Generating a Tool for Teaching Rule-based Design

Stef Joosten and Gerard Michels
Department of Computer Science, Open University The Netherlands, Valkenburgerweg 177, Heerlen, Netherlands
{stef.joosten, gerard.michels}@ou.nl

Keywords: Business Rules, Requirements Engineering, Software Generation, Computer Science Education.

Abstract: Software generation from a formal requirements specification has enabled a small research team to develop a tool set for educational design exercises and didactical research. This approach was needed to obtain a development environment, which is responsive to changing requirements due to maturing didactics and future research questions. We use Ampersand, a rule-based design methodology, to specify and generate the tool set called the Repository for Ampersand Projects (RAP). RAP is being used in a course on Ampersand for master students of computer science and business management. Analytic tools have been interconnected to RAP to obtain analytics about student activities in RAP. So, Ampersand is both the subject of teaching and research as well as an asset used to develop and maintain RAP. In this paper we present how RAP has been generated with Ampersand and reflect upon the value of this design choice.

1 INTRODUCTION

The authors have developed a Repository for Ampersand Projects (RAP) to facilitate teaching Ampersand as well as research on that subject. Ampersand is a methodology to design information systems and business processes as a formal collection of rules. RAP is a tool set for Ampersand design exercises, which features rule-based interfaces to connect analytic tools in a way that preserves semantics of data. The analytic tools use semantic data in RAP to produce unambiguous measurement results for our research. The objective of our research is to understand how to teach Ampersand to master students of computer science and business management. The first study using data from RAP has been accepted for publication (Michels and Joosten, 2013).

This paper reflects on our choice to use Ampersand to generate RAP from functional requirements. Ampersand was a prerequisite in this choice c.q. the question is whether rule-based design has brought us closer to our research objective. The answer is positive. By using Ampersand we have obtained a development environment for RAP, that enables us to respond to changing requirements in a controlable and timely fashion. We can adopt new didactical insights in the exercise tool and facilitate our current and future studies with unambiguous analytics from RAP. Thus, this paper presents the fundament of our environment to study and improve the teaching of Ampersand i.e. the generation of RAP with Ampersand.

Section 2 provides a background on using rules as requirements. Section 3 introduces Ampersand as a rule-based approach on a formal language to design information systems. Section 4 describes the generic and specific issues of generating RAP with Ampersand e.g. rule-compliant processes, customized data access. We conclude with a reflection on our accomplishments with RAP up to date and expectations for the future.

2 RULES AS REQUIREMENTS

This paper argues that a rule-based design approach can reduce the gap between the owners of requirements (end users, patrons, auditors, etc.) and the design of information systems, by giving compliance guarantees to these owners and by giving the appropriate tools to requirements engineers. The Business Rules Community (Ross, 2003b) has argued since a long time that natural language provides a better means for discussing requirements than graphical models (e.g. UML (Rumbaugh et al., 1999)). This vision is the fundament of profound assets like the Business Rules Manifesto (Ross, 2003a), the Business Rules Approach (Ross, 2003b), the SBVR standard (Object Management Group, Inc., 2008) and RuleSpeak (Business Rule Solutions, LLC., 2013). Rule-based design goes beyond requirements by formalizing business rules and using them as requirements to partially automate information system de-

Joosten S. and Michels G.
Generating a Tool for Teaching Rule-based Design.
DOI: 10.5220/0004775902300236
In *Proceedings of the Third International Symposium on Business Modeling and Software Design (BMSD 2013)*, pages 230-236
ISBN: 978-989-8565-56-3
Copyright © 2013 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

velopment.

Rule-based design uses the word *business rule* to denote a formal representation of a business requirement. Business rules have been represented in many different ways. Prolog (Clocksin and Mellish, 1981) is an early rule-based approach that uses Horn-clauses as a means to write computer programs. Widespread are also Event-Condition-Action (ECA) rules of active databases, such as (Dayal et al., 1988; Paton and Díaz, 1999; Widom, 1996), which represent successful results of earlier research into functional dependencies between database transactions in the seventies and eighties. Expert systems and other developments founded in ontology (Gruber, 1993; Berners-Lee et al., 2001) can be regarded as ways to represent business processes by means of rules. Approaches based on event traces, such as Petri Nets (Reisig, 1985) and ARIS (Scheer, 1998), have dominated the 90's and are still popular to date (Green and Rosemann, 2000). The information systems community is aware (e.g. (Ram and Khatri, 2005)) that mathematical representations of business rules can be useful. For example, Date's criticism of SQL for being unfaithful to relational algebra (Date, 2000) advocates a more declarative approach to information system design, and puts business rules in the center of the design process.

