; Entry \times Folder \tag{6.2}$$

This rule can now be expressed as:

$$name; name^{\smile} \cap folder; folder^{\smile} \;\vdash\; \mathbb{I}_{[Entry]} \tag{6.3}$$

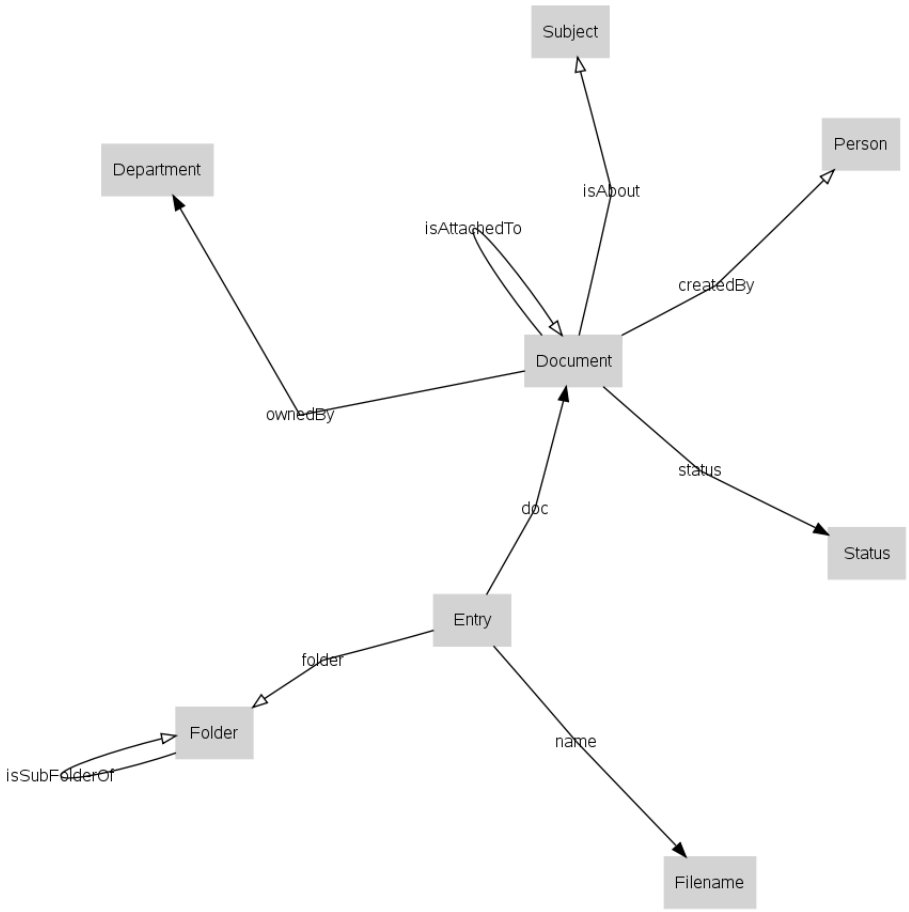

Figure 6.2: Conceptual diagram of the basic pattern

Folder hierarchy Folders can have subfolders, and the folders are organized in a hierarchical fashion. In order to formalize this, we introduce a relation among folders:

$$isSubFolderOf \quad : \quad Folder \times Folder \tag{6.4}$$

The expression F *isSubFolderOf* G means that folder F is a subfolder of the enveloping folder G. As was explained in chapter 3, this hierarchic organization of folders is achieved by a subfolder relation that is reflexive, asymmetric, and transitive.

If a document, or more precise an entry, is in a subfolder, then by implication it resides in the enveloping folder too. As a consequence, an entry 'resides' in many folders simultaneously. This folder hierarchy is regulated via the relation $folder$ (see 6.2):

$$folder \,;\, isSubFolderOf \vdash folder \tag{6.5}$$

Notice that some computer operating systems use other folder organizations (e.g. Microsoft Windows).

People create documents Documents are created by people. We want the person(s) who created a document to be traceable in the company. 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 captured by:

$$createdBy \quad : \quad Document \times Person \tag{6.6}$$

Now, we would like *createdBy* to be a total relation: every document is present in at least one tuple of this relation, meaning that at least one person is recorded as a creator. But we do not want an immediate-enforcement strategy imposed. That would imply that a person who is sole creator of some documents, cannot leave the enterprise; or at least, her data cannot be deleted from the database. Instead, we like any anonymous document, not traceable to its creator, to be signalled. The signal is produced for documents that appear as tuples in the relation:

$$\mathbb{I}_{[Document]} \cap \overline{(createdBy; createdBy^{\smile})} \tag{6.7}$$

Attachments A document can have other documents attached to it. Of course, no staples are used to attach anything to digital documents. Instead, a binary relation is set up to refer to attached documents. So, the sentence: "Document E is attached to document D" is meaningful. And a document F may be attached to E, G to F, and so forth, creating a chain of attachments.

In the real world, using paper and staples, it is physically impossible to have a document attached to itself But in the binary relation, an explicit rule is needed to prevent a document from being attached to itself.

This relation and restriction are captured by:

$$isAttachedTo \quad : \quad Document \times Document \tag{6.8}$$

plus the demand that if one document, G is attached to another, say D, then D may not be attached to the first one, G. Which is to say that the relation must be asymmetric:

$$isAttchedTo \ \cap \ isAttachedTo^{\smile} = \emptyset \tag{6.9}$$

6.3 Software Applications

Documents are created, edited, and viewed by software applications, but which applications can handle which documents? Generally speaking, applications can deal with certain document formats, and handle all documents that comply with the format. But over time, applications may fall into disuse, become obsolete, or even stop working altogether, due to software or hardware innovation. Good document management means that every document can be accessed by at least one application, even if the original editor application is no longer available. Furthermore, applications require access to the folders where the document (entries) reside. This section introduces several relations and rules that help to capture and record this.

– An *application* is software that provides document handling services to the user. Examples: 'MS Word 2007', 'Adobe Acrobat 2.0', or 'Realplayer 2010'. *application*

format

– A *format* describes how the document's data is technically stored by an editing application, in such a way that the original document can later be reproduced from the data. Examples: '.doc for MS Word 2000 version 9.0', '.pdf PS-Adobe-2.0', or 'neato version 1.7.6'.

Figure 6.3 is a conceptual diagram of this pattern. The rules in this pattern are

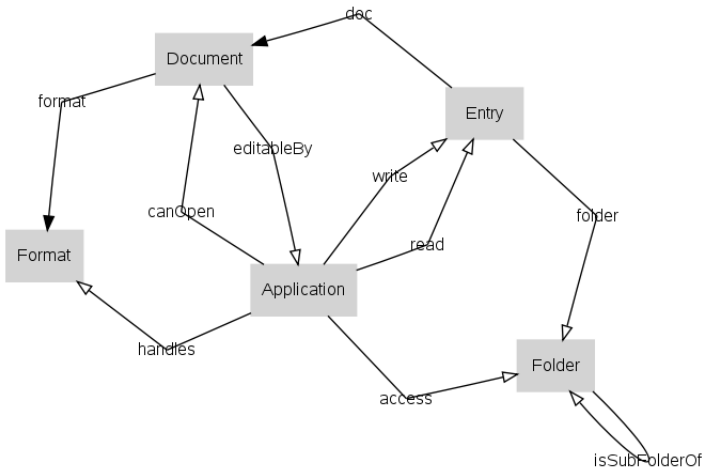

Figure 6.3: Conceptual diagram of applications to handle documents

rather straightforward. Interestingly, the homogeneous relation *isSubFolderOf*, introduced in the previous pattern, is seen to have an impact in this pattern too.

Open documents based on format For applications to open a document (i.e. read, print, scan, etc.), its format must be known. When an application can handle the format, and the document data is stored in that format, then the application can open that document and view its content. Which is not to say that the document is editable by the application. So, the sentence: "Application A can open the document D" is meaningful (i.e. it is either true or false) for any application A and document D. The formalization (equation 6.13) requires relations:

$$format \;:\; Document \to Format \tag{6.10}$$

$$handles \;:\; Application \times Format \tag{6.11}$$

$$canOpen \;:\; Application \times Document \tag{6.12}$$

Every application can handle at least one format. A company should also make sure that at least one application is available for any format in use, to prevent unreadable documents. So every relevant format should be handled by at least one application, i.e. the *handles* relation must be surjective. Thus, we can formalize the rule that any document can be opened by virtue of its format.

$$handles;\ format^{\smile} \;\vdash\; canOpen \tag{6.13}$$

Open any document In fact, the rule above can be weakened. The requirement is only that every document can always be opened, i.e., regardless of its format. To secure the contents of documents, we require that there is always at least one application that can open it. Phrased like this, the requirement comes down to:

$$\mathbb{I}_{Document} \vdash canOpen^{\smile}; canOpen \tag{6.14}$$

Editing implies opening To edit a digital document, an application must be used. Sometimes, several applications may be used for editing. If a document can be edited by an application, then obviously the application can open it. So if we have the relation:

$$editableBy \quad : \quad Document \to Application \tag{6.15}$$

then we have as a rule:

$$editableBy \vdash canOpen^{\smile} \tag{6.16}$$

Access to folders To work properly, applications must access the folders where document entries are located. So, the sentence: "Application A accesses folder F" is meaningful (i.e. it is either true or false) for any application A and folder F. The meaning of this sentence is that the application has the capability to retrieve data for any entry stored in that folder. This is formalized as:

$$access \quad : \quad Application \times Folder \tag{6.17}$$

For any folder, at least one application should exist that has access to it. If not, the folder is inexistent for all practical use. Therefore, we require that the *access* relation is surjective. Who has access to which documents and folders is controlled by way of user permissions, which is the subject of the next sections.

Access to folder hierarchy If an application has access to a folder, then by implication, it has access to the subfolders too. The relation *isSubFolderOf* is used to express this rule in a formal manner:

$$access; isSubFolderOf^{\smile} \vdash access \tag{6.18}$$

Accessing an entry implies accessing the folder If an application accesses an entry, i.e. it reads or writes the entry, then that document entry is situated in some folder that is accessible to the application. Two relations are required to capture this rule:

$$read \quad : \quad Application \times Entry \tag{6.19}$$
$$write \quad : \quad Application \times Entry \tag{6.20}$$

The rule is simply:

$$read \cup write \vdash access; folder^{\smile} \tag{6.21}$$

If writing does imply reading, then the rule can be simplified by omitting the *write* relation from the lefthand side of the rule.

