Development Environment for Rule-based Prototyping

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Open Universiteit
op gezag van de rector magnificus
prof. mr. A. Oskamp
ten overstaan van een door het
College voor promoties ingestelde commissie
in het openbaar te verdedigen

op vrijdag 19 juni 2015 te Heerlen
om 16.00 uur precies

door

Gerardus Michels

geboren op 27 februari 1980 te Utrecht

Printed by GVO drukkers & vormgevers B.V., Ede.
Cover by Floor Geluk.

ISBN: ??-??-??-??-??

Contents

Dankwoord

Stef Joosten is veel meer geweest dan dat je wensen mag van een promotor. Ik dank hem voor zijn geloof en motiverende betogen, zijn geduld en vasthoudendheid, zijn inspiratie en kritieken, zijn ondersteuning en enthousiasme, want deze invloeden zijn allen onmisbaar geweest voor het kunnen produceren van dit proefschrift.

Vervolgens dank ik Lex Wedemeijer, Jaap van der Woude en Lex Bijlsma, omdat zij altijd tijd hadden om mee te denken en commentaar te geven op dat wat mij bezighield.

Mijn dank gaat uit naar de mensen van het Centrum voor Software Technologie van de Universiteit Utrecht en met name Doaitse Swierstra, Sean Leather en Bastiaan Heeren. Zij hebben mij gastvrij ontvangen en barmhartig geholpen bij mijn onderzoek.

Martijn Schrage, Han en Bas Joosten dank ik voor hun ontwikkelwerkzaamheden aan de Ampersand compiler, waardoor ik de ontwikkelomgeving (RAP) heb kunnen maken met alle functionaliteiten die het nu heeft.

Bert Zwaneveld, Harrie Swart en Jeroen Merriënboer dank ik voor de gesprekken, die ik met ze heb gehad.

Tenslotte bedank ik mijn gezin. Mijn vrouw Floor voor het luisterend oor dat ze me geboden heeft. Mijn dochter Veerle omdat ze zo fijn heeft meegewerkt. Mijn zoon Ruben voor de zelfreflectie. Mijn pasgeboren zoon Roy, we gaan het beleven!

Chapter 1

INTRODUCTION

This dissertation provides insight into a didactic problem: How to teach Ampersand, which is a rule-based approach to design information systems and business processes (IS&BP). This research addresses difficulties to design IS&BP, which are specific to a rule-based design approach, namely the difficulty:

- to understand and agree upon requirements, that is, what do the stakeholders want? For which we have found inspiration in the Business Rules Approach [32].
- to implement requirements, that is, how to develop software from requirements? For which we have found inspiration in model-driven software engineering [25].
- to cope with changing requirements, that is, how to maintain requirements and software? For which we have found inspiration in the work of Date [5].
- to formulate and communicate requirements, which is needed to manage requirements and software in a structural manner. For which we have found inspiration in the Business Rules Manifesto [31] and relation algebra [20].

This research has been performed in the context of the Ampersand project [17], which aims at developing a fully automated, requirements-to-software method for designing IS&BP, called **Ampersand**. The idea of Ampersand is to make the design of IS&BP more concrete by letting requirements engineers produce working software. This is done by automating the programming and documentation chores in the software development process. Ampersand produces design artefacts and working software directly from formal, functional requirements. Data models or process models are derived automatically, without human effort. The price is paid by requirements engineers, who have to learn how to formalize functional requirements. We categorize Ampersand as a **rule-based design** approach, because Ampersand uses a relational language to express functional requirements as a model of rules.

The development and teaching of Ampersand is being complicated by our target audience: students interested in requirements engineering. Many of these students are interested in the business application of information technology. They typically have little background in the formal thinking needed to specify working software. For that type of thinking, which is more akin to programmers', we have found that a minimal amount of mathematical training is prerequisite. Nevertheless, we could achieve a lot with tools and acquired didactical insights to accommodate such students.

This dissertation reports on four research activities that were done to contribute to the Ampersand project:

- The development of tools for Ampersand with Ampersand [section 1.1.1];
- Finding ways of tool support to teach a formal language [section 1.1.2];
- Finding ways to automate design tasks for education [section 1.1.3];
- An exploration of didactic research with the developed tools [section 1.1.4].

The following section 1.1 explains how our interest in teaching IS&BP design has led to the above research activities centred around the development of tools to teach Ampersand.

1.1 From didactic problem to research

An **information system** can be defined as interrelated, IT-based components to collect, process, store and disseminate information for business process support [19]. Designing information systems requires both IT and organizational skills. Learning design skills is a complex learning task [21], which can be addressed by means of practical exercises and experiments [28]. An obvious approach is therefore to make an educational design exercise environment for students to practice IT and organizational skills. That is precisely what has been chosen to do in our research plan.

Merrienboer [21] argues that a learning environment should let students focus on essential tasks by taking away secondary tasks. In the IS&BP research group at the computer science department of the Open University of the Netherlands, we believe that business rules capture the essence of a business process. Consequently, the essential learning task is how to model a business process in terms of business rules. The learning environment used in this dissertation is a development environment for rule-based prototyping called the **Repository of Ampersand Projects**, abbreviated to **RAP**. RAP automates the design of an information system by producing two artefacts: design documentation and working business process software (prototypes). The design automation comes for free, because RAP supports the Ampersand [39] method, which generates documentation and software on the basis of business rules alone. The underlying assumption is that students will focus on their essential task of business rule elicitation, because RAP automates a number of secondary design tasks, such as computer programming and data modelling. This motivates our choice to use RAP as a learning platform.

RAP is an evolutionary product of which the first developments at the OUNL started in 2002, which has been given the name RAP in 2010. This initiative from 2002 originates from the Calculating with Concepts (CC) method [8], which was conceived in 1996 as a formal approach to reason about business processes and software applications. The CC-method and related knowledge grew through practical uses of the method in industry, e.g. [15, 1, 16]. The potential of rule violations to drive business processes, an important concept of Ampersand today, became apparent in 1998. The first violation detector was written in 2000 for a project of the ING Bank. In 2002 research at the OUNL was launched to further the works on the CC-method into the direction of an educative tool, that is, RAP. The research group at the OUNL first introduced the Ampersand language and later in 2006 they refined the CC-method into the Ampersand approach. The toolset for Ampersand in 2008 included a compiler for the Ampersand language, called the **Ampersand compiler**, and a web application for education at the OUNL [section 6.1], called the **Atlas**. The Ampersand compiler of 2008 could already generate functional specifications and prototype software from a set of rules expressed in the Ampersand language, a so-called **Ampersand-model**. The Atlas of 2008 was a web interface to examine rule violations of a single data population for an Ampersand-model. In 2010, the Ampersand compiler and Atlas were combined and centred around a repository for populated Ampersand-models, which became the first version of RAP. The second version of RAP appeared in 2012, which is a product of this dissertation fully described in chapter 5. In the meantime, the design of the Ampersand compiler has been revised and its Ampersand-model processing functions have been extended and improved. By being the Ampersand-model processor of RAP, the capabilities of the Ampersand compiler have a direct, positive influence on the capabilities of RAP.

In 2009, we acknowledged the need to learn more about the didactics of Ampersand, based on early experiences with our course on Ampersand [section 6.1]. For example, a significant number of our students have issues with learning the formal language of Ampersand, which distracts them from learning design skills. Therefore, we envision didactic research based on student behaviour in RAP. We have investigated whether student behaviour can be analysed based on student activity logs from RAP.

From a practical perspective, the research behind this dissertation seeks for answers to one central question: *What are requirements for RAP to teach Ampersand?* The requirements for RAP yield what RAP is (1), how RAP is developed and maintained (2) and how RAP is used (3):

(1) RAP is a development environment for Ampersand, which targets students who are learning Ampersand to improve their skills with respect to IS&BP design. An educational challenge of Ampersand is its use of a formal language [section 1.1.2]. Automated design tasks in Ampersand may serve education as well as designing [section 1.1.3].

(2) RAP has been developed with Ampersand, which makes it an interesting case to validate and refine the process of tool development with Ampersand [section 1.1.1]. A reason to use Ampersand is that Ampersand requires little development capacities to respond to changing requirements of RAP in a timely fashion.

(3) Student usage of RAP in combination with being responsive to changing requirements of RAP allows us to repeatedly: harvest data about students, define metrics on that data, formulate observations, learn lessons and refine RAP [section 1.1.4].

1.1.1 Development of RAP with Ampersand

Ampersand is not only the subject of RAP, but also an essential asset for the development of RAP. Because RAP has been developed with Ampersand, we:

- have formal - unambiguous - requirements
- could manage and investigate requirements with tools
- could manage and compare different versions of RAP
- could (re)process requirements with tools
- could (re)generate working software instantly

How has Ampersand given us these possibilities? To develop RAP with Ampersand, the functional requirements of RAP need to be captured in an Ampersand-model, which we call the **RAP model**. Requirements in the RAP model are expressed in the Ampersand language. Any requirement that is in the RAP model is instantly realized in the software of RAP, because that software is generated from the RAP model with the Ampersand compiler. The Ampersand compiler has more options to process Ampersand-models, like analysing requirements or generating software models and requirements documentation. For example, chapter 5 is the generated requirements documentation of the latest RAP. The Ampersand compiler is our primary medium to implement **automated (design) tasks** [section 1.1.3]. Thus, Ampersand has given me the possibilities mentioned above by given me a formal language to express requirements and tools to process the requirements in that language.

1.1.2 Teaching a formal language

The Ampersand language is based on relation algebra [20]. With relation algebra, a student needs to learn only a few, but powerful relational operators to express business rules. However, a small, powerful language is still a formal language, which requires mathematical skills of students. Our personal experience with teaching Ampersand shows that students have difficulties with learning and using the Ampersand language, which they blame on the mathematical aspect of Ampersand. Possibly RAP can support students with the mathematical aspects in learning and using the Ampersand language. With respect to our central question, this dissertation explicitly addresses *requirements for RAP to support students in learning and using the Ampersand language.* For example, a feedback system has been added to RAP, which gives correct and to-the-point feedback on type errors in relation algebraic expressions from students [section 3.5].

1.1.3 Automated design tasks for education

The possibility to automate design tasks with the Ampersand compiler is an opportunity to support the learning and teaching process. Therefore we have designed RAP to be a development environment in which automated design tasks for education can be implemented. With respect to our central question: *What are requirements for RAP to implement automated design tasks for education?*

RAP targets the following uses of automated tasks for education:

- A course developer may delegate secondary learning tasks to automated tasks such that a student can focus on their primary tasks. Prominent features of RAP for students are visualizing and editing Ampersand-models in RAP.
- Give intelligent feedback by means of an automated process to diagnose or give educational feedback on the design efforts of a student. As a rule-based method, Ampersand supports constraint-based feedback in RAP [24] out-of-the-box. That is, constraints and related feedback can be expressed in the Ampersand language.
- Provide information, like learning analytics [36], for researchers, teachers and students by means of an automated task to measure, analyse or visualize the

learning behaviour of an individual or group of students. The source of these analytics can be any data stored in RAP like Ampersand-models or user activities in RAP.

1.1.4 Didactic research with RAP

We have explored whether analytics in RAP can improve our knowledge about student behaviour related to learning and using Ampersand with RAP [section 6.5]. In other words, we have explored the possibilities of research with RAP to develop tool-based didactics for Ampersand. Research in RAP has been combined with rapid development of RAP with Ampersand, which has resulted in a cyclic research-driven development approach for RAP and didactics of Ampersand. Cycles are formed through gaining new insights from the usage of improved development tools and improvement of development tools is based on new insights. This research approach might be used for certain other subjects of computer science, but it requires developing a development environment like RAP with a method like Ampersand. We have verified the viability of the research approach on RAP through an experimental study with analytics from RAP and the development of a next version of RAP [chapter 6]. With respect to our central question, we ask ourselves: *What are requirements for RAP to analyse learning behaviour?*

1.2 Contribution

The primary contribution of this dissertation is RAP, a development environment for rule-based prototyping. This environment is currently being used in computer science education at the Open University of the Netherlands.

A contribution of this dissertation is that RAP has been generated as a prototype with RAP itself. The requirements, from which RAP has been generated, are given in chapter 5. This achievement validates the theory that functional requirements can be defined by means of rules as explained in chapters 2 and 4. The lessons learned from developing RAP have contributed to a major revision in progress of the type system and a refactoring of the Ampersand compiler.

The contribution of RAP to the problem of teaching rule-based design is threefold.

Firstly, RAP offers a development environment for students that allows them to do exercises with formal specifications written in Ampersand. The environment supports the automation of secondary learning tasks within such exercises, which is the responsibility of a course developer. Design exercises may target the learning of IS&BP design tasks ranging from requirements elicitation up to programming, which is a step towards a broader applicability of formal specification.

Secondly, RAP offers a platform for researchers to study the way students are learning this specific topic. The platform supports the automation of the collection and analysis of semantic data produced by students doing their exercises. A measurement framework in RAP [chapter 7] lets a researcher add metrics to RAP with an unambiguous meaning. Analysis functions, e.g. measurements and visualizations, may be implemented in RAP by means of functions in the Ampersand compiler or outside of RAP, e.g. in a spreadsheet.

Thirdly, a first study with RAP has been conducted [section 6.3], which has resulted in the formulation of observations based on measurements in RAP [section 6.4]. This study demonstrates our envisioned cyclic research and development approach and yields empirical validation of possibilities and limitations of that method [section 6.5].

1.3 Dissertation outline

This dissertation reports on the development of RAP and how RAP can and has been used for education.

Chapter 2 introduces Ampersand, which is the method for rule-based prototyping that RAP supports. Chapter 3 defines the rule-based language of Ampersand that RAP supports. This contains material published in [22, 38].

In chapter 4 we present the design and development of RAP. RAP has been developed with Ampersand, which means that there is a specification of RAP in the Ampersand language from which RAP has been generated as a prototype, namely the RAP model. Chapter 4 first presents the logical design of RAP and an impression based on screen shots. After that, RAP is presented as an Ampersand-generated prototype. The chapter elaborates on the required functions to generate RAP with RAP, which was published before [18]. Chapter 5 gives the detailed requirements specification of the second version of RAP. This chapter has been generated from the RAP model like RAP itself. The requirements in the specification are examples of what can be defined in Ampersand. Appendix A contains the script of the RAP model including user interface definitions.

RAP has been released to students as a tool for rule-based design exercises in a course we teach at the Open University of the Netherlands. Chapter 6 presents the work we have done to show that RAP is suitable as a platform to study the learning behaviour of students. It shows that a variety of lessons could have been learned by analysing student usage data in RAP, which was published before [23]. Chapter 7 gives a specification for an extension on RAP to implement metrics on usage data in RAP. Such metrics have been used to study learning behaviour.

1.4 Origin of content

This dissertation has been constructed from subproducts of my research including the RAP model and the following peer-reviewed and published papers.

(I) Gerard Michels, Sebastiaan Joosten, Jaap van der Woude, and Stef Joosten. Ampersand: Applying relation algebra in practice. In *Proceedings of the 12th conference on Relational and Algebraic Methods in Computer Science*, Lecture Notes in Computer Science 6663, pages 280–293, Berlin, 2011. Springer-Verlag

(II) Gerard Michels and Stef Joosten. Progressive development and teaching with RAP. In *Proceedings of the Computer Science Education Research Conference 2013*, pages 33–43, Heerlen, 2013. Open Universiteit

(III) Stef Joosten and Gerard Michels. Generating a tool for teaching rule-based design. In *Proceedings of the Third International Symposium on Business Modeling and Software Design*, pages 230–236. SCITEPRESS, 2013

Stef Joosten, the supervisor of this research, has helped me to rephrase sentences and outline the content of this dissertation and the three papers.

Chapter 2 has been constructed from material related to paper III. Sections 2.1 and 2.3 contain revised text from that paper. The example in section 2.2 has been used in the presentation of that paper at the BMSD conference 2013.

Chapter 3 has been constructed from the contents of paper I. Paper I presents a formal definition of the syntax, semantics and type system of the Ampersand language, which has been defined together with the co-authors of that paper. The introduction to the Ampersand language [section 3.1] originates from paper II. Section 3.5.3 introduces the type of conceptual diagrams used in the Ampersand approach, which is content that has already been covered by the book on Ampersand [39].

Chapter 4 restructures and extends contents of paper III with more detailed information on the design of RAP and prototype generation functions of Ampersand used to generate RAP.

Chapter 5 has been generated from the RAP model with the Ampersand compiler. The development of RAP is the key contribution of this dissertation.

Chapter 6 has been constructed from the contents of paper II. Sentences and terms have been rephrased to fit the terminology and other statements in this dissertation.

Chapter 7 has been generated with the Ampersand compiler from the Ampersand-model of the measurement framework. The model of the measurement framework is an extension of the RAP model, which contributes to the evidence of the claim that RAP is useful as a platform to validate didactical insights. Section 7.2 is new content added to this dissertation.

Chapter 2

METHOD FOR RULE-BASED PROTOTYPING

Basic knowledge of Ampersand is required to understand the user functions, design and development process of RAP. This chapter gives an introduction to Ampersand. Section 2.1 describes the Ampersand method for requirements engineers to define a system of rules to process information. Students use RAP to practice the Ampersand method. Also, RAP has been developed with the Ampersand method. Section 2.2 gives a schoolbook example for the Ampersand method. Section 2.3 describes the rule-compliant run-time system, which can be generated with the Ampersand compiler from an Ampersand-model. RAP is an example of such a rule-compliant run-time system, which has been generated from the RAP model.

2.1 Rule-based design for requirements engineers

Ampersand is a rule-based design approach on concepts known from the Business Rules Community [32] and Date [5]. The Business Rules Community has argued since a long time that natural language provides a better means for discussing requirements than graphical models (e.g. UML [33]). This vision is the foundation of profound assets like the Business Rules Manifesto [31], the Business Rules Approach [32], the SBVR standard [26] and RuleSpeak [4]. Ampersand goes beyond requirements by formalizing business rules and using them as requirements to partially automate information system development. There is no well-rooted research community that targets rule-based design approaches for requirements engineers as a whole. We find inspiration in the work of Date, which covers ways to develop information systems by compiling declarative rules for the presentation, application and database logic.

Ampersand shares various concepts with a variety of other approaches. Therefore, research on Ampersand relates to and may find inspiration in many other fields of research. We distinguish the following approaches, which we take into account in our research:

- the rule-based approach to model organizations e.g. rule complexes [3].
- the relational approach to model data structures e.g. Object Role Modelling [12], Alloy [14].
- the declarative approach to model data processes e.g. [11].
- the ontological approach to store information e.g. Semantic Web [2].
- the enterprise modelling approach to integrate process and information design e.g. Demo [7].
- the rule-based approach to separate business logic from code e.g. Object Constraint Language [27], active databases [6, 29, 40].
- the model-driven approach to generate code from models e.g. Model Driven Approach [25].

Ampersand uses the word **business rule** to denote a formal representation of a business requirement. An **Ampersand design** holds definitions of business rules in a relation algebra [20]. Rules are defined as expressions on declared binary relations between two concepts. There is an analogy between rules, relations, and concepts in Ampersand and rules, facts and terms from the Business Rules Manifesto.

We believe, supported by Date [5], that rules are sufficient to generate a functional prototype of an information system. Evidence of that claim is given by the Ampersand software generator, which compiles a set of rules (in the form of an Ampersand-model) into working software. The Ampersand generator also produces design artefacts, such as data models. As a consequence, data models need no longer be an input of the design process, but they are design artefacts that can be generated from an Ampersand analysis of the business rules. Having a computer generate data models makes validation of data models by of the business representatives redundant. Instead, discussions with representatives of the business can focus on the intended interpretation of business rules. In this way, Ampersand shifts the focus of the design process to requirements engineering, which we see as a result of automating an important part of the design process.

Controlling design processes directly by means of business rules has consequences for requirements engineers, who will encounter a simplified design process. From

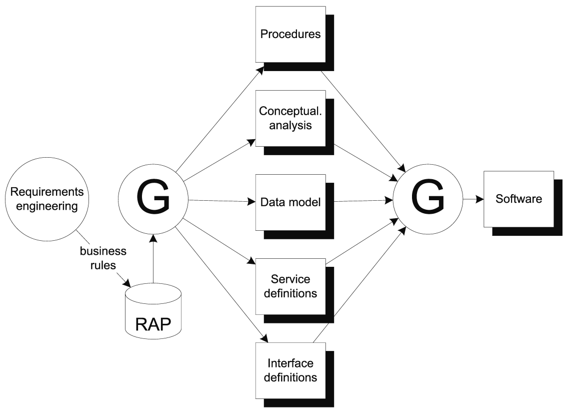


Figure 2.1: Rule-based design process (engineer).

their perspective, the design process is depicted in figure 2.1. The main task of a requirements engineer is to collect rules to be maintained. These rules are to be managed in a repository (RAP). From that point onwards, a first generator (G) produces various design artefacts, such as data models, process models, etc. These design artefacts can then be fed into a second generator that produces a rule-compliant software system. That second generator is typically a software development environment, of which many exist and are heavily used in the industry. Alternatively, the design can be built in the conventional way as a database application. A graphical user interface on the repository and generator functions will help the requirements engineer by storing, managing and checking rules, to generate specifications, analyse rule violations, and validate the design.

From the perspective of an organization, the design process looks like figure 2.2. At the focus of attention is the dialogue between a problem owner and a requirements engineer. The former decides which requirements he wants and the latter makes sure they are captured accurately and completely. The requirements engineer helps the problem owner to make requirements explicit. Ownership of the requirements remains in the business. The requirements engineer can tell with the help of his tools whether the requirements are sufficiently complete and concrete to make a buildable specification. The requirements engineer maintains a clear mapping between the formal requirements specification and the informal business requirements. This mapping is needed to explain the correspondence of the specification to the business for necessities like traceability and validation.

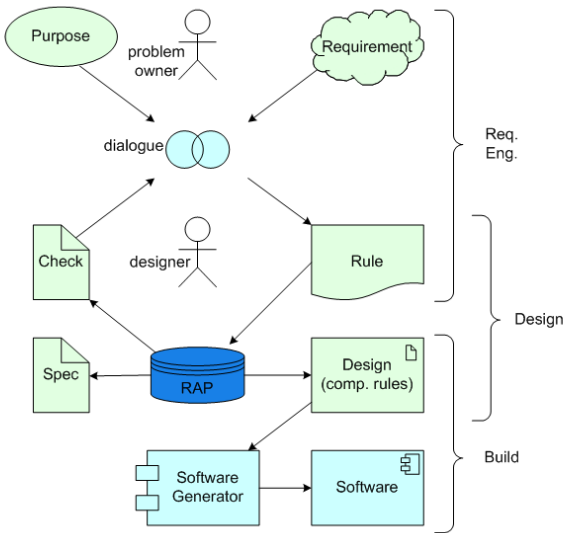


Figure 2.2: Rule-based design process (organization).

2.2 Example

To illustrate how a requirements engineer may use Ampersand, let us produce an Ampersand design by the process in figure 2.2.

In a dialogue between a problem owner and the requirements engineer, the problem owner pronounces a business requirement: ”We use a card for accounts.” The engineer asks: ”Why do you use cards for accounts?” The problem owner answers: ”To identify accounts.”

The requirements engineer needs to capture this requirement in a formal language like Ampersand. In the Ampersand language, which will be introduced in chapter 3, the requirements engineer codes an Ampersand-model in a script, an **Ampersand-script**. This script is a closed context - a relation algebra - of populated binary relations on concepts and declarative rules. A rule is expressed in relation algebra and enforced in a way, e.g. as a constraint on the population of the relations.

To continue the example, the requirements engineer creates a first attempt to formalize the business requirement in Ampersand. For example, the engineer defines a context with one relation `identifies :: Card * Account` and one rule `RULE I |- identifies;identifies˜`.

The engineer attaches the intended meaning to the relation: *a card identifies an account*. Pragmatically this means that an instance of `identifies` e.g.

`("card #1", "5115")` is interpreted as the administrative fact that *card #1 identifies account 5115*. By the meaning given to `identifies`, the rule means that *each card identifies at least one account*.

In an ASCII-encoded Ampersand-script the above formalization looks like:

```
CONTEXT Example

  identifies :: Card * Account
    MEANING IN ENGLISH
    "A card identifies an account."

  POPULATION identifies CONTAINS
     [("card #1","5115")
     ;("card #2","777")
     ;("card #3","7081")]

  RULE  I |- identifies;identifies~
    MEANING IN ENGLISH
    "Each card identifies at least one Account."

ENDCONTEXT
```

The engineer commits this script to the rules repository, RAP. Now, the engineer can use the generator functions as depicted in figure 2.1.

First, the engineer needs to check in a dialogue with the problem owner whether the declarative statements in the script are true and sufficiently complete with respect to the purpose of the requirement. Generator functions can be implemented to present the script in a way that supports the dialogue. For example, the engineer may benefit from support which translates the formalism back to a business language, or support which manages the rules to discuss.

The engineer has designed the requirement ”We use a card for accounts” as ”Each card identifies at least one Account”. The engineer asks: ”Is it true that each card identifies at least one account?” The problem owner says: ”Partially, each card identifies exactly one account.” The engineer instantly adds a rule to the script that means ”Each card identifies at most one Account”. The new version of the script is committed to RAP.

Now, the engineer is confident about the correctness and completeness of the design. The engineer hits a button to fire the generator function that generates a prototype with default user forms to view and change cards linked to accounts. The problem owner and engineer play with the prototype to test its functionality. They try to add a new card linked to a new account, which succeeds. They try to add another account to the same card, which fails. They try to add another card to the account that already has a card, which succeeds. The problem owner says: ”Perfect, you may indeed have two cards for the same account.”

2.3 Rule-compliant run-time system

Whenever and wherever people work together, they connect to one another by making agreements and commitments. These agreements and commitments constitute the rules of the business. A logical consequence is that the business rules must be known and understood by all who have to live by them. From this perspective business rules are the cement that ties a group of individuals together to form a genuine organization. In practice, many rules are documented, especially in larger organizations.

The role of information technology is to help *maintain* business rules. This is what compliance means. If any rule is violated, a computer can signal the violation and prompt people or trigger a computer to resolve the issue. This can be used 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. Rules maintained by people are called **process rules**, because they produce signals for people to act. Rules maintained by computers are called **integrity rules**, because the computer maintains them as invariants of data and processes.

Since all formalized 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 those rules. This establishes compliance. Business process management (BPM) is also included, based on the assumption that 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, which implements Shewhart’s Plan-Do-Check-Act cycle (often attributed to Deming) [35]. Figure 2.3 illustrates the principle. Assume the existence of an electronic infrastructure that contains data collections, functional components, user interface components and whatever is necessary to support the work. An adapter observes the business by drawing digital information from any available source (e.g. a data warehouse, interaction with users, or interaction with information systems). The observations are fed to a rule engine, which checks them against business rules in a repository. If rules are found to be violated, the rule engine 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. This principle rests solely on rules, yielding

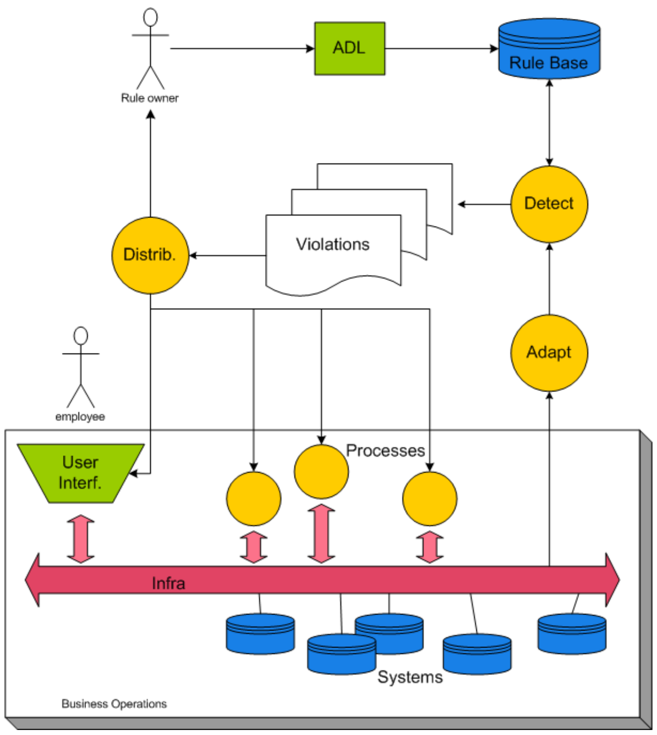


Figure 2.3: Principle of rule-based process management

implicit business processes which directly follow from the rules of the business. In comparison: workflow management derives actions from a workflow model, which models the procedure in terms of actions. Workflow models are built by modellers, who transform the rules of the business into actions and place these actions in the appropriate order to establish the desired result.

2.4 Conclusion

This chapter has introduced the idea that information system software can be derived from functional requirements alone. This idea has been implemented by the method Ampersand. Ampersand has been described from the perspective of a requirements engineer. A requirements engineer uses Ampersand to elicit business rules, formalize those rules and maintain them in a rule repository, RAP. The Ampersand language to define systems of rules has been built on relation algebra and the Business Rules Approach. A requirements engineer processes rules in RAP to develop rule-compliant run-time systems. In such a run-time system, rules can be guards of the integrity of business data and processes (integrity rules), or drivers of business processes and workflows (process rules). An engineer may use the Ampersand compiler to generate rule-compliant run-time systems, validate rules, visualize a system's design, produce functional specifications, and more.

Ampersand plays two roles in this dissertation in which RAP is the central subject. One, RAP is a development tool for Ampersand, which includes the Ampersand compiler and a rule repository as described. Two, RAP has been developed with Ampersand and is a generated rule-compliant run-time system. Thus, RAP could be used to develop RAP. Moreover, the RAP model is being maintained in RAP and committing changes to the rules in the RAP model causes an instant update of RAP itself.

Chapter 3

RULE-BASED LANGUAGE FOR RAP

This chapter describes the Ampersand language, which relates to this dissertation in three ways. One, RAP supports viewing and editing models in the Ampersand language, which yields requirements for RAP. Two, the Ampersand language is used to define the RAP model and the model for RAP's measurement framework. The functional specifications of RAP [chapter 5] and the measurement framework [chapter 7] have been generated from these two models. Three, students of the RBD course need to learn and use the Ampersand language, which characterizes the didactics of Ampersand we study. For example, we have identified the need for proper student feedback on type errors, which has resulted in a feedback system on type errors described in section 3.5.

3.1 Introduction

The Ampersand language is a computer language, that can be compiled into a working information system. It uses a relation algebra [20] as a language in which to express business rules. A tiny example of an Ampersand-script is given here, merely to communicate the flavour of the language. A formal definition is provided in the sequel.

```
PATTERN Hello
 identifies :: Card * Account

 RULE  I |- identifies;identifies~
   MEANING IN ENGLISH
   "Each card identifies at least one Account."
ENDPATTERN
```

Students specify a domain language in terms of binary relations (e.g. "`identifies`"). Students also provide rules. Rules consist of relations that are combined by means of relational algebra operators, such as "`|-`" , "`;`" , and "`~`". The example shows one rule only. It describes a property of a single relation c.q. a multiplicity rule of "`identifies`". Similarly, other rules can be defined to restrict the contents of different relations further. Relation algebras have been studied extensively and are well known for over a century [34]. The use of existing and well described theory brings the benefit of a well conceived set of operators with well known properties.

Students can specify a rule in natural language as well as in formal language. This allows peers to review and scrutinize their work.

Ampersand is a typed formalism. Every relation has a type. For example, the relation `identifies` in the example is given type `Card*Account`. This means that the every tuple $\langle i, b \rangle$ contained in relation `identifier` is a `Card`-`Account` pair, meaning that i is a `Card` and b an `Account`. The interpretation of relations in terms of sets of tuples lies at the heart of Ampersand. Experience so far strongly suggests that this interpretation makes the language concrete in the eyes of students, and therefore understandable.

A type system also means that Ampersand gives feedback to students with respect to type errors. Each expression that is accepted by the type checker makes sense in that it can be used to define the semantics specified in the student's script. A statement that makes no sense is rejected by the type checker, allowing tutors to focus on meaning and helping students to understand it. In a way, the type checker takes over the task of uncovering the "nitty-gritty" mistakes, freeing the tutors to do more intelligent work.

Van der Woude and Joosten [38] have enhanced the type system with subtyping of concepts. Subtyping is useful to confront two different, but comparable concepts without being rejected as a type error. For example, a requirements engineer can model the concept `Client` and `Provider` to be comparable over a more general concept. This might make sense when a client can be a provider as well. A course book on Ampersand [39] also discusses alternative patterns in Ampersand to model apparent subtypes of concepts.

3.2 Syntax

The Ampersand syntax consists of constant symbols for (business) concepts, (business) elements, relations and relation operators. Relation terms can be constructed with relations and relation operators.

Let $\mathbb{C}$ be a set of concept symbols. A concept is represented syntactically by an alphanumeric character string starting with an upper case character. In this paper, we use A, B, and C as concept variables.

Let $\mathbb{U}$ be a set of atom symbols. An atom is represented syntactically by an ASCII string within single quotes. All atoms are an element of a concept, e.g. `'Peter'` is an element of `Person`. We use a, b, and c as atom variables.

Let $\mathbb{D}$ be a set of relation symbols. A relation symbol is represented syntactically by an alphanumeric character string starting with a lower case character. For every $A, B \in \mathbb{C}$, there are special relation symbols, $\mathbb{I}_A$ and $V_{A\times B}$. For every $a \in \mathbb{U}$, there is a special relation symbol, a_A which implies that a is an element of A. We use r, s, and t as relation variables to denote elements of $\mathbb{D}$.

Let $^\smile$, $-$, $\cup$, and ; be relation operators of arity 1, 2, 2, and 2 respectively. The unary relation operator $\neg$ and binary relation operators $\cap$, $\subseteq$ and $\equiv$ are cosmetic and only defined on the interpretation function (see definition 3).

Definition 1 (relation terms).

Let $R, S \in \mathbb{R}, r \in \mathbb{D}$, $A, B \in \mathbb{C}$ and $a \in \mathbb{U}$. Then $\mathbb{R}$ is the set of relation terms, which is defined recursively by

$$r, \mathbb{I}_A, V_{A\times B}, a_A, R_{A\sim B}, R^\smile, R - S, R \cup S, R;S \in \mathbb{R}$$

We use R, S, and T as variables to denote relation terms.

Ampersand uses the notion of *context* to allow a user to distinguish true statements from other statements. Truth is consequently treated within a particular context, which means we have to define the notion of context.

Definition 2 (context).

A context $\mathfrak{C}$ consists of three sets: RUL, REL, *and* POP, *where*

- RUL $\subseteq \mathbb{R}$ *is a collection of relation terms called rule statements.*
- REL *is a collection of triples $r : A \sim B$ in which $r \in \mathbb{D}$ and $A, B \in \mathbb{C}$. Each instance of* REL *is called a relation declaration.*
- POP *is a collection of triples $a\ r\ b$ in which $a \in A, b \in B$ and $(r : A \sim B) \in$* REL. *Each instance of* POP *is called a relation element.*

The relation declarations define the conceptual structure and scope of $\mathfrak{C}$. Each relation element holds in $\mathfrak{C}$ by definition. *POP* is called the *population* of $\mathfrak{C}$ and represents administrative facts. A rule statement is a relation term $R_{A\sim B}$ that must be read as $[R_{A\sim B}]$. $[R_{A\sim B}]$ holds within $\mathfrak{C}$ if and only if $R_{A\sim B} = V_{A\times B}$. By defining a rule statement that must hold at all times, an integrity rule [section 2.3], the requirements engineer puts a constraint on *POP*. By defining a rule statement that may not hold, a process rule [section 2.3], the requirements engineer sets a directive to change *POP*.

An Ampersand-script is a user-defined collection of relation declarations, relation elements and rule statements. It describes a (business) context $\mathfrak{C}$. The Ampersand compiler contains a parser, which extracts *RUL*, *REL*, and *POP* from an Ampersand-script that defines $\mathfrak{C}$.

This dissertation includes Ampersand-scripts written in Ampersand's ASCII-syntax, which is equivalent to but differs from the Ampersand syntax described above. For example, the symbol for $\cup$ is `\/` in the ASCII-syntax. Scripts in the ASCII-syntax are printed verbatim. We use the ASCII-syntax in section 2.2 and 3.1 and in chapter 4, which contains snippets copied from appendix A. For a full description of the ASCII-syntax we refer to the course book [39], which is available online.

3.3 Semantics

The previous section defines the syntactic structure of Ampersand and the interpretation of relation elements and rule statements. This section introduces an interpretation $\mathfrak{I}(R)$ and a type $\mathfrak{T}(R)$, that define the semantics of relation term R. The interpretation function $\mathfrak{I}()$ interprets relation terms based on *POP*. It is a commonly known interpretation of representable heterogeneous relation algebras [20]. The relation symbols used in a relation term are either $\mathbb{I}_A$ or $V_{A\times B}$ or declared by the user in *REL*. So, the relation algebra on $\mathbb{R}$ can be configured by the user through *REL*. The interpretation of all relation symbols in $\mathbb{D}$ is completely user-defined through *POP*. Thus, given some *REL* and some *POP*, $\mathfrak{I}(R)$ determines whether some relation holds between two elements.

Definition 3 (interpretation function).

Given some $\mathfrak{C}$, the interpretation function of relation terms is defined by

$$\begin{aligned}
&\textit{relation} && \mathfrak{I}(r) = \{\langle a,b\rangle \mid a\ r\ b \in \mathrm{POP}\} && (3.1)\\
&\textit{identity} && \mathfrak{I}(\mathbb{I}_A) = \{\langle a,a\rangle \mid a \in A\} && (3.2)\\
&\textit{universal} && \mathfrak{I}(V_{A\times B}) = \{\langle a,b\rangle \mid a \in A,\ b \in B\} && (3.3)\\
&\textit{atom} && \mathfrak{I}(a_A) = \{\langle a,a\rangle\} && (3.4)\\
&\textit{disambiguate} && \mathfrak{I}(R_{A\sim B}) = \{\langle a,b\rangle \mid \langle a,b\rangle \in \mathfrak{I}(R),\ a \in A,\ b \in B\} && (3.5)\\
&\textit{converse} && \mathfrak{I}(R^{\smile}) = \{\langle b,a\rangle \mid \langle a,b\rangle \in \mathfrak{I}(R)\} && (3.6)\\
&\textit{union} && \mathfrak{I}(R \cup S) = \{\langle a,b\rangle \mid \langle a,b\rangle \in \mathfrak{I}(R) \textit{ or } \langle a,b\rangle \in \mathfrak{I}(S)\} && (3.7)\\
&\textit{difference} && \mathfrak{I}(R - S) = \{\langle a,b\rangle \mid \langle a,b\rangle \in \mathfrak{I}(R) \textit{ and } \langle a,b\rangle \notin \mathfrak{I}(S)\} && (3.8)\\
&\textit{composition} && \mathfrak{I}(R;S) = \{\langle a,c\rangle \mid \exists b \textit{ such that } \langle a,b\rangle \in \mathfrak{I}(R) \textit{ and } \langle b,c\rangle \in \mathfrak{I}(S)\} && (3.9)
\end{aligned}$$

Section 3.1 informally described that relations need to have a type. The user must declare the existence of a relation by stating $r : A \sim B$. This declaration means that there is a relation r with type $A \sim B$. The supertype relation $\leq$ between concepts is introduced later in section 3.4. Definition 4 defines the type of relation terms by means of a type function. $\mathfrak{T}(R)$ is inspired on Hattensperger's typing function for heterogeneous relation algebra [13].