This paper argues that business requirements are sufficient to generate a functional prototype of an information system. The word 'sufficient' suggests that requirements engineers need not communicate with the business in any other way. This suggestion is seriously flawed. Models are still useful, but their role changes. In Ampersand, models are artifacts, preferrably produced automatically, that document the design. If desired, a requirements engineer can avoid to discuss these artifacts (data models, etc.) with the business, but they are available and undeniably useful as documentation in the design process. Ampersand shifts the focus of the design process to requirements engineering, because a larger part of the process is automated.

Controlling design processes directly by means of business rules has consequences for requirements engineers, who will encounter a simplified design process. From their perspective, the design process is depicted in figure 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 artifacts, such as data models, process models etc. These design artifacts can then be fed into another generator (an information system development environment), that produces the actual system. That sec-

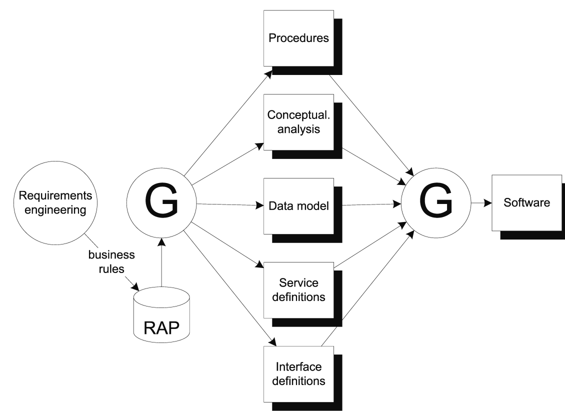

Figure 1: Rule-based design process (engineer).

ond 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 will help the requirements engineer by storing, managing and checking rules, to generate specifications, analyze rule violations, and validate the design.

From the perspective of an organization, the design process looks like figure 2. At the focus of at-

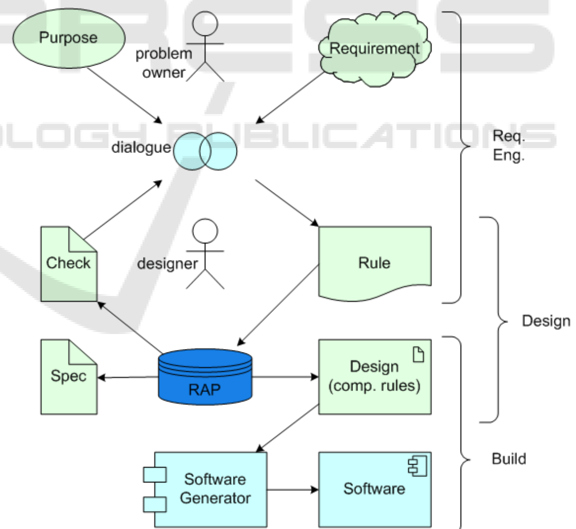

Figure 2: Rule-based design process (organization).

tention 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

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

make a buildable specification. The requirements engineer sticks to the principle of one-requirement-one-rule, enabling him to explain the correspondence of the specification to the business.

3 AMPERSAND

The purpose of Ampersand is to have the right interaction with stakeholders to define the right business rules and represent them unambiguously. Ampersand looks at business rules not only as an agreement between parties, but uses them as functional requirements to automate an information system as well. That is: these rules are to be maintained by all parties at all times and the IT must support that. Maintaining these rules is done either by people (of any party) or by computers.

A requirements engineer using Ampersand defines a suitable relational information structure to define rules upon. A meaningful specification of rules in Ampersand can be incomplete or even contain no business rule at all. In that matter, Ampersand clearly differs from more common relational specification languages like Alloy (Jackson, 1999). No business rules on an information structure represent ultimate freedom when using that information structure in business processes. This kind of freedom is explicitly useful in experimental contexts like the educational design exercises in RAP. For example, a student can first define an information structure and add rules one-by-one while studying the impact of adding another rule. More practical reasons to omit a business rule are disagreement between stakeholders, irrelevance of the rule, or unawareness of the rule.