A complication arises if the folder is located on removable media. Then we only know that the folder was accessible at the time of the transaction, but perhaps not before or after. For now, we prefer to ignore this complicating aspect of time.

Reading implies ability to open If an application reads an entry, then we know that the application could open the document. And if an application writes an entry, then the corresponding document is editable by the document. These claims, however, ignore that software utilities exist, such as Microsoft Explorer, that can handle datafiles, copying or deleting them, without being able to read or write the document contents. The two rules are captured as follows

$$read \vdash canOpen; doc^{\smile} \tag{6.22}$$
$$write \vdash editableBy^{\smile}; doc^{\smile} \tag{6.23}$$

6.4 Permissions

Who has access to which documents and folders? We need to ensure that users may gain access to the right documents in all circumstances. In document management systems, people can access folders and documents only if they have a permission to do so. Permissions enable us to implement access rights to documents in many different ways in various circumstances. This section on permissions introduces rules and relations to help organize this. To keep things simple, the granting of permissions, and the time restrictions on permissions, is relegated to subsequent sections.

permission

– A *permission* is a business agreement to allow or prohibit something. More exact, a permission grants the privilege to use particular documents or folders in a specific manner (e.g., viewing only), to one or several person(s), called the permissee, A permission without permissee is meaningless, and without a known permissee, the permission itself ceases to exist.

Figure 6.4 shows a conceptual diagram of this pattern.

For designers, the repetitive nature of the various "may do" relations is remarkable. Moreover, they should remark on the close connection between the "may do" relations and the homogeneous "is for" relations. This will be explained as we go along.

Permissions regarding documents A permission for viewing, editing etc., is granted to individuals, whom we call the permissee. A permission pertains to at least one, but possibly multiple documents. Indeed, a viewing permission without a document to view, is meaningless, and so is an editing permission without a document to edit. For one document, any number (0,1,more) of permissions can be granted. So we have four relations:

$$permissee : Permission \times Person \tag{6.24}$$
$$pertainsTo : Permission \times Document \tag{6.25}$$
$$mayView : Document \times Person \tag{6.26}$$
$$mayEdit : Document \times Person \tag{6.27}$$

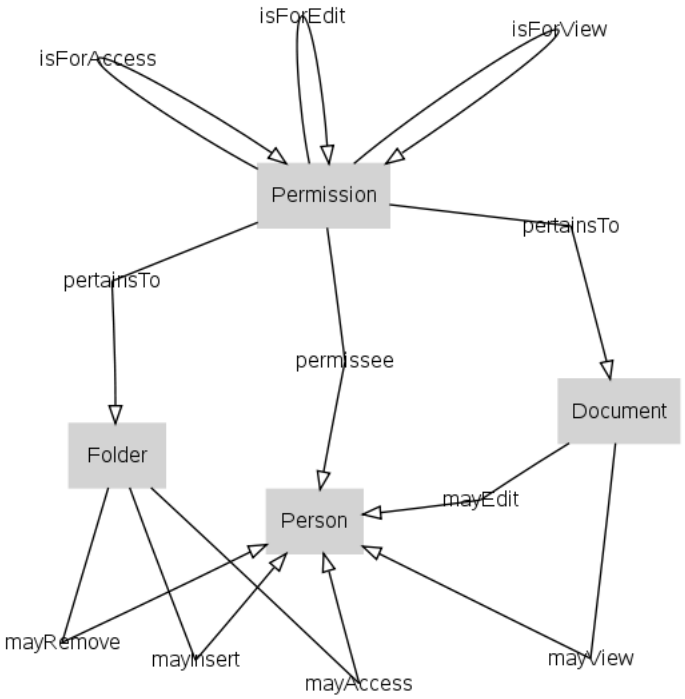

Figure 6.4: Conceptual diagram of Permissions

The rule is that a person may view (or edit) a document if and only if she has the appropriate permission for viewing (or editing). And likewise, edit is permitted only if the permissee has the proper edit permission. So something more is needed: is the permission for viewing, or for editing? This is captured by homogeneous relations on permissions:

$$isForView \quad : \quad Permission \times Permission \tag{6.28}$$
$$isForEdit \quad : \quad Permission \times Permission \tag{6.29}$$

Both are property relations: symmetric and asymmetric at the same time. The permission rules for viewing and editing documents can now be formalized:

$$mayView \quad = \quad pertainsTo^{\smile}; isForView; permissee \tag{6.30}$$
$$mayEdit \quad = \quad pertainsTo^{\smile}; isForEdit; permissee \tag{6.31}$$

Edit, create and viewing permissions We may assume that an edit permission automatically implies permission for viewing:

$$isForEdit \quad \vdash \quad isForView \tag{6.32}$$
$$mayEdit \quad \vdash \quad mayView \tag{6.33}$$

The rules here demonstrate the nature of the connection between the "may do" relations and the homogeneous "is for" relations. It creates redundancy in the rules: a single business rule forces us to write two rule assertions. In a later stage, the rule designer may attempt to eliminate

the redundancy, an activity that may call for a number of changes in the conceptual model.

Can we also assume that the person creating a document has edit permissions? Probable, but perhaps not. An assistant may be asked to create the annual corporate balance sheet. But will she have permission to edit it thereafter? Let us assume that yes, permission is automatic:

$$createdBy \vdash mayEdit \tag{6.34}$$

Permissions regarding folders A similar line of reasoning applies to permissions for folders. We also need the *permissee*, and the *pertainsTo* relations. For access, removal, and insertion of entries in folders, we need three distinct property relations (to avoid clutter, the diagram only shows one). And we will have three rules.

One last detail requires attention. The *pertainsTo* relation for permissions on documents, was defined as total. But we have expanded permissions to also cover folder permissions. The implication is that some permissions may pertain to folders, and not to a document: that relation is no longer total. Still, every permission pertains to something, either a document or folder:

$$\mathbb{I} \vdash pertainsTo_{[..,Document]}; pertainsTo^{\smile} \cup pertainsTo_{[..,Folder]}; pertainsTo^{\smile} \tag{6.35}$$

And likewise, every permission is for some kind of operation, i.e. every permission participates in at least one of the many "is for" homogeneous relations:

$$\mathbb{I}_{Permission} \vdash isForEdit \cup isForView \cup isForAccess\,(... \cup isFor...etc.) \tag{6.36}$$

6.5 Permission Authority

Documents and folders are under control of some authority.

authority

- An *authority* is a person or group of persons that, in an enterprise, is responsible to supervise and control the proper access to all documents and folders This is achieved by granting, and perhaps withdrawing, permissions to the company workers.

The conceptual diagram for authorities is shown in figure 6.5.

singleton concept

Multiple controlling authorities may be involved, for instance in multinational corporations. In small enterprises though, a single authority will suffice. In the latter case, it is a *singleton concept* in the diagram, having only one atom. It is quite common for designers to omit such a singleton concept from the model, but that may cause problems, as the quantity may change later on.

Permissions not granted to self A permission is granted by exactly one authority. The people acting as authority are not allowed to grant permissions to themselves. To formalize this, we use the *isA* relation for persons acting as authority.

$$grantedBy \quad : \quad Permission \rightarrow Authority \tag{6.37}$$

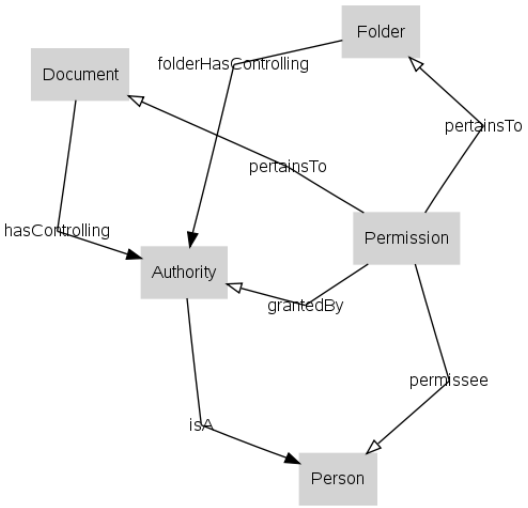

Figure 6.5: Conceptual diagram of Permission Authority

The corresponding rule reads

$$grantedBy; isA \cap permissee = \emptyset \tag{6.38}$$

Controlling authority An authority should only grant permissions for what is under its control. Every document and folder should be under the control of exactly one authority, so the relations expressing the facts about control, *hasControlling*, are functions.

$$hasControlling \quad : \quad Document \rightarrow Authority \tag{6.39}$$
$$folderHasControlling \quad : \quad Folder \rightarrow Authority \tag{6.40}$$

The rule is that if an authority issues a permission, and the permission pertains to some document (or folder), then the issuing authority is the one who has control over that document (or folder):

$$grantedBy^{\smile}; pertainsTo \quad \vdash \quad hasControlling^{\smile} \tag{6.41}$$
$$grantedBy^{\smile}; pertainsTo \quad \vdash \quad folderHasControlling^{\smile} \tag{6.42}$$

We remark here that the Ampersand tool will try to fit the relation types to suit the rule. That is, it will automatically pick the correct *grantedBy* and *pertainsTo* relations at the lefthand sides, to match the type of the relation at the righthand side of the rule assertions.

6.6 Versioning

From a business point of view, documents are not static objects. Rather, a business document evolves through a number of versions. The document management system is expected to keep track of versions from its creation, as it changes, until it is deleted. The actual opening, edit, or deletion of documents

is of course subject to the permissions, but that will be discussed in the next section.

Figure 6.6 shows a conceptual diagram of this pattern.

business document

– A *business document* is a set of information that is not fixed, but that may evolve over time to suit the business purpose.