Definition 4 (typing function).

Given some $\mathfrak{C}$, the partial typing function of relation terms is defined by

$$\begin{aligned}
\mathfrak{T}(r) &= A \sim B && , \textit{if } r : A \sim B \in \mathrm{REL} && (3.10)\\
\mathfrak{T}(\mathbb{I}_A) &= A \sim A && , \textit{if } A \in \mathbb{C} && (3.11)\\
\mathfrak{T}(V_{A\times B}) &= A \sim B && , \textit{if } A, B \in \mathbb{C} && (3.12)\\
\mathfrak{T}(a_A) &= A \sim A && , \textit{if } a \in A,\ A \in \mathbb{C} && (3.13)\\
\mathfrak{T}(R_{A\sim B}) &= A \sim B && , \textit{if } \mathfrak{T}(R) = \mathtt{A'} \sim \mathtt{B'}, A \leq A' or A' \leq A, B \leq B' or B' \leq B && (3.14)\\
\mathfrak{T}(R^{\smile}) &= B \sim A && , \textit{if } \mathfrak{T}(R) = A \sim B && (3.15)\\
\mathfrak{T}(R \cup S) &= \mathfrak{T}(R) && , \textit{if } \mathfrak{T}(R) = \mathfrak{T}(S) && (3.16)\\
\mathfrak{T}(R - S) &= \mathfrak{T}(R) && , \textit{if } \mathfrak{T}(R) = \mathfrak{T}(S) && (3.17)\\
\mathfrak{T}(R;S) &= A \sim C && , \textit{if } \mathfrak{T}(R) = A \sim B, \mathfrak{T}(S) = B \sim C && (3.18)
\end{aligned}$$

For any relation term R with a defined type, it can be proven that

$$\mathfrak{T}(R) = A \sim B \Rightarrow \mathfrak{I}(R) \subseteq \mathfrak{I}(V_{A\times B})$$

As a consequence, a relation declared as $r : A \sim B$ can only contain elements $a\ r\ b$ for which $a \in A$ and $b \in B$. If a relation term has no type, it is said to have a *type error*. Any script that contains relation terms with a type error is rejected by the Ampersand compiler, accompanied by appropriate error messages as feedback to the user.

From the operators defined in Ampersand, some others can be derived. They are called *cosmetic*, because they have been included in the language for user convenience only.

Definition 5 (cosmetic operators)**.**

$$\textit{complement} \qquad \neg R = V_{A\times B} - R,\ \textit{defined if}\ \mathfrak{T}(R) = A \sim B \tag{3.19}$$

$$\textit{intersection} \qquad R \cap S = \neg(\neg R \cup \neg S) \tag{3.20}$$

$$\textit{implication} \qquad R \subseteq S = \neg R \cup S \tag{3.21}$$

$$\textit{equivalence} \qquad R \equiv S = (R \subseteq S) \cap (S \subseteq R) \tag{3.22}$$

3.4 Type System

Only type correct expressions are allowed in Ampersand-scripts. Ampersand allows overloading of relation symbols. This means that relation term names need not be unique. Overloading is necessary for practical reasons only, in order to give users more freedom to choose names. In particular, requirements engineers may choose short uniform names, such as *aggr*, *has*, *in*, to represent different relations for different types. The type system deduces all possible types for any given relation term name. If there are multiple possibilities, a type error is given. The script writer can disambiguate any relation term name by adding type information explicitly.

Ampersand also allows that two different concepts overload an atom symbol, i.e., an $a \in A$ has an interpretation different from $a \in B$. Atoms are only interpreted in the context of typed relation terms, preventing an ambiguous identity of atoms. In practice, business concepts may be overlapping, e.g., `'Peter'` the `Person` has the same practical identity as `'Peter'` the `Auctioneer`. Details are provided in [38], a conference paper in which van der Woude and Joosten embed generalization of concepts in Ampersand. They introduce supertype relations ($\leq$), called embeddings, to define total extensions of the partial (heterogeneous) relational operators except for the complement. These extensions give a requirements engineer the possibility to express sub- and superconcepts. The supertype relation between subconcept A and superconcept B is expressed as $\leq : A \sim B \in REL$.

The use of negation, which is problematic in heterogeneous relation algebra [38], is allowed in Ampersand as syntactic sugar for the difference from the total relation. If Ampersand can derive from the script that $\mathfrak{T}(R) = A \sim B$, the unary operator $\neg R$ may be used as an abbreviation of $V_{A\times B} - R$.

Altogether, the problem that the type system must solve is twofold. The type system must deduce one relation term from a relation term name, and check the type of relation terms simultaneously. We use R', S', T' as variables to denote relation term names.

The type function $\mathfrak{T}'(R')$ is based on the partial typing function $\mathfrak{T}(R)$. The type system examines $\mathfrak{T}'(R')$ to mark the name R' as bound to a relation, ambiguous, undeclared or undefined.

- If some r is not declared with a type in *REL*, then r is said to be *undeclared.*
- If some R' can only refer to one type correct relation term R, then R' is said to *bind to R*.
- If some R' does not refer to any type correct relation term, then R' is said to be *undefined*.
- If some R' can refer to more than one type correct relation term, i.e., an *alternative*, then R' is said to be *ambiguous*.

Definition 6 (type function)**.**

Given some $\mathfrak{C}$, the type function of relation term names is defined by

$$\mathfrak{T}'(r) = \{A \sim B \mid (r : A \sim B) \in \text{REL}\ \} \tag{3.23}$$
$$\mathfrak{T}'(\mathbb{I}_A) = \{A \sim A\} \tag{3.24}$$
$$\mathfrak{T}'(V_{A\times B}) = \{A \sim B\} \tag{3.25}$$
$$\mathfrak{T}'(a_A) = \{A \sim A\} \tag{3.26}$$
$$\mathfrak{T}'({R'}_{A\sim B}) = \mathfrak{T}'(R') \cap \{A \sim B\} \tag{3.27}$$
$$\mathfrak{T}'(\neg R') = \mathfrak{T}'(R') \tag{3.28}$$
$$\mathfrak{T}'(R'^{\smile}) = \{B \sim A | A \sim B \in \mathfrak{T}'(R')\} \tag{3.29}$$
$$\mathfrak{T}'(R' \cup S') = \mathfrak{T}'(R') \cap \mathfrak{T}'(S') \tag{3.30}$$
$$\mathfrak{T}'(R';S') = \{A \sim C \mid \sharp\{B \mid A \sim B \in \mathfrak{T}'(R'), B \sim C \in \mathfrak{T}'(S')\} = 1\} \tag{3.31}$$

Rule 3.27 enables the requirements engineer to disambiguate an ambiguous relation term name with type information. Rule 3.31 ensures that if $R';S'$ yields more than one alternative with type $A \sim C$, then $A \sim C \notin \mathfrak{T}'(R';S')$.

This type system is not monotone, i.e., a well typed program can become ill-typed by adding rules for different, independent relations. When this happens to a requirements engineer, he will notice that the introduction of a new relation forces him to add more type information to other parts in his script.

3.5 Feedback system

The quality of feedback on type errors deserves attention, because it lets students focus on learning rule-based design. The type system is embedded in the feedback system of the Ampersand compiler. The feedback system must give an error message if and only if a relation term name cannot be bound to a relation term. The error messages of the compiler must be concise, i.e., correct and kept short, precise and relevant. Conceptual diagrams are introduced in subsection 3.5.3, which are a visual aid to read and analyse error messages.

The feedback system is implemented in Haskell [30] based on an attribute grammar [9] Haskell library developed at Utrecht University [37]. The feedback system checks all relation term names R' used in any statement in *RUL* or *POP*, given the conceptual structure of $\mathfrak{C}$ represented by *REL*. *REL* is an inherited attribute called *context structure*. Within the scope of a statement, given the context structure, R' is bound, ambiguous, undeclared or undefined.

A bound R' implies the existence of one suitable alternative for any subname S' of R'. The reader may verify this surjective function from subname S' to bound R' in rule 3.23-3.31. However, $\mathfrak{T}'(S')$ may yield more than one alternative. The feedback system uses two attributes to determine the type of any S' within the scope of R'. $\mathfrak{T}'(S')$ is a synthesized attribute called *pre-type*. Some $X \in \mathfrak{T}'(S')$ is an inherited attribute called *automatic type directive*. The automatic type directive is based on Definition 6.

For example, consider two statements a rule and a relation element, both yielding relation *rel1*:

$$\begin{aligned} \mathit{rel1} \cup \mathit{rel2} &\in \mathit{RUL} \\ \texttt{'atom1'}\ \mathit{rel1}\ \texttt{'atom2'} &\in \mathit{POP} \end{aligned}$$

where

$$\mathit{REL} = \{\mathit{rel1} : \texttt{Cpt1} \sim \texttt{Cpt2}, \mathit{rel1} : \texttt{Cpt1} \sim \texttt{Cpt3}, \mathit{rel2} : \texttt{Cpt1} \sim \texttt{Cpt2}\}$$

The rule binds to a typed relation term, although $\mathfrak{T}'(rel1)$ yields more than one alternative:

$$\begin{aligned}
\mathfrak{T}'(\mathit{rel1}) &= \{\texttt{Cpt1} \sim \texttt{Cpt2},\ \texttt{Cpt1} \sim \texttt{Cpt3}\} && \text{(rule 3.23, ambiguous)}\\
\mathfrak{T}'(\mathit{rel2}) &= \{\texttt{Cpt1} \sim \texttt{Cpt2}\} && \text{(rule 3.23, bound)}\\
\mathfrak{T}'(\mathit{rel1} \cup \mathit{rel2}) &= \mathfrak{T}'(\mathit{rel1}) \cap \mathfrak{T}'(\mathit{rel2}) && \text{(rule 3.30)}\\
&= \{\texttt{Cpt1} \sim \texttt{Cpt2},\ \texttt{Cpt1} \sim \texttt{Cpt3}\} \cap \{\texttt{Cpt1} \sim \texttt{Cpt2}\}\\
&= \{\texttt{Cpt1} \sim \texttt{Cpt2}\} && \text{(bound)}\\
&= \mathfrak{T}'((\mathit{rel1} \cup \mathit{rel2})_{\texttt{Cpt1}\sim\texttt{Cpt2}}) && \text{(auto)}\\
&= \mathfrak{T}'((\mathit{rel1}_{\texttt{Cpt1}\sim\texttt{Cpt2}} \cup \mathit{rel2}_{\texttt{Cpt1}\sim\texttt{Cpt2}})_{\texttt{Cpt1}\sim\texttt{Cpt2}}) && \text{(auto)}\\
\mathfrak{T}'(\mathit{rel1}_{\texttt{Cpt1}\sim\texttt{Cpt2}}) &= \{\texttt{Cpt1} \sim \texttt{Cpt2}\} && \text{(rule 3.23, bound)}\\
\mathfrak{T}'(\mathit{rel2}_{\texttt{Cpt1}\sim\texttt{Cpt2}}) &= \{\texttt{Cpt1} \sim \texttt{Cpt2}\} && \text{(rule 3.23, bound)}\\
& && \text{(done)}
\end{aligned}$$

The relation element is ambiguous, because $\mathfrak{T}'(\mathit{rel}1)$ yields more than one alternative:

$$\begin{aligned}
\mathfrak{T}'(\mathit{rel1}) &= \{\texttt{Cpt1} \sim \texttt{Cpt2},\ \texttt{Cpt1} \sim \texttt{Cpt3}\} && \text{(rule 3.23, ambiguous)}\\
& && \text{(done)}
\end{aligned}$$

The requirements engineer should have specified a type directive

$$\texttt{'atom1'}\ \mathit{rel1}_{\texttt{Cpt1}\sim\texttt{Cpt2}}\ \texttt{'atom2'} \in \mathit{POP}$$

or

$$\texttt{'atom1'}\ \mathit{rel1}_{\texttt{Cpt1}\sim\texttt{Cpt3}}\ \texttt{'atom2'} \in \mathit{POP}$$

Detailed information can be obtained through synthesized attributes of the attribute grammar. Three synthesized attributes are defined in our implementation. One attribute holds either $\mathfrak{T}(R)$ if R' is bound to R and $\mathfrak{T}(R)$ is defined, or an error message as described in Section 3.5.1 otherwise. Another attribute holds a fully typed relation term name $R_{\mathfrak{T}(R)}$ if the previous attribute holds $\mathfrak{T}(R)$, and is undefined otherwise. The third attribute holds an extensive LaTeX report for complete detail on type errors or binding relation term names. Such a report contains equational traces like presented for the examples in this section. This report is generated by the Ampersand compiler.

3.5.1 Error messages

If a relation term has an error, then an error message must be composed. We have designed templates for error messages which relate to the type system rule at which an error first occurs. The error messages are defined short but complete and specific. The templates are presented in the same order of precedence as the operators they relate to. If subterms of a relation term have error messages, then the error message of the relation term is the union of all the error messages of subterms.

relation undeclared If $\mathfrak{T}'(r)$ yields no types, then there is no relation declaration for r.

relation undeclared: r

relation undefined Requirements engineers may use the unique name of some R e.g. R'_X. This may cause different kinds of errors which are checked in the same chronological order as described here.

If $X = A \sim B$ and A or B does not occur in the type signature of any relation declaration, then there is probably a typo in the concept name.

unknown concept: A, or

unknown concept: B, or

unknown concepts: A and B

If $X \notin \mathfrak{T}'(R')$, then there is no R such that $\mathfrak{T}(R) = X$. If $R' = r$ then r_X is undeclared.

relation undeclared: r_X

In all other cases:

relation undefined: r_X

possible types are: $\mathfrak{T}'(R')$

incompatible/ambiguous composition Let $\mathfrak{T}'(R')$ and $\mathfrak{T}'(S')$ yield alternatives. If $\mathfrak{T}'(R';S')$ yields no types, then there is no alternative for $R';S'$. In case $\sharp\{B \mid A \sim B \in \mathfrak{T}'(R'), B \sim C \in \mathfrak{T}'(S')\} = 0$, then R' and S' are incompatible for composition.

incompatible composition: $R';S'$

possible types of R': $\mathfrak{T}'(R')$

possible types of S': $\mathfrak{T}'(S')$

In case $\sharp\{B \mid A \sim B \in \mathfrak{T}'(R'), B \sim C \in \mathfrak{T}'(S')\} > 1$, then the composition of R' and S' is ambiguous.

ambiguous composition: $R';S'$

possible types of R': $\{A \sim B \mid A \sim B \in \mathfrak{T}'(R'), B \sim C \in \mathfrak{T}'(S')\}$

possible types of S': $\{B \sim C \mid A \sim B \in \mathfrak{T}'(R'), B \sim C \in \mathfrak{T}'(S')\}$

incompatible comparison Let $\mathfrak{T}'(R')$ and $\mathfrak{T}'(S')$ yield alternatives. If $\mathfrak{T}'(R' \cup S')$ yields no types, then R' and S' are incompatible for comparison.

incompatible comparison: $R' \cup S'$

possible types of R'*:* $\mathfrak{T}'(R')$

possible types of S'*:* $\mathfrak{T}'(S')$

ambiguous type If R' is ambiguous within the scope of a statement, then the ambiguity is reported as an error.

ambiguous relation: R'

possible types: $\mathfrak{T}'(R')$

3.5.2 Demonstration

Let us demonstrate a typical constraint that some relation is contained within another relation.

$$(\mathit{rel1} \cap \mathit{rel2} \subseteq \mathit{rel0}) \in \mathit{RUL}$$

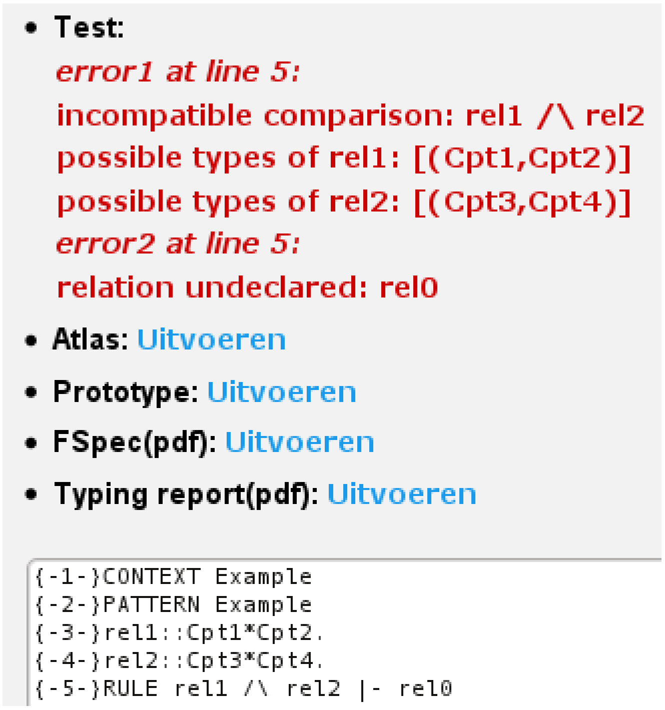
where

$$\begin{aligned}
\mathit{REL} &= \{\mathit{rel1} : \texttt{Cpt1} \sim \texttt{Cpt2}, \mathit{rel2} : \texttt{Cpt3} \sim \texttt{Cpt4}\} \\
R \cap S &= \neg(\neg R \cup \neg S) \\
R \subseteq S &= \neg R \cup S
\end{aligned}$$

The rule contains two errors. Both will be mentioned in the error message as depicted in Figure 3.1.

These messages provide the requirements engineer with relevant and sufficient information which they understand. For complete detail, the requirements engineer requests a LaTeXreport containing a trace like:

$$\begin{aligned}
\mathfrak{T}'(\mathit{rel1}) &= \{\texttt{Cpt1} \sim \texttt{Cpt2}\} && \text{(rule 3.23, bound)}\\
\mathfrak{T}'(\neg\mathit{rel1}) &= \mathfrak{T}'(\mathit{rel1}) && \text{(rule 3.28)}\\
&= \{\texttt{Cpt1} \sim \texttt{Cpt2}\} && \text{(bound)}\\
\mathfrak{T}'(\mathit{rel2}) &= \{\texttt{Cpt3} \sim \texttt{Cpt4}\} && \text{(rule 3.23, bound)}\\
\mathfrak{T}'(\neg\mathit{rel2}) &= \mathfrak{T}'(\mathit{rel2}) && \text{(rule 3.28)}\\
&= \{\texttt{Cpt3} \sim \texttt{Cpt4}\} && \text{(bound)}\\
\mathfrak{T}'(\neg\mathit{rel1} \cup \neg\mathit{rel2}) &= \mathfrak{T}'(\neg\mathit{rel1}) \cap \mathfrak{T}'(\neg\mathit{rel2}) && \text{(rule 3.30)}\\
&= \{\} && \text{(undefined)}\\
\mathfrak{T}'(\mathit{rel0}) &= \{\} && \text{(rule 3.23, undeclared)}\\
& && \text{(done)}
\end{aligned}$$



- **Test:**
  error1 at line 5:
  incompatible comparison: rel1 /\ rel2
  possible types of rel1: [(Cpt1,Cpt2)]
  possible types of rel2: [(Cpt3,Cpt4)]
  error2 at line 5:
  relation undeclared: rel0
- **Atlas:** Uitvoeren
- **Prototype:** Uitvoeren
- **FSpec(pdf):** Uitvoeren
- **Typing report(pdf):** Uitvoeren

```
{-1-}CONTEXT Example
{-2-}PATTERN Example
{-3-}rel1::Cpt1*Cpt2.
{-4-}rel2::Cpt3*Cpt4.
{-5-}RULE rel1 /\ rel2 |- rel0
```

Figure 3.1: compiler screen snippet with type error (Dutch)

3.5.3 Conceptual diagrams

Ampersand makes use of conceptual diagrams, which the Ampersand compiler constructs from a context using the program neato of Graphviz[1]. A conceptual diagram in Ampersand is a simple visualization of typed relations defined in a context. The requirements engineer can see in the diagram which relations are defined and how they connect over their types. This kind of easy-to-access information makes a conceptual diagram useful for the engineer to read and write type-correct relation expressions. Or, in case the type error has already been made, to understand the error message and solve the mistake.

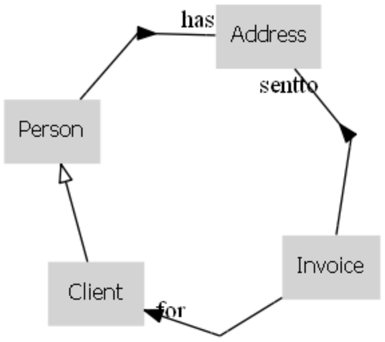


Figure 3.2: Example of conceptual diagram.

A conceptual diagram is constructed as follows. Take a subset of *REL*, for example *REL* itself. For each $r : A \sim B \in REL$ draw a node for the source A and target B if has not been drawn yet. And draw a vector r from the source node to the target node. The vector of a functional relation has a filled arrow head, which is connected to the target node. The vector of a supertype relation has an open arrow head, which is connected to the target node. The vector of any other relation has a filled arrow head, which is placed in the middle of the vector and points to the target node.

For example, figure 3.2 is the conceptual diagram of a context where $REL = \{has : \texttt{Person} \sim \texttt{Address}, for : \texttt{Invoice} \rightarrow \texttt{Client}, sentto : \texttt{Invoice} \sim \texttt{Address}, \leq : \texttt{Client} \sim \texttt{Person}\}$. It is likely that a requirements engineer prefers the diagram over the above definition of *REL* to determine that the following rule is type-correct: $sentto \subseteq for;has$.

[1]`http://www.graphviz.org/`

3.6 Conclusion

In order to make things work, it is imperative that the Ampersand language is well-defined. The main requirement for the Ampersand language is that it is easy to learn by requirements engineers, but sufficiently expressive to define real-life information systems.

To meet this requirement, the Ampersand language adopts relation algebra and respects the statements in the Business Rules Manifesto of the Business Rules Approach. A requirements engineer needs to learn a handful of relational operators and syntactic elements to define an Ampersand-model. The learning challenge that remains is how to define a meaningful Ampersand-model, one that formalizes the business rules of a business context. For that challenge, a requirements engineer needs to learn how to translate an Ampersand-model of concepts, relations and rules to a business language of terms, facts and business rules.

The learning challenge turns out to be significant, which we have concluded from unpublished student evaluations and a study on student behaviour in RAP [23].

The formal character of the Ampersand language allows us to address the learning challenge by means of feedback to the requirements engineer. The kinds of feedback presented in this chapter are conceptual diagrams and feedback on type errors. A conceptual diagram gives the requirements engineer a visual overview over the relations he has declared. A conceptual diagram uses the types of the relations to relate them to each other. An Ampersand-model must be free of type errors to be meaningful. We can guarantee that a requirements engineer gets feedback on each type error.

In the end of this dissertation, we will have proven that the Ampersand language is sufficiently expressive to define real-life information systems, namely by the case of RAP.

Chapter 4

DESIGN OF RAP

This chapter presents the design of our development environment for Ampersand that targets students doing design exercises. The design of RAP is not only a product of what it should be, namely a development environment for Ampersand, but also how it should be realized, namely being generated with the Ampersand prototype generation function from the Ampersand-model for RAP. Recall that we refer to this model as the RAP model. The requirement on how to realize RAP means that what-requirements must find their implementation in a combination of prototype generation function and rules in the RAP model.

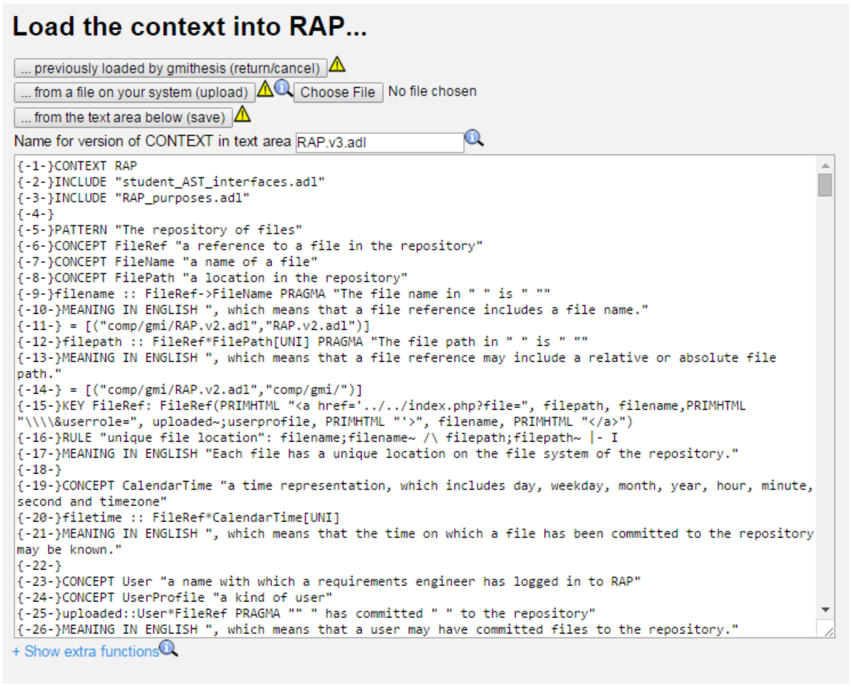
This chapter first presents the logical design of RAP. Next, we present how RAP has been built on Ampersand generated software components. The Ampersand prototype generation function subject to this chapter has been implemented in version 711-2191 of the Ampersand compiler and the RAP model of the second student release of RAP (**RAPv2**). After that, we elaborate on different parts of the generation function to describe how RAP has been generated from its specification in the Ampersand language. Examples of the RAP specification have been used. The entire RAP specification can be found in the next chapter, chapter 5.

4.1 Logical design of RAP

RAP consists of a repository of Ampersand-scripts on a file system, an Ampersand compiler to process Ampersand-scripts, a MySql database, PHP web pages with process and presentation logic on that database and one PHP web page to get access to RAP and upload Ampersand-scripts to the repository.

A web user of RAP starts on the above mentioned **upload page**, see figure 4.1 for a screen shot. The user uses one of the options to select an Ampersand-script to create in RAP.

After committing the selected script, that script is added to the repository. The user cannot edit or delete files in the repository, such that RAP keeps track of all versions of a script that a user commits to RAP.


Load the context into RAP...
... previously loaded by gmithesis (return/cancel)
... from a file on your system (upload)
Choose File
No file chosen
... from the text area below (save)
+ Show extra functions

Name for version of CONTEXT in text area RAP.v3.adl

```
CONTEXT RAP
INCLUDE "student_AST_interfaces.adl"
INCLUDE "RAP_purposes.adl"

PATTERN "The repository of files"
CONCEPT FileRef "a reference to a file in the repository"
CONCEPT FileName "a name of a file"
CONCEPT FilePath "a location in the repository"
filename :: FileRef->FileName PRAGMA "The file name in " " is " ""
MEANING IN ENGLISH ", which means that a file reference includes a file name."
 = [("comp/gmi/RAP.v2.adl","RAP.v2.adl")]
filepath :: FileRef*FilePath[UNI] PRAGMA "The file path in " " is " ""
MEANING IN ENGLISH ", which means that a file reference may include a relative or absolute file path."
 = [("comp/gmi/RAP.v2.adl","comp/gmi/")]
KEY FileRef: FileRef(PRIMHTML "<a href='../../index.php?file=", filepath, filename,PRIMHTML "\\\\&userrole=", uploaded~;userprofile, PRIMHTML "'>", filename, PRIMHTML "</a>")
RULE "unique file location": filename;filename~ /\ filepath;filepath~ |- I
MEANING IN ENGLISH "Each file has a unique location on the file system of the repository."

CONCEPT CalendarTime "a time representation, which includes day, weekday, month, year, hour, minute, second and timezone"
filetime :: FileRef*CalendarTime[UNI]
MEANING IN ENGLISH ", which means that the time on which a file has been committed to the repository may be known."

CONCEPT User "a name with which a requirements engineer has logged in to RAP"
CONCEPT UserProfile "a kind of user"
uploaded::User*FileRef PRAGMA "" " has committed " " to the repository"
MEANING IN ENGLISH ", which means that a user may have committed files to the repository."
```

Figure 4.1: Screen shot of upload page

The web pages of RAP let a user browse through the repository - figure 4.3 for a screen shot - and through the details of a script interpreted as an Ampersand-model, see for example figure 4.6, which is a screen shot that displays the details of a relation declaration $r : A \sim B$ from some script. To have data content on the web pages, the user needs to load a script in the repository into the database of RAP. Loading a script means that RAP feeds that script to a function of the Ampersand compiler, which produces all data content for the web pages in the database of RAP. A script is automatically loaded when added to RAP. Scripts already in the repository can be reloaded through a so-called *operation* named *loading into Atlas*, see figure 4.3.

Likewise, any compiler function can be added to RAP as an operation on Ampersand-

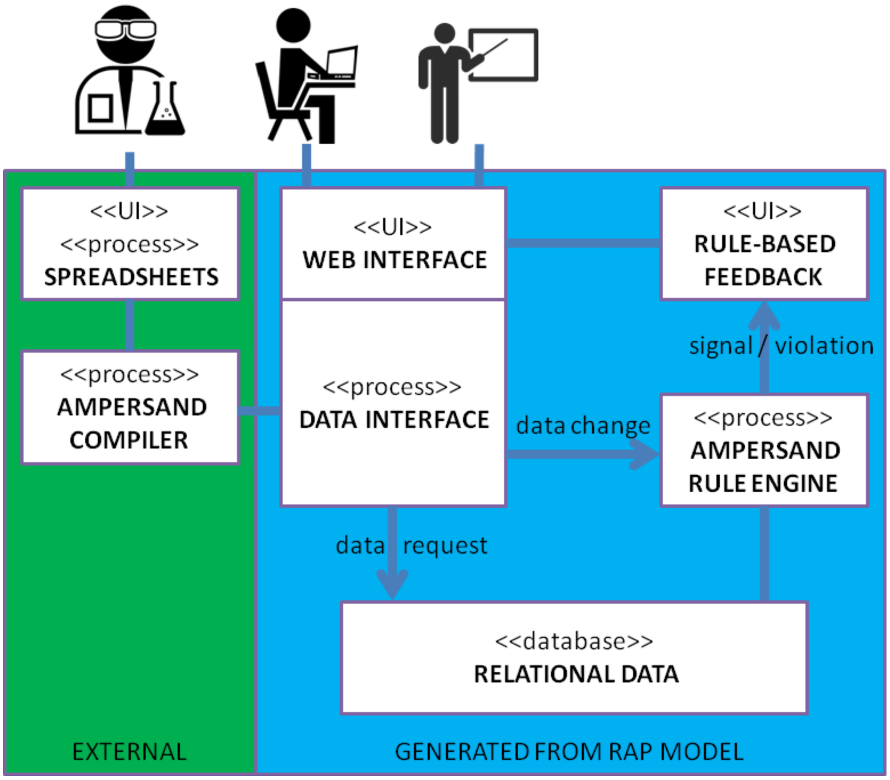


Figure 4.2: Software design of RAP

scripts in the repository e.g. generate a specification document or generate a prototype. Section 4.7 explains how this has been done. To adapt to the experience level of the user, RAP distinguishes three user roles - beginner, advanced student and professional designer - each with access to a certain set of operations.

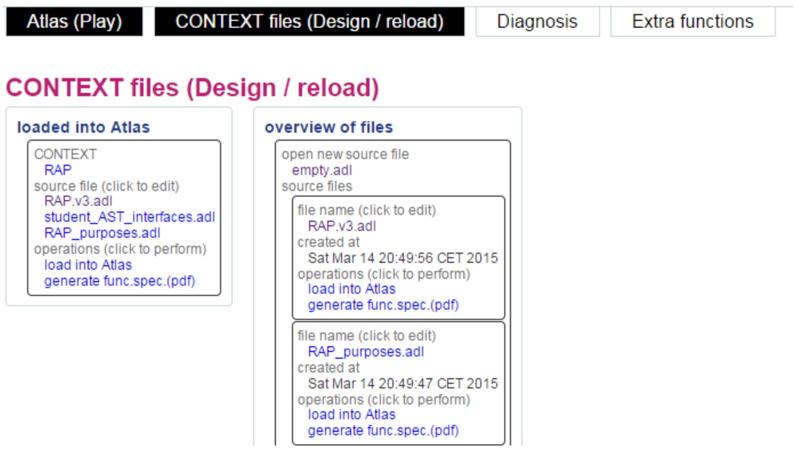
Summarizing, RAP features the following user functions:

- Upload Ampersand-scripts to the repository
- Run a compiler function on a script in the repository and navigate to the function result.
- Load a script from the repository as an Ampersand-model in the database

- Edit the population of an Ampersand-model
- Get feedback on rule violations in an Ampersand-model
- Get feedback on script errors
- Browse through data objects in the database

Data objects in the database include:

- Most of the objects in an Ampersand-script e.g. relations, the meaning of relations, rules, the Ampersand expression of rules, etcetera.
- The entries in the repository c.q. files with Ampersand-scripts
- Data about these entries e.g. script errors, date of creation, version of Ampersand compiler at date of creation, etcetera.
- (Derived) objects of an Ampersand model e.g. the relation type of rules, conceptual diagrams, rule violations, etcetera.
- A few metrics on an Ampersand-model, that is, the number of rules, relations and concepts in a model.



CONTEXT files (Design / reload)

loaded into Atlas

CONTEXT
RAP
source file (click to edit)
RAP.v3.adl
student_AST_interfaces.adl
RAP_purposes.adl
operations (click to perform)
load into Atlas
generate func.spec.(pdf)

overview of files

open new source file
empty.adl
source files

file name (click to edit)
RAP.v3.adl
created at
Sat Mar 14 20:49:56 CET 2015
operations (click to perform)
load into Atlas
generate func.spec.(pdf)

file name (click to edit)
RAP_purposes.adl
created at
Sat Mar 14 20:49:47 CET 2015
operations (click to perform)
load into Atlas
generate func.spec.(pdf)

Figure 4.3: Screen shot of view on repository

4.2 Design of Ampersand-generated prototypes

The previous section presented the logical design of RAP. Each one of the remaining sections in this chapter describes an aspect of how the Ampersand prototype generation function has been used to implement that design. This section 4.2 describes RAP as a system built on an Ampersand-generated prototype. Section 4.3 describes how some of the user functions have been implemented by means of process rules defined in the RAP model. Section 4.4 describes how data content on the web pages is implemented in the database of RAP; and how integrity rules defined in the RAP model are enforced upon that database. Section 4.5 describes how the presentation and process logic in the web pages is generated from so-called interfaces that have been defined in the RAP model. Section 4.6 describes how the layout of the web pages has been implemented. Section 4.7 describes how the Ampersand compiler has been integrated in the interfaces of RAP.

Figure 4.2 represents RAP as a system on an Ampersand-generated prototype. An Ampersand-generated prototype - or just *prototype* - consists of three components: a relational database, so-called **interfaces** and the Ampersand rule engine.

Each interface is a web page that consists of a presentation layer, called the **web interface**, and a process layer, called the **data interface**. The presentation of rule-based feedback is accessible on all web pages by means of two text boxes: one grey/red box for feedback on integrity rules (violations) - see figure 4.5 for screen shot, one yellow/black box for feedback on process rules (signals) - see figure 4.4 for screen shot. The Ampersand rule engine detects all violations and signals for all the rules defined in the RAP model.

Interfaces and process rules are defined for one or more types of users in RAP. Integrity rules apply to all types of users in RAP. RAP distinguishes one type of user, namely requirements engineers in Ampersand e.g. students. We could have defined interfaces and process rules for other types of users like teachers who need to examine the designs made by students. Another type of user could have been the analyst of data in RAP.

The execution of metrics on data in RAP to obtain the measurement results used for the research described in chapter 6 has still been implemented in external tools connected to RAP, namely a combination of visualisation functions in spreadsheets and calculation functions in the Ampersand compiler.

The Ampersand compiler has been connected to RAP to support Ampersand as illustrated in figure 2.1. In that figure RAP is a repository of rules on which generation functions are executed to produce design artefacts and prototype software. The Ampersand compiler implements such functions.

One of the compiler functions is the function to load an Ampersand-script from the repository into the RAP database. This function takes the RAP model and the selected Ampersand-script to regenerate the RAP prototype - database, interfaces and rule engine - with a new data population that includes information about the loaded Ampersand-script.

In a next version of RAP, metrics can be defined in the RAP model using the RAP extension presented in chapter 7. Each metric can be executed on and implemented as a function in the Ampersand compiler

4.3 Rule-based processes

RAP implements some of the user functions by means of process rules defined in the RAP model. We say that such a user function is a **rule-based process** or activity in RAP.

A rule-based process is built on rule-based feedback or signals, which guides a user through that process or activity. Figure 4.4 shows how RAP gives signals to a user, that is, in a yellow/black box which appears on all user interfaces. Each signal is constructed from one violation of a process rule. The Ampersand rule engine detects all violations of all process rules. The signal is visible for as long as the violation exists. A signal contains a message, prompting the user to act.

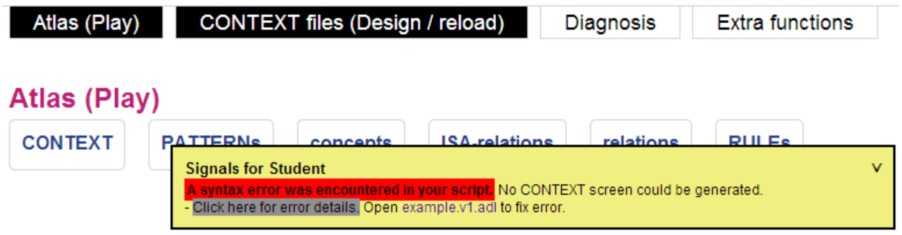
For example, a process rule is that a script error may not exist. If a script error exists then that is a violation of the rule. The corresponding signal contains the error message and prompts the user to solve the error. Figure 4.4 shows a signal for a script error.