Ampersand uses a relation algebra (Maddux, 2006) as a language in which to express business rules. Relation algebras have been studied extensively and are well known for over a century (Schröder, 1895). The use of existing and well described theory brings the benefit of a well conceived set of operators with well known properties.

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. 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, I_A and $V_{A\times B}$. We use r, s, and t as relation variables.

Let $\neg$, $^{\sqcup}$, $\sqcup$, and $⨾$ be relation operators of arity 1, 1, 2, and 2 respectively. The binary relation operators $\sqcap$,$\Rightarrow$ and $\equiv$ are cosmetic and only defined on the interpretation function.

Let $\mathbb{R}$ be the set of relation terms. We use R, S, and T as variables that denote relation terms.

$\mathbb{R}$ is recursively defined by

$$I_A, V_{A\times B}, r, \neg R, R^{\sqcup}, R \sqcup S, R ⨾ S \in \mathbb{R}$$

provided that $R,S \in \mathbb{R}, r \in \mathbb{D}$, and $A,B \in \mathbb{C}$

An Ampersand script for context $\mathfrak{C}$ is a user-defined collection of statements where

- $RUL \subseteq \mathbb{R}$ is a collection of relation terms called rules.
- REL is a collection of $r : A \sim B$ for all $r \in \mathbb{D}$ such that $A,B \in \mathbb{C}$. The instances of REL are called relation declarations.
- POP is a collection of $a\ r\ b$ such that $a \in A, b \in B$ and $(r : A \sim B) \in REL$.

The relation declarations define the conceptual structure and scope of $\mathfrak{C}$. Relation elements define facts in $\mathfrak{C}$. POP is called the population of $\mathfrak{C}$. Rules are constraints on that population.

The interpretation function $\mathfrak{I}(R)$ defines the semantics of a relation term R. This function interprets relation terms based on POP and a relation algebra $\langle \mathbb{R}, \cup, \bar{\ }, ;, \smile, \mathbb{I} \rangle$ where $\mathbb{R} \subseteq \mathcal{P}(\mathbb{U})$. All relation symbols used in a relation term are either declared by the user in REL or I_A or $V_{A\times B}$. So, the relation algebra on $\mathbb{R}$ is 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.

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

relation $$\mathfrak{I}(r) = \{\langle a,b \rangle \mid a\ r\ b \in POP\} \quad (1)$$

identity $$\mathfrak{I}(I_A) = \{\langle a,a \rangle \mid a \in A\} \quad (2)$$

universal $$\mathfrak{I}(V_{A\times B}) = \{\langle a,b \rangle \mid a \in A,\ b \in B\} \quad (3)$$

complement $$\mathfrak{I}(\neg R) = \overline{\mathfrak{I}(R)} \quad (4)$$

converse $$\mathfrak{I}(R^{\sqcup}) = \mathfrak{I}(R)^{\smile} \quad (5)$$

union $$\mathfrak{I}(R \sqcup S) = \mathfrak{I}(R) \cup \mathfrak{I}(S) \quad (6)$$

composition $$\mathfrak{I}(R ⨾ S) = \mathfrak{I}(R);\mathfrak{I}(S) \quad (7)$$

(the interpretation of the mentioned cosmetic relation operators)

$$\text{intersection} \quad \mathfrak{I}(R \sqcap S) = \mathfrak{I}(\neg(\neg R \sqcup \neg S)) \quad (8)$$

$$\text{implication} \quad \mathfrak{I}(R \Rightarrow S) = \mathfrak{I}(\neg R \sqcup S) \quad (9)$$

$$\text{equivalence} \quad \mathfrak{I}(R \equiv S) = \mathfrak{I}((R \Rightarrow S) \sqcap (S \Rightarrow R)) \quad (10)$$

Relations need to have a type to be more practical for requirements engineers (Michels et al., 2011). This way, only a relation term $R \in \mathbb{R}$ with a type $\mathfrak{T}(R)$ has an interpretation. A relation term without a type $\mathfrak{T}(R)$ is said to have a type error or to be (semantically) incorrect. Compilers for Ampersand scripts must reject relation terms with a type error. A requirements engineer might call a relation term without a type *nonsense*.

Van der Woude and Joosten (van der Woude and Joosten, 2011) have enhanced the typing function 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 (Wedemeijer et al., 2010) also discusses alternative patterns in Ampersand to model apparent subtypes of concepts.