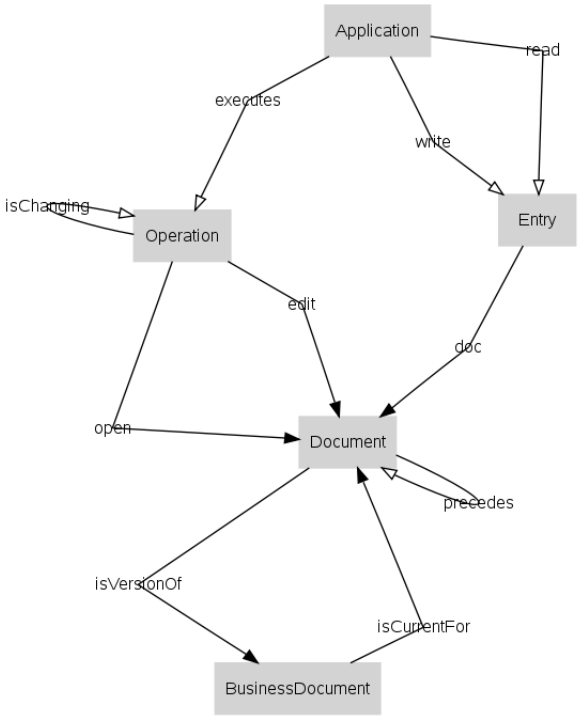

Figure 6.6: Conceptual diagram of business documents and its versions

The design of this pattern is interesting, because it addresses some dynamic issues, and it does so without an explicit notion of time. The idea is to match the (dynamic) business document to its successive (static) versions. And to realize that in fact, we already know about versions: it is just another name for document!

Document as version of a business document Earlier, we pointed out that a proper naming scheme for documents is lacking. The reason is that meaningful names for documents derive from the business. But the business rarely names the individual versions of the document, rather, one name is used to indicate all versions. Therefore, we now understand the concept 'document' to be the fixed version of an evolving business document. Over time, as the business document evolves, as it is being created, altered and used, consecutive versions of the business document may exist. Two relations express this:

$$isVersionOf \quad : \quad Document \to BusinessDocument \tag{6.43}$$

$$precedes \quad : \quad Document \times Document \tag{6.44}$$

The *precedes* relation records how documents, being versions of a business document, follow upon one another. This relation is defined as an ordering relation (asymmetric and transitive). Users may assign version numbers to documents, constituting a naming scheme, but the *precedes* relation does not require or enforce it. We do require however that a document and its predecessor both belong to the same business document:

$$precedes; isVersionOf \vdash isVersionOf \tag{6.45}$$

One version is current A business document has exactly one current version.

$$isCurrentFor \quad : \quad BusinessDocument \to Document \tag{6.46}$$

The rule is that the current version is indeed a version of the business document. Moreover, the current one is the latest one: there is no newer version. This is expressed by the rule

$$isCurrentFor^{\smile} \vdash isVersionOf \backslash (precedes; isVersionOf) \tag{6.47}$$

Operations on documents The business document evolves if an operation opens the document, and after editing, the operation returns a document with changed contents. Whether these operations are allowed is controlled by permissions, but that is dealt with by another pattern, to be discussed in the next section.

$$open \quad : \quad Operation \to Document \tag{6.48}$$

$$edit \quad : \quad Operation \times Document \tag{6.49}$$

The two operations 'open' and 'edit' engage the document as such, and not just an isolated entry. Opening a document is defined as a function: every operation opens exactly one document. To contrast, edit is univalent, but not total. Many operations will just view or reproduce the document, but do not result in a changed version.

Changing operation The operation on a document is said to change it if the originally opened version is unequal to the edited version afterwards. Of course, both belong to the same business document by virtue of the *precedes* relation.

$$isChanging = open; (isVersionOf; isVersionOf^{\smile} \backslash \mathbb{I}_{Document}); edit^{\smile} \tag{6.50}$$

Applications execute operations Most operations on documents require support by some software application. Not always, as a user may read a printed copy of the balance sheet without support. However, if an application is used to open a document, then in fact that application will read some entry of the document, and show that to the user. If the document is edited, then some application must have been used to write a first entry for the document.

A new relation is required:

$$executes \quad : \quad Application \to Operation \tag{6.51}$$

The rule for opening operations should apply only if an application is used. It reads:

$$executes; open = read; doc \tag{6.52}$$

The rule for writing is slightly different. It assumes that an application must always be used for editing (no editing on paper!). But it does not assume that every write is always a changing operation. This is because a document can be copied by writing extra entries for it, without changing the document content. So the rule is that editing a document implies (but not equals) writing of an entry:

$$write \vdash executes^{\smile}; writedoc \tag{6.53}$$

6.7 Permissions are temporary

So far, permissions were discussed as capabilities only, and we did not model the actual work being done on documents and folders. The reason is the temporal aspect: the operation on a document occurs at a certain moment in time, and right at that time, the appropriate permission must be valid to allow it.

Operations, and permissions for them, involve time, and lifespans. Referential integrity is a guarantee that the lifespan of a permission is restricted by both the lifespan of the corresponding documents, and the permissees. But authorities may grant and withdraw permissions, thus reducing the permission lifespans. And the system must ensure that operations may only proceed if there is a valid permission for it.

Figure 6.7 shows a conceptual diagram of this pattern.

operation
- An *operation* represents the (successful completion of) an actual piece of work done on a document or folder, at a certain moment in time, and by a certain person.

timestamp
- A *timestamp* is a moment in time when a single event occurs, a single piece of work is being completed. Do not confuse this notion with timespan, which has a beginning (timestamp), a duration, and an ending (timestamp).

This pattern is interesting for designers because it shows one way to deal with temporal issues. To keep things simple, we only consider documents, opening and editing operations on them, and permissions controlling those operations. The same approach however can deal with other types of operations, and with permissions for folders and other data objects. By incorporating time in the model, dynamic issues can be addressed in a more general way than was shown in the previous section.

Earlier, we indicated that relation algebra is not well suited to deal with time. The reason is that time has a continuous, linear ordering. This is hard to capture by way of binary tables with a finite, and often small number of instances, to which the current Ampersand tool is restricted. Moreover, it cannot reason with timestamps and durations, e.g. "March 29 + 4 days = April 2" Such limitations may be acceptable during design and test phases of a project, but never in a full production environment.

Time as a relation For now, we capture the flow of time using an antisymmetric, transitive relation *before*. Which is a bit clumsy: we want to use

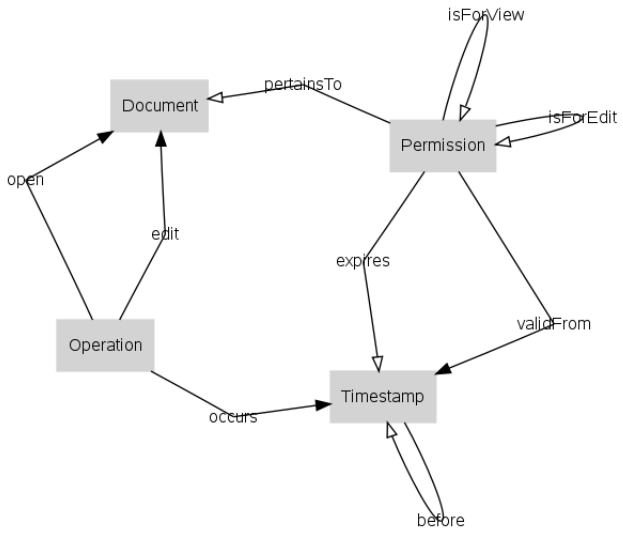

Figure 6.7: Conceptual diagram of temporary permissions

timestamps, but we do not want to provide a population of timestamps!

$$before \quad : \quad Timestamp \times Timestamp \tag{6.54}$$

Valid lifespan of permissions Each permission is granted at one time, and may expire after a while. A permission may, or may not expire, but if it does, then it is after it was granted. Two temporal relations are involved to express this rule.

$$validFrom \quad : \quad Permission \rightarrow Timestamp \tag{6.55}$$
$$expires \quad : \quad Permission \rightarrow Timestamp \tag{6.56}$$

and the rule is simply

$$expires \vdash validFrom; before \tag{6.57}$$

Valid timestamp of permissions The valid lifespan of permissions is determined by the two relations above. Together, they determine if a certain permission is valid at time t. We can formulate a relation to capture this:

$$validAt \quad : \quad Permission \rightarrow Timestamp \tag{6.58}$$

But this relation is redundant: its population is fully determined by the $validFrom$ and $expires$ relations. For that reason, we decided not to depict it in the conceptual diagram of this pattern. The determination is easy: a permission is valid at time t if it has been valid from a timestamp that lies before t, and it has not expired before time t. Which is to say:

$$validAt = validFrom; before \cap \overline{(expires; before)} \tag{6.59}$$

Operations on documents require valid permission Operations open and edit documents, and permissions control the legitimacy of the operations. The open and edit operations here, engage the document as such, and not the isolated entry.

$$occur \quad : \quad Operation \rightarrow Timestamp \tag{6.60}$$

The rule is that an operation may take place only if the appropriate permission is valid at the time. In the case of an edit operation, we specificially want it to be a permission for the Edit type of operation. So, the rule for an edit operation to be permitted is:

$$edit \vdash occurs; validAt^{\smile}; isForEdit; pertainsTo \tag{6.61}$$

A similar rule can be written for permissions controlling the operations that open a document.

6.8 Composing the patterns

The previous sections in this chapter have introduced a large number of rules, together with the vocabulary (concepts and relations) to express these rules.

The combined rules from the previous sections translate to a conceptual diagram shown in figure 6.1. All diagrams depicted in this chapter, and the engineers functional specifications (not shown) derive from a single Ampersand script that describes all the concepts, relations and rules. The script, containing less than 400 lines of code including explanations, is available on the website accompanying this course book.

If you want to use patterns of this DocPAD example, feel free to do so. Of course, you need to adjust and extend it to suit your business purposes. Some directions may be:

Generalize Concepts like documents and folders can be generalized into something called "data object". While you are at it, you may generalize relations such as "may view" and "may edit" too. Or, you may wish to define a Type-of-Operation concept to replace the "is For operation ..." relations, as explained in chapter 5. Another candidate for generalization is Operation itself, which can be broadened to *event*. Events may capture anything with a definite timestamp. Not only the work on data objects, but also the granting and withdrawal of permissions, or even the times when employees enter or leave the corporate work force.

event

Beware, however, that *generalisation* is something that designers like to do. Business stakeholders do not care for the abstract concepts that result, they need the easy names for terms and facts to which they are accustomed.

generalisation

Extend You may wish to capture topics closely related to document management, such as hardware devices and media. Applications may require specific device types, such as printers, scanners, or telephone, for a call centre application. A medium is the physical object that stores a datafile (document entry) which may be either machine- or human-readable, e.g.

paper, an old-fashioned floppy disk, or USB stick. Device types must match the media storing the datafiles. A document management system must keep track of the media that store folders and datafiles of documents, and must know which applications, device types, and actual devices are available.