The following rule-based processes exist in RAP:

- Solving script errors [section 5.4]. A script may contain syntax or type errors, which need to be solved. RAP gives rule-based feedback to help a user to solve script errors.

- Testing the rules in an Ampersand-model [section 5.15]. Rules are tested against a population. The violations resulting from that test are reported to the user. Such a violation report serves various activities of a requirements engineer e.g. validate rules or simulate a business scenario.

- Committing the changes made through the interfaces of RAP [section 5.16]. A user can use the interfaces to make certain changes to the data in the database. The user must decide when to commit these changes to a new entry in the RAP repository. RAP reports on the changes made by a user and gives directions to commit or cancel those changes.



Signals for Student

A syntax error was encountered in your script. No CONTEXT screen could be generated.

- Click here for error details. Open example.v1.adl to fix error.

Figure 4.4: Screen shot of signal

4.4 Rule-based database

The RAP model defines the database of RAP by means of relations and integrity rules in the RAP model. The data in that database is the population of the RAP model. Like all Ampersand-models, the integrity rules in the RAP model are enforced upon that population.

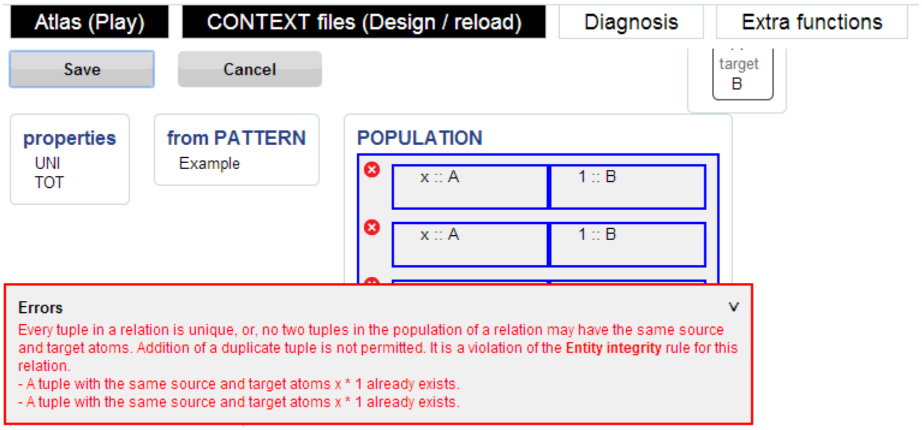
Integrity rules are like process rules, but enforced more strictly. That is, if a user tries to commit data to the database, which results in any violation of any integrity rule, then that transaction will be blocked. The Ampersand rule engine detects all violations of all integrity rules. The user receives a signal when a transaction has been blocked. Figure 4.5 shows how RAP sends such a signal to a user, that is, in a grey/red box which appears on all user interfaces. The user can either cancel the transaction or try to make another change.

4.4.1 Relational data

The database holds relational data, that is, the population of relations in a script. Earlier versions of the Ampersand compiler derived a simple database model, where each relation links to a table with two columns and each record is an element of the population of that relation. The current compiler takes integrity rules into account to derive a more complex database model, a rule-based structure. A population in a script serves as an initial population of a generated database, provided that that population violates none of the integrity rules.

RAP needs to administer data for the sake of user activities in RAP. To summarize these activities, RAP is used by:

- (student) requirements engineers as a design exercise tool. The user has an interface to upload an Ampersand-script to RAP, user interfaces to work on a script in the RAP repository, and access to Ampersand compiler functions to



Errors

Every tuple in a relation is unique, or, no two tuples in the population of a relation may have the same source and target atoms. Addition of a duplicate tuple is not permitted. It is a violation of the **Entity integrity** rule for this relation.
- A tuple with the same source and target atoms x * 1 already exists.
- A tuple with the same source and target atoms x * 1 already exists.

Figure 4.5: Screen shot of blocked transaction

run on a script. A user profile gives access to certain compiler functions. Three user profiles have been configured in RAP: a student, an advanced student and a regular requirements engineer. The user can only edit the population of an Ampersand-model through user interfaces;

- researchers as a data source for measurements to study student behaviour like the study published [23] and described in chapter 6.

The following sets of data are administered in RAP:

Details of Ampersand-model RAP analyses the text of an Ampersand-script, and stores the individual elements (such as relations, concepts, rules, etc.) and the relations between them into the database. Thus, the data dictionary of the RAP database contains a metamodel of an Ampersand-model that originates from an Ampersand-script. User interfaces have been defined to give the user a structured view on an Ampersand-model. The relational structure of an Ampersand-model has been derived from the Ampersand compiler code. This compiler, being coded in Haskell, has a transparent structure which was easy to map to the relations in the RAP-model. For example, the data structure `Concept` in Haskell has an attribute `cptos` of type `[String]`, which has been mapped to a relation *cptos* : `Concept` $\sim$ `AtomID` in the RAP-model.

Relations for process rules Relations have been declared to define the process rules and signal messages for those rules [section 4.3]. For example, a relation from a script to a parse error (*parseerror* : `FileRef` $\sim$ `ParseError`) is needed to express the process rule that parse errors need to be solved by the user ($\neg$*parseerror*). The signal message for this rule requires relations that hold the details of an error e.g. the file position of the error (*pe_position* : `ParseError`$\rightarrow$`String`). In this example, error details are shown on a user interface after clicking the link "Click here for error details" in the signal message, see figure 4.4.

Design of RAP repository Ampersand-script files in the RAP repository are the source of data and the subject of activities. Relations exist in the RAP-model to define that, a file contains an Ampersand-model (*sourcefile* : `Context`$\rightarrow$`AdlFile`); a compiler function runs on a file (*applyto* : `CCommand`$\rightarrow$`AdlFile`); a user commits a file to the RAP repository (*uploaded* : `User` $\sim$ `FileRef`); etcetera. These relations constitute the foundation of RAP, which is a repository of files with compiler functions on those files.

Metrics There is a measurement framework for RAP, which can be used to define metrics and store measurement results in RAP. Section 7.2 describes a design pattern on the measurement framework [chapter 7] to define a metric and store the results in RAP. Three metrics, which count concepts, relations and rules, have been defined in the RAP model using this design pattern. Through user interfaces, a student gets access to those counts, which helps the student with an activity called cycle-chasing. The other metrics we used for our study [section 6] could be, but are not, included in the RAP-model. The researcher used spreadsheets instead of user interfaces to get metric-based measurement results.

4.4.2 Integrity rules

In this subsection we refer to relations and rules defined in chapter 5.

Integrity rules are required for those relations that a user can edit through the user interfaces, because those rules can be violated by the user. Other integrity rules can only be violated through mistakes in the software. The RAP model does define rules that can only be violated by software, which are strictly speaking not required, namely: rule 5.19, 5.47, 5.55, 5.73, 5.74 and 5.83.

A user can edit three sets of relations through user interfaces, which requires integrity rules:

- the population of an Ampersand-model by editing the following relations: relation 5.49, 5.54, 5.60, 5.61 and 5.72. The following rules are required to let a user edit the population: rule 5.56, 5.57, 5.75, 5.76 and 5.77.

- the user profile to use by editing relation 5.9, which is an attribute of a user, that is, a univalent relation from user to user profile that requires no extra rules.
- a file name to save to. Users may export the population to a separate file. They can choose a file name before starting the export. Rule 5.18 has been defined to enforce a new, unique file name before the user starts an export. Strictly speaking, the definition of this rule in the RAP model is not required, because the file system enforces the same rule.

4.5 User interfaces

Ampersand allows its users to construct user interfaces, giving access to data. The following example shows an **interface definition** from RAP's Ampersand-script. A listing of all interface definitions of RAP is given in appendix A.2. This example generates a user interface for students to view or edit a relation. Figure 4.6 is a screen shot of this user interface, which displays an example relation, namely $r : A \sim B$.

```
INTERFACE Relation(decpopu,left,right) FOR Student:I[Declaration]
BOX ["PURPOSEs":decpurpose
    ,"MEANING":decmean
    ,"example of basic sentence":decexample
    ,"name":decnm
    ,"type":decsgn
     BOX ["source":src
          ,"target":trg
          ]
    ,"properties":decprps;declaredthrough
    ,"from PATTERN":ptdcs~
    ,"POPULATION":decpopu
     BOX ["source":left
          ,"target":right
          ]
    ,"used in RULEs":(rrexp;rels;reldcl)~;
                     (I[Rule] /\ -I[PropertyRule])
    ]
```

An interface definition is a tree of label-relation pairs, which maps to a web interface. A branch is called a `BOX` by its appearance as a box on the web interface. Each node has a full relation expression, which is the composition of the full expression of the parent node and the expression defined on the node. For example, the expression defined on the node labelled `"source"` is *decsgn*. The (full) expression of its parent - the root node - is $\mathbb{I}_{\texttt{Declaration}}$. Thus ,the full expression of `"source"` is

$\mathbb{I}_{\texttt{Declaration}};decsgn$. A web interface is a tree of label-atoms pairs as defined by the interface definition, where each full expression has been evaluated resulting in a set of atoms.

The evaluation function $eval(a)$ of a full expression R is $eval(a) = codomain(a;R)$, where atom $a \in source(R)$. All full expressions in an interface have the same source concept. One could say that an interface gives a view on the atoms of that source concept e.g. the example gives a view on relation declarations. A special concept, $\texttt{ONE} = \{1\}$, exists to define an interface on no particular concept, that is, the parameter is always set to $a = 1$. The source of such an interface definition is `ONE` e.g. the expression of the root node is $\mathbb{I}_{\texttt{ONE}}$.

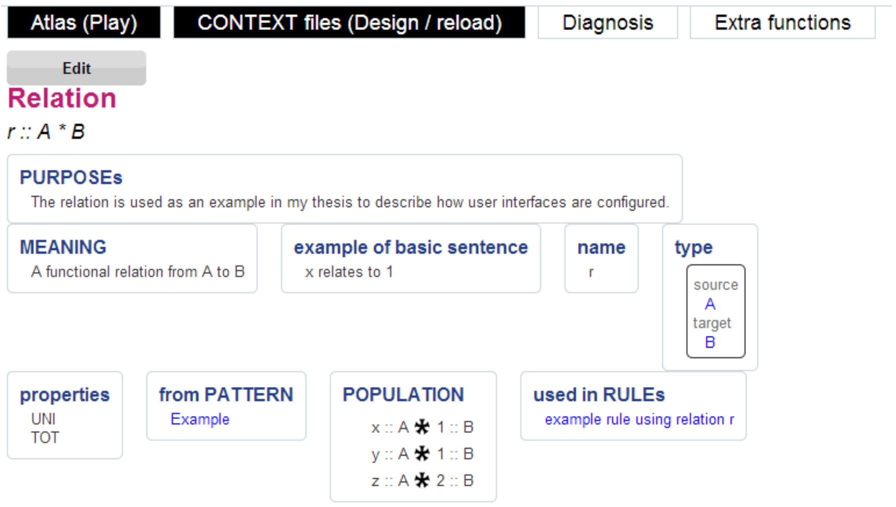
A user navigates through web interfaces by clicking a hyperlink on the menu-bar or an atom printed as a hyperlink. An atom is printed as a hyperlink, if there is a web interface to view that atom. Such a hyperlink points to that web interface with the parameter set to the atom that has been clicked. For example, to get to the exact page of the screen shot, the user has clicked the atom `'r::A*B'` on one of the web interfaces. On the screen shot, the user can click the `'example rule using relation r'` to get a view on that rule, or `'A'` to get a view on that concept. A hyperlink to an interface on `ONE` can be printed on the menu-bar, which is available on any web interface. For example, the menu-bar of RAP is printed at the top of the screen shot and contains hyperlinks to four interfaces, namely “Atlas (Play)”, “CONTEXT files (Design / reload)”, “Diagnosis” and “Extra functions”.

A web interface can be configured to have a create- or edit-mode. Most interfaces of RAP only have a view-mode. In edit-mode, the user can change certain sets of atoms as configured on the interface. In the example interface, a user may edit three relations, namely `(decpopu,left,right)` Three interfaces have an edit-mode, that is, interface `Relation` from the example, interface `Concept` and interface `Extra functions`. The create-mode is like the edit-mode, only with a new, created atom a, which is set to the parameter. A user may get access to the create-mode of an interface by means of an option on the menu-bar. The users of RAP are not given this option to the create-mode of interfaces.

RAP uses one more attribute to configure the interface, the `FOR`-attribute. The example sets the `FOR`-attribute to “Student”, which means that users in the role of “Student” have access to the interface.

4.6 User interface layout

The web layout of a user interface is configured in Cascading Style Sheets (CSS). The main content on a web interface is the tree of labels and sets of atoms, which are textual elements. Javascript code embeds the sets of atoms in HTML-elements to obtain the required functionality. For example, an atom is a HTML-link in view-



Relation

r :: A * B

PURPOSEs

The relation is used as an example in my thesis to describe how user interfaces are configured.

MEANING

A functional relation from A to B

example of basic sentence

x relates to 1

name

r

type

source A target B

properties

UNI TOT

from PATTERN

Example

POPULATION

x :: A ✱ 1 :: B
y :: A ✱ 1 :: B
z :: A ✱ 2 :: B

used in RULEs

example rule using relation r

Figure 4.6: Screen shot of user interface

mode and an HTML-textbox in edit-mode; a set of atoms in edit-mode has a handler to add a new atom to the set.

The Ampersand compiler ships with a set of CSS-files for a default layout of user interfaces. Little efforts were needed to customize these files to obtain the layout of RAP. We have experienced that concessions have to be made at the expense of user experience, because user interfaces are generated instead of hand-coded. Most noticeable is the positioning of labels and the sets of atoms, which cannot be configured optimally. Further research is needed to solve the problem of positioning this kind of dynamic content.

The RAP model contains **key definitions** to define how atoms are printed on a web interface. A key definition is a language construct on a concept that defines a HTML template by which atoms of that concept are printed. For example, the RAP-model has a concept `Image`, which represents images. An image has a relation to its location on the web, namely *imageurl* : `Image` $\sim$ `URL`. We want a web interface to print atom $a \in$ `Image` as an HTML-img element of which the src-attribute is the location of that image, such that the web interface shows the actual image. For that we have defined the following key on `Image` [appendix A]:

```
-390- KEY Image: Image(PRIMHTML "<img src='", imageurl
                      , PRIMHTML "'>")
```

The keyword `KEY` indicates that this is a key, followed by a name for the key, namely `Image`, and a colon. The `Image` after the colon is the concept to which this key applies, followed by a list of textual elements to print sequentially. The keyword `PRIMHTML` on an element indicates html code. The keyword `TXT` indicates text that is not html code. Elements without a keyword are relation expressions such as *imageurl*, which means take an atom from the codomain of a;*imageurl*, where $a \in$ `Image` is the atom to print.

To demonstrate the key-mechanism by the example of images, assume

`'diagram1'` *imageurl* `'http:://is.cs.ou.nl/image/diagram1.jpg'`

When a web interface needs to print `'diagram1'` $\in$ `Image`, it will print the following html code:

```
<img src='http:://is.cs.ou.nl/image/diagram1.jpg'>
```

Without a key definition it would have printed `diagram1`.

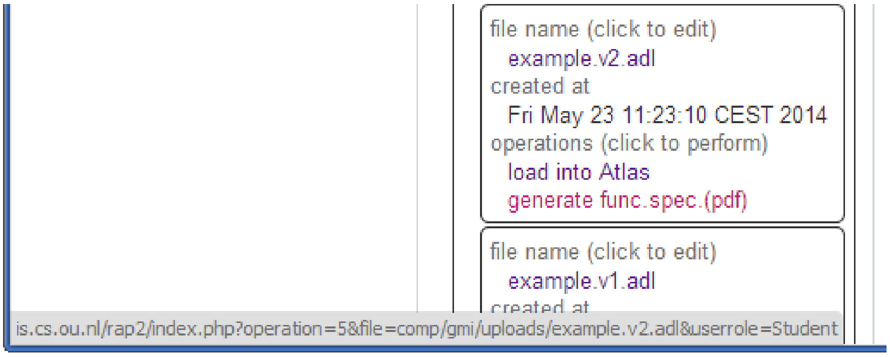
4.7 Ampersand compiler

Key definitions are also used to trigger functions, for example, an Ampersand compiler command. All Ampersand compiler commands are run by a special manually coded web page (index.php). A key has been defined on a concept `CCommand` to print its atoms as links to that special page, such that clicking an atom triggers a command.

This is the key definition on *CCommand*.

```
-54- KEY CompilerCommand:
     CCommand
       (PRIMHTML "<a href='../../index.php?operation="
       ,operation
-55-   ,PRIMHTML "&file="
       ,applyto;filepath  ,applyto;filename
-56-   ,PRIMHTML "&userrole="
       ,applyto;uploaded[User*AdlFile];userrole
-57-   ,PRIMHTML "'>"
       ,functionname
       ,PRIMHTML "</a>"
       )
```

Figure 4.7 shows a screen shot of a user interface that prints atoms of `CCommand` under the label "operations (click to perform)". The screen shot is taken when hoovering the HTML-link "generate func.spec.(pdf)", which shows the value of the href-attribute of that link at the bottom.



file name (click to edit)
example.v2.adl
created at
Fri May 23 11:23:10 CEST 2014
operations (click to perform)
load into Atlas
generate func.spec.(pdf)

file name (click to edit)
example.v1.adl
created at

is.cs.ou.nl/rap2/index.php?operation=5&file=comp/gmi/uploads/example.v2.adl&userrole=Student

Figure 4.7: Screen shot with compiler commands

4.8 Conclusion

This chapter describes the logical design of RAP and how RAP has been generated from the RAP model. We have demonstrated how a model in Ampersand presents itself as a working application to users. Although a fair number of working applications have been made, we cannot claim that Ampersand produces industry strength applications. For that, further engineering on the Ampersand compiler is required. This explains why the Ampersand toolset is currently being used in the design phase, for the purpose of prototyping and generating documentation.

Chapter 5

DETAILED SPECIFICATION OF RAP

The central contribution of this thesis is RAP, a development environment for rule-based prototyping. This chapter specifies RAP in its entirety, complete and in detail, which serves several purposes. One, the reader can validate all formalized requirements of RAP. Two, each requirement yields an example of what can be implemented with Ampersand. Three, the chapter is an example of a requirements specification generated from an Ampersand-model, which illustrates that generation function of Ampersand. Four, a subset of the requirements formally define aspects of Ampersand, because the RAP model includes those requirements to support Ampersand. Five, the chapter is documentation of RAP.

This chapter is not intended to be read from beginning to end. To guide the reader to potential points of interest:

> The rule-based design process of Ampersand in figure 2.1 has been formalized in section 5.2. This topic relates to section 4.7 and the definitions of relations 5.12 and 5.13.
>
> The data structure of type error messages discussed in chapter 3 has been formalized in section 5.3.
>
> How parse and type error messages are presented to the user has been formalized in section 5.4. This section relates to 4.3.
>
> Violations in the user-defined Ampersand-model are presented to the user no differently than parse and type error messages, which can be validated in section 5.15. This section relates to 4.3.

Most sections define Ampersand language concepts, relations and rules, namely sections 5.5 to 5.13. These requirements definitions correspond with the language definitions in chapter 3. These sections relate to 4.4.

Section 5.14 and 5.17 show how data derived from an Ampersand-model like conceptual diagrams or measurements are stored in the repository of RAP. These sections relate to 4.4.

Section 5.16 shows how rules can be defined to support a process of the user like editing the population of an Ampersand-model. This section relates to 4.3.

5.1 Overview

RAP is meant to support students who are doing Ampersand exercises as a means to become better requirement engineers. In Ampersand, we would say that RAP supports the business process of learning Ampersand in the course "Rule-Based Design" [section 6.1]. Hence RAP itself is a fine example of something Ampersand aims to design. So besides being a research product, RAP can be perceived as an example of the use of Ampersand as well.

The source code of RAP is an Ampersand-script. We call this script the RAP model. It contains the meta model of Ampersand and user interface specifications to practice and learn Ampersand. The RAP model is printed verbatim in appendix A. RAP has been built by compiling the RAP model into a working PHP-MySQL-application. This (generated) application is the very tool that students of course [section 6.1] use on the web. That same RAP model has been compiled to generate the documentation of RAP, as illustrated by figure 2.2. In fact, the generated documentation for RAP is this very chapter, which you are reading right now.

This chapter is organized in sections by the themes that occur as patterns and processes in the RAP model. Thus, within this chapter the terms section and theme are equivalent. And there are two kinds of themes, namely patterns and processes. Each theme starts with a description and purpose of that theme. Next, the theme is illustrated by a conceptual diagram [section 3.5.3], which provides a visual cue to the topic of that theme and its dependency on other themes. Short statistics are given, providing the number of concepts, relations and rules defined in that particular theme. Cross-reference information is provided wherever appropriate.

Model elements of an Ampersand-model, such as rules, relations, patterns, interfaces, concepts, roles, etcetera, are concepts in the RAP model. These concepts, the relations between them and the rules they satisfy are specified in themes, each of which is documented in a separate section of this chapter. The following themes constitute an Ampersand-model: Contexts, Patterns, Generalization rules, Concepts, Relation type signatures, Relation declarations, Expressions, Rules, Symbols and Calculated details. The first three themes discussed in this chapter specify fundamental functionality of RAP, that is, the repository. The last three themes specify specific user functions for student requirements engineers.

1. Repository
   (a) The repository of files pattern
   RAP is a file repository. A user can commit text files containing Ampersand-scripts to the repository. After committing, he can run the Ampersand compiler, which provides various functions to study, manipulate, and verify that script. One of these functions is opening a script to view and edit the Ampersand-model. Committed files cannot be changed or deleted. In this way, all previous submissions by users are being retained. They constitute traces of student actions, which have been used in our research.
   (b) Committed files pattern
   A committed Ampersand-script may or may not contain script errors, that is, parse or type errors. Only a script free of script errors is meaningful and yields an Ampersand-model. This theme specifies the data that is used in the report on script errors. This theme also specifies a cache for the changes made by a user, who is editing an Ampersand-model through the user interfaces.
   (c) Handling script errors process
   A user gets a browsable report on script errors.
2. Ampersand-models
   (a) Contexts pattern
   A context is the root model element, which sets the name and scope of an Ampersand-model.
   (b) Patterns pattern
   A requirement engineer specifies rules and relations in patterns and processes.
   (c) Generalization rules pattern
   A requirement engineer defines a generalization rule to specify a supertype relation between a specific and general concept.
   (d) Concepts pattern
   A requirement engineer specifies concepts of which the population is a set of atoms. RAP assigns a concept to the right partial order of concepts, based on the generalization rules in a script.

  (e) Relation type signatures pattern
  Pairs of concepts are used as types for declared relations and relation expressions, the logic of which is supported in RAP. Pairs of atoms are used as instances in the population of a declared relation or as violations of a rule.

  (f) Relation declarations pattern
  A requirement engineer declares relations and defines their population.

  (g) Expressions pattern
  Complex relations can be expressed on declared relations and relational operators. These relation expressions are used in, for example, rule definitions and user interface specifications.

  (h) Rules pattern
  A requirement engineer defines a rule in a pattern to specify the desired semantics of relations. A requirement engineer defines a rule in a process to specify a business processes.

  (i) Symbols pattern
  A requirement engineer may choose the name of user-defined model elements like concepts, relations and rules. Those names appear in the Amper-sand-script and need to follow the syntactic rules. For example, the name of a concept is of type `Conid`, which is a string symbol that starts with an uppercase. This theme introduces just five concepts for the five different syntactic domains for user-defined names.

  (j) Calculated details pattern
  This theme introduces calculated elements of an Ampersand-model: rule violations, conceptual diagrams and pragma-sentences.

3. User functions

  (a) Testing rules process
  RAP checks the rules in an Ampersand-model and reports rule violations. From the perspective of a requirements engineer, he tests the rules that he has defined in his model. RAP supports testing rules in an educational setting by means of violation reports that target students.

  (b) Editing a population process
  A user may test an Ampersand-model by repeatedly editing the population and testing the rules. A user can edit the population of an Ampersand model either through committing a new script or through the user interfaces. The advantage of using the interfaces to edit is that they give support while editing. For example, the interfaces prevent script errors from

occurring and give feedback and directions to students while editing a population.

(c) Metrics pattern
We use a framework for metrics [chapter 7] to add metrics to RAP. Such metrics are primarily used to produce information for research as described in chapter 6. RAP can be configured to give a user access to this kind of - or exact same - metric-based information. This theme adds metrics to RAP for students, which gives a student the required information for an activity called cycle chasing.

5.2 PATTERN: The repository of files

This theme defines RAP as a rule repository for requirements engineers - students in particular - to produce and process rules as illustrated by figures 2.1 and 2.2.

The rule repository stores rules in adl-files. An **adl-file**[1] is the file format of an Ampersand-script. The rules are defined in a model element called a context [section 5.5], which sets the name and scope of an Ampersand-model. Thus, the terms adl-file, Ampersand-script and context are different forms to refer to an Ampersand model, which may all be used as such.

Users have their own isolated workspace - a personalized copy of RAP - with their own adl-files. A requirements engineer creates a workspace with a user name to login to that workspace. A user configures the workspace by selecting a user profile to have access to the set of functionality configured for that kind of user. Three user profiles are available in RAP: a student - the default -, an advanced student, and a requirements engineer.

A user uploads Ampersand-scripts to the repository using a special web page, which called **the upload page**. On the upload page, a user can browse and upload an adl-file from his computer or the user can write down and upload a script using the integrated 'script editor'. RAP has been configured - using key definitions [section 4.6] to display an adl-file as a hyperlink - such that, when clicked, the script opens in the script editor on the upload page [section 4.7]. Committing a script from the script editor results in a new adl-file in the repository, because adl-files in the repository cannot be deleted or changed.

RAP supports two kinds of processing of adl-files in the repository: run commands on the Ampersand compiler or view and edit a script through the user interfaces.

[1] ADL is an abbreviation of *A Description Language*, which is a deprecated name for the Ampersand language.

A user can open an adl-file to view and edit its contents through the user interfaces [section 4.5]. To opening an adl-file, RAP runs a certain command on the Ampersand compiler. Changes made through the interfaces are cached until the user commits the changes to the repository, or opens an adl-file, which clears the cache. Changes are committed to a new adl-file, because, as mentioned before, adl-files in the repository cannot be deleted or changed. In this version of RAP, the interfaces have been defined such that a user can only edit the population of a context through the interfaces.

Recall from chapter 2 that an Ampersand-script includes a population. A user can put the population in a separate file in the repository, a **pop-file**. A user may, for example, use pop-files to switch easily between different populations by changing a single include-statement in the master script.

This theme introduces 11 concepts, 4 generalization rules, 13 relations with properties and 2 rules related to the repository of files in RAP. Figure 5.1 shows a conceptual diagram, which includes the elements introduced in this section; a concept `Context` [section 5.5]; a concept `String` [section 5.13]; a concept `Int` [section 5.17].

5.2.1 Defined concepts

This subsection defines 11 concepts and 4 generalization rules. Each definition defines a concept followed by how the user interfaces display an atom *a* of that concept. This display value of *a* is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on *a*. Relation expressions are shaded. A key may be defined such that *a* is printed as an HTML-element or as regular text. If *a* prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view *a* [section 4.5].

Definition: A `FileRef` is a reference to a file in the repository. Atom $a \in$ `FileRef` is displayed - using relations 5.6, 5.5, 5.9, 5.8 - as:

<a href='../../index.php?file= $a;\mathit{filepath}$ $a;\mathit{filename}$

&userrole= $a;\mathit{uploaded}^{\smile};\mathit{userprofile}$ '> $a;\mathit{filename}$ </a>

Definition: A `FileName` is a name of a file. Atom $a \in$ `FileName` is displayed as is.

Definition: A `FilePath` is a location in the repository. Atom $a \in$ `FilePath` is displayed as is.

Definition: A `CalendarTime` is a time representation, which includes day, weekday, month, year, hour, minute, second and timezone. Atom $a \in$ `CalendarTime` is displayed as is.

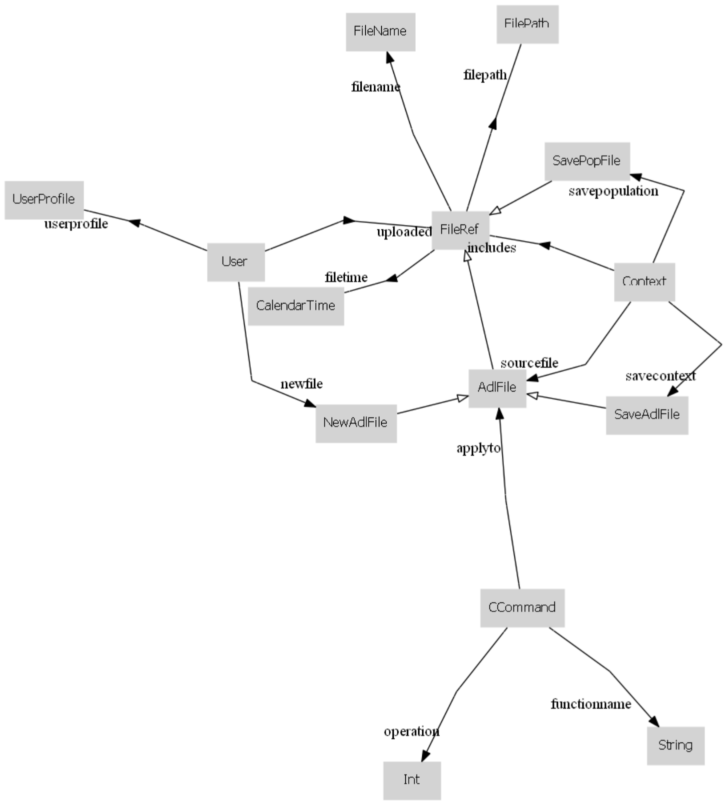


Figure 5.1: Concept diagram of The repository of files

Definition: A `User` is a name with which a requirements engineer has logged in to RAP. Atom $a \in$ `User` is displayed as is.

Definition: A `UserProfile` is a kind of user. Atom $a \in$ `UserProfile` is displayed as is.

Definition: A `AdlFile` is a file of the adl-format, which is the format for Ampersand-scripts. Atom $a \in$ `AdlFile` is displayed as a `FileRef`.

Rule: `AdlFile` is a kind of `FileRef`, which is formalized by the following generalization rule:

$$\texttt{AdlFile} \leq \texttt{FileRef} \tag{5.1}$$

Compiler commands to run functions of the Ampersand compiler are configured in RAP as atoms of concept `CCommand`. A typical function of the Ampersand compiler is a composition of the functions parse, typecheck, and interpret a script as an Ampersand model, followed by some specific processing function on that model e.g. generate software. A key has been defined on `CCommand` such that a command executes when a user clicks on that command in the user interfaces.

Definition: A `CCommand` is a command for the Ampersand compiler. Atom $a \in \texttt{CCommand}$ is displayed - using relations 5.14, 5.6, 5.12, 5.5, 5.9, 5.8, 5.13 - as:

<a href='../../index.php?operation= $a;\textit{operation}$

&file= $a;\textit{applyto};\textit{filepath}$ $a;\textit{applyto};\textit{filename}$

&userrole= $a;\textit{applyto};\textit{uploaded}_{\texttt{User}\sim\texttt{AdlFile}}{}^{\smile};\textit{userprofile}$ '

> $a;\textit{functionname}$ </a>

A user can open a new context in the script editor on the upload page. If a user clicks on an atom $a \in \texttt{NewAdlFile}$, then the script in the predefined file to which a refers opens in the script editor. One such file has been defined in RAP, which contains an empty context.

Definition: A `NewAdlFile` is a predefined adl-file, which opens as a new script in the script editor on the upload page. Atom $a \in \texttt{NewAdlFile}$ is displayed - using relation 5.5 - as:

<a href='../../index.php'> $a;\textit{filename}_{\texttt{NewAdlFile}\sim\texttt{FileName}}$ </a>

Rule: `NewAdlFile` is a kind of `AdlFile`, which is formalized by the following generalization rule:

$$\texttt{NewAdlFile} \leq \texttt{AdlFile} \tag{5.2}$$

A user can save the population of a context. If a user clicks on the link to the pop-file to save to, then RAP creates that file in the repository and saves the population to that file.

Definition: A `SavePopFile` is a file to which a user can save the population of a context. Atom $a \in$ `SavePopFile` is displayed - using relations 5.6, 5.5 - as:

<a href='../../index.php?operation=4
&file= $a;\mathit{filepath}_{\texttt{SavePopFile}\sim\texttt{FilePath}}$ $a;\mathit{filename}_{\texttt{SavePopFile}\sim\texttt{FileName}}$ '
> $a;\mathit{filename}_{\texttt{SavePopFile}\sim\texttt{FileName}}$ </a>

Rule: `SavePopFile` is a kind of `FileRef`, which is formalized by the following generalization rule:

$$\texttt{SavePopFile} \leq \texttt{FileRef} \tag{5.3}$$

A user can commit changes made to a context. If a user clicks on the link to the adl-file to save to, then RAP creates that file in the repository and saves the context to that file.

Definition: A `SaveAdlFile` is a file to which a user can save the changes made to a context. Atom $a \in$ `SaveAdlFile` is displayed - using relations 5.6, 5.5, 5.9, 5.8, 5.10, 5.17 - as:

<a href='../../index.php?operation=2
&file= $a;\mathit{filepath}_{\texttt{SaveAdlFile}\sim\texttt{FilePath}}$ $a;\mathit{filename}_{\texttt{SaveAdlFile}\sim\texttt{FileName}}$
&userrole= $a;\mathit{savecontext}^{\smile};\mathit{sourcefile};\mathit{uploaded}_{\texttt{User}\sim\texttt{AdlFile}}{}^{\smile};\mathit{userprofile}$ '
> $a;\mathit{filename}_{\texttt{SaveAdlFile}\sim\texttt{FileName}}$ </a>

Rule: `SaveAdlFile` is a kind of `AdlFile`, which is formalized by the following generalization rule:

$$\texttt{SaveAdlFile} \leq \texttt{AdlFile} \tag{5.4}$$

5.2.2 Declared relations

This subsection introduces 13 declared relations with properties and a meaning.

The following univalent, total relation has been declared

$$\mathit{filename} \quad : \quad \texttt{FileRef} \rightarrow \texttt{FileName} \tag{5.5}$$

, which means that a file reference includes a file name.

For example, `'comp/gmi/RAP.v2.adl'` *filename* `'RAP.v2.adl'` means:

The file name in `'comp/gmi/RAP.v2.adl'` is `'RAP.v2.adl'`.

The following univalent relation has been declared

$$\textit{filepath} \quad : \quad \texttt{FileRef} \sim \texttt{FilePath} \tag{5.6}$$

, which means that a file reference may include a relative or absolute file path.

For example, `'comp/gmi/RAP.v2.adl'` *filepath* `'comp/gmi/'` means:

The file path in `'comp/gmi/RAP.v2.adl'` is `'comp/gmi/'`.

The following univalent relation has been declared

$$\textit{filetime} \quad : \quad \texttt{FileRef} \sim \texttt{CalendarTime} \tag{5.7}$$

, which means that the time on which a file has been committed to the repository may be known.

The following relation has been declared

$$\textit{uploaded} \quad : \quad \texttt{User} \sim \texttt{FileRef} \tag{5.8}$$

, which means that a user may have committed files to the repository.

For example, `'gmi'` *uploaded* `'comp/gmi/RAP.v2.adl'` means:

`'gmi'` has committed `'comp/gmi/RAP.v2.adl'` to the repository.

The following univalent relation has been declared

$$\textit{userprofile} \quad : \quad \texttt{User} \sim \texttt{UserProfile} \tag{5.9}$$

, which means that a user may have a user profile, which gives him access to the set of compiler commands for that user profile.

For example, `'gmi'` *userprofile* `'Student'` means:

`'gmi'` has access to the compiler commands for a `'Student'`.

The following univalent, total relation has been declared

$$\textit{sourcefile} \quad : \quad \texttt{Context} {\rightarrow} \texttt{AdlFile} \tag{5.10}$$

, which means that a context originates from an adl-file in the repository.

For example, `'RAP'` *sourcefile* `'comp/gmi/RAP.v2.adl'` means:

Context `'RAP'` originates from `'comp/gmi/RAP.v2.adl'`.

An Ampersand-script may include scripts in other files. The context in such a script originates from the master file [relation 5.10] and included files. For example, a script may include the population in a previously saved pop-file.

For the above, the following relation has been declared

$$\textit{includes} \quad : \quad \texttt{Context} \sim \texttt{FileRef} \tag{5.11}$$

, which means that the adl-file from which a context originates may include other files.

For example, `'RAP'` *includes* `'comp/gmi/RAP.v77.pop'` means:

Context `'RAP'` partially originates from `'comp/gmi/RAP.v77.pop'`.

The following univalent, total relation has been declared

$$\textit{applyto} \quad : \quad \texttt{CCommand} {\rightarrow} \texttt{AdlFile} \tag{5.12}$$

, which means that a compiler command applies to an adl-file.

For example, `'1(comp/gmi/RAP.v2.adl)'` *applyto* `'comp/gmi/RAP.v2.adl'` means:

Command `'1(comp/gmi/RAP.v2.adl)'` applies to `'comp/gmi/RAP.v2.adl'`.

The following univalent, total relation has been declared

$$\textit{functionname} \quad : \quad \texttt{CCommand} \rightarrow \texttt{String} \tag{5.13}$$

, which means that a compiler command uses a compiler function that has a user-friendly name.

For example, `'1(comp/gmi/RAP.adl)'` *functionname* `'load into Atlas'` means:

Command `'1(comp/gmi/RAP.adl)'` uses the function `'load into Atlas'`.

The following univalent, total relation has been declared

$$\textit{operation} \quad : \quad \texttt{CCommand} \rightarrow \texttt{Int} \tag{5.14}$$

, which means that a compiler command uses a compiler function that has a technical identifier.

For example, `'1(comp/gmi/RAP.v2.adl)'` *operation* `'1'` means:

Command `'1(comp/gmi/RAP.v2.adl)'` uses the function `'1'`.

The following univalent, total relation has been declared

$$\textit{newfile} \quad : \quad \texttt{User} \rightarrow \texttt{NewAdlFile} \tag{5.15}$$

, which means that a user has an option to open a new script.

For example, `'gmi'` *newfile* `'empty.adl'` means:

`'gmi'` has an option to open `'empty.adl'` in the script editor.

The following univalent, total relation has been declared

$$\textit{savepopulation} \quad : \quad \texttt{Context} \rightarrow \texttt{SavePopFile} \tag{5.16}$$

, which means that there is an option to export the population of a context to a pop-file.

For example, `'RAP'` *savepopulation* `'comp/gmi/RAP.v78.pop'` means:

If the user exports the population of `'RAP'` then that population will be saved to a new pop-file `'comp/gmi/RAP.v78.pop'` in the repository.

The following univalent, total relation has been declared

$$\mathit{savecontext} \;:\; \texttt{Context}{\rightarrow}\texttt{SaveAdlFile} \tag{5.17}$$

, which means that there is an option to commit changes on a context to an adl-file.

For example, `'RAP'` *savecontext* `'comp/gmi/RAP.v3.adl'` means:

If the user commits the changes made to `'RAP'` then that context including the changes will be saved to a new adl-file `'comp/gmi/RAP.v3.adl'` in the repository.

5.2.3 Defined rules

This subsection defines 2 formal rules.

unique file location - The following requirement has been defined:
Each file has a unique location on the file system of the repository.
This is formalized - using the declared relations 5.6, 5.5 - as

$$\mathit{filename};\mathit{filename}^{\smile} \cap \mathit{filepath};\mathit{filepath}^{\smile} \subseteq \mathbb{I} \tag{5.18}$$

user profiles - The following requirement has been defined:
There are three user profiles: Student, StudentDesigner and Designer.
This is formalized - using the declared relations - as

$$\texttt{'Student'} \cup \texttt{'StudentDesigner'} \cup \texttt{'Designer'} \subseteq \mathbb{I}_{\texttt{UserProfile}} \tag{5.19}$$

5.3 PATTERN: Committed files

A user has two ways to commit an adl-file: upload a script or commit changes made to a context.

On a commit, a new file in the repository is created with a property that holds the current version of the Ampersand compiler at the time of the commit. This property documents how to interpret an adl-file in the repository, that is, parse, typecheck and interpret the file like that version of the compiler did. This property can, for example, be used to determine compatibility with other versions of the Ampersand compiler.

A script may contain script errors, which makes those scripts uninterpretable. Uninterpretable scripts are accepted in the repository so that RAP can provide support to analyse the script error.

Changes made on a context through the user interfaces are cached before the user commits them. The changes made are the difference between the current context and the initial context in the repository. Changes can only be made to those relations of RAP that are editable through user interfaces. This theme introduces uneditable copies of editable relations. These copies hold the initial population of their editable counterpart. The copies are used to express "the changes made" in Ampersand, which is needed to define the process rules for editing [section 5.16]. In this version of RAP, a user can only edit the population of a context.

This theme introduces 8 concepts, 0 generalization rules, 15 relations with properties and 0 rules related to committed files in RAP. Figure 5.2 shows a conceptual diagram, which includes the elements introduced in this section; 2 concepts `FileRef`, `AdlFile` [section 5.2]; 3 concepts `Concept`, `AtomID`, `Atom` [section 5.8]; a concept `PairID` [section 5.9]; a concept `Declaration` [section 5.10].

5.3.1 Defined concepts

This subsection defines 8 concepts and 0 generalization rules. Each definition defines a concept followed by how the user interfaces display an atom a of that concept. This display value of a is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on a. Relation expressions are shaded. A key may be defined such that a is printed as an HTML-element or as regular text. If a prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view a [section 4.5].

Definition: A `AdlVersion` is a version of the Ampersand compiler. Atom $a \in$ `AdlVersion` is displayed as is.

Definition: A `ParseError` is an error in the syntax of a script. Atom $a \in$ `ParseError` is displayed as:

Click here for error details

Definition: A `TypeError` is an error concerning the type of a relation declaration or relation expression in the script. Atom $a \in$ `TypeError` is displayed - using relation 5.32 - as:

Click here for details of error at $a;te\_position$

Definition: A `ErrorMessage` is a description of an error. Atom $a \in$ `ErrorMessage` is displayed as is.

Definition: A `FilePos` is a position in a file. Atom $a \in$ `FilePos` is displayed as is.

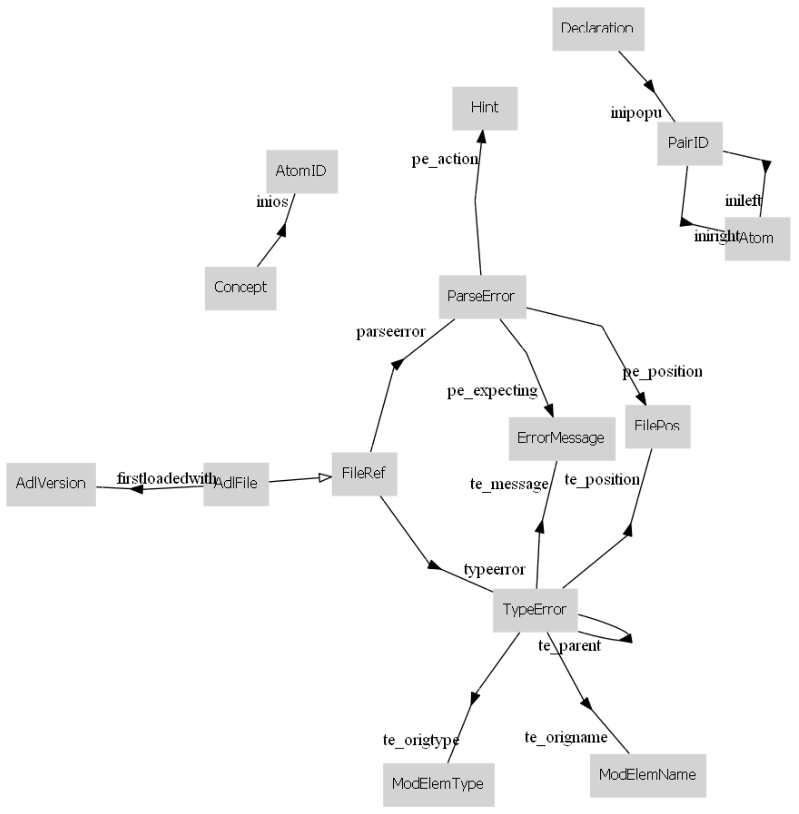


Figure 5.2: Concept diagram of Committed files

Definition: A `Hint` is a description of possible actions which may resolve an error. Atom $a \in$ `Hint` is displayed as is.

Definition: A `ModElemType` is a type of model element e.g. rule definition, user interface definition, key definition. Atom $a \in$ `ModElemType` is displayed as is.

Definition: A `ModElemName` is the name of a model element. Atom $a \in$ `ModElemName` is displayed as is.

5.3.2 Declared relations

This subsection introduces 15 declared relations with properties and a meaning.

The following univalent relation has been declared

$$\mathit{firstloadedwith} \ : \ \texttt{AdlFile} \sim \texttt{AdlVersion} \tag{5.20}$$

, which means that the version of the Ampersand compiler at the time that an adl-file was committed to the repository may be known.
For example, `'comp/gmi/RAP.v2.adl'` *firstloadedwith* `'v2.2.711-2191'` means:
At the time that `'comp/gmi/RAP.v2.adl'` was committed, RAP used compiler version `'v2.2.711-2191'`.

The following relation has been declared

$$\mathit{inios} \ : \ \texttt{Concept} \sim \texttt{AtomID} \tag{5.21}$$

, which means that the initial population of a concept may contain atoms.
For example, `'AdlFile'` *inios* `'comp/gmi/RAP.v2.adl'` means:
Initially, concept `'AdlFile'` contained an atom `'comp/gmi/RAP.v2.adl'`.

The following relation has been declared

$$\mathit{inipopu} \ : \ \texttt{Declaration} \sim \texttt{PairID} \tag{5.22}$$

, which means that the initial population of a relation may contain pairs of atoms.
For example, `'userprofile::User*UserProfile'` *inipopu* `'523422885'` means:
Initially, `'userprofile::User*UserProfile'` contained a pair with id `'523422885'`.

The following relation has been declared

$$\mathit{inileft} \ : \ \texttt{PairID} \sim \texttt{Atom} \tag{5.23}$$

, which means that a pair of atoms in the initial population of relations had an initial left atom.
For example, `'523422885'` *inileft* `'gmi'` means:
Initially, the pair with id `'523422885'` had a left atom `'gmi'`.

The following relation has been declared

$$\textit{iniright} \quad : \quad \texttt{PairID} \sim \texttt{Atom} \tag{5.24}$$

, which means that a pair of atoms in the initial population of relations had an initial right atom.
For example, `'523422885'` *iniright* `'Student'` means:
Initially, the pair with id `'523422885'` had a right atom `'Student'`.
The following univalent relation has been declared

$$\textit{parseerror} \quad : \quad \texttt{FileRef} \sim \texttt{ParseError} \tag{5.25}$$

, which means that the Ampersand-script in a file may have a parse error.
The following univalent, total relation has been declared

$$\textit{pe\_action} \quad : \quad \texttt{ParseError} \rightarrow \texttt{Hint} \tag{5.26}$$

, which means that a parse error describes possible actions which may resolve the error.
The following univalent, total relation has been declared

$$\textit{pe\_position} \quad : \quad \texttt{ParseError} \rightarrow \texttt{FilePos} \tag{5.27}$$

, which means that a parse error has occurred on a file position.
The following univalent, total relation has been declared

$$\textit{pe\_expecting} \quad : \quad \texttt{ParseError} \rightarrow \texttt{ErrorMessage} \tag{5.28}$$

, which means that a parse error has a message, which describes what was expected by the parser.
The following relation has been declared

$$\textit{typeerror} \quad : \quad \texttt{FileRef} \sim \texttt{TypeError} \tag{5.29}$$

, which means that the Ampersand-script in a file may have type errors.

The following univalent relation has been declared

$$te\_message \quad : \quad \texttt{TypeError} \sim \texttt{ErrorMessage} \tag{5.30}$$

, which means that a type error may have a message that gives a complete description of the error.

The following univalent relation has been declared

$$te\_parent \quad : \quad \texttt{TypeError} \sim \texttt{TypeError} \tag{5.31}$$

, which means that a type error may be nested in another type error that describes the same mistake on a higher level.

The following univalent relation has been declared

$$te\_position \quad : \quad \texttt{TypeError} \sim \texttt{FilePos} \tag{5.32}$$

, which means that a type error may have occurred on a file position.

The following univalent relation has been declared

$$te\_origtype \quad : \quad \texttt{TypeError} \sim \texttt{ModElemType} \tag{5.33}$$

, which means that a type error may have occurred in a certain type of model element.

The following univalent relation has been declared

$$te\_origname \quad : \quad \texttt{TypeError} \sim \texttt{ModElemName} \tag{5.34}$$

, which means that a type error may have occurred in a model element that has a certain name.

5.4 PROCESS: Handling script errors

The user needs to solve script errors. RAP provides the user with error information to do so. User interfaces have been defined for parse and type errors so that the user can browse through error details as desired.

To illustrate by example how RAP handles script errors. Figure 4.4 shows how a parse error $a \in \texttt{ParseError}$ is presented to the user based on the report for violations of process rule 5.35. *a* displays as “Click here for error details” as defined by the key on `ParseError`. “Click here for error details” is a hyperlink to the user interface that shows the details of *a*, namely the interface defined on line 43 to 47 in appendix A.2. `'comp/gmi/uploads/example.v1.adl'` $\in$ `FileRef` is the file in which the error occurred. This file is printed as a hyperlink “example.v1.adl” as defined by the key on `FileRef`. The user may click that hyperlink to open the script in that file in the script editor on the upload page.

This section gives a purpose to introduce 2 rules related to handling script errors in RAP. Figure 5.3 shows a conceptual diagram, which includes the declared relations used to define the 2 process rules.

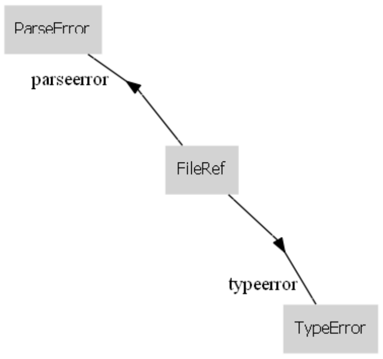


Figure 5.3: Concept diagram of Handling script errors

5.4.1 Defined rules

This subsection defines 2 formal process rules for the user.

noparseerror - The following requirement has been defined:
The requirements engineer needs to solve all parse errors.
This is formalized - using the declared relation 5.25 - as

$$\neg parseerror \tag{5.35}$$

RAP reports all rule violations $\langle a,b\rangle \in \neg(noparseerror)$ to the user. This report[2] starts with the following message

[2]This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.

> **A syntax error was encountered in your script.** No CONTEXT screen could be generated.

, followed by the following message for each $\langle a,b\rangle$.

> b Open a to fix error.

notypeerror - The following requirement has been defined:
The requirements engineer needs to solve all type errors.
This is formalized - using the declared relation 5.29 - as

$$\neg typeerror \tag{5.36}$$

RAP reports all rule violations $\langle a,b\rangle \in \neg(notypeerror)$ to the user. This report[3] starts with the following message

> **Type error(s) were encountered in your script.** A CONTEXT screen was generated with concepts and relation declarations only, which may be useful to understand and fix the errors.

, followed by the following message for each $\langle a,b\rangle$.

> b. Open a to fix error.

5.5 PATTERN: Contexts

A context is the root element of an Ampersand model. A context directly links to the model elements of the types pattern [section 5.6] and concept [section 5.8]. A context indirectly links to other model elements through patterns and concepts. For example, a pattern `'P'` in context $\mathfrak{C}$ contains a relation declaration $r : A \sim B$, which implies that $r : A \sim B$ is an element of $\mathfrak{C}$. Another example, $r : A \sim B$ in $\mathfrak{C}$ contains a relation element $a\ r\ b$, which implies that $a\ r\ b$ is an element of $\mathfrak{C}$.

A context has a user-defined name, which may be used to refer to a context. However the name of a context does not necessarily identify a context; an adl-file

[3]This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.

does identify a context. For analytics with RAP we have more purposes for the name of a context e.g. we assume that two contexts with identical names of the same user are different - versions of - scripts for the same business context.

This theme introduces 1 concept, 0 generalization rules, 3 relations with properties and 0 rules related to contexts in RAP. Figure 5.4 shows a conceptual diagram, which includes the elements introduced in this section; a concept `Pattern` [section 5.6]; a concept `Concept` [section 5.8]; a concept `Conid` [section 5.13].

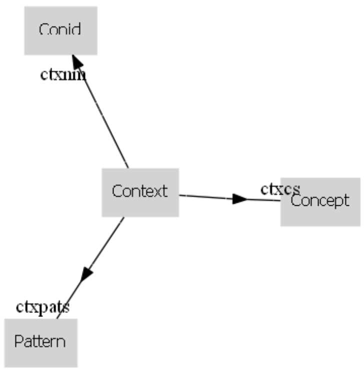


Figure 5.4: Concept diagram of Contexts

5.5.1 Defined concepts

This subsection defines 1 concept and 0 generalization rules. Each definition defines a concept followed by how the user interfaces display an atom *a* of that concept. This display value of *a* is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on *a*. Relation expressions are shaded. A key may be defined such that *a* is printed as an HTML-element or as regular text. If *a* prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view *a* [section 4.5].

Definition: A `Context` is a model element, which defines the name and scope of an Ampersand model. Atom $a \in$ `Context` is displayed - using relation 5.37 - as:

$a; ctxnm$

5.5.2 Declared relations

This subsection introduces 3 declared relations with properties and a meaning.

The following univalent, total relation has been declared

$$ctxnm \quad : \quad \texttt{Context}{\rightarrow}\texttt{Conid} \tag{5.37}$$

, which means that a context has a user-defined name.
For example, `'comp/gmi/RAP.v2.adl'` *ctxnm* `'RAP'` means:
The context in `'comp/gmi/RAP.v2.adl'` is called `'RAP'`.
The following relation has been declared

$$ctxpats \quad : \quad \texttt{Context} \sim \texttt{Pattern} \tag{5.38}$$

, which means that a context may contain patterns.
For example, `'RAP'` *ctxpats* `'Committed files'` means:
Context `'RAP'` contains a pattern `'Committed files'`.
The following relation has been declared

$$ctxcs \quad : \quad \texttt{Context} \sim \texttt{Concept} \tag{5.39}$$

, which means that a context may contain concepts.
For example, `'RAP'` *ctxcs* `'UserProfile'` means:
Context `'RAP'` contains a concept `'UserProfile'`.

5.6 PATTERN: Patterns

A requirements engineer uses pattern and process elements to group certain elements, e.g. rules and relations, in a logical manner. For example, the elements in a group may all relate to a single theme. Typically, patterns contain invariant rules to specify the semantics of information. Processes contain process rules to specify the semantics of processes. Process rules are defined like invariant rules, but then assigned to a role.

This version of the RAP model does not define the process element. Process elements have not been released to students as course material yet. RAP treats process elements as if they were patterns.[4]

[4]Note that RAP interprets scripts, it does not silently alter scripts e.g. change a process element in a script into a pattern.

A pattern relates to the following model elements: user-defined rules [section 5.12], user-defined generalization rules [section 5.7], relation declarations [section 5.10] and purposes of the pattern. Other kinds of elements may exist in a pattern or process in an Ampersand-script, but they are ignored or modelled elsewhere in this RAP model. For example, patterns and processes may contain a definition of a concept, which has been directly linked to the concept `Concept` by means of *cptdf* : `Concept` $\sim$ `Blob` [relation declaration 5.50]. And for example, a process may assign rules to a role, role elements are completely ignored in this RAP model.

This theme introduces 1 concept, 0 generalization rules, 5 relations with properties and 0 rules related to patterns in RAP. Figure 5.5 shows a conceptual diagram, which includes the elements introduced in this section; a concept `Gen` [section 5.7]; a concept `Declaration` [section 5.10]; a concept `Rule` [section 5.12]; 2 concepts `Conid`, `Blob` [section 5.13].

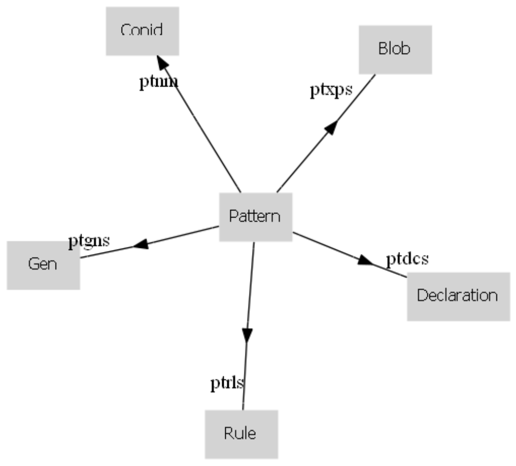


Figure 5.5: Concept diagram of Patterns

5.6.1 Defined concepts

This subsection defines 1 concept and 0 generalization rules. Each definition defines a concept followed by how the user interfaces display an atom *a* of that concept. This display value of *a* is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on *a*. Relation expressions are shaded. A key may be defined such that *a* is printed as an HTML-element or as regular text. If a prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view a [section 4.5].

Definition: A `Pattern` is a model element, in which a requirements engineer groups certain model elements in a logical manner. Atom $a \in$ `Pattern` is displayed - using relation 5.40 - as:

$a; ptnm$

5.6.2 Declared relations

This subsection introduces 5 declared relations with properties and a meaning.

The following univalent, total relation has been declared

$$ptnm \quad : \quad \texttt{Pattern} \rightarrow \texttt{Conid} \tag{5.40}$$

, which means that a pattern has a name.
The following relation has been declared

$$ptrls \quad : \quad \texttt{Pattern} \sim \texttt{Rule} \tag{5.41}$$

, which means that a pattern may contain rule definitions.
The following relation has been declared

$$ptgns \quad : \quad \texttt{Pattern} \sim \texttt{Gen} \tag{5.42}$$

, which means that a pattern may contain generalization rule definitions.
The following relation has been declared

$$ptdcs \quad : \quad \texttt{Pattern} \sim \texttt{Declaration} \tag{5.43}$$

, which means that a pattern may contain relation declarations.
The following relation has been declared

$$ptxps \quad : \quad \texttt{Pattern} \sim \texttt{Blob} \tag{5.44}$$

, which means that a pattern may have purpose descriptions in a natural language.

5.7 PATTERN: Generalization rules

A requirements engineers specifies a partial order of concepts [section 5.8]. They do so by defining is-a-relationships between two concepts. The transitive reflexive closure of these user-defined is-a-relationships results in a set of total orders of concepts. Two concepts in different total orders are disjoint. The model element to define an is-a-relationship between two concepts is called a generalization rule or just gen.

This theme introduces 1 concept, 0 generalization rules, 2 relations with properties and 1 rule related to generalization rules in RAP. Figure 5.6 shows a conceptual diagram, which includes the elements introduced in this section; a concept `Concept` and a relation *cptnm* [section 5.8]; a concept `Conid` [section 5.13].

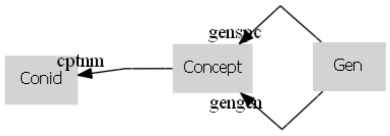


Figure 5.6: Concept diagram of Generalization rules

5.7.1 Defined concepts

This subsection defines 1 concept and 0 generalization rules. Each definition defines a concept followed by how the user interfaces display an atom *a* of that concept. This display value of *a* is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on *a*. Relation expressions are shaded. A key may be defined such that *a* is printed as an HTML-element or as regular text. If *a* prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view *a* [section 4.5].

Definition: A `Gen`, or generalization rule, is a model element to define the is-a-relation-ship between a more specific and a more generic concept. Atom $a \in$ `Gen` is displayed - using relations 5.48, 5.46, 5.45 - as:

SPEC $a;genspc;cptnm$ ISA $a;gengen;cptnm$

5.7.2 Declared relations

This subsection introduces 2 declared relations with properties and a meaning.

The following univalent, total relation has been declared

$$gengen \quad : \quad \texttt{Gen}{\rightarrow}\texttt{Concept} \tag{5.45}$$

, which means that a generalization rule has a generic concept.
For example, `'SPEC Horse ISA Animal'` *gengen* `'Animal'` means:
In `'SPEC Horse ISA Animal'` , `'Animal'` is the generic concept.

The following univalent, total relation has been declared

$$genspc \quad : \quad \texttt{Gen}{\rightarrow}\texttt{Concept} \tag{5.46}$$

, which means that a generalization rule has a specific concept.
For example, `'SPEC Horse ISA Animal'` *genspc* `'Horse'` means:
In `'SPEC Horse ISA Animal'` , `'Horse'` is the specific concept.

5.7.3 Defined rules

This subsection defines 1 formal rule.

eq gen - Identical user-defined generalization rules are not allowed, because it may confuse a requirements engineer without a valid reason. For example, a requirements engineer removes one of the identical gens under the assumption that the gen is actually removed, which is not the case. The engineer may experience unexpected behaviour and has to spend time to identify and correct the problem.

For the above, the following requirement has been defined:

There is only one generalization rule between a certain specific concept and a certain generic concept.

This is formalized - using the declared relations 5.46, 5.45 - as

$$gengen;gengen^{\smile} \cap genspc;genspc^{\smile} \subseteq \mathbb{I} \tag{5.47}$$

5.8 PATTERN: Concepts

Concepts are essential elements of declared relations [section 5.10] and relation expressions [section 5.11]. A concept is an abstract business term of which the population represents a certain set of business objects. Whether to model a concept or not is up to the requirements engineer. A requirements engineer may annotate a concept with a business definition or purposes.

The population of a concept is a set of atomic terms, also called atoms. An atom represents a business object. An atom of a concept is said to be of that kind in the business context. An atom that is not in a concept is said to be not of that kind of concept, because Ampersand assumes a closed world. For example, `'horse'` $\in$ `Animal` means that there are animals in the business context and there is a horse, which is a kind of animal.

In this example, “horse”' is an atom. The requirements engineer may also use the same term “Horse” as a concept, an abstract term, in order to talk about specific horses like `'Jolly Jumper'` $\in$ `Horse`. This kind of congruence between an atom and a concept cannot be expressed in Ampersand e.g. `'horse'` $\cong$ `Horse`. What can be expressed are generalization rules [section 5.7] like `Horse` $\leq$ `Animal`. Which means that, in that case, $a \in$ `Horse` implies $a \in$ `Animal`.

The structure and properties of Ampersand’s order of concepts do not have to be fully specified in the RAP model. RAP must only be able to determine whether a concept belongs to some order or not. e.g. `Horse` belongs to *the order of animals*. To determine that, we have specified the relation *order* : `Concept` $\sim$ `Order`, which is basically an element-of-relation. RAP names an order after the most generic concept in that order.

Let me give some remarks to avoid confusion with respect to the concepts `AtomID` and `Atom` and the abstract term “atom”. The abstract term “atom” applies to both concepts, the suffix “ID”'in “AtomID” exists just to make a distinction. RAP specifies an atom as a key-value pair. `AtomID` is the term for the keys, which represents the actual atoms c.q. the business object is an atom element. `Atom` is the term for the values, which represents the value of an atom element in an Ampersand-model c.q. the business object is a string recognized as the value of an atom. For example, a context has an atom `'Jolly Jumper'`, which represents a horse named Jolly Jumper. When that context has been opened in RAP, then `'Jolly Jumper'` $\in$ `AtomID`, which represents the atom element that represents the horse named Jolly Jumper in that context. And, `'Jolly Jumper'` $\in$ `Atom`, which represents the string “Jolly Jumper” recognized as the value of an atom. The key of an atom is unique in RAP, the value of an atom is unique within the unified population of all concepts in an order.

A user can edit the population of an Ampersand-script through the user interfaces [section 5.16]. Those who edit a population through the interfaces get feedback based on rule violations. Some of that feedback follows from rules defined in this pattern.

This theme introduces 4 concepts, 0 generalization rules, 7 relations with properties and 3 rules related to concepts in RAP. Figure 5.7 shows a conceptual diagram, which includes the elements introduced in this section; a concept `Gen` and 2 relations *gengen*, *genspc* [section 5.7]; 2 concepts `PairID`, `Sign` and 4 relations *src*, *trg*, *left*, *right* [section 5.9]; a concept `Declaration` and 2 relations *decsgn*, *decpopu* [section 5.10]; 3 concepts `String`, `Conid`, `Blob` [section 5.13].

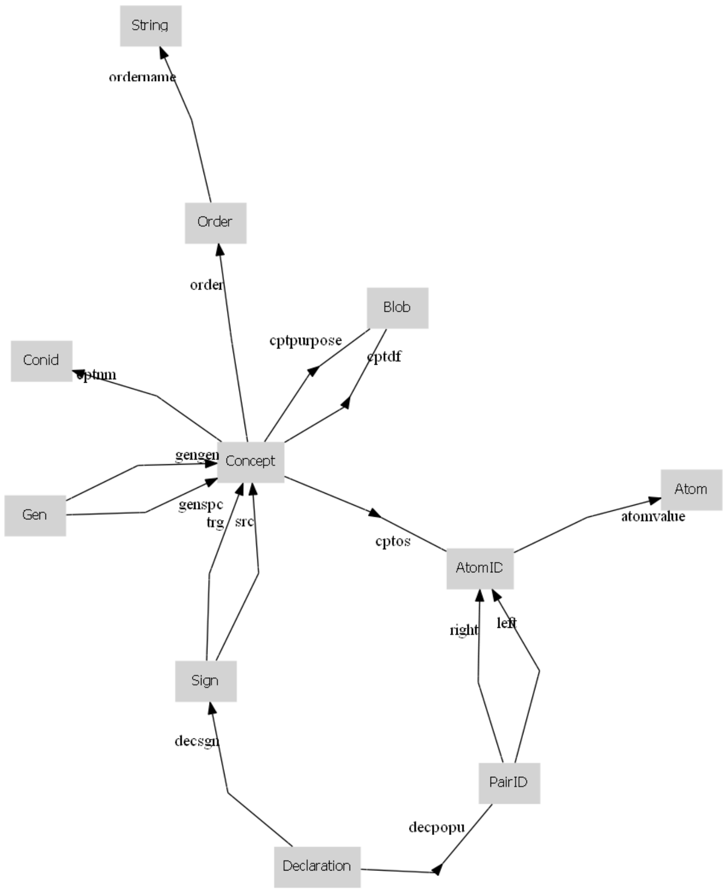


Figure 5.7: Concept diagram of Concepts

5.8.1 Defined concepts

This subsection defines 4 concepts and 0 generalization rules. Each definition defines a concept followed by how the user interfaces display an atom *a* of that concept. This display value of *a* is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on *a*. Relation expressions are shaded. A key may be defined such that *a* is printed as an HTML-element or as regular text. If *a* prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view *a* [section 4.5].

Definition: A `Concept` is a model element for an abstract business term of which the population represents a certain set of business objects. Atom $a \in$ `Concept` is displayed - using relation 5.48 - as:

$a; cptnm$

Definition: A `Order` is a structure for a group of is-a-related concepts. Atom $a \in$ `Order` is displayed - using relation 5.52 - as:

$a; ordername$

Definition: A `AtomID` is a model element for an atom. Atom $a \in$ `AtomID` is displayed - using relations 5.54, 5.52, 5.53, 5.49 - as:

$a; atomvalue$ :: $a; cptos^{\smile}; order; ordername$

Definition: A `Atom` is the value of an atom and an identifier of an atom within an order. Atom $a \in$ `Atom` is displayed as is.

5.8.2 Declared relations

This subsection introduces 7 declared relations with properties and a meaning.

The following univalent, total relation has been declared

$$cptnm \quad : \quad \texttt{Concept} \rightarrow \texttt{Conid} \tag{5.48}$$

, which means that a concept has a name.

The following relation has been declared

$$\mathit{cptos} \quad : \quad \mathtt{Concept} \sim \mathtt{AtomID} \tag{5.49}$$

, which means that the population of a concept may contain atom elements.
For example, `'Horse'` *cptos* `'443859690'` means:
The population of `'Horse'` contains an atom element identified by `'443859690'`.

The following relation has been declared

$$\mathit{cptdf} \quad : \quad \mathtt{Concept} \sim \mathtt{Blob} \tag{5.50}$$

, which means that a concept may have definitions in a natural language.

The following relation has been declared

$$\mathit{cptpurpose} \quad : \quad \mathtt{Concept} \sim \mathtt{Blob} \tag{5.51}$$

, which means that a concept may have purpose descriptions in a natural language.

The following univalent, total relation has been declared

$$\mathit{ordername} \quad : \quad \mathtt{Order} \rightarrow \mathtt{String} \tag{5.52}$$

, which means that an order has a name.

The following univalent, total relation has been declared

$$\mathit{order} \quad : \quad \mathtt{Concept} \rightarrow \mathtt{Order} \tag{5.53}$$

, which means that a concept belongs to an order.
For example, `'Horse'` *order* `'order of animals'` means:
Concept `'Horse'` belongs to the `'order of animals'`.

The following univalent, total relation has been declared

$$\mathit{atomvalue} \quad : \quad \mathtt{AtomID} \rightarrow \mathtt{Atom} \tag{5.54}$$

, which means that an atom element has a value.
For example, `'443859690'` *atomvalue* `'Jolly Jumper'` means:
The value of atom `'443859690'` is `'Jolly Jumper'`.

5.8.3 Defined rules

This subsection defines 3 formal rules.

order - The following requirement has been defined:
Is-a-related concepts belong to the same order.
This is formalized - using the declared relations 5.53, 5.45, 5.46 - as

$$order^{\smile};genspc^{\smile};gengen;order \subseteq \mathbb{I} \tag{5.55}$$

referential integrity - The following requirement has been defined:
An atom in the domain or codomain of a relation is an instance of a concept from the same order as the source respectively the target of that relation.
This is formalized - using the declared relations 5.49, 5.53, 5.61, 5.72, 5.64, 5.59, 5.60, 5.58 - as

$$\begin{split} src^{\smile};decsgn^{\smile};decpopu;left \cup trg^{\smile};decsgn^{\smile};decpopu;right \\ \subseteq order;order^{\smile};cptos \end{split} \tag{5.56}$$

RAP reports all rule violations $\langle a,b\rangle \in \neg(referentialintegrity)$ to the user. This report[5] starts with the following message

> If an atom is in some tuple of a relation, then that atom must exist in the concept that is the source respectively target of that relation. Deletion of an atom from a concept is not permitted if that atom is still present in some tuple of some relation. Nor is addition of a tuple permitted if the source or target atom is not present in the related concept. It is a violation of **Referential integrity** rule for a relation.

, followed by the following message for each $\langle a,b\rangle$.

> The tuple a refers to a source or target atom that does not exist.

entity integrity concept - The following requirement has been defined:
An atom of a concept is unique within the order of that concept.
This is formalized - using the declared relations 5.49, 5.53, 5.54 - as

$$atomvalue;atomvalue^{\smile} \cap cptos^{\smile};order;order^{\smile};cptos \subseteq \mathbb{I} \tag{5.57}$$

[5]This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.

RAP reports all rule violations $\langle a,b\rangle \in \neg(entityintegrityconcept)$ to the user. This report[6] starts with the following message

> Every atom of a concept is unique, or, no two atoms in the population of a concept have the same name. Addition of a duplicate atom is not permitted. It is a violation of the **Entity integrity** rule for this concept. Please refer to book *Rule Based Design*, page 43 and 52, *entity integrity*.

, followed by the following message for each $\langle a,b\rangle$.

> An atom with name a already exists.

5.9 PATTERN: Relation type signatures

Each declared relation [section 5.10] or relation expression [section 5.11] has a source concept and a target concept. A pair of a source and target concept is called a relation type signature, or just a sign.

The population of a relation is a typed collection of pairs of atoms. Each pair of atoms has a left atom of the relation's source and a right atom of the the relation's target. A pair only has meaning as an element in the population of a relation. So, to be meaningful, a pair must reflect to which relation it belongs in order to determine the meaning of a pair from a pair alone. For that reason, RAP specifies a pair in a similar way as an atom, that is, as a key-value pair. The key of a pair identifies a pair element in the population of a certain relation. The value of a pair is a pair of atoms.

This theme introduces 2 concepts, 0 generalization rules, 4 relations with properties and 0 rules related to relation type signatures in RAP. Figure 5.8 shows a conceptual diagram, which includes the elements introduced in this section; 3 concepts `Concept`, `AtomID`, `Atom` and 2 relations *cptnm*, *atomvalue* [section 5.8]; a concept `Conid` [section 5.13].

5.9.1 Defined concepts

This subsection defines 2 concepts and 0 generalization rules. Each definition defines a concept followed by how the user interfaces display an atom a of that concept. This display value of a is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on a. Relation

[6]This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.

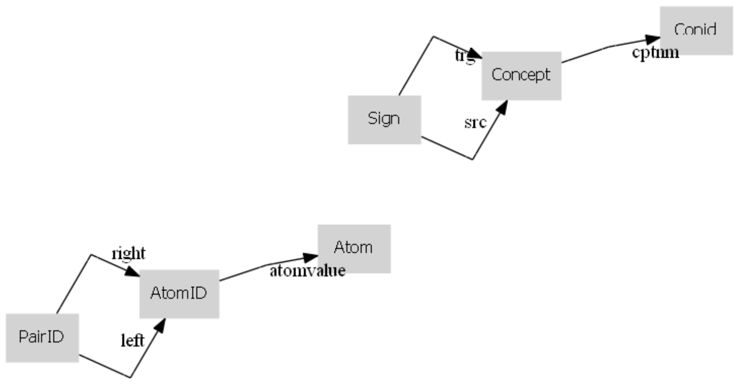


Figure 5.8: Concept diagram of Relation type signatures

expressions are shaded. A key may be defined such that *a* is printed as an HTML-element or as regular text. If *a* prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view *a* [section 4.5].

Definition: A `Sign`, or a relation type signature, is a model element for relation types.. Atom $a \in$ `Sign` is displayed - using relations 5.48, 5.58, 5.59 - as:

$a;src;cptnm$ * $a;trg;cptnm$

Definition: A `PairID` is a model element for a pair of atoms. Atom $a \in$ `PairID` is displayed - using relations 5.54, 5.60, 5.61 - as:

$a;left;atomvalue$ * $a;right;atomvalue$

5.9.2 Declared relations

This subsection introduces 4 declared relations with properties and a meaning.

The following univalent, total relation has been declared

$$src \quad : \quad \texttt{Sign} \rightarrow \texttt{Concept} \tag{5.58}$$

, which means that a sign has a source concept.

The following univalent, total relation has been declared

$$\mathit{trg} \quad : \quad \texttt{Sign}{\rightarrow}\texttt{Concept} \tag{5.59}$$

, which means that a sign has a target concept.
The following univalent, total relation has been declared

$$\mathit{left} \quad : \quad \texttt{PairID}{\rightarrow}\texttt{AtomID} \tag{5.60}$$

, which means that a pair of atoms has a left atom.
The following univalent, total relation has been declared

$$\mathit{right} \quad : \quad \texttt{PairID}{\rightarrow}\texttt{AtomID} \tag{5.61}$$

, which means that a pair of atoms has a right atom.

5.10 PATTERN: Relation declarations

The requirement engineer creates relation terms to use them in relation expressions [section 5.11], for example, to define rules [section 5.12]. A relation term is created by means of a relation declaration.

The requirements engineer may define certain properties of a relation as rules on its declaration e.g. the totality of a relation. The Ampersand language has predefined symbols for relation properties e.g. `'TOT'` for totality.

A requirements engineer may annotate a relation declaration with a template for example sentences, meaning and purpose. The template for example sentences defines a prefix text, infix text and suffix text, which are called a **pragma**. An example sentence is constructed from a pair $a\ r\ b$ by concatenating the pragma prefix, a, pragma infix, b and pragma suffix.

A requirement engineer defines and alters the population of declared relations for purposes like model validation and simulation. Because that is how business users would run a data administration and business processes on an Ampersand model, by maintaining the population of declared relations. A user can edit the population of declared relations through the user interfaces [section 5.16]. Those who edit a population through the interfaces get feedback based on rule violations. Some of that feedback is configured by means of rules defined in this pattern.

This theme introduces 3 concepts, 1 generalization rule, 10 relations with properties and 5 rules related to relation declarations in RAP. Figure 5.9 shows a conceptual diagram, which includes the elements introduced in this section; 3 concepts `Concept`, `AtomID`, `Order` and 3 relations *cptnm*, *cptos*, *order* [section 5.8]; 2 concepts `PairID`, `Sign` and 4 relations *src*, *trg*, *left*, *right* [section 5.9]; a concept `Rule` [section 5.12]; 4 concepts `String`, `Conid`, `Blob`, `Varid` [section 5.13].