Refine New relations can be added to provide more detail, and improved, and more refined rules may be formulated. For instance, a rule might be that legal contracts cannot be drawn up using a mere HTML format. Or financial documents must be secured by password permissions that change every 2 weeks.

This raises the question: when are we done inventing rules and scrutinizing the available ones? In practice, there is always room for new rules, and there will always be discussion. A design is never finished.

Instead, you must pick a moment to stop discussions and freeze the deliverable. That point can be reached for mundane reasons, such as a deadline imposed by management or customer contract.

The Ampersand method comes with a tool, also called Ampsersand, that allows to generate a functional specification for an arbitrary ruleset. As a consequence, you can have a partial specification in printable format at any moment during the requirements elicitation process. When you decide that the requirements have been scrutinized and discussed in sufficient detail, you can use the last functional specification as the final document. For database design, workflow design, user interface design etc, the software engineers can obtain other output from the tool that meet their design needs. The Ampersand tools produces all the required information, but limited to platform independent characteristics.

6.9 Conclusion

This chapter has introduced a partial specification of a document management system. The specification consists only of concepts, relations and rules.

composition of patterns Various patterns were discussed in the sections of this chapter. In practice, each pattern, each theme will be discussed by many people, all of who represent different concerns in the business. The composition of their requirements must yield a single, coherent design.

incremental design The presentation of rules in this chapter shows that rules can be discussed one at a time. The tools allow rules to be added one at a time to an existing rule base. The analysis supported by the tools can validate the entire design with the new rule in it.

rules with an impact across different patterns Several rules had an impact across different themes simultaneously, for instance the rules about an application writing a document entry, which calls for an appropriate edit permission for the document involved. Like any other rule, it was added incrementally to the rule base. This demonstrates that rules with wider impact require no other treatment than rules restricted to a single pattern.

keep the amount of rules small The close correspondence between rules and business requirements keeps the total number of rules to a bare minimum, as shown in this chapter. In practice, keeping the number down, is achieved by keeping requirements and rules in sync. Throughout the process of requirements elicitation, every suggestion and idea can be formalized and analyzed with immediate feedback to the business stakeholders.

reuse The rules shown in this chapter can be reused in other projects. As you do so, the rules presented here can be complemented by other rules. In that way you can build specifications that suit other, more specific or more general purposes.

This chapter showed an extensive example in order to demonstrate useful features of the Ampersand approach. The next chapter discusses how Ampersand played an important role in a large IT project.

Chapter 7

INDiGO

Abstract

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. The case discussed here is INDiGO, a project where Ampersand proved to be a major asset during the tendering phase. In INDiGO, Ampersand enabled designers to isolate the key concepts and discuss them with stakeholders in a very early stage. As a consequence, a comprehensive system architecture was already available at the start of system design. The contribution of Rule Based Design was to maintain correctness and consistency of this system architecture under the extreme time pressure that is common to tenders.

7.1 Introduction

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

[1]This name was chosen by IND after a naming contest among IND personnel

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 a 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 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.

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 every single action taken any day on any location is compliant with then-current regulations, and traceable to the original motivation and legal evidence. IND wanted to eliminate even the possibility to make legal mistakes. But even if a mistake was made, it was required that all actions are recorded in a legal procedure file.

Efficiency means to IND that no unnecessary 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, duplicate 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 quantifiable, 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, with no tailor-made 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.

Conceptual Model INDiGO employs an enterprise service bus (ESB) to overcome mismatches in the internal data models of the various COTS-components. A comprehensive Conceptual Model is central in that discussion. The conceptual model produced by Ampersand and the data

model that was derived from it, 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 performed, 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 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. The Conceptual Model will be briefly discussed in section 7.5.

When studying these examples, it is worth to notice that our business rules have three important properties:

- Every rule is valid throughout the entire scope (in this example: INDiGO).
- Each rule can be considered independent from other rules.
- The number of rules is small, as compared to the size and complexity of the organization.

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

[2]at an earlier stage, this model was called Common Message Model, following the conventions of service orientation in the UML

formalization of rules that govern the process of deciding about the IND's decision to grant residence permits (green cards).

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.

The generic procedure for dealing with administrative procedures in the Netherlands is laid down in the AWB ("Algemene Wet Bestuursrecht"). The specific issues of immigration regulations are described in the immigration law, called VW ("Vreemdelingenwet"), and related regulations.

Article 1:3 AWB introduces the ideas of *application* and *decision*, which lay 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).

application
decision

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.

alien

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 its own set of activities to be performed, as outlined by the law. Thus, INDiGO has to act as if each decision is unique, the single instance of a unique procedure. A *procedure* is a set of activities performed to produce a decision on the application of an alien.

procedure

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.

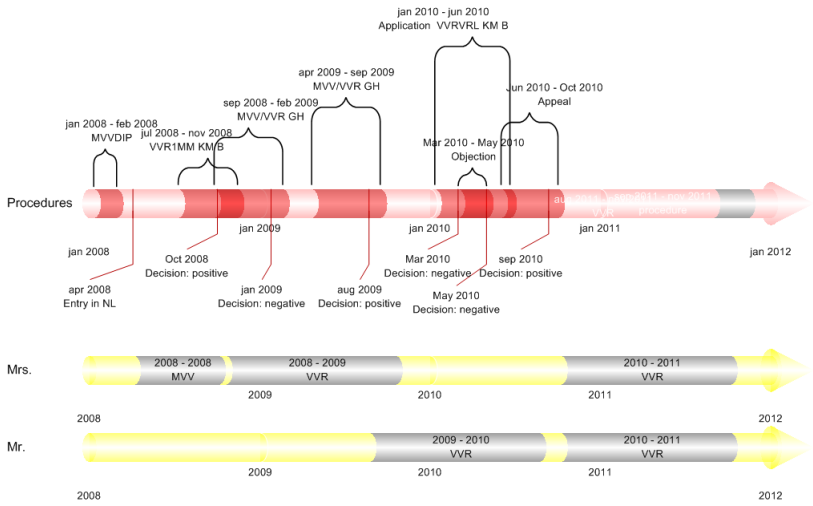

Figure 7.1: Example: related cases going through different procedures.

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 the functions:

$$reDescr \quad : \quad Procedure \rightarrow Description \tag{7.1}$$

$$reVal \quad : \quad Procedure \rightarrow DecisionValue \tag{7.2}$$

$$decisionName \quad : \quad DecisionValue \rightarrow Description \tag{7.3}$$

The rule is to maintain:

$$reDescr \;=\; reVal; decisionName \tag{7.4}$$

The formalization reduces the process to a simple lookup of standard texts. In many specifications, not only those of the IND, this kind of simplification is a common side effect of the formalization process. It shows how the mental efforts to come up with good specifications will pay off.

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:

$$decisionDescr \quad : \quad Procedure \rightarrow Description \tag{7.5}$$

The rule is to maintain:

$$decisionDescr \;=\; decisionVal; decisionName \tag{7.6}$$

Definitions 7.2 and 7.3, introduced in the previous rule, are reused for this purpose. This rule is similar to the previous rule 7.4, illustrating the point that such lookups are common.

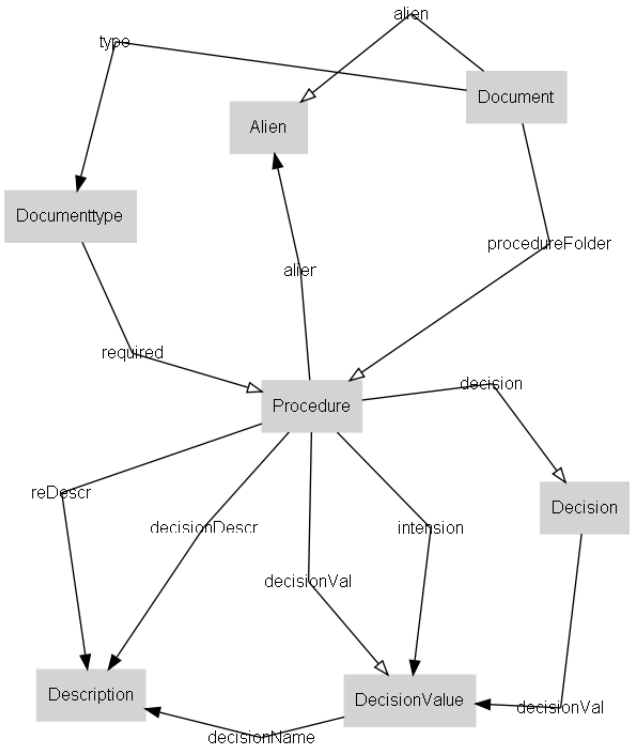

Figure 7.2: Conceptual Model of Decision making

Rule: deficiencies The IND starts by checking whether an application for residence is complete. Required documents that the IND is still missing must be signalled, so that an employee can request the additional information and documents. This check can well be automated by formulating a rule about required, but missing documents. Which (types of) documents are required, is determined by the type of application.

document

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. The formal document is also a physical product, being issued to someone as a result of a decision. The formalization of the rule (equation 7.12) requires the following five relations.

$$\begin{aligned} \mathit{required} &: \mathit{Documenttype} \times \mathit{Procedure} && (7.7)\\ \mathit{type} &: \mathit{Document} \to \mathit{Documenttype} && (7.8)\\ \mathit{docForProcedure} &: \mathit{Document} \times \mathit{Procedure} && (7.9)\\ \mathit{docAboutAlien} &: \mathit{Document} \times \mathit{Alien} && (7.10)\\ \mathit{alien} &: \mathit{Procedure} \to \mathit{Alien} && (7.11) \end{aligned}$$

The rule that every application shall be complete can be formalized as follows:

$$\mathit{required} \vdash \mathit{type}^{\smile}; (\mathit{docForProcedure} \cap \mathit{docAboutAlien}; \mathit{alien}^{\smile}) \quad (7.12)$$

This rule says that if a document type is required for a particular pro-

cedure, then there must actually be a document about the alien in the procedure's set of documents. To signal any missing documents, the system can compute:

$$\overline{required \cap (type^{\smile}; (docForProcedure \cap docAboutAlien; alien^{\smile}))} \quad (7.13)$$

First discrepancy rule 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 7.17, which requires the following three relations.

$$decision \;:\; Procedure \times Decision \quad (7.14)$$

$$decisionVal \;:\; Decision \rightarrow DecisionValue \quad (7.15)$$

$$intension \;:\; Procedure \rightarrow DecisionValue \quad (7.16)$$

The rule that says the two must be equal is:

$$decision; decisionVal \;=\; intension \quad (7.17)$$

This rule may be violated, but all discrepancies between the IND's decisions and the rule engine's computations are reported, for accountability purposes. The discrepancies can be computed by:

$$(decision; decisionVal \cup intension) \cap \overline{((decision; decisionVal) \cap intension)} \quad (7.18)$$

Second discrepancy rule Similar to the previous rule, the intended decision may deviate from the final decision. We use definitions 7.14 and 7.16, to state as a rule that the two must be equal:

$$intension \;=\; decisionVal \quad (7.19)$$

This rule reports the following signals:

$$(intension \cup decisionVal) \cap (\overline{intension} \cup \overline{decisionVal}) \quad (7.20)$$

If you do not understand this line of reasoning, check back to the section in chapter 4 that explains how violations of an assertion (like 7.19) can be computed as the extension of an expression (like 7.20). So far, the formalization defines (a selection of) the rules that govern the decision process at IND.

7.5 Conceptual Model

This section discusses the key principles that govern the conceptual 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 diagram of the Conceptual Model, as a work in progress: the diagram is neither complete nor correct in its finer details. This analysis contains the concepts discussed in this section.

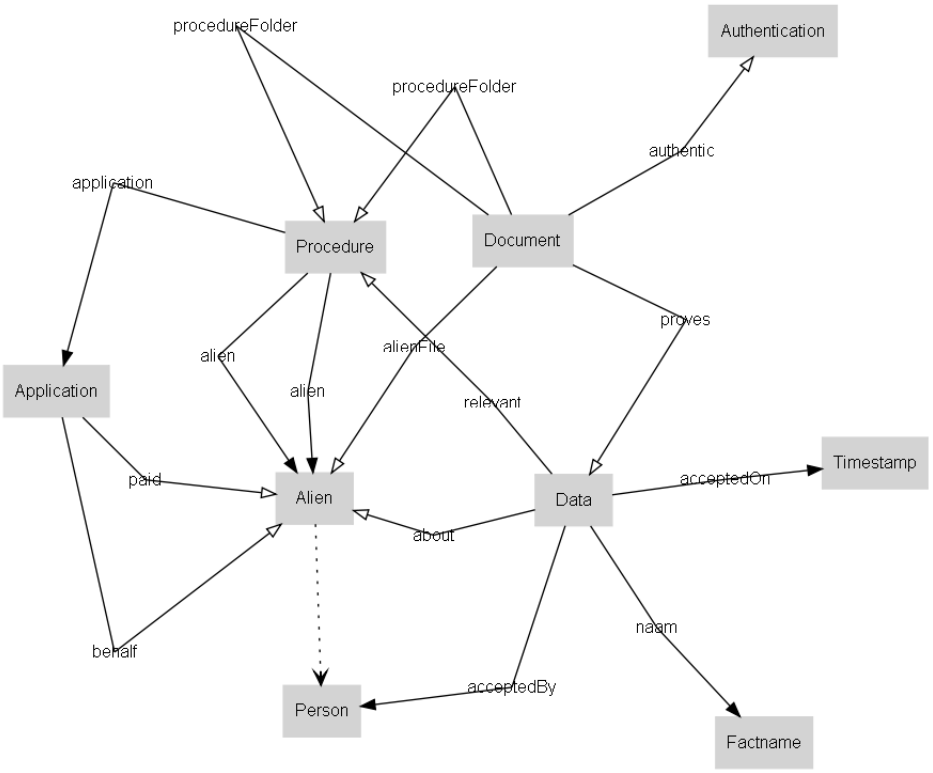

Figure 7.3: Conceptual diagram of IND Applications, as a work in progress

initiate procedures Every application has been submitted on behalf of just a single alien. It is conceivable to combine applications of different persons, for instance to keep groups and families together. However, the IND processes every application separately, and does not combine them. For each application, a procedure is conducted that leads to a decision. The formalization (equation 7.23) requires the following two relations.

$$\mathit{behalf} \quad : \quad \mathit{Application} \times \mathit{Alien} \tag{7.21}$$

$$\mathit{application} \quad : \quad \mathit{Procedure} \rightarrow \mathit{Application} \tag{7.22}$$

Besides, we use relation 7.11 linking the procedure to the alien. The rule maintains:

$$\mathit{behalf} \vdash \mathit{application}^{\smile}; \mathit{alien} \tag{7.23}$$

alien file Documents that are in the folder of a particular procedure, reside in the alien's folder by implication. We use definitions 7.9, 7.10, and 7.11, to state the rule to be maintained as:

$$\mathit{docForProcedure}; \mathit{alien} \vdash \mathit{docForAlien} \tag{7.24}$$

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 7.27, we need to introduce two relations:

$$about \quad : \quad Data \times Alien \tag{7.25}$$
$$relevant \quad : \quad Data \times Procedure \tag{7.26}$$

Again using relation 7.11, we come up with the rule:

$$about; alien^{\smile} \ \vdash \ relevant \tag{7.27}$$

procedure file All evidence corroborating a fact is relevant in the procedure in which that fact is used. In order to formalize this, we introduce:

$$proves \quad : \quad Document \times Data \tag{7.28}$$

Combining this with relations 7.26 and 7.9 we formalize the requirement 7.3 (page 150): This rule maintains:

$$proves; relevant \ \vdash \ docForProcedure \tag{7.29}$$

fees to be paid 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 7.30:

$$paid \quad : \quad Application \times Alien \tag{7.30}$$

Using definitions 7.11 and 7.22, we formalize the requirement that fees be paid as a rule:

$$application^{\smile}; alien \ \vdash \ paid \tag{7.31}$$

This rule reports the following signals:

$$application^{\smile}; alien \cap \overline{paid} \tag{7.32}$$

authentication All documents in a folder must eventually be attested by some authority as being authentic. Documents that have not yet been authenticated need to be signalled. In order to formalize this, we introduce:

$$authentic \quad : \quad Document \times Authority \tag{7.33}$$

Again using relation 7.9 that gives us all documents involved in a procedure, we may formalize the authentication requirement as:

$$docForProcedure; docForProcedure^{\smile} \ \vdash \ authentic; authentic^{\smile} \tag{7.34}$$

This rule reports the following signals:

$$docForProcedure; docForProcedure^{\smile} \cap \overline{(authentic; authentic^{\smile})} \tag{7.35}$$

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 unnecessary 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 Ampersand 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.

7.6 Data Analysis

The IND requirements, presented in the introduction in an off-hand manner, was translated into an Ampersand script. The script contains over 30 concepts and relations. From this script, a data model can be derived in which all rules from the Ampersand script can be represented. A provisional class diagram based on the script is shown in figure 7.4.

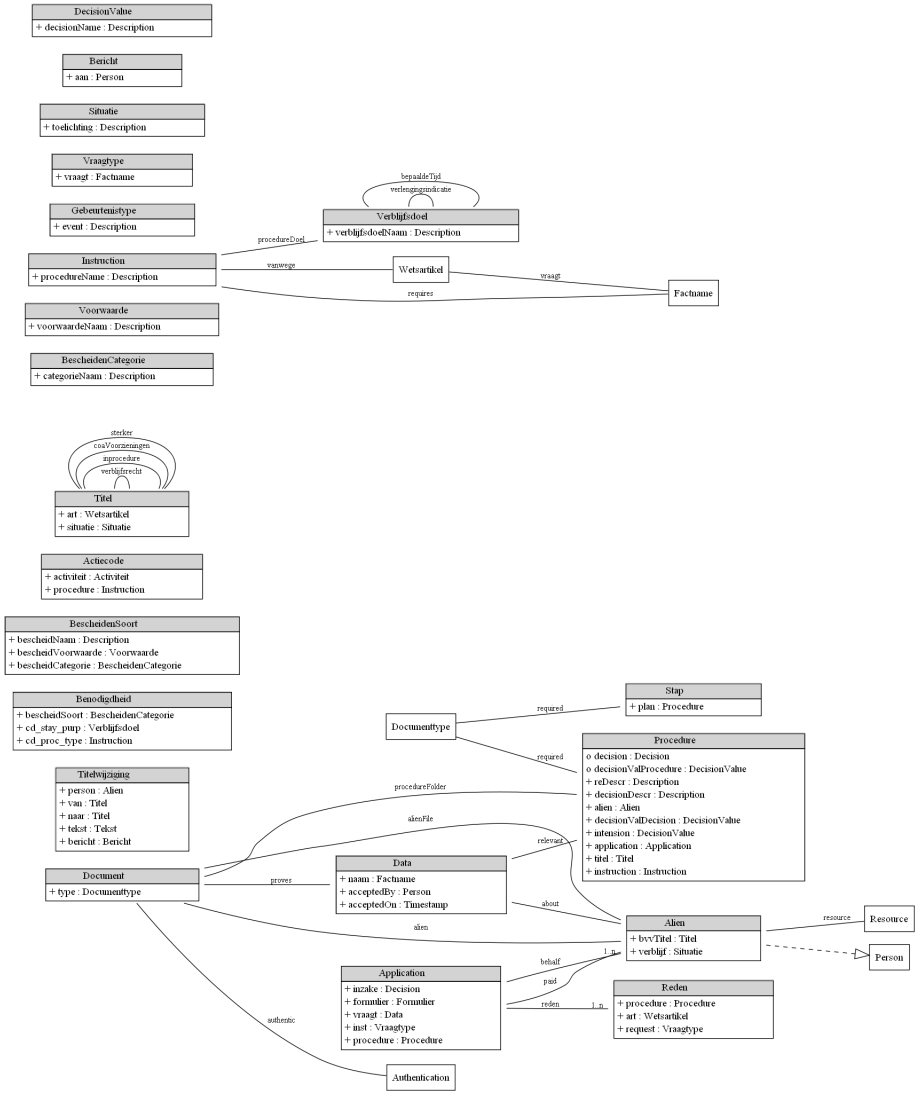

Figure 7.4: Provisional Class Diagram of IND

Taking it from here, software analysts and programmers can proceed to build an elegant information system that has a complete and correct set of business rules at its core. In practice, they can and will extend this model with more attributes and other classes. Removing attributes is not allowed however, as that would jeopardize compliance to the rules.