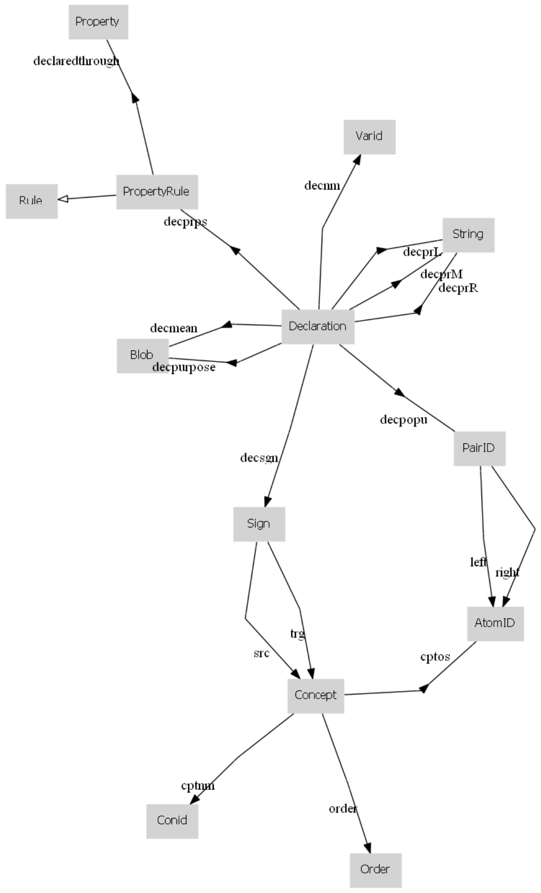


Figure 5.9: Concept diagram of Relation declarations

5.10.1 Defined concepts

This subsection defines 3 concepts and 1 generalization rule. Each definition defines a concept followed by how the user interfaces display an atom a of that concept. This display value of a is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on a. Relation expressions are shaded. A key may be defined such that a is printed as an HTML-element or as regular text. If a prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view a [section 4.5].

Definition: A `Declaration`, or a relation declaration, is a model element to declare a relation. Atom $a \in$ `Declaration` is displayed - using relations 5.63, 5.48, 5.58, 5.64, 5.59 - as:

$$a;decnm \ :: \ a;decsgn;src;cptnm \ * \ a;decsgn;trg;cptnm$$

Definition: A `Property` is a predefined symbol to define property rules. Atom $a \in$ `Property` is displayed as is.

Definition: A `PropertyRule` is a rule, that has been defined as a property. Atom $a \in$ `PropertyRule` is displayed as a `Rule`.

Rule: `PropertyRule` is a kind of `Rule`, which is formalized by the following generalization rule:

$$\texttt{PropertyRule} \leq \texttt{Rule} \tag{5.62}$$

5.10.2 Declared relations

This subsection introduces 10 declared relations with properties and a meaning.

The following univalent, total relation has been declared

$$decnm \ : \ \texttt{Declaration} \rightarrow \texttt{Varid} \tag{5.63}$$

, which means that a relation is declared with a name.
For example, `'r::A*B'` *decnm* `'r'` means:
The name of `'r::A*B'` is `'r'`.

The following univalent, total relation has been declared

$$decsgn \ : \ \texttt{Declaration} \rightarrow \texttt{Sign} \tag{5.64}$$

, which means that a relation is declared with a sign.

For example, `'r::A*B'` *decsgn* `'A*B'` means:
The sign of `'r::A*B'` is `'A*B'`.
The following injective relation has been declared

$$\mathit{decprps} \quad : \quad \texttt{Declaration} \sim \texttt{PropertyRule} \tag{5.65}$$

, which means that a relation may be declared with property rules.
For example, `'r::A*B'` *decprps* `'TOT r::A*B'` means:
`'r::A*B'` has a property `'TOT r::A*B'` .
The following total relation has been declared

$$\mathit{declaredthrough} \quad : \quad \texttt{PropertyRule} \sim \texttt{Property} \tag{5.66}$$

, which means that a property rule is defined by means of predefined symbols.
For example, `'TOT r::A*B'` *declaredthrough* `'TOT'` means:
Rule `'TOT r::A*B'` is defined by means of property symbol `'TOT'`.
The following univalent relation has been declared

$$\mathit{decprL} \quad : \quad \texttt{Declaration} \sim \texttt{String} \tag{5.67}$$

, which means that the meaning of a relation may be clarified with an example sentence that has a prefix text.
The following univalent relation has been declared

$$\mathit{decprM} \quad : \quad \texttt{Declaration} \sim \texttt{String} \tag{5.68}$$

, which means that the meaning of a relation may be clarified with an example sentence that has an infix text.
The following univalent relation has been declared

$$\mathit{decprR} \quad : \quad \texttt{Declaration} \sim \texttt{String} \tag{5.69}$$

, which means that the meaning of a relation may be clarified with an example sentence that has a suffix text.
The following relation has been declared

$$\mathit{decmean} \quad : \quad \texttt{Declaration} \sim \texttt{Blob} \tag{5.70}$$

, which means that a relation may have descriptions of its meaning in a natural language.

The following relation has been declared

$$decpurpose \quad : \quad \texttt{Declaration} \sim \texttt{Blob} \tag{5.71}$$

, which means that a relation may have purpose descriptions in a natural language.

The following relation has been declared

$$decpopu \quad : \quad \texttt{Declaration} \sim \texttt{PairID} \tag{5.72}$$

, which means that the population of a relation may contain pairs of atoms.

5.10.3 Defined rules

This subsection defines 5 formal rules.

eq declaration - The following requirement has been defined:
A declared relation can be identified by a name, a source concept, and a target concept.
This is formalized - using the declared relations 5.59, 5.64, 5.58, 5.63 - as

$$decnm;decnm^{\smile} \cap decsgn;src;(decsgn;src)^{\smile} \cap decsgn;trg;(decsgn;trg)^{\smile} \subseteq \mathbb{I} \tag{5.73}$$

property enum - The following requirement has been defined:
There are eleven predefined symbols to define property rules: -> means univalent and total; UNI means univalent; TOT means total; INJ means injective; SUR means surjective; RFX means reflexive; IRF means irreflexive; SYM means symmetric; ASY means antisymmetric; TRN means transitive; and PROP means symmetric and antisymmetric.
This is formalized - using the declared relations - as

$$\begin{aligned}\mathbb{I}_{\texttt{Property}} \subseteq {} & \texttt{'->'} \cup \texttt{'UNI'} \cup \texttt{'TOT'} \cup \texttt{'INJ'} \cup \texttt{'SUR'} \\ & \cup \texttt{'RFX'} \cup \texttt{'IRF'} \cup \texttt{'SYM'} \cup \texttt{'ASY'} \cup \texttt{'TRN'} \cup \texttt{'PROP'}\end{aligned} \tag{5.74}$$

entity integrity of relation - The following requirement has been defined:
There cannot be two pairs in a declared relation with the same left and same right.

This is formalized - using the declared relations 5.72, 5.61, 5.60 - as

$$left;left^{\smile} \cap right;right^{\smile} \cap decpopu^{\smile};decpopu \subseteq \mathbb{I} \tag{5.75}$$

RAP reports all rule violations $\langle a,b\rangle \in \neg(entityintegrityofrelation)$ to the user. This report[7] starts with the following message

> Every tuple in a relation is unique, or, no two tuples in the population of a relation may have the same source and target atoms. Addition of a duplicate tuple is not permitted. It is a violation of the Entity integrity rule for this relation.

, followed by the following message for each $\langle a,b\rangle$.

> A tuple with the same source and target atoms a already exists.

typed domain - The following requirement has been defined:
The left atoms of pairs in a declared relation belong to the same order as the source of that relation.
This is formalized - using the declared relations 5.53, 5.58, 5.64, 5.49, 5.60, 5.72 - as

$$decpopu;left;cptos^{\smile};order \subseteq decsgn;src;order \tag{5.76}$$

RAP reports all rule violations $\langle a,b\rangle \in \neg(typeddomain)$ to the user. This report[8] starts with the following message

> You try to add a tuple with a source atom, that is not in the population of the source of the relation. This is a violation of the type of the tuple. TIP: enter text in the left input field to get a shorter pick list. Note on ISA-relations: You can make an atom more specific by moving it to the population of a more specific concept.

, followed by the following message for each $\langle a,b\rangle$.

[7] This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.
[8] This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.

> Source atom b is not in the population of $a;\mathit{decsgn};\mathit{src}$

typed codomain - The following requirement has been defined:
The right atoms of pairs in a declared relation belong to the same order as the target of that relation.
This is formalized - using the declared relations 5.53, 5.59, 5.64, 5.49, 5.61, 5.72 - as

$$\mathit{decpopu};\mathit{right};\mathit{cptos}^{\smile};\mathit{order} \subseteq \mathit{decsgn};\mathit{trg};\mathit{order} \tag{5.77}$$

RAP reports all rule violations $\langle a,b\rangle \in \neg(\mathit{typedcodomain})$ to the user. This report[9] starts with the following message

> You try to add a tuple with a target atom, that is not in the population of the target of the relation. This is a violation of the type of the tuple. TIP: enter text in the right input field to get a shorter pick list. Note on ISA-relations: You can make an atom more specific by moving it to the population of a more specific concept.

, followed by the following message for each $\langle a,b\rangle$.

> Target atom b is not in the population of $a;\mathit{decsgn};\mathit{trg}$

5.11 PATTERN: Expressions

The declared relations are used in relation expressions e.g. to define rules [section 5.12]. The relation term to refer to a declared relation is the name of that relation. An explicit relation type signature might be needed to determine the intended type of a relation. The sign of a relation in an expression might differ from its declaration when is-a-relations are involved. For example, a relation *name* : `Animal` $\sim$ `Name` can be used to express the more specific relation $\mathit{name}_{\texttt{Horse}\times\texttt{Name}}$ in case `Horse` $\leq$ `Animal`. An expression has not been modelled in full detail yet. RAP only links relation declarations to the expressions in which they are used.

This theme introduces 3 concepts, 0 generalization rules, 5 relations with properties and 1 rule related to expressions in RAP. Figure 5.10 shows a conceptual diagram,

[9] This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.

which includes the elements introduced in this section; a concept `Concept` and a relation *cptnm* [section 5.8]; a concept `Sign` and 2 relations *src*, *trg* [section 5.9]; a concept `Declaration` and a relation *decnm* [section 5.10]; 2 concepts `Conid`, `Varid` [section 5.13].

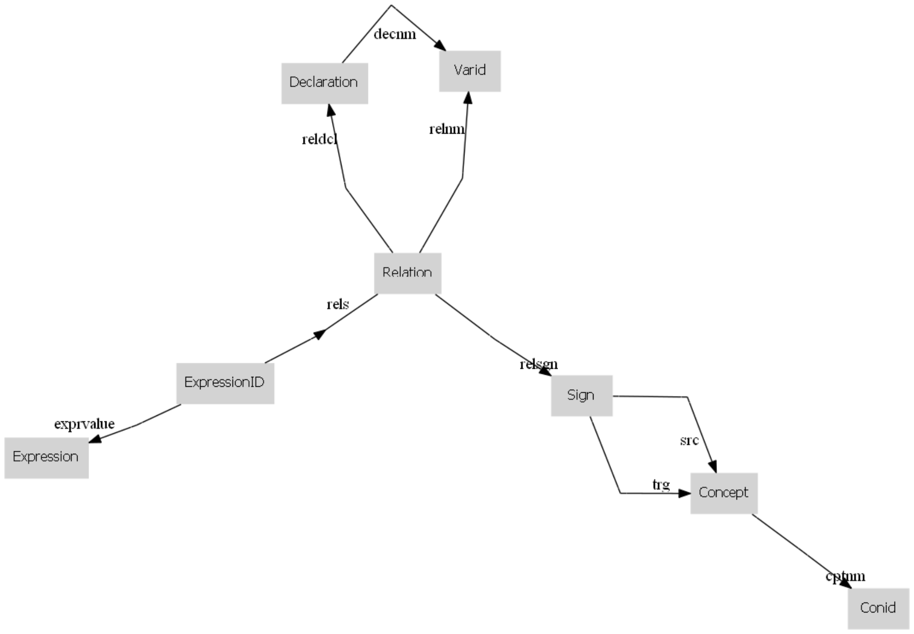


Figure 5.10: Concept diagram of Expressions

5.11.1 Defined concepts

This subsection defines 3 concepts and 0 generalization rules. Each definition defines a concept followed by how the user interfaces display an atom *a* of that concept. This display value of *a* is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on *a*. Relation expressions are shaded. A key may be defined such that *a* is printed as an HTML-element or as regular text. If *a* prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view *a* [section 4.5].

Definition: A `ExpressionID` is a model element for a relation. Atom $a \in$ `ExpressionID` is displayed - using relation 5.78 - as:

$a;exprvalue$

Definition: A `Expression` is the relation expression, which is a relation expression written in Ampersand ASCII syntax. Atom $a \in$ `Expression` is displayed as is.

Definition: A `Relation` is a relation term in an expression that has a declaration. Atom $a \in$ `Relation` is displayed - using relations 5.80, 5.48, 5.58, 5.81, 5.59 - as:

$a;relnm$ [$a;relsgn;src;cptnm$ * $a;relsgn;trg;cptnm$]

5.11.2 Declared relations

This subsection introduces 5 declared relations with properties and a meaning.

The following univalent, total relation has been declared

$$exprvalue \quad : \quad \texttt{ExpressionID}{\rightarrow}\texttt{Expression} \tag{5.78}$$

, which means that an expression element has a value, which is a relation expression.
For example, `'RULE RFX r::A*A: r |- I'` *exprvalue* `'r |- I'` means:
Relation `'RULE RFX r::A*A: r |- I'` is expressed as `'r |- I'`.

The following relation has been declared

$$rels \quad : \quad \texttt{ExpressionID} \sim \texttt{Relation} \tag{5.79}$$

, which means that an expression uses relations that have been declared.
For example, `'RULE RFX r::A*A: r |- I'` *rels* `'r[A*A]'` means:
`'RULE RFX r::A*A: r |- I'` uses `'r[A*A]'` in its expression.

The following univalent, total relation has been declared

$$relnm \quad : \quad \texttt{Relation}{\rightarrow}\texttt{Varid} \tag{5.80}$$

, which means that a relation has a name.
For example, `'r[A*A]'` *relnm* `'r'` means:
The name of `'r[A*A]'` is `'r'`.

The following univalent, total relation has been declared

$$relsgn \quad : \quad \texttt{Relation}\rightarrow\texttt{Sign} \tag{5.81}$$

, which means that a relation has a sign.
For example, `'r[A*A]'` *relsgn* `'A*A'` means:
The sign of `'r[A*A]'` is `'A*A'`.

The following univalent, total relation has been declared

$$reldcl \quad : \quad \texttt{Relation}\rightarrow\texttt{Declaration} \tag{5.82}$$

, which means that a relation has a declaration.
For example, `'r[A*A]'` *reldcl* `'r::A*A'` means:
`'r[A*A]'` has been declared by `'r::A*A'`.

5.11.3 Defined rules

This subsection defines 1 formal rule.

rel name is decl name - The following requirement has been defined:
The name of a relation is the same as the name in its declaration.
This is formalized - using the declared relations 5.63, 5.82, 5.80 - as

$$relnm \equiv reldcl;decnm \tag{5.83}$$

5.12 PATTERN: Rules

A requirements engineer defines a rule as a relation expression. A rule definition $R_{A\sim B}$ means that, given a population of the context, $[R_{A\sim B}]$ holds if and only if $R = V_{A\times B}$. Given a population of the context, any pair $\langle a,b\rangle \in V_{A\times B}$, but not $\langle a,b\rangle \in R_{A\sim B}$ is a violation of the rule [section 5.14]. Thus, the violations of a rule can be defined by the complement of its relation expression ($\langle a,b\rangle \in \neg R_{A\sim B}$). The requirements engineer may annotate a rule with a meaning and purpose.

This theme introduces 1 concept, 0 generalization rules, 4 relations with properties and 0 rules related to rules in RAP. Figure 5.11 shows a conceptual diagram, which includes the elements introduced in this section; a concept `ExpressionID` [section 5.11]; 2 concepts `Blob`, `ADLid` [section 5.13].

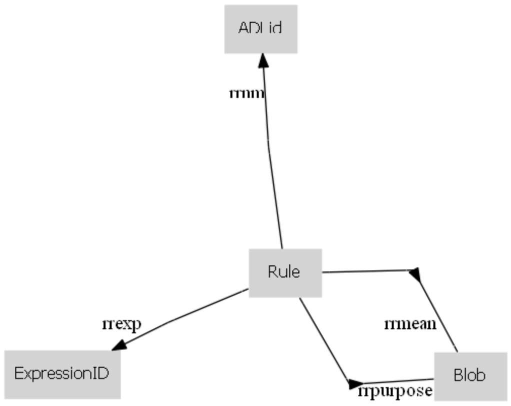


Figure 5.11: Concept diagram of Rules

5.12.1 Defined concepts

This subsection defines 1 concept and 0 generalization rules. Each definition defines a concept followed by how the user interfaces display an atom *a* of that concept. This display value of *a* is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on *a*. Relation expressions are shaded. A key may be defined such that *a* is printed as an HTML-element or as regular text. If *a* prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view *a* [section 4.5].

Definition: A `Rule` is a model element, which defines a rule by means of a relation expression. Atom $a \in$ `Rule` is displayed - using relation 5.84 - as:

$a; rrnm$

5.12.2 Declared relations

This subsection introduces 4 declared relations with properties and a meaning.

The following univalent, total relation has been declared

$$rrnm \quad : \quad \texttt{Rule} \rightarrow \texttt{ADLid} \tag{5.84}$$

, which means that a rule has a name.

For example, `'RULE RFX r::A*A'` *rrnm* `'RFX r::A*A'` means:
The name of `'RULE RFX r::A*A'` is `'RFX r::A*A'`.

The following univalent, total relation has been declared

$$rrexp \ : \ \texttt{Rule} \rightarrow \texttt{ExpressionID} \tag{5.85}$$

, which means that a rule has a relation expression to express that rule.
For example, `'RULE RFX r::A*A'` *rrexp* `'RULE RFX r::A*A: r |- I'` means:
The relation expression of `'RULE RFX r::A*A'` is `'RULE RFX r::A*A: r |- I'`.

The following relation has been declared

$$rrmean \ : \ \texttt{Rule} \sim \texttt{Blob} \tag{5.86}$$

, which means that a rule may have descriptions of its meaning in a natural language.
For example, `'RULE RFX r::A*A'` *rrmean* `'r::A*A is reflexive'` means:
`'RULE RFX r::A*A'` means `'r::A*A is reflexive'`.

The following relation has been declared

$$rrpurpose \ : \ \texttt{Rule} \sim \texttt{Blob} \tag{5.87}$$

, which means that a rule may have purpose descriptions in a natural language.

5.13 PATTERN: Symbols

The Ampersand language has five syntactic domains for user-defined model element names: String, Blob, Conid, Varid and ADLid.

This theme introduces 5 concepts, 0 generalization rules, 0 relations with properties and 0 rules related to symbols in RAP.

5.13.1 Defined concepts

This subsection defines 5 concepts and 0 generalization rules. Each definition defines a concept followed by how the user interfaces display an atom *a* of that concept. This display value of *a* is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on *a*. Relation expressions are shaded. A key may be defined such that *a* is printed as an HTML-element or as regular text. If *a* prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view *a* [section 4.5].

Definition: A `String` is text restricted to a maximum of 256 characters. Atom $a \in$ `String` is displayed as is.

Definition: A `Blob` is text, which may exceed 256 characters. Atom $a \in$ `Blob` is displayed as is.

Definition: A `Conid` is a string starting with an uppercase. Atom $a \in$ `Conid` is displayed as is.

Definition: A `Varid` is a string starting with a lowercase. Atom $a \in$ `Varid` is displayed as is.

Definition: A `ADLid` is string, which may be of the kind Varid, Conid, or String. Atom $a \in$ `ADLid` is displayed as is.

5.14 PATTERN: Calculated details

An Ampersand-model includes calculated model elements like rule violations, conceptual diagrams and example sentences for relations. The Ampersand compiler derives such elements from a script. There is an intuitive distinction between calculated model elements and other compiler-calculated elements e.g. metrics [section 5.17] or design artefacts.

Rule violations are the drivers of the generic process of Ampersand-based systems [section 2.3]. RAP lets a user step through that process [section 5.15], which requires a concept of rule violations.

The purpose of a conceptual diagram is to give people a visual overview over relations and concepts. For example, the diagrams used throughout this chapter are conceptual diagrams, which provide a visual cue to read a section.

An example sentence is a textual aid to understand or memorize the meaning of a relation. An example sentence is based on the pragma in a relation declaration [section 5.10].

More kinds of calculated model elements exist, but are not used in RAP.

This theme introduces 4 concepts, 1 generalization rule, 6 relations with properties and 0 rules related to calculated details in RAP. Figure 5.12 shows a conceptual diagram, which includes the elements introduced in this section; a concept `Pattern` [section 5.6]; a concept `Concept` [section 5.8]; a concept `PairID` [section 5.9]; a concept `Declaration` [section 5.10]; a concept `Rule` [section 5.12].

5.14.1 Defined concepts

This subsection defines 4 concepts and 1 generalization rule. Each definition defines a concept followed by how the user interfaces display an atom a of that concept.

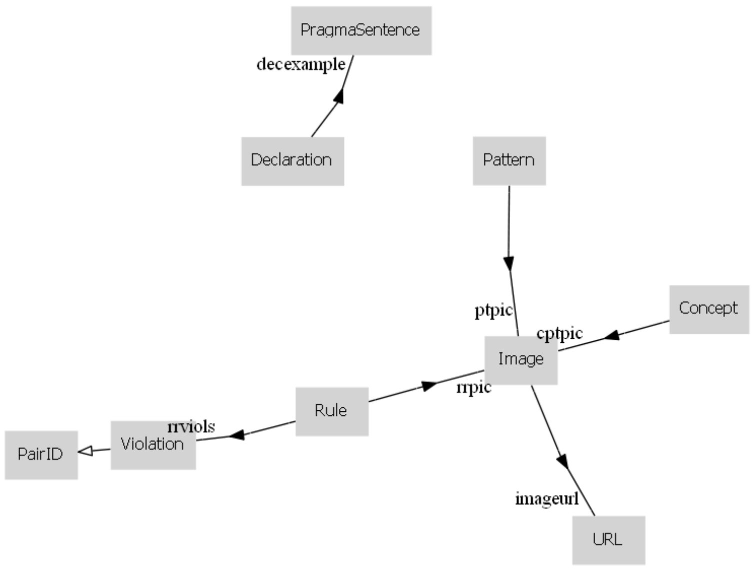


Figure 5.12: Concept diagram of Calculated details

This display value of *a* is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on *a*. Relation expressions are shaded. A key may be defined such that *a* is printed as an HTML-element or as regular text. If *a* prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view *a* [section 4.5].

Definition: A `Violation` is a pair of atoms of a special kind of relation, that is, the complement of a rule expression. Such a pair violates the rule. Atom $a \in$ `Violation` is displayed as a `PairID`.

Rule: `Violation` is a kind of `PairID`, which is formalized by the following generalization rule:

$$\texttt{Violation} \leq \texttt{PairID} \tag{5.88}$$

Definition: A `Image` is a digital representation of a diagram or figure. Atom $a \in$ `Image` is displayed - using relation 5.93 - as:

<img src=' *a*;*imageurl* '>

Definition: A `URL`, or unified resource location, is a web address. Atom $a \in$ `URL` is displayed as is.

Definition: A `PragmaSentence` is an example sentence for a relation to clarify its meaning. Atom $a \in$ `PragmaSentence` is displayed as is.

5.14.2 Declared relations

This subsection introduces 6 declared relations with properties and a meaning.

The following relation has been declared

$$\mathit{rrviols} : \texttt{Rule} \sim \texttt{Violation} \tag{5.89}$$

, which means that a rule may have violations.
The following univalent relation has been declared

$$\mathit{ptpic} : \texttt{Pattern} \sim \texttt{Image} \tag{5.90}$$

, which means that a pattern may have a conceptual diagram to visualize that pattern.
The following univalent relation has been declared

$$\mathit{cptpic} : \texttt{Concept} \sim \texttt{Image} \tag{5.91}$$

, which means that a concept may have a conceptual diagram to visualize that concept.
The following univalent relation has been declared

$$\mathit{rrpic} : \texttt{Rule} \sim \texttt{Image} \tag{5.92}$$

, which means that a rule may have a conceptual diagram to visualize that rule.
The following relation has been declared

$$\mathit{imageurl} : \texttt{Image} \sim \texttt{URL} \tag{5.93}$$

, which means that an image may be found on web addresses.

The following relation has been declared

$$decexample : \texttt{Declaration} \sim \texttt{PragmaSentence} \tag{5.94}$$

, which means that a relation may have example sentences to clarify its meaning.

5.15 PROCESS: Testing rules

An Ampersand-model describes a rule-based system, which runs on an iterative process of three steps: test rules, distribute rule violations and act upon violations [section 2.3]. A user can step through this process as follows. A new process starts when the user commits an Ampersand-script with a population. RAP tests the rules in the script against the population. Next, RAP gives the user feedback on rule violations, which may include directives to distribute and act. The user distributes and acts upon violations by changing the population or the rules. The population can be changed through user interfaces [section 5.16] or in a text editor. The user commits the changes, which starts the next iteration.

This theme defines different process rules to give different kinds of feedback on different kinds of violations. More intelligent feedback can be defined likewise as we learn more about the didactics for Ampersand [chapter 6].

This section gives a purpose to introduce 5 rules related to testing rules in RAP. Figure 5.13 shows a conceptual diagram, which includes the declared relations used to define the 5 process rules.

5.15.1 Defined rules

This subsection defines 5 formal process rules for the user.

multviolations1 - The following requirement has been defined:

The user gets feedback on the violations of total and surjective property rules. This is formalized - using the declared relations 5.89, 5.66 - as

$$\begin{aligned}&\neg((\mathbb{I}_{\texttt{PropertyRule}} \cap declaredthrough;(\texttt{'TOT'} \cup \texttt{'SUR'})\\&;declaredthrough^{\smile});rrviols)\end{aligned} \tag{5.95}$$

RAP reports all rule violations $\langle a,b\rangle \in \neg(multviolations1)$ to the user. This report[10] starts with the following message

[10]This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.

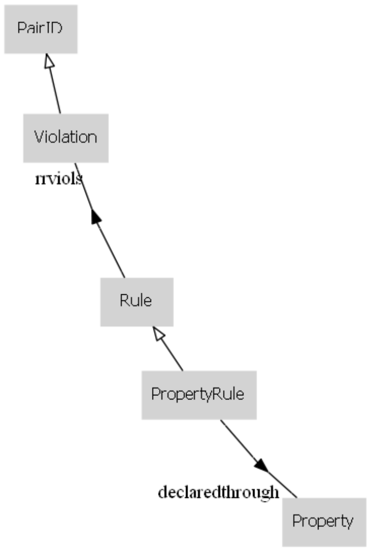


Figure 5.13: Concept diagram of Testing rules

A TOTal or SURjective multiplicity rule is violated for some relation(s).
Add tuple(s) in the relation(s) to correct the violation(s).

, followed by the following message for each $\langle a,b\rangle$.

RULE a is violated by the atom b; *left*.

multviolations2 - The following requirement has been defined:
The user gets feedback on the violations of univalent and injective property rules.
This is formalized - using the declared relations 5.89, 5.66 - as

$$\neg((\mathbb{I}_{\texttt{PropertyRule}} \cap declaredthrough;(\texttt{'UNI'} \cup \texttt{'INJ'}) \\ ;declaredthrough^{\smile});rrviols) \tag{5.96}$$

RAP reports all rule violations $\langle a,b\rangle \in \neg(multviolations2)$ to the user. This report[11] starts with the following message

> **A UNIvalent or INJective multiplicity rule is violated for some relation(s).** Delete tuple(s) in the relation(s) to correct the violation(s).

, followed by the following message for each $\langle a,b\rangle$.

> RULE a is violated by the tuple b in combination with some other tuple(s).

multviolations3 - The following requirement has been defined:
The user gets feedback on the violations of functional property rules.
This is formalized - using the declared relations 5.89, 5.66 - as

$$\neg((\mathbb{I}_{\texttt{PropertyRule}} \cap declaredthrough;\texttt{'->'};declaredthrough^{\smile});rrviols) \tag{5.97}$$

RAP reports all rule violations $\langle a,b\rangle \in \neg(multviolations3)$ to the user. This report[12] starts with the following message

> **A UNIvalent or TOTal multiplicity rule is violated for some relation(s).** Delete tuple(s) in the relation(s) to correct the violation(s).

, followed by the following message for each $\langle a,b\rangle$.

> RULE a is violated by the tuple b in combination with some other tuple(s).

homoviolations - The following requirement has been defined:
The user gets feedback on the violations of homogeneous property rules.
This is formalized - using the declared relations 5.89, 5.66 - as

$$\begin{aligned}&\neg((\mathbb{I}_{\texttt{PropertyRule}} \cap declaredthrough\\&;(\texttt{'RFX'} \cup \texttt{'IRF'} \cup \texttt{'SYM'} \cup \texttt{'ASY'} \cup \texttt{'TRN'} \cup \texttt{'PROP'})\\&;declaredthrough^{\smile});rrviols)\end{aligned} \tag{5.98}$$

[11]This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.
[12]This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.

RAP reports all rule violations $\langle a,b\rangle \in \neg(homoviolations)$ to the user. This report[13] starts with the following message

> **A rule for homogeneous relation(s) is violated.** Add or delete tuple(s) in the relation(s) to correct the violation(s).

, followed by the following message for each $\langle a,b\rangle$.

> RULE a is violated by the tuple b in combination with some other tuple(s).

otherviolations - The following requirement has been defined:
The user gets feedback on the violations of non-property rules.
This is formalized - using the declared relation 5.89 - as

$$\neg((\neg\mathbb{I}_{\mathtt{PropertyRule}} \cap \mathbb{I}_{\mathtt{Rule}});rrviols) \tag{5.99}$$

RAP reports all rule violations $\langle a,b\rangle \in \neg(otherviolations)$ to the user. This report[14] starts with the following message

> **A business rule that involves several relations is violated.** Add or delete tuple(s) in one or more of the relation(s) to correct the violation(s).

, followed by the following message for each $\langle a,b\rangle$.

> RULE a is violated by the tuple b in combination with some other tuple(s).

5.16 PROCESS: Editing a population

A user may edit the population of a context through the user interfaces or in a text editor.

In a text editor, a user directly edits the Ampersand-script. Some changes are better made in a text editor like copy-pasting larger pieces of code into a script.

The advantage of editing through the user interfaces is that these activities are governed by the rules of RAP. In other words, the user gets rule-based feedback while

[13]This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.
[14]This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.

editing. For that reason, we have the objective to edit and create the entire model through the user interfaces. The reason that this RAP model only allows for editing of the population is a matter of practical development capacity and priorities. Especially the editing of expressions requires significant development efforts like implementing a (visual) expression editor.

This section gives a purpose to introduce 1 rule related to editing a population in RAP. Figure 5.14 shows a conceptual diagram, which includes the declared relations used to define the 1 process rules.

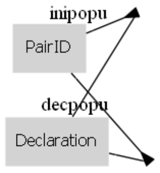


Figure 5.14: Concept diagram of Editing a population

5.16.1 Defined rules

This subsection defines 1 formal process rule for the user.

popchanged - This rule guides a user who has started to make changes to the population through the user interfaces. A user needs to commit those changes explicitly. Automatically committing each single change to a population is undesired, because a single change in a population may cause awkward violations in a context. For example, a requirements engineer tries to resolve a violation in a context. The engineer has planned to first add a pair to a relation and next delete a pair from another relation. If RAP would automatically commit after each change, then RAP may report all kinds of violations after adding the pair to a relation, which would make no sense to the requirements engineer. After-all the requirements engineer has already planned to make more changes. RAP cannot decide when to commit a set of changes, c.q. the user has to decide.

For the above, the following requirement has been defined:

The user gets feedback on uncommitted changes to a population.

This is formalized - using the declared relations 5.72, 5.22 - as

$$inipopu \equiv decpopu \tag{5.100}$$

RAP reports all rule violations $\langle a,b\rangle \in \neg(popchanged)$ to the user. This report[15] starts with the following message

You have made changes to the population. You can:
1) **enter more** change(s), or;
2) **undo** your changes **by (re)loading** any CONTEXT into Atlas, or;
3) Click here to **commit** the change(s) **and update** violations on your rules.

, followed by the following message for each $\langle a,b\rangle$.

added or deleted pair b of $b;(\mathit{inipopu} \cup \mathit{decpopu})^{\smile}$

5.17 PATTERN: Metrics

We use a framework for metrics [chapter 7] to add metrics to RAP. Such metrics are primarily used to produce information for research as described in chapter 6. For example, several counters, change vectors and metrical percepts are products of such metrics. Such information may be useful for requirements engineers as well. In other words, it may be useful to report on measurements to requirements engineers. Currently, RAP gives the requirements engineer information about the number of rules, concepts and relations in a context [section 5.5]. These metrics support a requirements engineer with an activity called cycle chasing [39].

Subsection 7.2 describes how to add a metric to RAP using the framework.

This theme introduces 1 concept, 0 generalization rules, 3 relations with properties and 0 rules related to metrics in RAP. Figure 5.15 shows a conceptual diagram, which includes the elements introduced in this section; a concept `Context` [section 5.5].

5.17.1 Defined concepts

This subsection defines 1 concept and 0 generalization rules. Each definition defines a concept followed by how the user interfaces display an atom a of that concept. This display value of a is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on a. Relation

[15]This report is the actual feedback to users of RAP and is illustrative to the reader of this thesis.

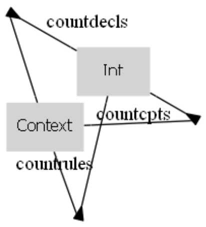


Figure 5.15: Concept diagram of Metrics

expressions are shaded. A key may be defined such that *a* is printed as an HTML-element or as regular text. If *a* prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view *a* [section 4.5].

Definition: A `Int` is a number. Atom $a \in$ `Int` is displayed as is.

5.17.2 Declared relations

This subsection introduces 3 declared relations with properties and a meaning.

The following univalent relation has been declared

$$\textit{countrules} \quad : \quad \texttt{Context} \sim \texttt{Int} \tag{5.101}$$

, which means that the number of rule definitions in a context may have been calculated.

The following univalent relation has been declared

$$\textit{countdecls} \quad : \quad \texttt{Context} \sim \texttt{Int} \tag{5.102}$$

, which means that the number of relation declarations in a context may have been calculated.

The following univalent relation has been declared

$$\textit{countcpts} \quad : \quad \texttt{Context} \sim \texttt{Int} \tag{5.103}$$

, which means that the number of concepts in a context may have been calculated.

5.18 Conclusion

This chapter is the documentation of RAP and has been generated from the RAP model. For that, we have used the same Ampersand compiler that generates RAP from the RAP model.

The documentation of RAP yields examples of what can be defined in Ampersand using the mechanisms presented in chapter 4. We claim that RAP implements Ampersand, for that reason we conclude that Ampersand can be defined in and documented by means of an Ampersand-model.

To generate this chapter shows how design automation can help to document the design. There is a guarantee that the documentation of RAP remains up-to-date with RAP, because both are generated from the same source. The documentation of RAP is complete, consistent and faultless e.g. correct cross-references and up-to-date diagrams, which can be validated by comparing this chapter to its source, the RAP model in appendix A. The thematic organization on patterns and processes in an Ampersand-model shows that complete and consistent documentation can be better organized for human digestion.

Chapter 6

USING RAP FOR EDUCATION

In order to validate whether RAP supports learning, measurements have been performed in a course on rule-based design. For this reason, quantitative data was collected during the learning process. The purpose was to find out whether this data contains meaningful information about the learning process, in order to establish the usability of RAP as a platform for studying the didactics of Ampersand. Throughout the operational life of RAP, data has been collected automatically, providing us with longitudinal data and an objective means to collect these data.

This chapter studies our claim that RAP is useful for studying the learning behaviour of students, who are learning to design information systems with Ampersand. This claim has been studied by observing how students formalize requirements in design exercises.

6.1 Background on the course

RAP has been in use since 2010 in the master course at the OUNL called "Ontwerpen met Bedrijfsregels" ("Rule-Based Design"). This course teaches students to design information systems by means of business rules. They use Ampersand as a vehicle to model business rules, using RAP as their development platform. While working on a project, they commit subsequent versions of their Ampersand-model to their own workspace in RAP. They use it to validate a model, to obtain diagnostic reports, to generate functional requirements specifications, or to generate a prototype which exhibits behaviour that Ampersand derives from the rules.

Differences among the student population of the RBD course complicate the teaching process. Our students are typically 25 to 40 years of age and work a full time job. The learning context is individual learning in four different settings: physical classrooms, virtual classrooms, individual learning, and in-company. Another difference is the curriculum in which the course is taught. The RBD course is taught in the Master of Computer Science and in the Master of Business Process Management and IT, both at the Open University of the Netherlands (OUNL). The in-company courses have been taught outside a formal curriculum, with at most 12 students per course. These differences explain why students start with very different levels of knowledge and interests. Nevertheless, all students must satisfy the same criteria at the end of the course.

During the course a student learns

- the formal language of Ampersand, which is built on relation algebra;
- to understand the meaning of their Ampersand-model, which is a specific interpretation of the algebra;
- how to design information structures and business processes as a collection of rules;
- how to express those rules in the Ampersand language.

A student gets instructions through course books and a wiki environment. The course book on Ampersand [39] is available online for free. A student is taught to produce a model, which contains rules that represent functional requirements. As the course progresses, the student works on a single requirements model of a business context from his own professional environment. 80% of the final grade is based on this design exercise to which we simply refer as the **Design Exercise**. Most of the data in RAP has been identified and related to the Design Exercise of a student. In our study we have focused on the data set that we could identify and relate to the Design Exercise.

The Design Exercise formulates two deliverables: One, a document with a description in natural language of a business situation where certain business rules are relevant The document must describe useful business vocabulary and relevant business rules. Two, an Ampersand-model that formalizes that business situation.

The following is a typical example of a business situation for the Design Exercise: the processing of a Request For Change (RFC). (1) A Requester creates an RFC, which gets assigned to a Change Manager instantly. (2) The Change Manager rejects or approves the RFC. (3) The Change Manager assigns the RFC to a Developer. (4) The Developer processes the request and closes the RFC. (5) The Requester gets notified about all status changes of the RFC.

A student is instructed to put deliverables on the wiki environment, mentioned earlier. Deliverables on the wiki are available to other students and the teachers, who may put comments on the work or learn from what others have created. We have used the deliverables on the wiki to better identify and relate the data in RAP to the Design Exercise of a particular student.