The relevance of conceptual modeling using Ampersand is in its comprehensiveness that enables analysis across the various COTS components in INDiGO. For example, data about aliens is 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 that involves documents that are related to decisions that are about aliens, affects all three components.

Rules such as we discussed here, will typically affect multiple components of the corporate system architecture. The rules provide the semantic consistence between components, which is most visible on the service bus level. Using the script, the Ampersand tool can take all concepts and relations, multiplicity constraints and rule assertions, and turn it into a complete and formal requirements specification. This comprehensive document is for the convenience of programmers, who use it to configure the data stores; workflow routines, message patterns, user interfaces, and much more.

7.7 Way of Working

The INDiGO project marked a first occasion in which Ampersand 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. It came as a surprise that some preconceived opinions about information system design were revised. So we present each lesson as a reaction to the preconceived opinion, that is contested by this experience.

Our people are not trained in formal specifications. The formulas and rules that appear in the functional specifications may look impressive, but they are computer generated. The most desired skill designers must have is to elicit requirements. This skill to write Ampersand is of secondary importance and can be learnt in approximately 100 hours. The functional specifications serve different stakeholders, of whom only the requirements engineer (designer) will ever see or use the formal language. The following specifications are generated by Ampersand:

- a specification intended for business stakeholders, meant to sign-off requirements in natural language only;
- a specification intended for requirements engineers, meant to establish the correct correspondence of natural language requirements and formalized rules;
- a specification intended for builders, containing data models and other artifacts that are useful for realizing an information system that fulfills the requirements.

The lesson is that Ampersand eliminates the need to share any formalism with user communities, officials, and members of the demand organisation. In English: the customer does not have to learn formal specifications at all. That is the task of requirements engineers and designers.

We don't have time to formalize. The lesson is that formalization saves time. The formalization of INDiGO was made as early as the tendering stage, way ahead of the start of the project. The resulting INDiGO

architecture has proved robust, and little needed to be changed in the course of time. (Of course, many additional features were added, but the core idea remained stable.) This is a direct result of its consistency, which eliminates the need to change the architecture in hindsight. The large amount of concrete details, that were already captured in the design during the tendering phase, is also a consequence of doing things right the first time.

Why bother about correctness? Correctness of an Ampersand script, and the specifications derived from it, means several things. The type checker ensures absence of type errors. It means that the specification is representable on a database system, so the system can actually be built.

In the system specification two types of rule are encountered. First, rules that must be maintained rigorously, and the information system must ensure that the rule is adhered to, anywhere and at all times. Second, rules whose violations are signalled. These 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. System correctness 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. The lesson is that we need to bother about correctness.

Isn't it difficult? Yes, writing good rules is hard. The difficulty lies in analyzing a problem and making it concrete. Writing it in relation algebra helps, because it allows only concrete rules to be written. In the INDiGO project, this feature has contributed considerably in detailing the initial architecture document.

Chapter 8

RAP: a Repository for Ampersand Projects

Abstract

This chapter is an excursion through the development of a tool set that supports Ampersand. This tool set automates the design process of the Ampersand method (chapter 2) as illustrated by figure 2.11.

The design process of Ampersand is a business process like any other one. Ampersand is a method to design information systems that support business processes. So Ampersand itself has been used to design and implement the tool, which is a repository in which students can store and study their Ampersand projects. This Repository for Ampersand Projects (RAP) stores the rules of which a project consists in a database (a repository). Students can use an application on top of that database.

RAP was designed by describing the design process in terms of rules that must remain satisfied. These rules are discussed in this chapter for two different reasons. The first is to illustrate a live application of Ampersand, which is the RAP repository. The second is to discuss the design of RAP in terms of its requirements, which have been formalized.

8.1 Introduction

We will present how we used Ampersand for the development of the RAP tools. In order to prevent confusion in reading, we introduce a convention to distinguish between the Ampersand project of the RAP tools and Ampersand projects using RAP tools. Ampersand concepts related to a tool project get the qualification *tool* or the name of some specific tool component e.g. *atlas* e.g. tool project, atlas, Ampersand script. Ampersand projects using RAP tools are assumed to be projects of students and are qualified by *student* e.g. student project, student Ampersand script.

The RAP tool set consists of:

1. A *control component*, which is a user interface to invoke RAP-functions on an Ampersand project in RAP. It is discussed in section 8.2. *control component*
2. A *command-line component*, which packages a number of RAP-functions for generating and validating specifications from an Ampersand project. These functions are available through command-line options. It is discussed in section 8.3. *command-line component*
3. An *atlas*, which enables students to analyze and explore their Ampersand projects in RAP. It is discussed in section 8.4. *atlas*

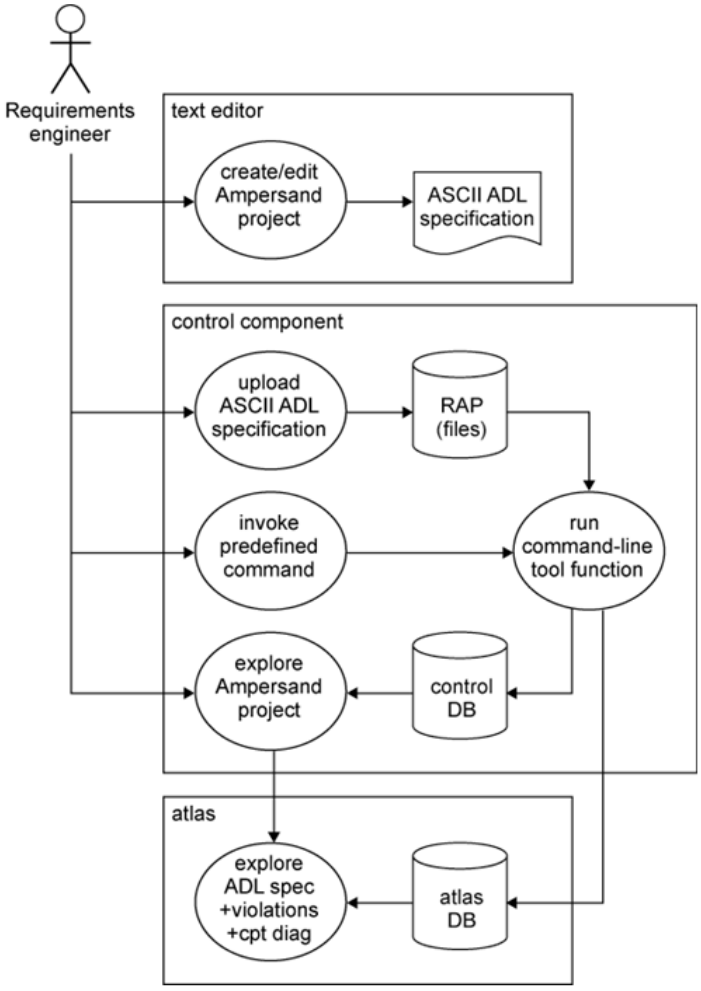

Figure 8.1: RAP tool set 1.1

This chapter is based on version 1.1 of RAP, which is the current version at the time of writing. In this version students use a text editor to create and edit their Ampersand scripts in plain ASCII format. Each script corresponds to an Ampersand project. Students can upload a script in the control component, which has a web interface for this purpose. The control component invokes the

command-line component to compile scripts and store them in the repository. When a student wants to analyze his Ampersand script, inspect results, or generate functional specifications and prototypes, he invokes the atlas by clicking a button in the control component.

First the development of the tools is discussed. Next we conclude in section 8.5 with a preview on future developments, towards an integrated development environment (IDE) for Ampersand.

8.2 Control component

The control component exists as a user interface, meant to offer self-evident functionality to students from a single web page. It is a web application with

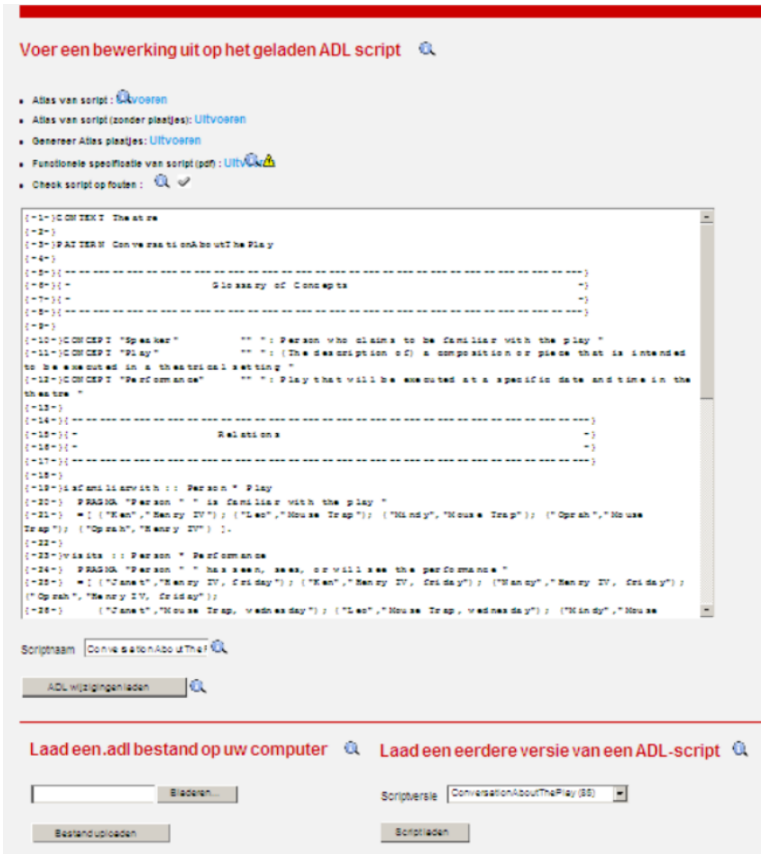

Figure 8.2: Screenshot of control page

one page only (see figure 8.2). This component invokes Ampersand functions on scripts provided by students. These functions are implemented as a sequence of one or more functions of the command-line component. The following functions are made available to students:

1. Generate a functional specification in pdf-format comprising a conceptual diagram and data model.

2. Translate the ASCII script to database content of the atlas, together with rule violations and hyperlinks from this data to generated conceptual diagrams (png).

The control component enables students to enter versions of student scripts, either by typing code directly in the screen editor, or by uploading an ASCII file. The main function of this component is to enable students to invoke the predefined commands on the command-line tool. An invocation generates data objects that can be accessed through an URL link. Invoking the generator may fail if a script contains mistakes. This results in on-screen feedback, which is a piece of text that contains the error messages. Upon succesful compilation, data objects that represent the student's script are generated and stored in RAP. The control component registers the activity of students as events, by way of logging their design efforts.

In the remainder of this section we discuss the specification of the control component in Ampersand. It has been formalized for the purpose of generating a database application.

Whenever a student runs the compiler, we call this an invocation of RAP. Calling the compiler means to create an invocation. A relation *runsOn* is needed to say which invocation operates on which file. In Ampersand, every invocation works on one file only. The command which is invoked is registered by relation *runs*. Every command makes its output know through a unique URL, which is represented by relation *hasOutput*.

```
runsOn    :: Invocation -> File.
runs      :: Invocation -> Command.
hasError  :: Invocation * ErrorMsg [UNI].
hasOutput :: Command * URL [UNI].
```

To represent the fact that a compilation is still running, every invocation has a boolean flag:

```
isRunning :: Invocation -> Flag.
```

Students that use the tool are carrying out a session. For this purpose, we introduce the notion of *session* as a registration of all activities a student performs.

session

A *Session* relates to a *User*.

```
isFrom :: Session -> User.
```

Based on this fact students we can derive the files of a user.

Display names are defined for *File* and *Command* hiding their more technical identifying strings.

```
hasName      :: File -> FileName.
isaPath      :: File -> Path.
hasName      :: Command -> CmdName.
isaScript    :: Command -> CmdScript.
```

Recapitulating what we did. We described the Rule Based Design business from the perspective of the role that the computer takes. From this description the core unconstrained conceptual model (see section 4.1.4) is derived. Multiplicities are added to this model right after. We added more detail where needed.

Explicitly modeling the core first and adding detail with a motivation has advantages. At present it ensures a design with relevant details. It is very tempting to just model details, because it describes reality. Preserving motivations for details makes design choices traceable, which may be valuable for future design choices.

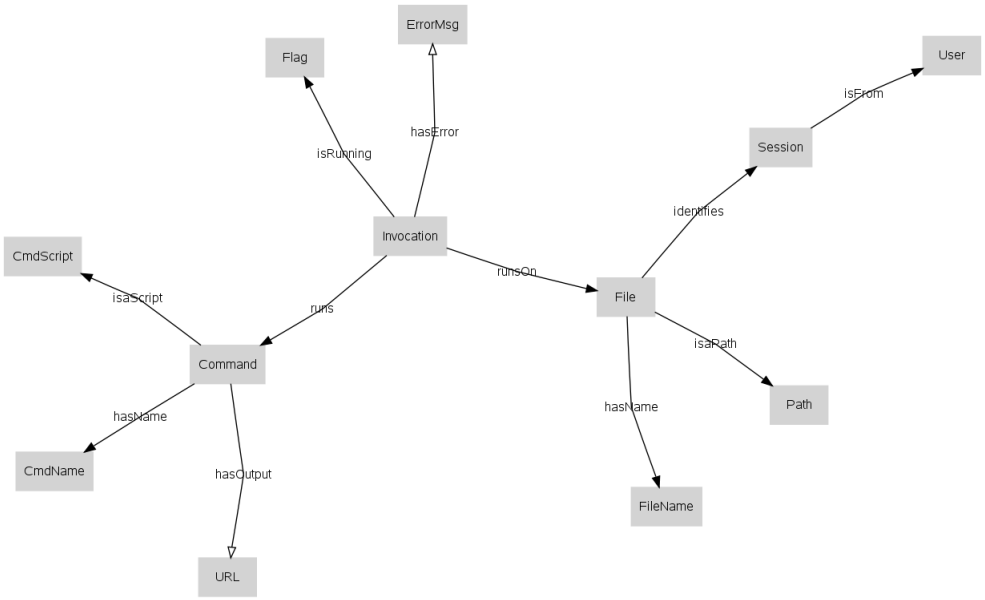

Figure 8.3: Conceptual diagram of control component

Besides multiplicities there are no other business rules. This can be verified by looking at the number of cycles or just looking at the conceptual diagram. There are 12 concepts and 11 relations: $1 - 12 + 11 = 0$.

The absence of business rules is strange and should be investigated for that reason. For example, you could expect some connection between the flag indicating that an invocation has been completed and the output channels. This rule sounds reasonable: *There can only be an error message for an invocation if the invocation is not running anymore.* Thus,

> The absence of a cycle in your model does not imply the absence of a rule.

> The reason why we did not define rules for the control component, is that we are not interested in them yet in the current version of the Ampersand tool. Our objective is to develop a tool to register facts without to much complexity i.e. just a database with function access. In future versions, business rules are to be expected.

8.3 Command-line component

We have constructed a Rule Based Design for the control component using Ampersand. This was the first activity of the Ampersand design process as illustrated by figure 2.11. The next activity in this business process is generating artifacts from this design using the command-line tool.

The design of the control component controls this flow of work. Students cannot generate artifacts if they do not have requirements in RAP, i.e. a script file must be provided:

```
runsOn    :: Invocation -> File.
```

If there is a script file, then the command can be executed to generate the desired output.

```
runs      :: Invocation -> Command.
hasOutput :: Command * URL [UNI].
```

We generated a prototype system from the control component script. The prototype system comprises SQL and PHP functions that enable to create and manipulate a database for recording and handling relations defined by students. These functions are accessed through a web browser interface, which is restricted for use by administrators only. The functions enable an administrator to maintain the population of the control component, e.g. to insert, alter, or remove the predefined commands made available to students. The browser page for students (figure 8.2) was designed as a conventional web page, using the PHP functions generated by Ampersand.

The control component offers a function (button) to generate a functional specification in .pdf format from an Ampersand script. Among others, this generated specification outlines the data model for the database of the prototype. To generate a functional specification, the following steps need to be taken.

1. Upload a script, such as
2. Invoke the command for generating the specification.
3. Check out the data model of the control component in the .pdf specification.

For readers who want to try this for themselves, we suggest uploading the script of figure 8.4 and generating a functional specification for it. This script represents the control component of RAP.

8.4 Atlas

The atlas is a web application enabling students to analyze and explore their Ampersand scripts. Through the control component the statements in a student script are loaded into the atlas database with the same granularity as the ASCII syntax. Additional information is generated like rule violations and

```
CONTEXT ControlApplication
PATTERN Control
  runsOn    :: Invocation -> File.
  runs      :: Invocation -> Command.
  hasError  :: Invocation * ErrorMsg [UNI].
  hasOutput :: Command * URL [UNI].
  isRunning :: Invocation -> Flag.
  isFrom    :: Session -> User.
  hasName   :: File -> FileName.
  isaPath   :: File -> Path.
  hasName   :: Command -> CmdName.
  isaScript :: Command -> CmdScript.
ENDPATTERN
ENDCONTEXT
```

Figure 8.4: Ampersand script for the control component

conceptual diagrams and links to them. After loading the data can be explored through web pages presenting the data as related concepts.

For example, in Ampersand a business rule is an assertion, a complex formula, defined within the scope of a single pattern. In the atlas script we can define this as a function from a concept *Rule* to a concept *Pattern*

```
definedIn :: Rule -> Pattern.
```

We have defined a web page for patterns that given some pattern from a student script presents all the rules defined in this pattern. The student can click on rules on the pattern page, because we have defined a page to display the concept *Rule*. The web page for rules presents information about a rule. Information about a rule is data that relates to a rule e.g. an explanation

```
describedIn :: Rule -> Explanation.
```

Students cannot click on the explanation, because we didn't define a page to display the concept *Explanation*.

Thus the atlas database contains syntax terms from student scripts that are related to each other by the syntax rules of Ampersand. Atlas pages may be defined to display certain kinds of Ampersand terms. All terms that can be displayed are clickable resulting in a web application in which students can click through their scripts.

Besides the statements in the student script, the atlas contains some derived or generated data that is related to some tool concept. For example a rule relates to its violations

```
violates :: Violation * Rule.
```

or a pattern relates to a conceptual diagram identified by an URL string.

```
linksTo :: Pattern -> Picture.
```

We discuss the development of the web page to display instances of the atlas concept *Concept*. Other pages are developed likewise. Instances of *Concept* are concepts within a student's specification, i.e. his Ampersand script. Figure 8.5 is a screenshot of this web page, which displays a student concept *Play*.

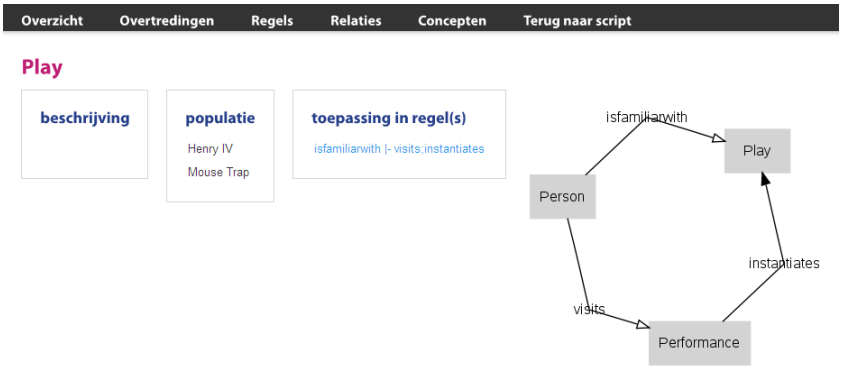

Figure 8.5: Screenshot of atlas displaying some student concept *Play*

The objective of the atlas is to define a read-only information system. In a read-only information system you cannot change facts, and therefore you can neither create nor remedy rule violations. Thus, for the atlas it is sufficient to define relations that are relevant to display information. For example the page for student concepts displays four related atlas concepts:

Explanation A description of the student concept. We define a relation that represents this information

```
describedIn :: Concept*Explanation [UNI].
```

There is no description defined for *Play*.