6.2 Research Approach

Two challenges complicate studying the learning behaviour of students. The first is not having a well established didactical structure for rule-based design. The second is the physical absence of students in a distance learning situation. So, the first question to be answered was whether it is possible at all to obtain meaningful information from the use of RAP. This requires an exploratory approach. Since RAP is a platform intended for non-classroom situations, the obvious choice was to collect data as students work their way through their assignments. Before asking any other research question, we first needed to know how meaningful that data is in the first place.

In order to study that question, the approach has been taken to analyse sequences of Ampersand-models committed to the repository of RAP by students. We have formed sequences by grouping Ampersand-models by their name and their creator, the student. Each sequence has been ordered by their creation time in the repository. As such each sequence represents a history of design activities of a student on an Ampersand-model with a particular name. That is way we call such a sequence a **trace** of the creation of an Ampersand-model and each element in a trace a **version** of that Ampersand-model.

In order to analyse traces, we have defined metrics on traces collected between April 2010 and April 2011. The resulting measurements have been analysed and observations about learning behaviour were formulated. By discussing these observations with teachers and practitioners, we have learned lessons about learning behaviour. This work involved speculating with experts about the didactics of formal methods in general, relational algebra and particularly Ampersand.

The observations and lessons learned prove the viability of RAP for studying the didactics of Ampersand, but says little about the truth of our observations. That is, the interpretation of our measurements as observations in terms of learning behaviour is not exact. However, proving the viability of RAP was precisely the purpose of our research. In order to find a correct interpretation of metrics as an observation about learning behaviour requires further research, which is beyond the scope of this dissertation due to time constraints. As the body of data grows, the same data or newly gathered data (or both) can be used later to conduct quantitative research on the didactical problems of teaching a formal method such as Ampersand.

To summarize the four steps of our approach described so far. One, we have formed traces from Ampersand-models committed to RAP. Two, we have defined metrics on those traces to obtain measurements to analyse. Three, we have analysed the measurements to formulate observations. And four, we have discussed these observations to learn lessons.

To accommodate these lessons, we have added a fifth step: refine RAP with lessons learned. The fifth step leads us back to step one in order to validate the refinements and learn more lessons.

During our research, one cycle by our approach has been completed. We have named the five steps: harvesting traces, defining metrics, formulating observations, learning lessons and refining RAP.

Harvesting traces Each time a student commits a change on an Ampersand-model, that code is stored in RAP as a new version of the model. This results in a construction trace of the model. Reasons for taking this approach were:

- Tracing is done completely. All students participate.
- This measurement is non-intrusive. Logging actions in RAP does not interfere with the learning process.
- The measurement is objective. The researcher has no influence on the data that is collected.

The conceptual design of a construction trace in RAP [chapter 7] is kept as a chronological sequence of files. Thus, common concepts in revision control systems[1] like branches and merges have been omitted. Each successive file in RAP holds the next version of an Ampersand-script.

Defining metrics In order to analyse traces we have defined metrics. The metrics have been implemented by combined functions in the Ampersand compiler and standard commercial spreadsheet software. The Ampersand compiler produced all quantitative measurements by executing the metrics on the traces. The spreadsheet software has been used to produce graphics and pivot tables from these measurements.

In order to get more variety in measurements, three types of metrics have been used:

- properties of one version of a model,
- differences between two subsequent versions, and
- representations of a sequence of versions.

[1]`http://en.wikipedia.org/wiki/Revision_control#Common_vocabulary`

In a next research cycle, we can define these three types of metrics in the RAP model and have measurements on traces in RAP itself. For that we have defined a measurement framework [chapter 7] that extends the RAP model. The definition of a metric in that framework adds functionality to store and present measurements in RAP, which are related to traces and Ampersand-models. Functions can be added to the Ampersand compiler to produce the measurements in RAP, which may include functions to produce graphics and pivot tables. Section 7.2 explains how to define and implement a metric in RAP.

Formulating observations and Learning lessons The exploratory measurements gave rise to various observations. These observations may be studied separately, to determine whether they can be accepted as lessons learned in the practice of teaching. In fact, this is the stage where most of the ideas for teaching rule-based design have been formed. For that reason, the observations made here are an interesting research result by themselves, which are of course in need of validation.

Refining RAP The observations and lessons learned give input to refine RAP. Such refinements may be validated in the next cycle of harvesting, defining, formulating, learning and refining. These refinements require much flexibility to adapt RAP to progressive insight. That flexibility is obtained by generating RAP from the RAP model, as described in chapter 4. A significant reduction of development efforts was gained by generating RAP. It has allowed us to modify RAP frequently without subjecting the students to error-infested software.

For the future we see potential to extend RAP with constraint-based feedback [24]. Such feedback can be defined in the RAP model by means of process rules [section 4.3] and integrity rules [section 4.4.2].

Another opportunity is the implementation of metrics in RAP. Only by adding a relation expression to an interface for students [section 4.5], students get access to the measurements in RAP that we are using for research. Sharing statistic data with students to enhance learning is a form of learning analytics [10].

6.3 A first exploration of traces in RAP

We have conducted an exploratory study following the proposed approach (section 6.2). This study has two interrelated objectives:

- validate whether RAP facilitates progressive research and development;
- formulate observations about the behaviour of individual students in order to contribute to teaching rule-based design.

To address the first objective, we describe how RAP has facilitated each step taken so far. Formulated observations and contributions for further research will be presented as results in the next section.

6.3.1 Step 1: harvesting traces

In order to obtain an empirical basis for this research, the first release of RAP (RAPv1) has been collecting versions of models since April 2010. After a year, that constituted a data set, consisting of 6947 versions in 172 traces of 52 identified students of the course linked to 72 RAP user accounts.

6.3.2 Step 2: defining metrics

Various counters and qualifiers have been defined in the measurement framework of RAP to explore the data set. For example, there are counters for the core elements of an Ampersand-model: concepts, relations and rules. The study has focused on design behaviour of students, coding, and usage of RAP.

To explore design behaviour, we have defined a heuristic visualization of a trace. It combines a variety of measures, which we call a **metrical percept**. Figure 6.1 is an example of a metrical percept, which plots five counters for five kinds of elements in an Ampersand-model. The x-axis represents a sequence of versions of a model. The y-axis represents the number of elements in a version. The label on the x-axis describes the change between two successive models with a *change vector*. The position in the vector relates to a collection of elements in the model; the symbol qualifies the change in that collection as only more elements ($+$), only less ($-$), more and less (1), or no change (0)[2]. For example, the 23rd label 13hour(+01-0) in figure 6.1 means: *after 13 hours the student committed the next correct model to RAP, which has more concepts, the same properties on previously existing relations, modified relations, less rules, and the same relationships on previously existing relations.*

With the available research data (6947 student models, organized in 172 traces), 18 metrical percepts could be obtained by filtering traces according to the following criteria:

- A trace can be related to the design exercise of a student in the course.
- There are more than 10 versions committed to that trace.
- The design exercise of a student has been published on the wiki environment of the course before we finalized the data set resulting from step 1.

[2]Equivalence of elements is defined on a semantic level e.g. the name of a concept, the signature of a relation.

The 18 percepts obtained turned out to be from 18 different students. Six of them were participants of a business and IT programme for self-tuition. Two were participants of a business and IT programme in a classroom setting. Ten students participated in a virtual class room setting. Figure 6.2 provides an overview that shows 18 miniatures of the percepts. We refer to a particular percept in figure 6.2 by its column-row position, e.g. A1 for the percept in column A on row 1.

6.3.3 Step 3 and 4: formulating observations and learning lessons

What does the evidence tell us? Discussing the metrical percepts with tutors and students has allowed us to formulate 6 observations. These are based on the counters, qualifiers, and metrical percepts of step 2.

1. A metrical percept exhibits different types of student behaviour.
2. Students have difficulty to write correct code.
3. The user-friendliness of RAP needs improvement.
4. Change vectors can be used to qualify the changes made by students for measuring the behaviour of students.
5. The start of a trace can predict some type of student behaviour.
6. A high activity level in the first session predicts successful completion of the course within a limited period of time.

These observations are true in the eyes of the researchers[3], which is based on the evidence and the evaluations with tutors and students. Since the evidence (i.e. 6947 versions of student scripts) was used to formulate these observations, and only 18 metric percepts could be harvested from them, more evidence must be collected to validate these observations. For this reason, we claim face validity only to the truth of these observations.

6.3.4 Step 5: refining RAP

The majority of adjustments to RAPv1 concerned including more structure and actions in the RAP model. This has yielded more information that can be linked to a trace. It has also reduced the amount of manually coded and maintained software in RAP. For example, managing model code files has become part of the generated software of RAPv2. The resulting RAP platform (RAPv2) is in use since May 2012.

[3]Truth of these observations requires a separate quantitative analysis, based on fresh data. That validation is beyond our scope and requires further research.

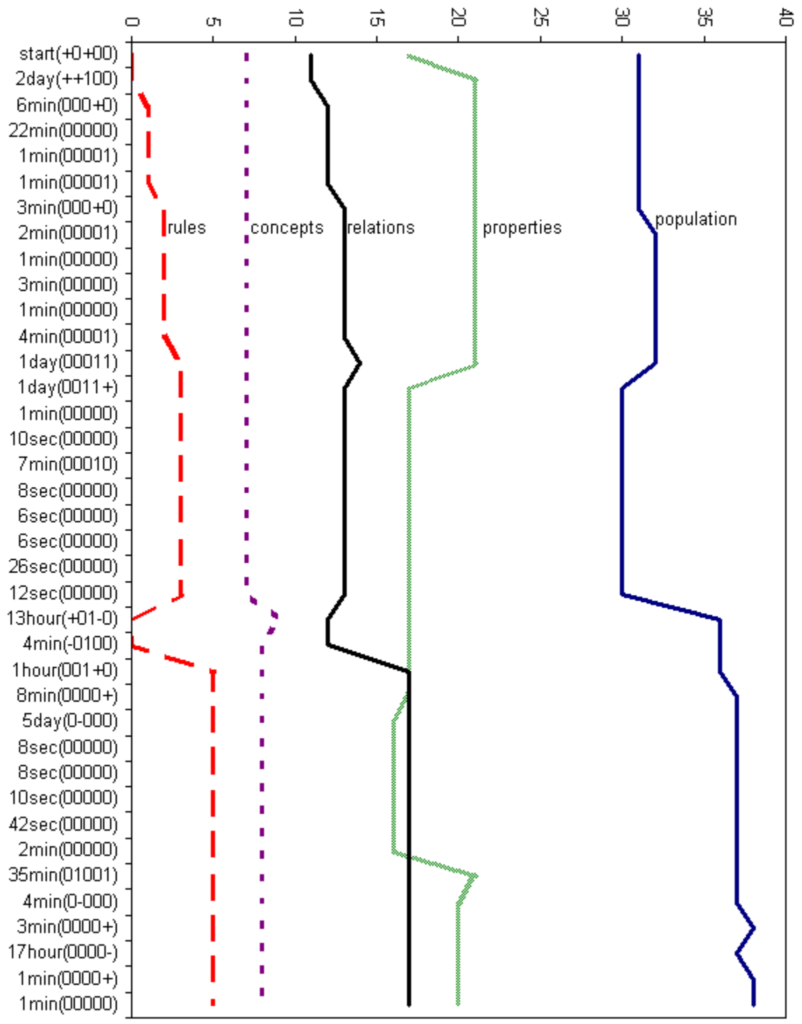


Figure 6.1: Enlarged metrical percept A5 of figure 6.2 (rotated)

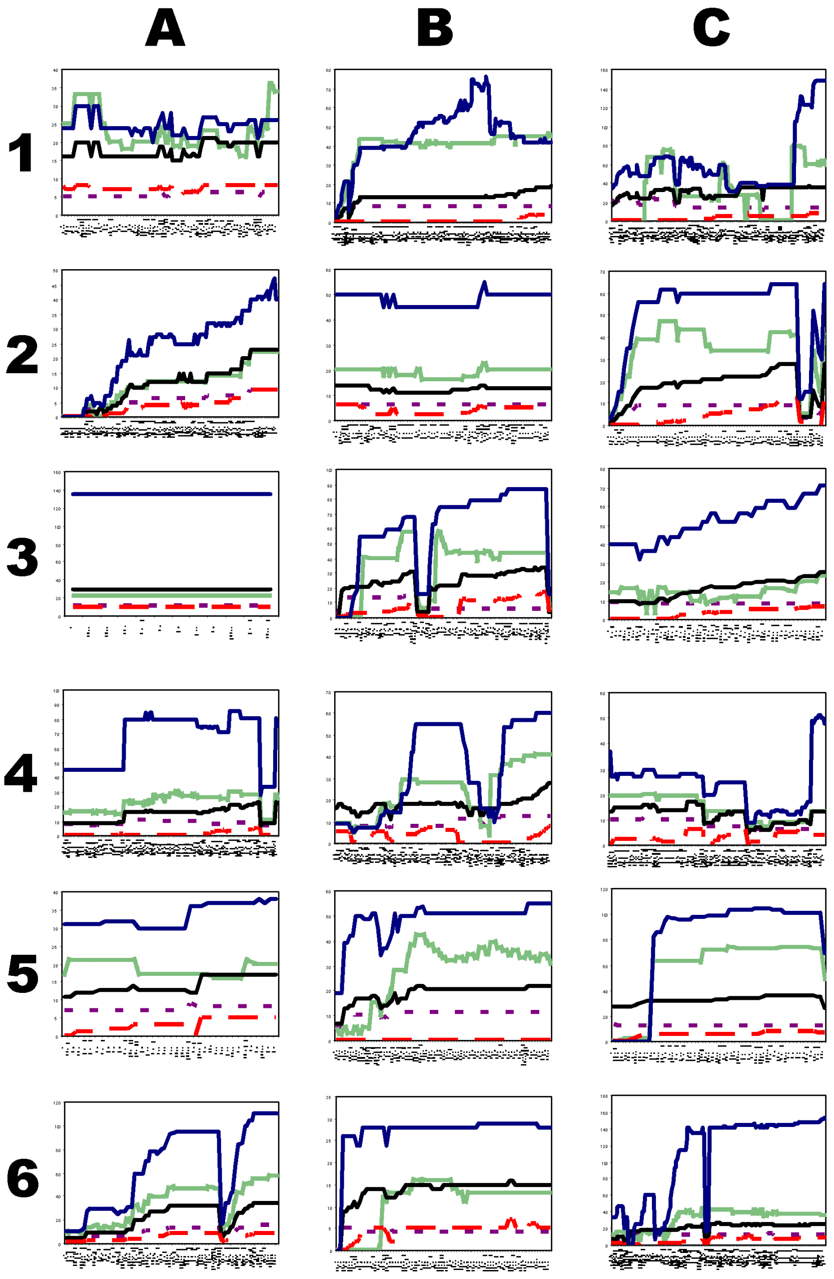


Figure 6.2: 18 metrical percepts of completed design exercises

In RAPv1, students must submit a new model if they wish to make changes. A student edits an Ampersand script in an ASCII text editor and commits that script for exploration in RAP. RAPv2 features editing of the population of a model through user interfaces c.q. with feedback. A student may alter the population to validate or play with a model. Feedback for changing the population has been defined to validate the concept of constraint based feedback in RAP. Didactically correct feedback needs to be accounted for.

Some of the observations of step 3 have led to specialistic adjustments, see section 6.4.

6.3.5 Further steps

The results so far require a larger empirical base for further investigation. For this reason, harvesting of new traces is continuing, yielding an increasing body of versions ready to be analysed. RAPv1 is still running in parallel and contains an unused data set equivalent to the one of from step 1. RAPv2 has opened possibilities to study modifications to constraint based feedback and two new design activities, testing and playing with a model.

6.4 Observations and Lessons Learned

This study has produced two relevant results. The first result is that RAP has been built, tried, tested and refined. It is in actual use in the course on rule-based design at the OUNL. It has proven to be usable for studying the learning behaviour in this course. RAP has also proven flexible enough to adapt to progressive understanding by the researchers. The traces and the metrical percepts derived from them demonstrate that meaningful observations can be formulated with RAP.

The second result relates to the observations about the student's learning behaviour. These observations were formulated while changes were being made to the metrics on traces.

Observation 1: A metrical percept exhibits different types of student behaviour

A metrical percept (i.e. a picture as shown in figure 6.1) is intended to be the initial input to cognitive processes. It results in a better understanding of student behaviour.

Based on our cognition of the 18 percepts in figure 6.2, we propose three kinds of behavioural perceptions upon a metrical percept:

- construction
  A students starts with a small foundation and expands, which visualizes as increasing lines e.g. percept A2;

- jump start
  A student defines a comprehensive model and adjusts, which visualizes as horizontal lines e.g. percept A1;
- various kinds of special behaviour, that emerge as a dip e.g. percept B3.

Inspired by this practical experience, we propose a fourth perception:

- reduction, define an over-specific model and reduce, which visualizes as a shrinking model.

We have investigated the scripts near dips in order to find causes of such dips. However, patterns of behaviour seem to be too diverse to draw general conclusions. Some examples of behaviour that has caused a dip are: remodelling or reconstruction; temporarily comment elements to isolate a coding or modelling problem; experiment with an isolated part of a model.

We would expect that metrical percepts show behaviour that complies with the Ampersand method. For example, Ampersand advocates to gradually extend a model with more structure and rules. We would expect to see percepts that show constructive behaviour. However the 18 percepts show diverse behaviour. So either students do not adhere to Ampersand or these percepts do not visualize the desired behaviour. That question is definitely interesting, but requires a separate experiment to answer. Our idea is to assess percepts of traces of qualified designers, who have done the design exercise while adhering to Ampersand. When those percepts do show common behaviour then a reference percept for Ampersand can be formulated. Such a reference percept might be a helpful tool for teachers in a distance learning situation.

The metrical percept and its details are modelled as measures in the measurement framework, which is part of the RAP model. Measurements in the framework can be stored in RAP. Recall that RAP is an open platform to connect analytic tools to the RAP database. Such tools may keep a metrical percept and its details up-to-date with the latest activities of a student. For example, RAPv1 and RAPv2 contain diagnostic information for students, which are real-time analytics kept up-to-date by the Ampersand compiler.

Observation 2: Students have difficulty to write correct code

Despite the fact that Ampersand has deliberately few syntactic constructs, only half of all code files is correct. Table 6.1 shows that 3760 of the 6947 versions in the RAP repository are correct. 1606 versions are syntactically correct, but not semantically. 1574 versions are not syntactically correct. Is this normal in programming practice? Do students require more support? Further investigation can answer such questions and reveal what causes this magnitude of errors.

correct code	3760
syntactic error	1574
semantic error	1606
compiler fatal	7
total	6947

Table 6.1: Correct scripts

Learning how to code in Ampersand is not a learning objective, but a necessity to design with Ampersand. Therefore students may be given maximal support to minimize the time to learn Ampersand. Michels et.al. [22] formulated clear, but concise error messages for meaningless code committed to RAPv1. RAPv2 includes a user interface to display script error messages to mitigate the trade-off that had to be made between clear and concise. This means that students get a general message from which they can navigate to more detailed information by their needs. This adjustment aims at a higher percentage of correct scripts in RAPv2.

Observation 3: The user-friendliness of RAP needs improvement

Table 6.2 shows that at least 987 out of 3760 changes are semantically identical. Two successive versions are semantically identical if the change vector between them is (00000), which we qualify as *nothing*. So what does this say about learning behaviour? Why do students submit semantically identical models? In practice, committing a (00000)-change may make sense incidentally e.g. rename a rule. Over 25% of the changes cannot be called incidentally any more, which implies malfunctions of students designing with RAP. This can be blamed on an unnatural user interface for RAP functions, bad instructions for designing with RAP, insufficient efforts by students to understand designing with RAP, or any combination of those three. For any of those causes, a decrease in (00000)-changes would mean an improvement.

Discussions with teachers and students suggest that the number of semantically identical changes may be a measure of the user-friendliness of Ampersand. The available evidence cannot corroborate this claim, so further research is needed.

Observation 4: Change vectors can be used to qualify the changes made by students for measuring the behaviour of students

An actual change is every change vector that is not (00000). A simple change is every change vector that contains exactly four zeros. For example, if a student has only changed the rules, the change vector is (00010). A complex change is every

nothing	987
population	964
rules	583
relations	425
concepts or properties	364
relations and rules	306
relations, rules and population	77
relations and population	54
total	3760

Table 6.2: Scope of change

change vector that contains less than four zeros. There are $3760 - 987 = 2773$ actual changes. 1905 of 2773 changes are simple. 868 of 2773 changes are complex.

Table 6.3 shows various kinds of interesting behaviour based on counting change vectors.

- Over 1350 simple changes concern changing the population, i.e. (00001) (0000+) (0000-), or rules, i.e. (00010) (000+0). Population is needed to valuate a rule in a meaningful way. Therefore, a common activity in RAP appears to be formulating or validating rules.

- The complex changes in the top-12 of changes all concern adding more elements to a model. Thus, at least 2071 of 2773 changes are either simple or concern adding elements. This is evidence that students mainly design online using RAP and not off-line using a local text editor. Because, if a student would mainly design off-line, one would expect more complex changes.

- Behaviour towards rules is different than behaviour towards properties. A property is a simple rule, which is a rule that concerns only one relation e.g. totality. The difference between a property and a simple rule is purely syntactic. Table 6.3 shows that changing (00010) and adding (000+0) rules is a common change, but deleting (000-0) them is not in the top-12 at all. Despite that properties are semantically like rules, they are treated differently. Deleting properties (0-000) is more common than changing (01000) or adding (0+000) them.

Change vectors can be correlated to a certain kind of impact on the design. For example, an experimental correlation of change vectors with impact on rule violations in a model has been realized. Interesting conclusions could not be made yet with the current number of traces.

(	0	0	0	0	1	)	407
(	0	0	0	1	0	)	333
(	0	0	0	0	+	)	277
(	0	0	0	+	0	)	203
(	0	-	0	0	0	)	153
(	0	0	0	0	-	)	136
(	+	0	+	0	0	)	131
(	0	1	0	0	0	)	108
(	0	+	0	0	0	)	103
(	+	0	+	+	0	)	94
(	0	0	+	0	0	)	68
(	0	0	1	0	0	)	58
							2071

Table 6.3: Top-12 changes

Printing the change vectors in the labels of the x-axis of a percept allows for manual investigation of a sequence of changes. Analytic functions might be implemented to search for patterns in sequences of change vectors.

Observation 5: The start of a trace can predict some type of student behaviour

If a kind of behaviour can be determined from change vectors in an early stage of the trace, then intelligent feedback in RAP may be tuned to that kind of behaviour. The first change vector may already hint to a certain kind of behaviour. For example, for the 18 traces of figure 6.2: 3 of 18 start with nothing; 14 of the remaining 15 start with a population; 6 of 15 start with rules; and 13 of 15 start with property rules. The 3 students who have started with nothing are candidates for construction behaviour. The 6 students who have started with rules are candidates for jump start behaviour. These observations suggest prediction of behaviour by early inspection of change vectors. That is interesting in a distance learning situation.

Observation 6: A high activity level in the first session predicts successful completion of the course within a limited period of time

4 of 18 percepts in figure 6.2 show a daily usage of RAP for around one week only. 1 of those 4 percepts shows 12 versions in total, which is significantly less than the other 17 percepts. The median for the other 14 percepts is a month, ranging from several weeks until almost a year. These 14 percepts can roughly be divided into halves. One

half shows a continuous usage of RAP on a less frequent basis than daily. The other half shows around three series of daily activity with pauses of several days or months. The 14 percepts show activity on 10 to 20 distinct days.

3 of 4 percepts that show usage of RAP for only one week show over 30 versions on the first session or two, the other 15 percepts do not. We have counted a new session when the next version is committed after a period of 5 hours or more. 9 of the other 15 percepts never show over 30 versions in one session. Therefore, we classify 30 versions in one session as *high activity*. We expect that a high activity at the start of a trace can determine whether a student has postponed the design exercise until the end of the course or not.

A perception of sessions could have been visualized in a metrical percept. For example, a bar can be drawn between two sessions. This adjustment has not been made yet.

6.5 Conclusion

This chapter aims at validation of our five step development cycle with RAP. The idea behind our approach is that usage of tools for design exercises gives rise to measuring facts about student behaviour. Such facts can be used to study student behaviour and enhance those tools for better education.

To execute that idea, RAP has been developed and used for a course on rule-based design. RAP integrates user interfaces for design exercises with analytics upon a trace of design products. RAP requires to be responsive to the maturing didactics of rule-based design with limited development capacity. Responsiveness has been mitigated to the design of a domain-specific ontology to which learning and analytic components can easily be connected up to semantics of rule-based design exercises.

Four conclusions can be drawn from this study:

Feasibility to study student behaviour In the first three steps of our study, RAP needed to support our search for initial metrics to obtain measured facts about student behaviour. A framework was added to configure metrics upon a trace of design products. The framework accounts for metrics with three types of input: a version of a model, two subsequent version, or a sequential trace of versions. We aimed at a rich visual representation of a trace, which resulted in a metrical percept. On a generic level, a percept is just a metric in the framework, which combines various other metrics in the framework e.g. change vectors, number of rules. On a domain-specific level, metrics need to be interpreted in the context of student behaviour. Student behaviour may concern coding, designing, learning, or tool handling. Our metrics are still raw products, which can be interpreted as facts about student behaviour in an informal manner. Metrics with a formal interpretation to student behaviour are required to discuss and study student behaviour. We envision studies to effects on student behaviour due to changes in RAP. For example, the effects of changes in constraint based feedback might be studied. Or, the counter of correct code can be used as a performance indicator.

Lessons about teaching Rule-Based Design The real reason for conducting this research was to learn more about teaching a formal method, Ampersand, to students. The observations made by analysing traces contain a number of relevant lessons, even though more validation is desirable:

1. A metrical percept exhibits different types of student behaviour.
   To have an instrument by which student behaviour can be observed is highly relevant in a distance learning environment. Additional observations of that behaviour is imperative, if this instrument is to be useful. Additional validation is required. Nevertheless, results so far are promising, and have lead to new understanding and significant improvements of the course already.

2. Students have difficulty to write correct code.
   This observation is not new to the authors. It was the real reason for taking the effort to do learning analytics in the first place. RAP has shown to provide more understanding as to where student difficulties lie. Besides, it gives tutors an instrument to measure all students, rather than paying attention to those who ask questions.

3. The user-friendliness of RAP needs improvement.
   Many improvements with respect to user friendliness have been made already. RAP will tell us whether these improvements have an effect on learning.

4. Change vectors can be used to qualify the changes made by students for measuring the behaviour of students.
   Working with change vectors has proven effective and useful. This aspect is making it possible to measure behaviour along multiple dimensions.

5. The start of a trace can predict some type of student behaviour.
   This observation is likely to enable early intervention by tutors.

6. A high activity level in the first session predicts successful completion of the course within a limited period of time.
   This observation too, is useful to support early impressions by tutors of how a student is doing.

The nature of these observations illustrates the capabilities of RAP to facilitate didactical studies with facts about student behaviour. The authors see potential in metrics like the “metrical percept” and the change vector to measure student behaviour. Eventually, a metrical percept might turn out to be a meaningful instrument for teachers and examiners of this course.

Evaluation of developments to learning environment Most adjustments to RAP concern enhanced functionality for design exercises. Minor adjustments have been made to RAP due to the six observations. RAP includes diagnostic feedback upon design products of a student, which has been configured on metrics in the framework. Like diagnostic feedback, a percept can be reported to a student via the user interfaces of RAP to enhance their learning by the ideas of learning analytics. Educative feedback can be configured in RAP as constraint based feedback based on knowledge obtained from studies to student behaviour with RAP. This has convinced the authors that RAP is responsive enough to support design exercises of the course on rule-based design, now and in the future.

Assessment of design The ontology of RAP is defined in Ampersand. An Ampersand compiler is used to generate software components for that ontology. An external component can be connected to interfaces generated from the ontology of RAP. We have connected custom programs in Haskell and spreadsheets to RAP. A component can be designed in Ampersand; included in the design of RAP; and generated. User interfaces and the framework to configure metrics are included in the design of RAP. We claim that without software generation, we could not have reached an acceptable level of responsiveness. Furthermore, the central ontology served as a backbone that allowed us to stay in control over semantic and software changes.

General conclusion RAP has fulfilled the requirements for progressive development in the course on rule-based design. Students of that course have a dedicated environment for design exercises with the infrastructure in place for constraint based feedback and learning analytics. The development of RAP with Ampersand is key to this success. RAP has potential to be used for studies to learning and design behaviour. Ampersand might be used to develop design exercise environments like RAP e.g. an environment to design an UML class diagram. Ampersand is less suitable for environments that require extensive development efforts to define a domain-specific ontology, such as an environment for Java programming exercises. Knowledge and tools of Ampersand are required and freely available via `wiki.tarski.nl`. The Ampersand-model and software of RAP is freely available at request.

Chapter 7

SPECIFICATION OF RAP EXTENSION FOR METRICS

Well-documented measurements are needed to study the learning behaviour of students as described in chapter 6. This chapter presents a framework for metrics on Ampersand projects in RAP, called **the framework**. The framework has been modelled in Ampersand and can be included by the RAP model, so that metrics can become part of the RAP model. The first section of this chapter has been generated from the Ampersand-model of the framework like the previous chapter has been generated from the RAP model. As part of the RAP model, the meaning of measurements and measurement results is documented and their semantics within RAP are formally defined. Since measurement results in RAP are no different than other data in RAP, any user of RAP can be given access to measurement results and use it to their benefit. For example, a student may reflect on its own learning behaviour, a requirements engineer may reflect on its design behaviour, or a teacher may monitor students. These examples are opportunities of the framework. For now, the framework has primarily been used to study the behaviour of students.

A researcher defines Ampersand projects on adl-files in RAP and metrics on Ampersand projects. An Ampersand project is basically a chain of adl-files, which we call a **trace** of adl-files. A metric or **metric relation** is a univalent relation on an element of an Ampersand project, which represents a measurement function on that element of a project. An element of a project can be an adl-file, anything in or related to an adl-file, two subsequent adl-files or a chain of adl-files. For example, the study described in chapter 6 uses several counters on adl-files, change vectors of subsequent adl-files and metrical percepts of traces. Also, the relations defined in section 5.17 are metrics. Section 7.2 explains how to add a metric to RAP.

7.1 PATTERN: Ampersand projects

This pattern defines the entire framework for metrics on Ampersand projects. The framework introduces concepts that represent elements of Ampersand projects like chains of adl-files in a project, two subsequent files in a project or two successive files in a project. These concepts are needed so that metric relations can be defined upon them, that is, such a concept can become the source of a metric relation.

An Ampersand project is a set of adl-files from a requirements engineer, in which each file contains an Ampersand-model for the same business context. We may speak of an adl-file in a project as a **version** of an Ampersand-model. Versions in RAP cannot be changed or removed from RAP, only new versions can be added to RAP. Therefore, RAP has similarities with source code repositories. However, RAP does not have all the characteristics of a source code repository like the ability to make a branch or merge.

A researcher can manually define traces, which might be projects. For the study in chapter 6, we have only defined traces, which we assumed to be Ampersand projects. For example, a trace of the error-free adl-files of a user with a common context name. In this example, we assume that the combination of a user and context name of an adl-file identifies a model, which makes that file a version of that model. The versions with script errors are simply left out of the trace.

In this section, we use a,b,c,d as variables for an adl-file and x as a variable for a trace.

A trace has been modelled as a set of links. A link is a triple of two adl-files and a trace, that represents successive versions in a trace. The researcher only needs to define the set of links to define a trace. For example, `'⟨ a,b,x ⟩'` $\in$ `Link` represents two successive versions in trace x, where b is the successor of a. a is called the left of the link and b the right. All chains and subsequent files in a trace are calculated from a trace by connecting the left of a link to an equal right of a link, that is, a reflexive transitive closure on the links of a trace. The reflexive transitive closure of a relation R is expressed in Ampersand as R^*. The longest chain in the reflexive transitive closure on the links of a trace represents the trace itself. So, a researcher needs to define a set of three links to define a trace x of four files (a,b,c,d), namely {`'⟨ a,b,x ⟩'`,`'⟨ b,c,x ⟩'`,`'⟨ c,d,x ⟩'`}.

`Chain` represents all chains that can be formed by connecting links of a certain trace. A chain has been modelled as a pair of adl-files, where the left file is the first file of the chain and the right file is the last. For example, `'⟨⟨ a,b,x ⟩,⟨ c,d,x ⟩⟩'` $\in$ `Chain` represents the chain from a to d.

`SubseqFiles` represents all pairs of two different, subsequent adl-files in a trace. `SubseqFiles` has been modelled as being equivalent to `Chain`. For example, `'⟨⟨ a,b,x ⟩,⟨ c,d,x ⟩⟩'` $\in$ `SubseqFiles` represents that a preceeds d.

A researcher needs to define the links of a trace such that they form one chain of links. Each chain of links is reflexive, antisymmetric, transitive and lineair. The following integrity rules have been defined to enforce that the links of a trace form one chain of files:

- Each link has one left file.
- Each link has one right file.
- Each two links in a trace cannot have the same left file.
- Each two links in a trace cannot have the same right file.
- The set of all chains is a reflexive transitive closure on the links of each trace, which connects a left file to a right file if these files are the same.
- The set of all chains is lineair. A chain can be formed between each two links in a trace.
- The set of all chains is antisymmetric. If a link a preceeds a link b, then b cannot preceed a unless $a = b$.

This theme introduces 5 concepts, 4 generalization rules, 5 relations with properties and 6 rules related to ampersand projects in the measurement framework. Figure 7.1 shows a conceptual diagram, which includes the elements introduced in this section.

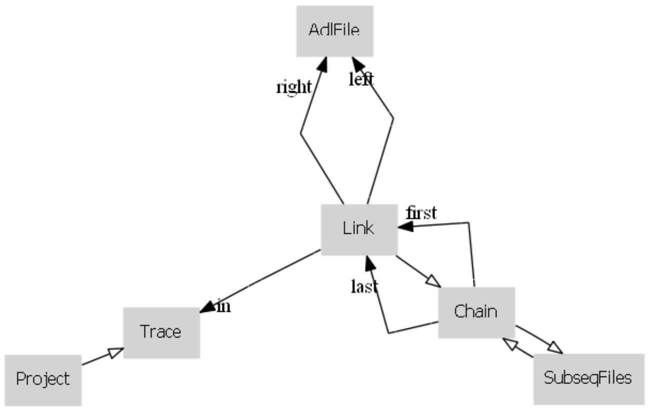


Figure 7.1: Concept diagram of Ampersand projects

7.1.1 Defined concepts

This subsection defines 5 concepts and 4 generalization rules. Each definition defines a concept followed by how the user interfaces display an atom a of that concept. This display value of a is based on a key definition [section 4.6]. The key is defined on the concept and may be parametrized with a relation expression on a. Relation expressions are shaded. A key may be defined such that a is printed as an HTML-element or as regular text. If a prints as regular text, then it will be embedded in a hyperlink in case a user interface exists to view a [section 4.5].

Definition: A `Trace` is a set of adl-files in RAP ordered by creation date. Atom $a \in$ `Trace` is displayed as is.

Definition: A `Project` is a trace of versions of an Ampersand-model from a requirements engineer.. Atom $a \in$ `Project` is displayed as a `Trace`.

Rule: `Project` is a kind of `Trace`, which is formalized by the following generalization rule:

$$\texttt{Project} \leq \texttt{Trace} \tag{7.1}$$

Definition: A `Link` is a successive relation between two adl-files in a trace. Atom $a \in$ `Link` is displayed - using relations 7.6, 7.7, 7.5 - as:

($a;left$, $a;right$, $a;in$)

Definition: A `Chain` is a chain of links in a trace. Atom $a \in$ `Chain` is displayed - using relations 7.6, 7.8, 7.7, 7.9, 7.5 - as:

($a;first;left$, $a;last;right$, $a;first;in \cap last;in$)

Definition: A `SubseqFiles` is a subsequent relation between two different files in a trace. Atom $a \in$ `SubseqFiles` is displayed as a `Chain`.

Rule: `SubseqFiles` is a kind of `Chain`, which is formalized by the following generalization rule:

$$\texttt{SubseqFiles} \leq \texttt{Chain} \tag{7.2}$$

Rule: `Chain` is a kind of `SubseqFiles`, which is formalized by the following generalization rule:

$$\texttt{Chain} \leq \texttt{SubseqFiles} \tag{7.3}$$

Rule: `Link` is a kind of `Chain`, which is formalized by the following generalization rule:

$$\texttt{Link} \leq \texttt{Chain} \tag{7.4}$$

7.1.2 Declared relations

This subsection introduces 5 declared relations with properties and a meaning.

The following univalent, total relation has been declared

$$\textit{in} \quad : \quad \texttt{Link} \rightarrow \texttt{Trace} \tag{7.5}$$

, which means that a link exists in one trace.
For example, `'(RAPv1,RAPv2,Project RAP)'` *in* `'Project RAP'` means:
`'(RAPv1,RAPv2,Project RAP)'` is a link in `'Project RAP'`.

The following univalent, total relation has been declared

$$\textit{left} \quad : \quad \texttt{Link} \rightarrow \texttt{AdlFile} \tag{7.6}$$

, which means that a link has a left file.
For example, `'(RAPv1,RAPv2,Project RAP)'` *left* `'RAPv1'` means:
The left file of `'(RAPv1,RAPv2,Project RAP)'` is `'RAPv1'`.

The following univalent, total relation has been declared

$$\textit{right} \quad : \quad \texttt{Link} \rightarrow \texttt{AdlFile} \tag{7.7}$$

, which means that a link has a right file.
For example, `'(RAPv1,RAPv2,Project RAP)'` *right* `'RAPv2'` means:
The right file of `'(RAPv1,RAPv2,Project RAP)'` is `'RAPv2'`.

The following univalent, total relation has been declared

$$\textit{first} \quad : \quad \texttt{Chain} \rightarrow \texttt{Link} \tag{7.8}$$

, which means that a chain has a first link.
For example, `'((a,b,x),(c,d,x))'` *first* `'(a,b,x)'` means:
The first link of `'((a,b,x),(c,d,x))'` is `'(a,b,x)'`.

The following univalent, total relation has been declared

$$\textit{last} \quad : \quad \texttt{Chain} \rightarrow \texttt{Link} \tag{7.9}$$

, which means that a chain has a last link.
For example, `'((a,b,x),(c,d,x))'` *last* `'(c,d,x)'` means:
The last link of `'((a,b,x),(c,d,x))'` is `'(c,d,x)'`.

7.1.3 Defined rules

This subsection defines 6 formal rules.

injective left - The following requirement has been defined:
Two links in a trace cannot have the same left file.
This is formalized - using the declared relations 7.6, 7.5 - as

$$in;in^{\smile} \cap left;left^{\smile} \subseteq \mathbb{I} \tag{7.10}$$

injective right - The following requirement has been defined:
Two links in a trace cannot have the same right file.
This is formalized - using the declared relations 7.7, 7.5 - as

$$in;in^{\smile} \cap right;right^{\smile} \subseteq \mathbb{I} \tag{7.11}$$

chain id - The following requirement has been defined:
A chain is identified by the first and last link of the chain.
This is formalized - using the declared relations 7.9, 7.8 - as

$$first;first^{\smile} \cap last;last^{\smile} \subseteq \mathbb{I} \tag{7.12}$$

all chains - The following requirement has been defined:
The set of all chains is a reflexive transitive closure on the links of each trace, which connects a left file to a right file if these files are the same.
This is formalized - using the declared relations 7.5, 7.6, 7.7, 7.9, 7.8 - as

$$first^{\smile};\mathbb{I}_{\texttt{Chain}};last \equiv (right;left^{\smile})^{*} \cap in;in^{\smile} \tag{7.13}$$

antisymmetric trace - The following requirement has been defined:
The set of all chains is antisymmetric.
This is formalized - using the declared relations 7.9, 7.8 - as

$$first^{\smile};\mathbb{I}_{\texttt{Chain}};last \cap (first^{\smile};\mathbb{I}_{\texttt{Chain}};last)^{\smile} \subseteq \mathbb{I} \tag{7.14}$$

lineair trace - The following requirement has been defined:
The set of all chains is lineair.
This is formalized - using the declared relations 7.9, 7.8, 7.5 - as

$$in;in^{\smile} \subseteq first^{\smile};\mathbb{I}_{\texttt{Chain}};last \cup (first^{\smile};\mathbb{I}_{\texttt{Chain}};last)^{\smile} \tag{7.15}$$

7.2 How to add a metric to RAP

For the research described in chapter 6, measurements by metrics were done in an offline batch on a dump of files from RAPv1 using a combination of the Ampersand compiler version of RAPv1 and spreadsheets. Metrics may be added to the RAP model for real-time information from measurements like those of section 5.17. Such real-time information can be used by (student) requirements engineers, researchers, teachers, or any other user role in RAP.

If you want to add metrics to the RAP model then you need to declare a univalent relation for each metric and implement a function to measure and update the population of the relation for a metric. The univalent relation must have a source concept that represents the required input for the metric and a target for the measurement result. As a source, one may use:

- an `AdlFile` or `Context`, an `AdlFile` might contain script errors, a `Context` cannot.
- a `Successive` or `Subsequent`, `Successive` is a special kind of `Subsequent`.
- a `Trace`.

For example, the three relations declared in section 5.17 each count the number of some kind of model element in a context. These relations are maintained by a function of the Ampersand compiler. When the user opens an Ampersand-script in RAP, the Ampersand compiler interprets the script, counts the numbers of elements and updates the population of the relations for the counters. For example, the user opens a context `'X'` $\in$ `Context`. The compiler counts five rules, three concepts and six relation declarations. The compiler updates the population of RAP such that `'X'` *countrules* `'5'`, `'X'` *countcpts* `'3'` and `'X'` *countdecls* `'6'`.

7.3 Conclusion

Metrics in RAP can be modelled like any other concept in RAP. The measurement framework is an example of how rules of a common concept, like a chain, can be defined as an extension for domain-specific Ampersand-models, like the RAP model.

The purpose of the measurement framework is to configure metrics in RAP to study student behaviour, like in chapter 6. By being an Ampersand-model, the framework guarantees the semantic integrity between the different kinds of metrics on traces of Ampersand-scripts. As a result we could rely on the formal semantics of our measurements and thereby focus on studying student behaviour.

Like the previous chapter, sections of this chapter have been generated from an Ampersand-model. To generate this chapter shows again how design automation can help to document the design.

Chapter 8

CONCLUSIONS

The introduction of this dissertation related this research to a central question: What are requirements for RAP to teach Ampersand? The requirements of RAP we seek yield what RAP is (1), how RAP is developed and maintained (2) and how RAP is used (3). Four research activities have been done to find answers, namely:

- The development of RAP with RAP, which relates to (2).
- Finding ways of support in RAP to teach the Ampersand language, which relates to (1).
- Finding ways to automate tasks in RAP to support the learning process of a student doing design exercises in RAP, which relates to (1).
- Finding ways to analyse usage data in RAP to get insight in the learning behaviour of students, which relates to (3).

8.1 Answer to research question

8.1.1 Requirements to develop RAP with RAP

We have succeeded to produce RAP as a rule-based prototype produced with Ampersand. To produce a rule-based prototype with Ampersand, one needs to define an Ampersand-model and apply the prototype generation function of Ampersand to that model. The evolution of the prototype generation function - which took place parallel to our research - has contributed to this success.

The only difference - apart from differences in the Ampersand-models - between RAP and any typical Ampersand-generated prototypes is the inclusion of a function processor in RAP, namely the Ampersand compiler. For any function in the compiler, we can implement a data item in RAP that applies data items in RAP to that function, namely as an atom of the concept `CCommand` [section 5.2]. The atoms of `CCommand` can be made available to the user by means of interfaces like any other data in RAP.

The compiler required the implementation of a special function to produce data in RAP from actions on the file repository of Ampersand-scripts. For example, when a user opens an Ampersand-script in the repository, then RAP applies the special function to that script, which produces a new population for the RAP model. Next, RAP is regenerated with the new population by applying the prototype generation function to the RAP model with the new population.

Each user has been given his own instance of RAP as a solution to regenerate RAP for that user without disturbing the other users. This solution on multiple physical instances of RAP requires a solution to obtain one virtual RAP again. Such a virtual, single RAP becomes needed when data from different physical instances need to be combined in RAP.

8.1.2 Requirements to teach the Ampersand language

The learning objective for students is how to define requirements for an information system and process those requirements into a design for that system. For that objective they must use a method for rule-based prototyping, namely Ampersand. Currently, students must learn the ASCII syntax of the Ampersand language for that and produce Ampersand-scripts. Creating such scripts looks and feels like programming and mathematics, which happens to distract students from the actual subject, namely rule-based prototyping.

A requirement for RAP is prevent such distractions.

For that purpose, we have defined a type system with correct and detailed feedback on type errors. RAP can give a student all the details he needs to understand an error, because the details do not have to be printed to the screen at once. That is, the student can browse through the information using the interfaces of RAP.

RAP can put a level of abstraction on scripts, such that students do not have to deal with scripts any more. The current interfaces of RAP present an Ampersand-model to the user instead of a script. However, editing the Ampersand-model is still limited to the population of the model. In the future, a student should be able to create and edit Ampersand-models entirely through RAP interfaces. Those Ampersand-models get stored in the repository as scripts out of sight of the user. In that future, a student cannot make script errors any more.

A point of discussion is whether RAP should put a level of abstraction on relation algebraic expressions or not e.g. a visual rule editor. Do we need to learn relation algebra to understand how to define requirements and process them into prototypes?

8.1.3 Requirements to automate tasks for design exercises

We have investigated how we could automate tasks for design exercises in RAP. And, we have automated certain tasks in RAP. However, we have neither identified those tasks explicitly, nor have we followed a structured approach to determine which tasks could or should be automated. We do consider our cyclic five-step approach to be a structured approach for that purpose.

We have demonstrated the following mechanisms to implement automated tasks:

- define process rules in the RAP model to guide a student through a design exercise by means of hints e.g. [section 5.16].
- define integrity rules in the RAP model to prevent a student from introducing errors in an Ampersand-model e.g. [section 5.10].
- implement automated tasks as functions of the Ampersand compiler e.g. generate conceptual diagram or generate a specification document.
- define metrics in the RAP model to have measurements in RAP and implement functions in the Ampersand compiler to obtain those measurements e.g. [section 5.17].
- define process rules or interfaces on measurements in RAP to give diagnostic feedback on the products or behaviour of a student e.g. the interface named *Diagnosis* [chapter A.2].

8.1.4 Requirements to analyse learning behaviour

The RAP model has been extended with a measurement framework to define metrics to analyse learning behaviour [chapter 7]. A metric defined in RAP can be implemented as a function in the Ampersand compiler.

The measurement framework defines the concept of traces in RAP. A researcher can compose traces from Ampersand-scripts in the repository of RAP. RAP may compose certain traces automatically e.g. a function in the compiler that creates a trace in RAP for all scripts of a single user without script errors. A trace is defined as a sequence of Ampersand-script ordered by creation time. The repository keeps track of how to parse and interpret each script by documenting the version of the Ampersand compiler, which yields a certain parse and interpretation function.

The researcher may define three types of metrics on the traces: metrics on each single Ampersand-script in the trace, metrics on each two subsequent scripts in the trace, or metrics on the entire trace. For that, the measurement framework defines the concept of subsequent scripts within a trace. The concept of an Ampersand-script has already been defined in the RAP model.

8.2 Contributions

The four research activities mentioned earlier have resulted in the following contributions, some of which are strongly related to each other:

A learning platform has been realised that enables research on the didactics of rule-based design. The feasibility to study student behaviour with the learning platform has been established by the study in chapter 6, which was published at CSERC13 [23]. RAP has the technical ability to analyse the data in RAP and process the results into feedback for students. That is, RAP has a measurement framework to add metrics to RAP [chapter 7] and RAP supports constraint-based feedback on data in RAP, which includes metrics. Note that the data in RAP are the results of all design actions of students performed in RAP. In chapter 6, metrics have been defined in a framework of the learning platform. The metrics have been interpreted as observations about student behaviour in an informal manner. Metrics of three different kinds were implemented, demonstrating the usability and flexibility of the platform for the purpose of studying the learning behaviour with respect to rule-based design. Based upon [23], it is reasonable to expect that many useful observations about improving the learning of rule-based design can be investigated on our platform.

Students are using RAP as a learning platform. One of the first contributions of this research was the release of RAP to students of the Open University course on rule-based design. If anything, the analysis of 6947 traces has shown that students can learn the Ampersand approach to rule-based design. To use emergent research results immediately in the course has certainly not harmed our understanding of the topic.

Lessons about teaching rule-based design have been drawn in [23]. These lessons vary from usability in a general sense to very specific ones, such as the prediction of behaviour that may lead to early intervention by tutors [chapter 6]. Each lesson is a subtle contribution of this research and a concrete direction to refine the learning platform or the didactics of rule-based design.

RAP as a learning platform is responsive enough to support design exercises of the course with small development capacities. Evidence is the evolution of the learning platform from 2010 until 2012. The first Ampersand-generated version of the learning platform was developed and released in 2010. That first version harvested student data for one year. In the meantime, the run-time system generation function of the Ampersand compiler has been improved. Analysis of the harvested data and the revised compiler resulted in new requirements for a second version of the learning platform. The second version of the learning platform was developed and released in 2012. A higher level of responsiveness can be reached by a continuous execution of each of the five steps of our development approach - harvesting traces, defining metrics, formulating observations, learning lessons and refining RAP. For the future, we expect this to be done in a repetitive cycle, consisting of harvest, define, formulate, learn and refine.

Ampersand's rule repository has been generated with Ampersand. All database software, required for the RAP repository, has been generated with Ampersand rather than written by hand. This includes web-based user interfaces, transactional functionality, and software to safeguard the semantic integrity. To generate a rule repository from an Ampersand-model [chapter 4] proves that the model contains sufficient knowledge about the rule repository; and proves that substantial knowledge about rule-based design is embedded in the code of the Ampersand compiler. Chapter 5 and 7, which have been generated by the compiler as well, show how design automation can help to document the design.

Ampersand fulfils its purpose to automate designing of systems. This dissertation presents the learning platform itself as evidence, because it has been designed and generated almost entirely with Ampersand. But the work performed by our students in the Rule-Based Design course is also evidence that Ampersand is up to its task. During our research, but outside the scope of this dissertation, Ampersand has also been used in industry, e.g. in projects for the judiciary, education inspection, immigration and university administration. That experience too shows how Ampersand helps to automate the design of information systems, and corroborates our knowledge of its limitations.

The consistency of the Ampersand language has been documented and exploited. In order to make things work, it is imperative that the Ampersand language is well-defined. Due to initial flaws in the language, substantial parts of the language had to be redefined. The formal language definition, especially the type system, has undergone thorough revision, which has led to [22, 38]. As a result, at the start of 2012 the Ampersand compiler was robust enough to meet its first real industrial challenge in the Dutch judiciary system.

8.3 Reflection

The contributions of this dissertation are focused on Ampersand. There are still only a few researchers on Ampersand, which makes the direct scientific target audience small. Indirectly, a much larger audience might be attracted, because Ampersand has a common base with hot topics like business rules, model-driven software development, constraint-based tutors, learning analytics, semantic web, relational models and enterprise modelling. But, further research on RAP is still needed before we can communicate our contributions to that larger audience in a natural manner.

Our research approach has been heavily based on practical experience, namely experience from the development and usage of RAP. This approach has pros and cons. The advantage is in my opinion that all kinds of new insights arise spontaneously, which can be validated instantly and simultaneously by means of concrete examples. The disadvantage is in my opinion the complexity of how all these new insights relate to the current research communities. So, with the approach we could learn those lessons we needed the most, but we did not have capacity left to reflect on all those lessons in a scientific manner and share them with the right research community. Our approach has resulted in an answer to our research question, namely validated requirements for the research and development infrastructure of Ampersand. And, as a bonus, we have learned valuable lessons for our practice of teaching Ampersand.

8.4 Further research and development

Our idea is that our research and development continues by repetition of the five steps: harvest traces, define metrics, formulate observations, learn lessons and refine RAP.

In a next cycle, we can define the metrics in the RAP model and have measurements on traces in RAP itself. Visualization functions, e.g. to produce metrical percepts and pivot tables, can be moved to the Ampersand compiler.

Having metrics defined in RAP for research comes with an opportunity for students. Only by adding a relation expression to an interface for students [section 4.5], students get access to the measurements in RAP that we are using for research. Sharing statistic data with students to enhance learning is a form of learning analytics [10].

The rule-based nature of RAP comes with an opportunity to extend RAP with constraint-based feedback [24]. Such feedback can be defined in the RAP model by means of process rules [section 4.3] and integrity rules [section 4.4.2]. However, further research will be needed to find an interpretation of rule violations such that it becomes useful feedback for students doing design exercises.

A development environment for other purposes than rule-based prototyping might be defined in an Ampersand-model. We have experimented shortly with a rough definition of a development environment for UML class diagrams, which has partly demonstrated that other development environments can be implemented like RAP. Our research group has no plans to implement other kinds of development environments.

The main focus of further development on RAP is on editing the entire Ampersand-model in RAP. This targets the concept in Ampersand of RAP being one big repository of rules. From a users perspective, this targets the completion of a development environment to work on Ampersand-models, professionally as well as in education.

Appendix A

The RAP model

A.1 RAP

```
CONTEXT RAP
INCLUDE "student_AST_interfaces.adl"
INCLUDE "RAP_purposes.adl"

PATTERN "The repository of files"
CONCEPT FileRef "a reference to a file in the repository"
CONCEPT FileName "a name of a file"
CONCEPT FilePath "a location in the repository"
filename :: FileRef->FileName PRAGMA "The file name in " " is " ""
MEANING IN ENGLISH ", which means that a file reference includes a file name."
 = [("comp/gmi/RAP.v2.adl","RAP.v2.adl")]
filepath :: FileRef*FilePath[UNI] PRAGMA "The file path in " " is " ""
MEANING IN ENGLISH ", which means that a file reference may include a relative or absolute file path."
 = [("comp/gmi/RAP.v2.adl","comp/gmi/")]
KEY FileRef: FileRef(PRIMHTML "<a href='../../index.php?file=", filepath, filename ,PRIMHTML "\\\\&userrole=", uploaded~; userprofile, PRIMHTML "'>", filename, PRIMHTML "</a>")
RULE "unique file location": filename;filename~ /\ filepath;filepath~ |- I
MEANING IN ENGLISH "Each file has a unique location on the file system of the repository."

CONCEPT CalendarTime "a time representation, which includes day, weekday, month, year, hour, minute, second and timezone"
filetime :: FileRef*CalendarTime[UNI]
MEANING IN ENGLISH ", which means that the time on which a file has been committed to the repository may be known."
```

```

CONCEPT User "a name with which a requirements engineer has logged in
     to RAP"
CONCEPT UserProfile "a kind of user"
uploaded::User*FileRef PRAGMA "" " has committed " " to the repository
     "
MEANING IN ENGLISH ", which means that a user may have committed files
      to the repository."
 = [("gmi","comp/gmi/RAP.v2.adl")]
userprofile::User*UserProfile [UNI] PRAGMA "" " has access to the
     compiler commands for a " ""
MEANING IN ENGLISH ", which means that a user may have a user profile,
      which gives him access to the set of compiler commands for that
     user profile."
 = [("gmi","Student")]
RULE "user profiles": 'Student' \/ 'StudentDesigner' \/ 'Designer' |-
     I[UserProfile]
MEANING IN ENGLISH "There are three user profiles: Student,
     StudentDesigner and Designer."

CONCEPT AdlFile "a file of the adl-format, which is the format for
     Ampersand-scripts"
SPEC AdlFile ISA FileRef
--KEY AdlFile: inherit from File
sourcefile::Context->AdlFile PRAGMA "Context " " originates from " ""
MEANING IN ENGLISH ", which means that a context originates from an
     adl-file in the repository."
 = [("RAP","comp/gmi/RAP.v2.adl")]
includes  ::Context*FileRef PRAGMA "Context " " partially originates
     from " ""
MEANING IN ENGLISH ", which means that the adl-file from which a
     context originates may include other files."
 = [("RAP","comp/gmi/RAP.v77.pop")]

CONCEPT CCommand "a command for the Ampersand compiler"
applyto::CCommand->AdlFile PRAGMA "Command " " applies to " ""
MEANING IN ENGLISH ", which means that a compiler command applies to
     an adl-file."
 = [("1(comp/gmi/RAP.v2.adl)","comp/gmi/RAP.v2.adl")]
functionname :: CCommand->String PRAGMA "Command " " uses the function
      " ""
MEANING IN ENGLISH ", which means that a compiler command uses a
     compiler function that has a user-friendly name."
 = [("1(comp/gmi/RAP.adl)","load into Atlas")]
operation :: CCommand->Int PRAGMA "Command " " uses the function " ""
MEANING IN ENGLISH ", which means that a compiler command uses a
     compiler function that has a technical identifier."
 = [("1(comp/gmi/RAP.v2.adl)","1")]
KEY CCommand: CCommand(PRIMHTML "<a href='../../index.php?operation=",
     operation
         ,PRIMHTML "\\\\&file=", applyto;filepath , applyto;filename
```

```
        ,PRIMHTML ”\\\\&userrole=”, applyto;uploaded[User*AdlFile]˜;userprofile
        ,PRIMHTML ”’\\\\>”, functionname, PRIMHTML ”</a>”)

CONCEPT NewAdlFile ”a predefined adl-file, which opens as a new script in the script editor on the upload page”
SPEC NewAdlFile ISA AdlFile
KEY NewAdlFile: NewAdlFile(PRIMHTML ”<a href=’../../index.php’>”, filename[NewAdlFile*FileName],PRIMHTML”</a>”)
 newfile::User->NewAdlFile PRAGMA ”” ” has an option to open ” ” in the script editor”
MEANING IN ENGLISH ”, which means that a user has an option to open a new script.”
 = [(”gmi”,”empty.adl”)]

CONCEPT SavePopFile ”a file to which a user can save the population of a context”
SPEC SavePopFile ISA FileRef
KEY SavePopFile: SavePopFile(PRIMHTML ”<a href=’../../index.php?operation=4\\\\&file=”, filepath[SavePopFile*FilePath] , filename[SavePopFile*FileName]
                                          ,PRIMHTML ”’\\\\>”, filename[SavePopFile*FileName] ,PRIMHTML ”</a>”)
 savepopulation::Context->SavePopFile PRAGMA ”If the user exports the population of ” ” then that population will be saved to a new pop-file ” ” in the repository”
MEANING IN ENGLISH ”, which means that there is an option to export the population of a context to a pop-file.”
 = [(”RAP”,”comp/gmi/RAP.v78.pop”)]

CONCEPT SaveAdlFile ”a file to which a user can save the changes made to a context”
SPEC SaveAdlFile ISA AdlFile
KEY SaveAdlFile: SaveAdlFile(PRIMHTML ”<a href=’../../index.php?operation=2\\\\&file=”, filepath[SaveAdlFile*FilePath] , filename[SaveAdlFile*FileName]
                                          ,PRIMHTML ”\\\\&userrole=”, savecontext˜;sourcefile;uploaded[User*AdlFile]˜;userprofile
                                          ,PRIMHTML ”’\\\\>”, filename[SaveAdlFile*FileName], PRIMHTML ”</a>”)
 savecontext::Context->SaveAdlFile PRAGMA ”If the user commits the changes made to ” ” then that context including the changes will be saved to a new adl-file ” ” in the repository”
MEANING IN ENGLISH ”, which means that there is an option to commit changes on a context to an adl-file.”
 = [(”RAP”,”comp/gmi/RAP.v3.adl”)]
ENDPATTERN

PATTERN ”Committed files”
CONCEPT AdlVersion ”a version of the Ampersand compiler”
 firstloadedwith :: AdlFile * AdlVersion [UNI] PRAGMA ”At the time that
```

```
      " " was committed, RAP used compiler version " ""
MEANING IN ENGLISH ", which means that the version of the Ampersand
     compiler at the time that an adl-file was committed to the
     repository may be known."
 = [("comp/gmi/RAP.v2.adl","v2.2.711-2191")]
inios::Concept*AtomID PRAGMA "Initially, concept " " contained an atom
      " ""
MEANING IN ENGLISH ", which means that the initial population of a
     concept may contain atoms."
 = [("AdlFile","comp/gmi/RAP.v2.adl")]
inipopu::Declaration*PairID PRAGMA "Initially, " " contained a pair
     with id " ""
MEANING IN ENGLISH ", which means that the initial population of a
     relation may contain pairs of atoms."
 = [("userprofile::User*UserProfile","523422885")]
inileft::PairID*Atom PRAGMA "Initially, the pair with id " " had a
     left atom " ""
MEANING IN ENGLISH ", which means that a pair of atoms in the initial
     population of relations had an initial left atom."
 = [("523422885","gmi")]
iniright::PairID*Atom PRAGMA "Initially, the pair with id " " had a
     right atom " ""
MEANING IN ENGLISH ", which means that a pair of atoms in the initial
     population of relations had an initial right atom."
 = [("523422885","Student")]
CONCEPT ParseError "an error in the syntax of a script"
CONCEPT TypeError "an error concerning the type of a relation
     declaration or relation expression in the script"
CONCEPT ErrorMessage "a description of an error" TYPE "Blob"
CONCEPT FilePos "a position in a file"
CONCEPT Hint "a description of possible actions which may resolve an
     error"
CONCEPT ModElemType "a type of model element e.g. rule definition,
     user interface definition, key definition"
CONCEPT ModElemName "the name of a model element"
KEY ParseError: ParseError(TXT "Click here for error details")
parseerror   :: FileRef * ParseError[UNI]
MEANING IN ENGLISH ", which means that the Ampersand-script in a file
     may have a parse error."
pe_action    :: ParseError -> Hint
MEANING IN ENGLISH ", which means that a parse error describes
     possible actions which may resolve the error."
pe_position  :: ParseError -> FilePos
MEANING IN ENGLISH ", which means that a parse error has occurred on a
      file position."
pe_expecting :: ParseError -> ErrorMessage
MEANING IN ENGLISH ", which means that a parse error has a message,
     which describes what was expected by the parser."
KEY TypeError: TypeError(TXT "Click here for details of error at ",
     te_position)
typeerror   :: FileRef * TypeError
```

```
MEANING IN ENGLISH ", which means that the Ampersand−script in a file
    may have type errors."
te_message  :: TypeError * ErrorMessage [UNI]
MEANING IN ENGLISH ", which means that a type error may have a message
     that gives a complete description of the error."
te_parent   :: TypeError * TypeError [UNI]
MEANING IN ENGLISH ", which means that a type error may be nested in
    another type error that describes the same mistake on a higher
    level."
te_position :: TypeError * FilePos [UNI]
MEANING IN ENGLISH ", which means that a type error may have occurred
    on a file position."
te_origtype :: TypeError * ModElemType [UNI]
MEANING IN ENGLISH ", which means that a type error may have occurred
    in a certain type of model element."
te_origname :: TypeError * ModElemName [UNI]
MEANING IN ENGLISH ", which means that a type error may have occurred
    in a model element that has a certain name."
ENDPATTERN

PROCESS "Handling script errors"
ROLE Student MAINTAINS noparseerror,notypeerror

RULE "noparseerror": −parseerror
MEANING IN ENGLISH "The requirements engineer needs to solve all parse
     errors."
MESSAGE IN ENGLISH LATEX "\\textbf{A syntax error was encountered in
    your script.} No CONTEXT screen could be generated."
VIOLATION (TGT I, TXT " Open ", SRC I,TXT " to fix error.")

RULE "notypeerror": −typeerror
MEANING IN ENGLISH "The requirements engineer needs to solve all type
    errors."
MESSAGE IN ENGLISH LATEX "\\textbf{Type error(s) were encountered in
    your script.} A CONTEXT screen was generated with concepts and
    relation declarations only, which may be useful to understand and
    fix the errors."
VIOLATION (TGT I, TXT ". Open ", SRC I,TXT " to fix error.")
ENDPROCESS


PATTERN Contexts
CONCEPT Context "a model element, which defines the name and scope of
    an Ampersand model"
               "DatabaseDesign.Ampersand.Core.AbstractSyntaxTree.
    A_Context(ctxnm,ctxpats,ctxcs):svn568"
KEY Context: Context(ctxnm)

ctxnm  ::Context−>Conid PRAGMA "The context in " " is called " ""
MEANING IN ENGLISH ", which means that a context has a user−defined
    name."
```

```
 = [("comp/gmi/RAP.v2.adl","RAP")]
ctxpats::Context*Pattern PRAGMA "Context " " contains a pattern " ""
MEANING IN ENGLISH ", which means that a context may contain patterns.
    "
 = [("RAP","Committed files")]
ctxcs   ::Context*Concept PRAGMA "Context " " contains a concept " ""
MEANING IN ENGLISH ", which means that a context may contain concepts.
    "
 = [("RAP","UserProfile")]
ENDPATTERN

PATTERN Patterns
CONCEPT Pattern "a model element, in which a requirements engineer
    groups certain model elements in a logical manner"
                "DatabaseDesign.Ampersand.Core.AbstractSyntaxTree.
    Pattern(ptnm,ptdcs,ptgns,ptrls):svn568"
KEY Pattern: Pattern(ptnm)

ptnm   :: Pattern->Conid
MEANING IN ENGLISH ", which means that a pattern has a name."
ptrls  :: Pattern*Rule
MEANING IN ENGLISH ", which means that a pattern may contain rule
    definitions."
ptgns  :: Pattern*Gen
MEANING IN ENGLISH ", which means that a pattern may contain
    generalization rule definitions."
ptdcs  :: Pattern*Declaration
MEANING IN ENGLISH ", which means that a pattern may contain relation
    declarations."
ptxps  :: Pattern*Blob
MEANING IN ENGLISH ", which means that a pattern may have purpose
    descriptions in a natural language."
ENDPATTERN

PATTERN "Generalization rules"
CONCEPT Gen ", or generalization rule, is a model element to define
    the is-a-relationship between a more specific and a more generic
    concept"
           "DatabaseDesign.Ampersand.Core.AbstractSyntaxTree.A_Gen(
    genspc,gengen):svn568"
KEY Gen: Gen( TXT "SPEC ", genspc;cptnm , TXT " ISA " , gengen;cptnm )

RULE "eq gen": gengen;gengen~ /\ genspc;genspc~ |- I
MEANING IN ENGLISH "There is only one generalization rule between a
    certain specific concept and a certain generic concept."

gengen :: Gen->Concept PRAGMA "In " ", " " is the generic concept"
MEANING IN ENGLISH ", which means that a generalization rule has a
    generic concept."
 = [("SPEC Horse ISA Animal","Animal")]
genspc :: Gen->Concept PRAGMA "In " ", " " is the specific concept"
```

```
MEANING IN ENGLISH ", which means that a generalization rule has a
    specific concept."
 = [("SPEC Horse ISA Animal","Horse")]

ENDPATTERN

PATTERN Concepts
CONCEPT Concept "a model element for an abstract business term of
    which the population represents a certain set of business objects"
              "DatabaseDesign.Ampersand.Core.AbstractSyntaxTree.
    A_Concept(cptnm,cptdf,cptos):svn568"
KEY Concept: Concept(cptnm)

cptnm :: Concept->Conid
MEANING IN ENGLISH ", which means that a concept has a name."
cptos :: Concept*AtomID PRAGMA "The population of " " contains an atom
     element identified by " ""
MEANING IN ENGLISH ", which means that the population of a concept may
     contain atom elements."
 = [("Horse","443859690")]
cptdf :: Concept*Blob
MEANING IN ENGLISH ", which means that a concept may have definitions
    in a natural language."
cptpurpose:: Concept*Blob
MEANING IN ENGLISH ", which means that a concept may have purpose
    descriptions in a natural language."

CONCEPT Order "a structure for a group of is-a-related concepts"
KEY Order: Order(ordername)

ordername :: Order -> String
MEANING IN ENGLISH ", which means that an order has a name."
order :: Concept -> Order PRAGMA "Concept " " belongs to the " ""
MEANING IN ENGLISH ", which means that a concept belongs to an order."
 = [("Horse","order of animals")]
RULE "order": order~;genspc~;gengen;order |- I
MEANING IN ENGLISH "Is-a-related concepts belong to the same order."
RULE "referential integrity": src~;decsgn~;decpopu;left \/ trg~;decsgn
    ~;decpopu;right |- order;order~;cptos
MEANING IN ENGLISH "An atom in the domain or codomain of a relation is
     an instance of a concept from the same order as the source
    respectively the target of that relation."
MESSAGE IN ENGLISH LATEX "If an atom is in some tuple of a relation,
    then that atom must exist in the concept that is the source
    respectively target of that relation. Deletion of an atom from a
    concept is not permitted if that atom is still present in some
    tuple of some relation. Nor is addition of a tuple permitted if
    the source or target atom is not present in the related concept.
    It is a violation of \\textbf{Referential integrity} rule for a
    relation."
VIOLATION (TXT "The tuple ", SRC I, TXT " refers to a source or target
      atom that does not exist." )


CONCEPT AtomID " a model element for an atom"
CONCEPT Atom "the value of an atom and an identifier of an atom within
      an order" TYPE "Blob"
               "DatabaseDesign.Ampersand.Input.ADL1.UU_Scanner.pAtom:
     svn568"
KEY AtomID: AtomID( atomvalue , TXT " :: ", cptos~;order;ordername)

atomvalue::AtomID->Atom PRAGMA "The value of atom " " is " ""
MEANING IN ENGLISH ", which means that an atom element has a value."
 = [("443859690","Jolly Jumper")]
RULE "entity integrity concept": atomvalue;atomvalue~ /\ cptos~;order;
     order~;cptos |- I
MEANING IN ENGLISH "An atom of a concept is unique within the order of
      that concept."
MESSAGE IN ENGLISH LATEX "Every atom of a concept is unique, or, no
     two atoms in the population of a concept have the same name.
     Addition of a duplicate atom is not permitted. It is a violation
     of the \\textbf{Entity integrity} rule for this concept. Please
     refer to book \\emph{Rule Based Design}, page 43 and 52, \\emph{
     entity integrity}. "
VIOLATION (TXT "An atom with name ", SRC I, TXT " already exists." )

ENDPATTERN

PATTERN "Relation type signatures"
CONCEPT Sign ", or a relation type signature, is a model element for
     relation types."
               "DatabaseDesign.Ampersand.Core.AbstractSyntaxTree.Sign:
     svn568"
KEY Sign: Sign( src;cptnm , TXT " * " , trg;cptnm )

src::Sign->Concept
MEANING IN ENGLISH ", which means that a sign has a source concept."
trg::Sign->Concept
MEANING IN ENGLISH ", which means that a sign has a target concept."

CONCEPT PairID "a model element for a pair of atoms"
KEY PairID: PairID( left;atomvalue , TXT " * " , right;atomvalue )
left::PairID->AtomID
MEANING IN ENGLISH ", which means that a pair of atoms has a left atom
     ."
right::PairID->AtomID
MEANING IN ENGLISH ", which means that a pair of atoms has a right
     atom."
ENDPATTERN

PATTERN "Relation declarations"
CONCEPT Declaration ", or a relation declaration, is a model element
```

```
    to declare a relation"
                     "DatabaseDesign.Ampersand.Core.AbstractSyntaxTree.
    Declaration(decnm,decsgn,decprps):svn568"
KEY Declaration: Declaration( decnm , TXT " :: ", decsgn;src;cptnm ,
    TXT " * ", decsgn;trg;cptnm )
RULE "eq declaration": decnm;decnm~ /\ decsgn;src;(decsgn;src)~ /\
    decsgn;trg;(decsgn;trg)~ |- I
MEANING IN ENGLISH "A declared relation can be identified by a name, a
     source concept, and a target concept."

decnm   ::Declaration->Varid PRAGMA "The name of " " is " ""
MEANING IN ENGLISH ", which means that a relation is declared with a
    name."
 = [("r::A*B","r")]
decsgn  ::Declaration->Sign PRAGMA "The sign of " " is " ""
MEANING IN ENGLISH ", which means that a relation is declared with a
    sign."
 = [("r::A*B","A*B")]
decprps::Declaration*PropertyRule [INJ] PRAGMA "" " has a property " "
    "
MEANING IN ENGLISH ", which means that a relation may be declared with
     property rules."
 = [("r::A*B","TOT r::A*B")]

CONCEPT Property "a predefined symbol to define property rules"
                 "DatabaseDesign.Ampersand.ADL1.Prop.Prop(..):svn568"
CONCEPT PropertyRule "a rule, that has been defined as a property"
                     "DatabaseDesign.Ampersand.ADL1.Rule.rulefromProp:
    svn568"
SPEC PropertyRule ISA Rule
RULE "property enum": I[Property] |- '->' \/ 'UNI' \/ 'TOT' \/ 'INJ'
    \/ 'SUR' \/ 'RFX' \/ 'IRF' \/ 'SYM' \/ 'ASY' \/ 'TRN' \/ 'PROP'
MEANING IN ENGLISH "There are eleven predefined symbols to define
    property rules: -> means univalent and total; UNI means univalent;
     TOT means total; INJ means injective; SUR means surjective; RFX
    means reflexive; IRF means irreflexive; SYM means symmetric; ASY
    means antisymmetric; TRN means transitive; and PROP means
    symmetric and antisymmetric."
declaredthrough::PropertyRule*Property [TOT] PRAGMA "Rule " " is
    defined by means of property symbol " ""
MEANING IN ENGLISH ", which means that a property rule is defined by
    means of predefined symbols."
 = [("TOT r::A*B","TOT")]

decprL  ::Declaration*String[UNI]
MEANING IN ENGLISH ", which means that the meaning of a relation may
    be clarified with an example sentence that has a prefix text."
decprM  ::Declaration*String[UNI]
MEANING IN ENGLISH ", which means that the meaning of a relation may
    be clarified with an example sentence that has an infix text."
decprR  ::Declaration*String[UNI]
```

```
MEANING IN ENGLISH ", which means that the meaning of a relation may
    be clarified with an example sentence that has a suffix text."
decmean ::Declaration * Blob
MEANING IN ENGLISH ", which means that a relation may have
    descriptions of its meaning in a natural language."
decpurpose::Declaration * Blob
MEANING IN ENGLISH ", which means that a relation may have purpose
    descriptions in a natural language."
decpopu ::Declaration*PairID
MEANING IN ENGLISH ", which means that the population of a relation
    may contain pairs of atoms."

RULE "entity integrity of relation": left;left~ /\ right;right~ /\
    decpopu~;decpopu |- I
MEANING IN ENGLISH "There cannot be two pairs in a declared relation
    with the same left and same right."
MESSAGE IN ENGLISH LATEX "Every tuple in a relation is unique, or, no
    two tuples in the population of a relation may have the same
    source and target atoms. Addition of a duplicate tuple is not
    permitted. It is a violation of the \\textsf{Entity integrity}
    rule for this relation."
VIOLATION (TXT "A tuple with the same source and target atoms ", SRC I
    , TXT " already exists." )

RULE "typed domain": decpopu;left;cptos~;order  |- decsgn;src;order
MEANING IN ENGLISH "The left atoms of pairs in a declared relation
    belong to the same order as the source of that relation."
MESSAGE IN ENGLISH LATEX "You try to add a tuple with a source atom,
    that is not in the population of the source of the relation. This
    is a violation of the type of the tuple. TIP: enter text in the
    left input field to get a shorter pick list. Note on ISA-relations
    : You can make an atom more specific by moving it to the
    population of a more specific concept."
VIOLATION (TXT "Source atom ", TGT I, TXT " is not in the population
    of ", SRC decsgn;src)

RULE "typed codomain": decpopu;right;cptos~;order |- decsgn;trg;order
MEANING IN ENGLISH "The right atoms of pairs in a declared relation
    belong to the same order as the target of that relation."
MESSAGE IN ENGLISH LATEX "You try to add a tuple with a target atom,
    that is not in the population of the target of the relation. This
    is a violation of the type of the tuple. TIP: enter text in the
    right input field to get a shorter pick list. Note on ISA-
    relations: You can make an atom more specific by moving it to the
    population of a more specific concept."
VIOLATION (TXT "Target atom ", TGT I, TXT " is not in the population
    of ", SRC decsgn;trg)
ENDPATTERN

PATTERN Expressions
CONCEPT ExpressionID "a model element for a relation"
CONCEPT Expression "the relation expression, which is a relation
    expression written in Ampersand ASCII syntax"
                  "DatabaseDesign.Ampersand.Input.ADL1.Parser.pExpr:
    svn568"
KEY ExpressionID : ExpressionID(exprvalue)

exprvalue :: ExpressionID->Expression PRAGMA "Relation " " is
    expressed as " ""
MEANING IN ENGLISH ", which means that an expression element has a
    value, which is a relation expression."
 = [("RULE RFX r::A*A: r |- I","r |- I")]
rels  :: ExpressionID*Relation PRAGMA "" " uses " " in its expression"
MEANING IN ENGLISH ", which means that an expression uses relations
    that have been declared."
 = [("RULE RFX r::A*A: r |- I","r[A*A]")]

CONCEPT Relation "a relation term in an expression that has a
    declaration"
KEY Relation: Relation( relnm , TXT "[" , relsgn;src;cptnm , TXT "*" ,
     relsgn;trg;cptnm , TXT "]")

relnm :: Relation -> Varid PRAGMA "The name of " " is " ""
MEANING IN ENGLISH ", which means that a relation has a name."
 = [("r[A*A]","r")]
relsgn:: Relation -> Sign PRAGMA "The sign of " " is " ""
MEANING IN ENGLISH ", which means that a relation has a sign."
 = [("r[A*A]","A*A")]
reldcl:: Relation -> Declaration PRAGMA "" " has been declared by " ""
MEANING IN ENGLISH ", which means that a relation has a declaration."
 = [("r[A*A]","r::A*A")]
RULE "rel name is decl name": relnm = reldcl;decnm
MEANING IN ENGLISH "The name of a relation is the same as the name in
    its declaration."
ENDPATTERN

PATTERN Rules
CONCEPT Rule "a model element, which defines a rule by means of a
    relation expression"
             "DatabaseDesign.Ampersand.Core.AbstractSyntaxTree.Rule(
    rrnm,rrexp):svn568"
KEY Rule: Rule(rrnm)

rrnm  :: Rule -> ADLid  PRAGMA "The name of " " is " ""
MEANING IN ENGLISH ", which means that a rule has a name."
 = [("RULE RFX r::A*A","RFX r::A*A")]
rrexp :: Rule -> ExpressionID PRAGMA "The relation expression of " "
    is " ""
MEANING IN ENGLISH ", which means that a rule has a relation
    expression to express that rule."
 = [("RULE RFX r::A*A","RULE RFX r::A*A: r |- I")]
rrmean:: Rule * Blob PRAGMA "" " means " ""
```

```
MEANING IN ENGLISH ", which means that a rule may have descriptions of
      its meaning in a natural language."
 = [("RULE RFX r::A*A","r::A*A is reflexive")]
rrpurpose:: Rule * Blob
MEANING IN ENGLISH ", which means that a rule may have purpose
     descriptions in a natural language."
ENDPATTERN

PATTERN Symbols
CONCEPT String "text restricted to a maximum of 256 characters"
               "DatabaseDesign.Ampersand.Input.ADL1.UU_Scanner.pString
     :svn568"
CONCEPT Blob "text, which may exceed 256 characters" TYPE "Blob"
             "DatabaseDesign.Ampersand.Input.ADL1.UU_Scanner.pString:
     svn568"
CONCEPT Conid "a string starting with an uppercase"
              "DatabaseDesign.Ampersand.Input.ADL1.UU_Scanner.pConid:
     svn568"
CONCEPT Varid "a string starting with a lowercase"
              "DatabaseDesign.Ampersand.Input.ADL1.UU_Scanner.pVarid:
     svn568"
CONCEPT ADLid "string, which may be of the kind Varid, Conid, or
     String"
              "DatabaseDesign.Ampersand.Input.ADL1.Parser.pADLid:
     svn568"
ENDPATTERN

PATTERN "Calculated details"
CONCEPT Violation "a pair of atoms of a special kind of relation, that
      is, the complement of a rule expression. Such a pair violates the
      rule"
SPEC Violation ISA PairID
rrviols::Rule*Violation
MEANING IN ENGLISH ", which means that a rule may have violations."

CONCEPT Image "a digital representation of a diagram or figure"
ptpic::Pattern*Image[UNI]
MEANING IN ENGLISH ", which means that a pattern may have a conceptual
      diagram to visualize that pattern."
cptpic::Concept*Image[UNI]
MEANING IN ENGLISH ", which means that a concept may have a conceptual
      diagram to visualize that concept."
rrpic::Rule*Image[UNI]
MEANING IN ENGLISH ", which means that a rule may have a conceptual
     diagram to visualize that rule."
CONCEPT URL ", or unified resource location, is a web address"
imageurl :: Image*URL
MEANING IN ENGLISH ", which means that an image may be found on web
     addresses."
KEY Image: Image(PRIMHTML "<img src='", imageurl , PRIMHTML "'>")