Atom All the atomic values that are contained in the student concept. This is modeled as

```
contains :: Concept*Atom.
```

'Henry IV' and 'Mouse trap' are all plays in the business context.

Rule The set of all the student rules that relate to the student concept in the sense that the student concept is the source or target of one of the relations used in a student rule. This is modeled as an expression

```
(source~ \/ target~);signat~;uses~
```

The expression uses four relations

```
source   :: Type->Concept.
target   :: Type->Concept.
signat   :: Relation*Type.
uses     :: Rule*Relation.
```

Picture A conceptual diagram limited to closely related relations and concepts of a student concept

```
linksTo :: Concept -> Picture.
```

8.5 Ampersand IDE

An integrated development environment (IDE, figure 8.6) for Ampersand is foreseen in the future. In the current toolset, students upload a script, which

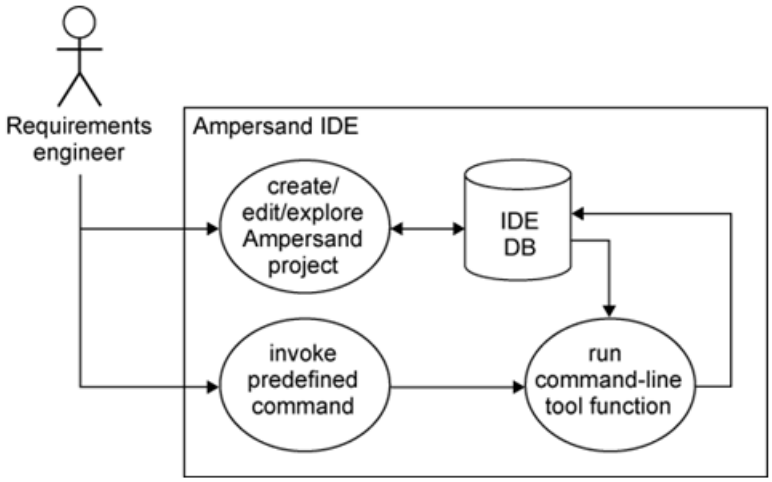

Figure 8.6: Ampersand IDE

is edited in a separate editor. The IDE will allow students to edit projects directly in RAP. This will eliminate the need for any other syntax than the rule syntax only, which makes learning the language easier for students. This IDE will become an information system for the Ampersand designers.

Developing the atlas as a read-only system is a measure to deal with business rules that have not yet been defined. In the IDE students have access to edit certain data in the IDE database. At first they will be able to edit populations of their models, which requires proper rules. Later they will be able to edit their models such that they do not need a text editor anymore, requiring even more rules. Certain data always remains read-only because it is derived, for example violations. To recalculate violations they need to invoke an action on the control component. Although not realized in practice, we describe the case of designing the Ampersand IDE.

What does it mean that the Ampersand IDE supports Ampersand? In section 5.6 Ampersand is defined as a way of working to deliver a Rule Based Design. Thus the Ampersand IDE supports a way of working. This is translated into the feature that the Ampersand IDE may run checks on a student model even as the model is under development and unfinished. So the student can be challenged to have *all concepts and relations to be populated in such a way that no violations emerge.* This sounds like a declarative business rule for the IDE, but to prevent confusion between the tool and student model we better qualify some terms: *all student concepts and student relations can be populated in such a way that no student violations emerge.*

This rule is based on concepts and relations in a student project that are related to a population in a way, and these facts are related to violations in some way. Violations are related to rules, rules are related to relations, a signature relates

a relation with a type, a type consists of a source and target concept, concepts are related to a population of atoms, relations are related to a population of tuples, a tuple consists of a left and right atom. We copy definitions from the atlas 1.1 project and add the concept *Tuple* with some relations:

```
violates :: Violation * Rule.
uses     :: Rule * Relation.
signat   :: Relation * Type.
contains :: Relation * Tuple.
left     :: Tuple -> Atom.
right    :: Tuple -> Atom.
source   :: Type -> Concept.
target   :: Type -> Concept.
contains :: Concept * Atom.
```

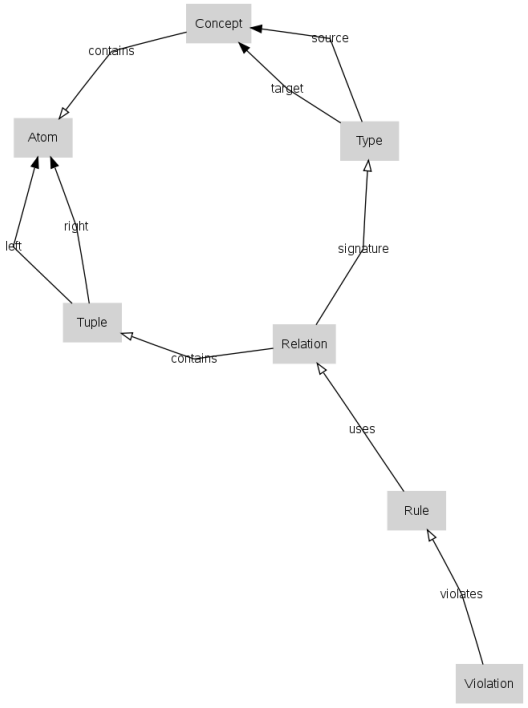

Figure 8.7: Partial conceptual diagram of IDE

The partial conceptual diagram shows three cycles, you may check that the cyclomatic number is indeed $1 - 7 + 9 = 3$. Hence, at least three rules can be expected to control correctness of the populations and to prevent violations. But in this case, not. The cycle involving types and concepts, is unconstrainted. No restriction is placed on the types that students may create from their concepts.

On the other hand, there is an important restriction to control the populations of concepts and relations. We described this demand as referential integrity in section 3.1.9. Two almost identical rule assertions can capture this single

requirement:

```
 RULE chksource MAINTAINS contains;left |- signat;source;contains

RULE chktarget MAINTAINS contains;right |- signat;target;contains
```

By maintaining these two rules, the IDE ensures referential integrity of the populations. In other words, it is safe to let student edit the populations of concepts in the tool.

Now the student wants to check whether they defined a population with violations or not. For that they need to invoke the violations check command in the control component. In version 1.1 this generation function is integrated in the synchronization command from file to atlas. Assume that students can only edit populations and not their models. Invocations on files are still needed to load student models with an initial population. Invocation of the validation check will run on a project in the database. We redefine some relations from version 1.1 and add the concept *Project*. Projects are the link between file-based RAP and the RAP database.

```
GEN File    ISA Project
GEN Context ISA Project
object    :: Invocation -> Project.
output    :: Command * URL [UNI].
user      :: RAP  -> User.
context   :: Pattern -> Context.
originates:: Context -> File.
```

A final remark on the relation *originates*. In the IDE version where students are allowed to edit their populations only, every context in the database relates to one script file only. In the future IDE version where students are allowed to edit their model and thus create new contexts, the concept *File* may turn out to be superfluous. At that point we do not need the text editor to create and edit scriptfiles anymore. Students can create and edit their Rule Based Design projects in the IDE, guided by the tertiary business rules defined in the Ampersand method.

The next step is proving students with a means to generate a provisional user interface (e.g. in a browser) for viewing and editing the populations of concepts, like we did in the atlas for the tool concepts. The interfacing, based on the rules defined by students, will support the inspection of violations, and selection and execution of edit actions to remedy them. By performing a sequence of such edit actions, all violations are resolved and a business process is completed. This will be experienced by students as a prototype information system that implements all the requirements of the business process, i.e. all the primary, secondary and tertiary rules. A single information system for the IT and business stakeholders striving to live by the rules and getting the job done.

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 M.M. 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] C. Brink and R. A. Schmidt. Subsumption computed algebraically. *Computers and Mathematics with Applications*, 23(2–5):329–342, 1992. Also in Lehmann, F. (ed.) (1992), *Semantic Networks in Artificial Intelligence*, Modern Applied Mathematics and Computer Science Series **24**, Pergamon Press, Oxford, UK. Also available as Technical Report TR-ARP-3/90, Automated Reasoning Project, Research School of Social Sciences, Australian National University, Canberra, Australia.

[7] Chris J. Date. *What not how: the business rules approach to application development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[8] 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.

[9] 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.

[10] R. Deen. Het nut van case-handling - flexibel afhandelen. *Workflow magazine*, 6(4):10–12, 2000.

[11] 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.

[12] 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.

[13] W. van Dommelen and Stef M.M. Joosten. Vergelijkend onderzoek hulpmiddelen beheersing bedrijfsprocessen. Technical report, Anaxagoras and EDP Audit Pool, 1999.

[14] Berco Beute Erik van Essen and Rieks Joosten. Service domain design. Technical Report Freeband/PNP2008/D3.2, TNO, The Netherlands, 2005.

[15] M. Fowler. *Analysis Patterns - Reusable Object Models*. Addison-Wesley, Menlo Park, 1997.

[16] 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.

[17] 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.

[18] Rieks Joosten and Berco Beute. Requirements for personal network security architecture specifications. Technical Report Freeband/PNP2008/D2.4, TNO, The Netherlands, 2005.

[19] 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.

[20] Stef M.M. Joosten. Workpad - a conceptual framework for workflow process analysis and design. Unpublished, 1996.

[21] Stef M.M. Joosten. Rekenen met taal. *Informatie*, 42:26–32, juni 2000.

[22] Stef M.M. Joosten and Sandeep R. Purao. A rigorous approach for mapping workflows to object-oriented i.s. models. *Journal of Database Management*, 13:1–19, October-December 2002.

[23] Stef M.M. Joosten et.al. *Praktijkboek voor Procesarchitecten*. Kon. van Gorcum, Assen, 1st edition, September 2002.

[24] Tony Morgan. *Business Rules and Information Systems: Aligning IT with Business Goals*. Addison Wesley, 2002.

[25] Ordina and Rabobank. Procesarchitectuur van het servicecentrum financieren. Technical report, Ordina and Rabobank, October 2003. presented at NK-architectuur 2004, www.cibit.nl.

[26] 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.

[27] 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.

[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] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, September 1941.

[34] 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.

[35] Joost van Beek. Generation workflow - how staffware workflow models can be generated from protos business models. Master's thesis, University of Twente, 2000.

[36] R. van der Tol. Workflow-systemen zijn niet flexibel genoeg. *AutomatiseringGids*, (11):21, 2000.

[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