```

```
CONCEPT PragmaSentence "an example sentence for a relation to clarify
    its meaning" TYPE "Blob"
decexample :: Declaration * PragmaSentence
MEANING IN ENGLISH ", which means that a relation may have example
    sentences to clarify its meaning."
ENDPATTERN

PROCESS "Testing rules"
ROLE Student MAINTAINS otherviolations, multviolations1,
    multviolations2,multviolations3, homoviolations

RULE "multviolations1": -((I[PropertyRule] /\ declaredthrough;('TOT'
    \/ 'SUR');declaredthrough~);rrviols)
MEANING IN ENGLISH "The user gets feedback on the violations of total
    and surjective property rules."
MESSAGE IN ENGLISH LATEX "\\textbf{A TOTal or SURjective multiplicity
    rule is violated for some relation(s).} Add tuple(s) in the
    relation(s) to correct the violation(s)."
VIOLATION (TXT "RULE ", SRC I, TXT " is violated by the atom ", TGT
    left, TXT ".")

RULE "multviolations2": -((I[PropertyRule] /\ declaredthrough;('UNI'
    \/ 'INJ');declaredthrough~);rrviols)
MEANING IN ENGLISH "The user gets feedback on the violations of
    univalent and injective property rules."
MESSAGE IN ENGLISH LATEX "\\textbf{A UNIvalent or INJective
    multiplicity rule is violated for some relation(s).} Delete tuple(
    s) in the relation(s) to correct the violation(s)."
VIOLATION (TXT "RULE ",SRC I, TXT " is violated by the tuple ", TGT I,
     TXT " in combination with some other tuple(s).")

RULE "multviolations3": -((I[PropertyRule] /\ declaredthrough;'->';
    declaredthrough~);rrviols)
MEANING IN ENGLISH "The user gets feedback on the violations of
    functional property rules."
MESSAGE IN ENGLISH LATEX "\\textbf{A UNIvalent or TOTal multiplicity
    rule is violated for some relation(s).} Delete tuple(s) in the
    relation(s) to correct the violation(s)."
VIOLATION (TXT "RULE ",SRC I, TXT " is violated by the tuple ", TGT I,
     TXT " in combination with some other tuple(s).")

RULE "homoviolations": -((I[PropertyRule] /\ declaredthrough;('RFX' \/
     'IRF' \/ 'SYM' \/ 'ASY' \/ 'TRN' \/ 'PROP');declaredthrough~);
    rrviols)
MEANING IN ENGLISH "The user gets feedback on the violations of
    homogeneous property rules."
MESSAGE IN ENGLISH LATEX "\\textbf{A rule for homogeneous relation(s)
    is violated.} Add or delete tuple(s) in the relation(s) to correct
     the violation(s)."
VIOLATION (TXT "RULE ",SRC I, TXT " is violated by the tuple ", TGT I,
     TXT " in combination with some other tuple(s).")
```

```

RULE "otherviolations": -((-I[PropertyRule] /\ I[Rule]);rrviols)
MEANING IN ENGLISH "The user gets feedback on the violations of non-
    property rules. "
MESSAGE IN ENGLISH LATEX "\\textbf{A business rule that involves
    several relations is violated.} Add or delete tuple(s) in one or
    more of the relation(s) to correct the violation(s)."
VIOLATION (TXT "RULE ",SRC I, TXT " is violated by the tuple ", TGT I,
    TXT " in combination with some other tuple(s).")
ENDPROCESS

PROCESS "Editing a population"
ROLE Student MAINTAINS popchanged
RULE "popchanged": inipopu = decpopu
MEANING IN ENGLISH "The user gets feedback on uncommitted changes to a
    population."
MESSAGE IN ENGLISH LATEX "You have made changes to the population. You
    can: \\\\ 1) \\textbf{enter more} change(s), or; \\\\ 2) \\textbf
    {undo} your changes \\textbf{by (re)loading} any CONTEXT into
    Atlas, or;\\\\ 3) \\textcolor{blue}{Click here} to \\textbf{commit
    } the change(s) \\textbf{and update} violations on your rules."
VIOLATION (TXT "added or deleted pair ",TGT I, TXT " of ", TGT (
    inipopu\/decpopu)~)
ENDPROCESS

PATTERN Metrics
CONCEPT Int "a number"
countrules :: Context*Int[UNI]
MEANING IN ENGLISH ", which means that the number of rule definitions
    in a context may have been calculated."
countdecls :: Context*Int[UNI]
MEANING IN ENGLISH ", which means that the number of relation
    declarations in a context may have been calculated."
countcpts  :: Context*Int[UNI]
MEANING IN ENGLISH ", which means that the number of concepts in a
    context may have been calculated."
ENDPATTERN

ENDCONTEXT
```

A.2 User interfaces for students

```
CONTEXT RAP
INTERFACE "Atlas (Play)" FOR Student:I[ONE]
BOX ["CONTEXT":V[ONE*Context]
  BOX ["name":ctxnm
      ,"number of RULEs":countrules
              ,"number of relations":countdecls
              ,"number of concepts":countcpts
              ]
```

```
     ,"PATTERNs":V[ONE*Context];ctxpats
     ,"concepts":V[ONE*Context];ctxcs
     ,"ISA-relations":V[ONE*Context];ctxpats;ptgns
     ,"relations":V[ONE*Context];ctxpats;ptdcs
  BOX ["relation":I
             ,"with properties":decprps;declaredthrough
             ]
     ,"RULEs":V[ONE*Context];ctxpats;ptrls;(I[Rule] /\ -I[PropertyRule
     ])
     ]
INTERFACE "Validate" FOR Student:I[ONE]
BOX ["Click to commit and validate":V[ONE*Context];savecontext
     ]
INTERFACE "CONTEXT files (Design / reload)" FOR Student:I[ONE]
BOX ["loaded into Atlas":V[ONE*Context]
  BOX ["CONTEXT":I
             ,"source file (click to edit)":sourcefile \/ includes
             ,"operations (click to perform)":sourcefile;applyto~
             ]
     ,"overview of files":V[ONE*User]
  BOX ["open new source file":newfile
             ,"source files":uploaded[User*AdlFile];I[AdlFile]
    BOX ["file name (click to edit)":I
                         ,"created at":filetime
                         ,"operations (click to perform)":applyto~
                         ]
             ]
     ]
INTERFACE "Diagnosis" FOR Student:I[ONE]
BOX ["concepts without definition":V[ONE*Concept];(-(cptdf;cptdf~) /\
     I)
     ,"relations without MEANING":V[ONE*Declaration];(-(decmean;decmean
     ~) /\ I)
     ,"RULEs without MEANING":V[ONE*Rule];(-(rrmean;rrmean~) /\ I)
     ,"populated relations":V[ONE*PairID];decpopu~
     ,"unpopulated relations":V[ONE*Declaration];(-(decpopu;decpopu~)
     /\ I)
     ]
INTERFACE "Syntax error" FOR Student:I[ParseError]
BOX ["expecting":pe_expecting
     ,"position":pe_position
     ,"try"     :pe_action
     ]
INTERFACE "Type error" FOR Student:I[TypeError]
BOX ["error":I
  BOX ["in":te_origtype
             ,"at":te_position
             ,"stating":te_origname
             ]
     ,"error message":te_message
     ,"declared relations":V[TypeError*Declaration]
     ]
INTERFACE "Extra functions"(filename,userprofile) FOR Student:I[ONE]
BOX ["Export POPULATIONs to...":V[ONE*Context];savepopulation
  BOX ["file (INCLUDE only)":I[SavePopFile]
            ,"type a file name":filename
            ]
     ,"User settings":V[ONE*User]
  BOX ["use role to load files":userprofile
            ]
     ,"files with only POPULATIONs":V[ONE*User];uploaded;(I[FileRef] /\
      -I[AdlFile])
  BOX ["file name":I
            ,"created at":filetime
            ]
     ]
INTERFACE "CONTEXT" FOR Student:I[Context]
BOX ["name":ctxnm
     ,"PATTERNs":ctxpats
     ,"concepts":ctxcs
     ,"ISA-relations":ctxpats;ptgns
     ,"relations":ctxpats;ptdcs
  BOX ["relation":I
            ,"with properties":decprps;declaredthrough
            ]
     ,"RULEs":ctxpats;ptrls;(I[Rule] /\ -I[PropertyRule])
     ]
INTERFACE "PATTERN" FOR Student:I[Pattern]
BOX ["PURPOSEs":ptxps
     ,"name":ptnm
     ,"RULEs":ptrls;(I[Rule] /\ -I[PropertyRule])
     ,"relations":ptdcs
  BOX ["relation":I
            ,"with properties":decprps;declaredthrough
            ]
     ,"ISA-relations":ptgns
     ,"diagram":ptpic
     ]
INTERFACE "ISA-relation" FOR Student:I[Gen]
BOX ["SPEC":genspc
     ,"ISA":gengen
     ,"in PATTERN":ptgns~
     ]
INTERFACE "Concept"(cptos,atomvalue) FOR Student:I[Concept]
BOX ["PURPOSEs":cptpurpose
     ,"CONCEPT definition":cptdf
     ,"name":cptnm
     ,"POPULATION":cptos
  BOX ["atom":atomvalue
            ]
     ,"POPULATION (through ISA)":(genspc~;gengen \/ genspc~;gengen;
     genspc~;gengen \/ genspc~;gengen;genspc~;gengen;genspc~;gengen)~
```

```
    --TODO closure
  BOX ["more specific concept":I
            ,"POPULATION":cptos;atomvalue
            ]
    ,"more generic concepts":genspc~;gengen \/ genspc~;gengen;genspc~;
    gengen \/ genspc~;gengen;genspc~;gengen;genspc~;gengen   --TODO
    closure
    ,"used in relations":(decsgn;(src \/ trg))~
    ,"used in RULEs":(relsgn;(src \/ trg))~;(rrexp;rels)~;(I[Rule] /\
    -I[PropertyRule])
    ,"diagram":cptpic
    ]
INTERFACE "Atom"(cptos) FOR Student:I[Atom];atomvalue~
BOX ["in POPULATION of":cptos~
    ]
INTERFACE Relation(decpopu,left,right) FOR Student:I[Declaration]
BOX ["PURPOSEs":decpurpose
    ,"MEANING":decmean
    ,"example of basic sentence":decexample
    ,"name":decnm
    ,"type":decsgn
  BOX ["source":src
            ,"target":trg
            ]
    ,"properties":decprps;declaredthrough
    ,"from PATTERN":ptdcs~
    ,"POPULATION":decpopu
  BOX ["source":left
            ,"target":right
            ]
    ,"used in RULEs":(rrexp;rels;reldcl)~;(I[Rule] /\ -I[PropertyRule
    ])
    ]
INTERFACE "RULE" FOR Student:I[Rule]
BOX ["PURPOSEs":rrpurpose
    ,"MEANING":rrmean
    ,"name":rrnm
    ,"assertion":rrexp
    ,"uses":rrexp;rels
  BOX ["relation":reldcl
            ,"with properties":reldcl;decprps;declaredthrough
            ,"source":relsgn;src
            ,"target":relsgn;trg
            ]
    ,"in PATTERN":ptrls~
    ,"diagram":rrpic
    ]
ENDCONTEXT
```

Bibliography

[1] R. Barends. Activeren van de administratieve organisatie. master thesis, Open Universiteit Nederland, 2003.

[2] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[3] Tom R. Burns and Anna Gomolińska. The theory of socially embedded games: The mathematics of social relationships, rule complexes, and action modalities. *Quality and Quantity*, 34:379–406, 2000.

[4] Business Rule Solutions, LLC. RuleSpeak. Retrieved Februari 27, 2013, from `http://www.rulespeak.com`, 2013.

[5] Chris J. Date. *What not how: The Business Rules Approach to application development*. Addison-Wesley Longman Publishing Co., Inc., Boston, 2000.

[6] Umeshwar Dayal, Alejandro P. Buchmann, and Dennis R. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented database system. In Klaus R. Dittrich, editor, *OODBS*, volume 334 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 1988.

[7] Jan L. G. Dietz. *Enterprise ontology - theory and methodology*. Springer, Heidelberg, 2006.

[8] Remco M. Dijkman, Luis Ferreira Pires, and Stef M.M. Joosten. Calculating with concepts: a technique for the development of business process support. In Andy Evans, editor, *Practical UML based rigorous development methods countering or integrating the extremists / Workshop of the PUML-Group held together with the "UML" 2001, October 1st, 2001 in Toronto, Canada. Gesellschaft für Informatik*, volume P-7 of *GI-Edition / Gesellschaft für Informatik $ Proceedings ; Vol. 7*, pages 87–98, Bonn, Germany, 2001. Gesellschaft für Informatik.

[9] Atze Dijkstra and S. Doaitse Swierstra. Typing Haskell with an attribute grammar. In Varmo Vene and Tarmo Uustalu, editors, *AFP 2004*, volume 3622 of *LNCS*, pages 1–72, Berlin, 2005. Springer-Verlag.

[10] Erik Duval. Attention please!: learning analytics for visualization and recommendation. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*, LAK '11, pages 9–17, New York, NY, USA, 2011. ACM.

[11] Stijn Goedertier and Jan Vanthienen. Declarative process modeling with business vocabulary and business rules. In *Proceedings of the 2007 OTM confederated international conference on On the move to meaningful internet systems - Volume Part I*, OTM'07, pages 603–612, Berlin, Heidelberg, 2007. Springer-Verlag.

[12] Terry Halpin. Object Role Modeling. `http://www.orm.net`, October 2012.

[13] Claudia Hattensperger and Peter Kempf. Towards a formal framework for heterogeneous relation algebra. *Inf. Sci.*, 119(3-4):193–203, 1999.

[14] Daniel Jackson. A comparison of object modelling notations: Alloy, UML and Z. Technical report, Retrieved Februari 27, 2013, from `http://people.csail.mit.edu/dnj/publications/alloy-comparison.pdf`, 1999.

[15] R. Joosten, J-W. Knobbe, P. Lenoir, H. Schaafsma, and G. Kleinhuis. Specifications for the RGE Security Architecture. Technical Report Deliverable D5.2 Project TSIT 1021, TNO Telecom and Philips Research, The Netherlands, August 2003.

[16] Stef Joosten. *Praktijkboek voor procesarchitecten*. Koninklijke Van Gorcum, 2002.

[17] Stef Joosten. Sustainable integration. Retrieved Februari 27, 2014, from `http://portal.ou.nl/documents/1466009/0/isbpplan.pdf`, June 2011.

[18] Stef Joosten and Gerard Michels. Generating a tool for teaching rule-based design. In *Proceedings of the Third International Symposium on Business Modeling and Software Design*, pages 230–236. SCITEPRESS, 2013.

[19] Kenneth C. Laudon and Jane P. Laudon. *Management information systems: New approaches to organization & technology*. Prentice Hall, New Jersey, 5 edition, 1998.

[20] Roger D. Maddux. *Relation Algebras*, volume 150 of *Studies in logic*. Elsevier, Iowa, 2006.

[21] Jeroen J.G. Merriënboer and Paul A. Kirschner. *Ten steps to complex learning: a systematic approach to four-component instructional design.* Lawrence Erlbaum Associates, Mahwah, New Jersey, 2007.

[22] Gerard Michels, Sebastiaan Joosten, Jaap van der Woude, and Stef Joosten. Ampersand: Applying relation algebra in practice. In *Proceedings of the 12th conference on Relational and Algebraic Methods in Computer Science*, Lecture Notes in Computer Science 6663, pages 280–293, Berlin, 2011. Springer-Verlag.

[23] Gerard Michels and Stef Joosten. Progressive development and teaching with RAP. In *Proceedings of the Computer Science Education Research Conference 2013*, pages 33–43, Heerlen, 2013. Open Universiteit.

[24] Antonija Mitrovic, Kenneth Koedinger, and Brent Martin. A comparative analysis of cognitive tutoring and constraint-based modeling. In Peter Brusilovsky, Albert Corbett, and Fiorella de Rosis, editors, *User Modeling 2003*, volume 2702 of *Lecture Notes in Computer Science*, pages 147–147. Springer, Heidelberg, 2003.

[25] Object Management Group, Inc. OMG model driven architecture. Technical report, `http://www.omg.org/cgi-bin/doc?omg/03-06-01`, June 2003.

[26] Object Management Group, Inc. Semantics of Business Vocabulary and Business Rules (SBVR), v1.0. Technical report, Retrieved Februari 27, 2013, from `http://www.omg.org/spec/SBVR/1.0/PDF`, 2008.

[27] Object Management Group, Inc. Object Constraint Language (OCL). Technical report, Retrieved Februari 27, 2014, from `http://www.omg.org/spec/OCL/`, 2012.

[28] Seymour Papert and Idit Harel. *Situating constructionism*, volume 36, pages 1–11. Ablex Publishing Corporation, New York, 1991.

[29] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, March 1999.

[30] Simon Peyton Jones, editor. *Haskell 98 language and libraries – The Revised Report.* Cambridge University Press, Cambridge, 2003.

[31] Ronald G. Ross. The Business Rules Manifesto. Retrieved Februari 27, 2013, from `http://www.businessrulesgroup.org/brmanifesto.htm`, November 2003.

[32] Ronald G. Ross. *Principles of the Business Rule Approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[33] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual.* Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[34] Ernst Schröder. *Algebra und logik der relative.* Vorlesungen über die Algebra der Logik (Exakte Logik) / Ernst Schröder. Teubner, Leipzig, 1895.

[35] Walter A.A. Shewhart and W. Edwards Deming. *Statistical methods from the viewpoint of quality control.* Dover Books on Mathematics Series. Dover Publications, Incorporated, 1939.

[36] George Siemens, Dragan Gasevic, Caroline Haythornthwaite, Shane Dawson, Simon Buckingham Shum, Rebecca Ferguson, Erik Duval, Katrien Verbert, and Ryan S. J. d. Baker. Open learning analytics: an integrated & modularized platform. Technical report, `http://solaresearch.org/OpenLearningAnalytics.pdf`, 2011.

[37] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. Designing and implementing combinator languages. In S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques, editors, *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206. Springer Berlin Heidelberg, 1999.

[38] Jaap van der Woude and Stef Joosten. Relational heterogeneity relaxed by subtyping. In *Proceedings of the 12th conference on Relational and Algebraic Methods in Computer Science*, Lecture Notes in Computer Science 6663, pages 347–361, Berlin, 2011. Springer-Verlag.

[39] Lex Wedemeijer, Stef Joosten, and Gerard Michels. *Rule Based Design.* Open Universiteit Nederland, Heerlen, 1st edition, 2010.

[40] Jennifer Widom. The starburst active database rule system. *IEEE Trans. on Knowl. and Data Eng.*, 8(4):583–595, August 1996.

Summary

Ampersand is a rule-based approach to design information systems and business processes (IS&BP). The idea of Ampersand is to make the design of IS&BP more concrete by letting requirements engineers produce working software. The requirements engineer builds software by formalizing functional requirements as a collection of rules, which is fed to a compiler to generate the software for those requirements.

This rule-based approach puts the emphasize of designing IS&BP on:

- understanding and obtaining agreement upon system requirements.
- translating system requirements to software.
- dealing with changing system requirements.
- formulating and communicating well-defined system requirements.

This dissertation described a development environment for rule-based prototyping (RAP) and its various roles in research on the didactics of Ampersand. The first role of RAP is that of a development environment for Ampersand, which is being used by students for Ampersand design exercises. The second role is the interesting case of using Ampersand to develop RAP. For the third role, we have implemented metrics in RAP to measure student behaviour in the development environment. Based on such measurements we have studied the didactics of Ampersand.

Chapter 2 has introduced the idea that information system software can be derived from functional requirements alone. This idea has been implemented by the method Ampersand. Ampersand has been described from the perspective of a requirements engineer. A requirements engineer uses Ampersand to elicit business rules, formalize those rules and maintain them in a rule repository, RAP. The Ampersand language to define systems of rules has been built on relation algebra and the Business Rules Approach. A requirements engineer processes rules in RAP to develop rule-compliant run-time systems. In such a run-time system, rules can be guards of the integrity of business data and processes (integrity rules), or drivers of business processes and workflows (process rules). An engineer may use the Ampersand compiler to generate rule-compliant run-time systems, validate rules, visualize a system's design, produce functional specifications, and more.

Ampersand plays two roles in this dissertation in which RAP is the central subject. One, RAP is a development tool for Ampersand, which includes the Ampersand compiler and a rule repository as described. Two, RAP has been developed with Ampersand and is a generated rule-compliant run-time system. Thus, RAP could be used to develop RAP. Moreover, the RAP model is being maintained in RAP and committing changes to the rules in the RAP model causes an instant update of RAP itself.

In order to make things work, it is imperative that the Ampersand language is well-defined. In chapter 3 we have defined the Ampersand language. The main requirement for the Ampersand language is that it is easy to learn by requirements engineers, but sufficiently expressive to define real-life information systems.

To meet this requirement, the Ampersand language adopts relation algebra and respects the statements in the Business Rules Manifesto of the Business Rules Approach. A requirements engineer needs to learn a handful of relational operators and syntactic elements to define an Ampersand-model. The learning challenge that remains is how to define a meaningful Ampersand-model, one that formalizes the business rules of a business context. For that challenge, a requirements engineer needs to learn how to translate an Ampersand-model of concepts, relations and rules to a business language of terms, facts and business rules.

The learning challenge turns out to be significant, which we have concluded from unpublished student evaluations and a study on student behaviour in RAP.

The formal character of the Ampersand language allows us to address the learning challenge by means of feedback to the requirements engineer. The kinds of feedback presented in chapter 3 are conceptual diagrams and feedback on type errors. A conceptual diagram gives the requirements engineer a visual overview over the relations he has declared. A conceptual diagram uses the types of the relations to relate them to each other. An Ampersand-model must be free of type errors to be meaningful. We can guarantee that a requirements engineer gets feedback on each type error.

This dissertation shows that the Ampersand language is sufficiently expressive to define real-life information systems, namely by the case of RAP.

Chapter 4 describes how Ampersand generates software, and how it is used to support business processes. We have demonstrated how a model in Ampersand presents itself as a working application to users. Although a fair number of working applications have been made, we cannot claim that Ampersand produces industry strength applications. For that, further engineering on the Ampersand compiler is required. This explains why the Ampersand toolset is currently being used in the design phase, for the purpose of prototyping and generating documentation.

Chapter 6 aims at validation of our five step development cycle with RAP. The idea behind our approach is that usage of tools for design exercises gives rise to measuring facts about student behaviour. Such facts can be used to study student behaviour and enhance those tools for better education.

To execute that idea, RAP has been developed and used for a course on rule-based design. RAP integrates user interfaces for design exercises with analytics upon a trace of design products.

Four conclusions can be drawn based on running one development cycle by our development approach:

- It is feasible to study student behaviour based on measurements in RAP.
- We have drawn multiple lessons about teaching Ampersand.
- The authors are convinced that, by using our development approach, continuous improvement of RAP can be realized in practice based on actual student behaviour.
- Door Ampersand te gebruiken in onze ontwikkelaanpak zijn we met beperkte middelen, op een gecontroleerde wijze tot een - naar onze mening - rijk product gekomen, de tweede versie van RAP.
- By using Ampersand in our development approach, we needed little resource to produce a - in our opinion - wealthy product (RAPv2) in a controlled manner.

Chapter 5 is the documentation of RAP and has been generated from the RAP model. For that, we have used the same Ampersand compiler that generates RAP from the RAP model.

The documentation of RAP yields examples of what can be defined in Ampersand using the mechanisms presented in chapter 4. We claim that RAP implements Ampersand, for that reason we conclude that Ampersand can be defined in and documented by means of an Ampersand-model.

To generate chapter 5 shows how design automation can help to document the design. There is a guarantee that the documentation of RAP remains up-to-date with RAP, because both are generated from the same source. The documentation of RAP is complete, consistent and faultless e.g. correct cross-references and up-to-date diagrams, which can be validated by comparing chapter 5 to its source, the RAP model in appendix A.

Chapter 7 shows that metrics in RAP can be modelled like any other concept in RAP. The measurement framework is an example of how rules of a common concept, like a chain, can be defined as an extension for domain-specific Ampersand-models, like the RAP model.

The purpose of the measurement framework is to configure metrics in RAP to study student behaviour, such that we can use our cyclic development approach more efficiently. As a result we could rely on the formal semantics of our measurements and thereby focus on studying student behaviour.

The dissertation contributes the following:

- A learning platform has been realised that enables research on the didactics of rule-based design.
- Students are enjoying the benefits of RAP as a learning platform.
- Ampersand's rule repository has been generated with Ampersand.
- Ampersand fulfils its purpose to automate designing of systems.
- The consistency of the Ampersand language has been documented and exploited.

Samenvatting

Ampersand is een regelgebaseerde aanpak voor het ontwerpen van informatiesystemen en bedrijfsprocessen (IS&BP). Het idee achter Ampersand is dat software voor IS&BP geproduceerd kan worden door requirements engineers, die de functionele systeemeisen vastleggen als een verzameling van formele regels. Deze (bedrijfs)regelgebaseerde aanpak legt de activiteit van het ontwerpen van IS&BP bij:

- het begrijpen en afspreken van systeemeisen.
- het vertalen van systeemeisen naar software.
- het kunnen omgaan met veranderende systeemeisen.
- het eenduidig formuleren en communiceren van systeemeisen.

Dit proefschrift beschrijft een ontwikkelomgeving voor regelgebaseerde prototypes (RAP) en haar multifunctionele rol in het onderzoek naar hoe Ampersand onderwezen kan worden De eerste rol van RAP is die van een ontwikkelomgeving voor Ampersand, waarmee studenten praktijkoefeningen kunnen maken. De tweede rol is de interessante casus over het gebruik van Ampersand, die voortkomt uit het feit dat RAP ontwikkeld is met Ampersand. Voor de derde rol hebben we metrieken in RAP geïmplementeerd om studentengedrag te kunnen meten. Op basis van die metingen hebben we de resultaten van de huidige didactiek van Ampersand kunnen bestuderen.

Hoofdstuk 2 heeft het idee geïntroduceerd dat informatie systeem software afgeleid kan worden van functionele systeemeisen. Dit idee is geïmplementeerd in de methode Ampersand. Ampersand is beschreven vanuit het perspectief van de requirements engineer. Een requirements engineer gebruikt Ampersand om bedrijfsregels naar boven te halen, die regels te formaliseren en te onderhouden in een verzamelbak voor regels. De Ampersandtaal, waarmee een systeem van regels gedefinïeerd kan worden, is gebouwd op relatie-algebra en de Business Rules Approach. Een requirements engineer verwerkt regels in RAP om operationele systemen te ontwikkelen die zich conformeren aan de regels. In een dergelijk systeem kunnen regels bewakers zijn van de integriteit van bedrijfsdata en -processen (integriteitsregels), of aandrijvers van bedrijfsprocessen en werkstromen (procesregels). Een engineer kan de Ampersand compiler gebruiken om operationele systemen te genereren, regels te valideren, een systeemontwerp te visualiseren, functionele specificaties te produceren, en meer.

Ampersand verhoudt zich op twee manieren tot RAP. Eén, RAP is een ontwikkelgereedschap voor Ampersand, waarvan de Ampersand compiler en de regelverzamelbak een onderdeel zijn. Twee, RAP is ontwikkeld met Ampersand en gegenereerd als een operationeel systeem, dat zich conformeert aan de regels. Dus, RAP kon gebruikt worden om RAP te ontwikkelen. Daarenboven, het RAP model wordt onderhouden in RAP en veranderingen aan de regels in het RAP model hebben direct effect in RAP zelf.

Om producten geautomatiseerd af te kunnen leiden, is het onontkoombaar dat de Ampersandtaal eenduidig gedefiniëerd is. In hoofdstuk 3 definiëren we de Ampersandtaal. De hoofdeis voor deze taal is dat zij eenvoudig te leren is door requirements engineers, maar voldoende expressief om een echt informatie systeem op te stellen.

Om aan deze eis te voldoen, adopteert de Ampersandtaal relatie-algebra en respecteert zij de uitspraken in het Business Rules Manifesto uit de Business Rules Approach. Een requirements engineer moet een handvol relationele bewerkingstekens en syntactische elementen leren om een Ampersandmodel op te kunnen stellen. De leeruitdaging die dan overblijft is, hoe kan een betekenisvol Ampersandmodel worden opgesteld, een model dat de bedrijfsregels uit een bedrijfscontext formaliseert. Voor die uitdaging moet een requirements engineer leren hoe dat een Ampersandmodel van concepten, relaties en regels vertaald kan worden naar een bedrijfstaal van termen, feiten en bedrijfsregels.

De leeruitdaging blijkt significant te zijn, hetgeen we geconcludeerd hebben uit niet gepubliseerde studentevaluaties en een gepubliseerde studie naar studentengedrag in RAP.

Het formele karakter van de Ampersandtaal maakt het mogelijk om ondersteuning te bieden tijdens het leren, ondersteuning in de vorm van terugkoppeling aan de requirements engineer. De soorten terugkoppeling die in hoofdstuk 3 worden gepresententeerd zijn conceptuele diagrammen en terugkoppeling op typefouten in relatie-expressies. Een conceptueel diagram geeft de requirements engineer een grafisch overzicht over de relaties tussen concepten die hij heeft bedacht. Een conceptueel diagram verbindt relaties met elkaar op basis van hun type. Een Ampersandmodel moet vrij zijn van typefouten om betekenis te hebben. We kunnen garanderen dat een requirements engineer terugkoppeling krijgt op iedere typefout.

RAP is het bewijs dat de Ampersandtaal voldoende expressief is om een echt informatie systeem op te stellen.

Hoofdstuk 4 beschrijft hoe Ampersand software genereert en hoe die software bedrijfsprocessen ondersteunt. We hebben gedemonstreerd hoe een Ampersandmodel zich kan presenteren als een operationeel systeem voor gebruikers. Ondanks dat we een redelijk aantal operationele systemen hebben gemaakt, kunnen we niet stellen dat Ampersand systemen voor industrieel gebruik kan produceren. Daarvoor is verdere ontwikkeling van de Ampersand compiler noodzakelijk. Dat verklaart waarom de huidige Ampersand gereedschappen gebruikt worden in de ontwerpfase, voor het ontwikkelen van prototypes en het genereren van documentatie.

Hoofdstuk 6 heeft als doel om onze cyclische ontwikkelaanpak met RAP te valideren. Het idee achter onze aanpak is dat het gebruik van RAP voor ontwerpopgaven de mogelijkheid schept om feiten over studentengedrag te meten. Dergelijke metingen kunnen gebruikt worden om studentengedrag te bestuderen en RAP te verbeteren voor educatie.

Om feiten over studentengedrag te verzamelen is RAP ontwikkeld en door studenten gebruikt in een cursus voor regelgebaseerd ontwerpen. RAP combineert software voor ontwerpopgaven met metrieken op een historie van ontwerpproducten.

Vier conclusies zijn getrokken op basis van één cyclus volgens onze ontwikkelaanpak:

- Het is mogelijk om studentengedrag te bestuderen op basis van metingen in RAP.
- We hebben meerdere lessen getrokken over het onderwijzen van Ampersand.
- De schrijvers zijn overtuigd dat onze ontwikkelaanpak in de praktijk kan zorgen voor een continue verbetering van RAP voor educatie, die gebaseerd is op feitelijk studentengedrag.
- Door Ampersand te gebruiken in onze ontwikkelaanpak zijn we met beperkte middelen, op een gecontroleerde wijze tot een - naar onze mening - rijk product gekomen, de tweede versie van RAP.

Hoofdstuk 5 documenteert de tweede versie van RAP volledig, zodat de lezer het product kan waarderen op wat het is. Dit hoofdstuk is gegenereerd uit het RAP model, waarmee ook RAP gegenereerd is.

Deze documentatie van RAP levert voorbeelden van dat wat geproduceerd kan worden met Ampersand, gebruikmakend van de mechanismes gepresenteerd in hoofdstuk 4. We stellen dat RAP Ampersand implementeert, waardoor we concluderen dat Ampersand gedefiniëerd en gedocumenteerd kan worden door middel van een Ampersandmodel.

Door hoofdstuk 5 te genereren laten we zien hoe geautomatiseerd ontwerpen kan helpen bij het documenteren van het ontwerp. We kunnen garanderen dat de documentatie van RAP actueel blijft, omdat RAP en de documentatie beide afgeleid zijn uit dezelfde bron. De documentatie van RAP is compleet, consistent en foutloos, wat gevalideerd kan worden door hoofdstuk 5 te vergelijken met zijn bron, het RAP model in de appendix.

Hoofdstuk 7 laat zien dat metrieken in RAP hetzelfde gemodelleerd worden als alle andere concepten in RAP. Het meetraamwerk is een voorbeeld van hoe regels op een algemeen concept, zoals een keten, gedefiniëerd kunnen worden als een extensie voor domeinspecifieke Ampersandmodellen, zoals het RAP model.

Het doel van het meetraamwerk is om metrieken in RAP te configureren om studentgedrag te meten, zodat we onze cyclische ontwikkelaanpak efficiënter kunnen toepassen. Omdat het raamwerk een Ampersandmodel is, kunnen we vertrouwen op de formele betekenis van onze metingen en ons bezighouden met het bestuderen studentengedrag.

Het proefschrift levert de volgende bijdragen:

- Een leerplatform is gerealiseerd, waarmee onderzoek naar het onderwijzen van Ampersand mogelijk is.
- Studenten gebruiken RAP en profiteren daarmee van de voordelen die RAP als leerplatform brengt.
- De regelverzamelbak van Ampersand is gegenereerd met Ampersand zelf.
- Ampersand voldoet aan zijn doel om systemen geautomatiseerd te ontwerpen.
- De eenduidige definitie van de Ampersandtaal is vastgesteld en benut.

Curriculum Vitae

GERARD MICHELS

27 februari 1980	Geboren te Utrecht
1992 - 1998	VWO, Jacob Roelandslyceum te Boxtel
1998 - 2003	WO - Bestuurlijke Informatiekunde, Universiteit van Tilburg
	Afstudeeronderzoek naar de toepasbaarheid van Rational Unified Process met een offshore ontwikkelteam, Infopulse B.V. te Best en te Kiev in Oekraïne
2004 - 2006	Technisch integratiespecialist, Atos Origin B.V. te Utrecht
2006 - 2008	Enterprise Applicatie Integratie consultant, Virtual Sciences BIS te Utrecht
2008 - 2013	Promovendus, faculteit Informatica, Open Universiteit
2014 - heden	Integratie Architect, ValueBlue B.V. te Utrecht

Index