4 GENERATING RAP

RAP includes first and second generator functions and the repository (RAP) of figure 1. The generator functions are part of a command-line tool called the Ampersand compiler. A web application of RAP for design exercises provides access to the repository and generator functions. The Ampersand compiler is manually coded in Haskell. Source and binary files of the Ampersand compiler are freely available via `wiki.tarski.nl`. The web application and repository are generated with the compiler from an Ampersand script for RAP. RAP and its full script are freely available at request.

We claim that the Ampersand compiler can generate a compliant information system from business requirements in an Ampersand script. Early versions of the Ampersand compiler could already generate a trivial, but compliant system. Such a trivial system consists of a relational database, a web interface, and a rule engine. The database model contains a table of two columns for each relation declaration in the script, to hold its population. An initial population for the database is checked to be free of violations. The web interface provides a user with basic data functions for the database. A rule engine checks all the rules before committing changes to the database in order to remain compliant with the rules c.q. the requirements. The current version of the Ampersand compiler takes rules into account and uses simple syntactic Ampersand constructs to derive more practical system components. Examples of Ampersand constructs for practical enhancements are: an interface construct to customize data access; attributes to customize violation handling; and plain text attributes to attach meaning or purpose to formal elements for traceability. In the next subsection we address a generic issue: How do business rules yield a compliant business process? In the remaining subsections we describe how a system, RAP, could be generated, which has shown to be sufficiently practical for studies on and design exercises of a course at the Open University. We use examples from the Ampersand script for RAP.

4.1 Compliant Processes

Whenever and whereever people work together, they connect to one another by making agreements and commitments. These agreements and commitments constitute the rules of the business. 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. A rule maintained by a computer is an automated activity within a business process.

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) (Shewhart and Deming, 1939).

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. 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 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 modelers, who transform the rules of the business into actions and place these actions in the appropriate order to establish the desired result.

4.2 Rule-based Structure of RAP

RAP is an application on a repository of Ampersand scripts. If RAP was just a storage for scripts, then its Ampersand structure could remain limited to only one concept $\mathit{Script} \in \mathbb{C}$; no rules; no relations. However, we want RAP to have functions to facilitate certain activities. For those functions, concepts, relations and rules need to be added to the script of RAP.

We have used RAP as:

- a design exercise tool for students. A student user of RAP has a facility to upload a script to RAP, a structured view on a script in RAP, and access to a few compiler functions to run on a script in RAP. The second release of the exercise tool anticipates on more roles than students, which are teachers, advanced students, and requirements engineers. The second release introduces rule-guided editing of a script in the structured view;
- a data source for measurements to study student behaviour;
- a data storage for measurement results to use in the exercise tool for learning.

No additional structure was needed in the script of RAP to use RAP as a data source for measurements.

If the results of measurements need to be stored in RAP, then we need at least a univalent relation from the source of the measure to the result. For example, the structured view on a script includes the counters of rules, relations and concepts, which are measurements needed for a design activity called cycle chasing. Each counter is a functional relation in the script of RAP e.g. $\mathit{count_rules} : \texttt{Script} \rightarrow \texttt{Number}$.

Most of the script of RAP relates to student activities in the exercise tool. The upload facility is a handcoded component of RAP, and is thus excluded from the script of RAP. The structured view follows the syntactic structure of a script as implemented by the script parser of the compiler. The compiler including parser is coded in Haskell, which is a functional programming language. The functional structure of a script in Haskell could easily be transposed into a relational structure in Ampersand, because a relation is less strict than a function. The view also contains derivates from a script like conceptual diagrams, a diagnosis on the typing function(Michels et al., 2011), or a report of rule violations. For example, the source concept of a relation declaration is a functional relation in the script of RAP, $\mathit{source} : \texttt{Concept} \rightarrow \texttt{Declaration}$. In the Haskell code of the compiler, the data structure *Declaration* has an attribute of type *Concept* to hold its source concept. A compiler function on a script is implemented as a functional relation from a script to a web location where the compiler output is accessible. For example, the script of RAP contains a relation to generate a functional specification from a script, $\mathit{gen_fspec} : \texttt{Script} \rightarrow \texttt{FSpec}$.

4.3 Custom Data Access

An interface construct exists in the Ampersand language to access the population of relations by means of relation algebraic expressions. An interface is a tree of labeled expressions where each node connects a parent R and child S with a composition operator $R ; S$. The interface provides an easy way to configure data access with the full power of the relation algebra. The relation algebra also has restrictions, for instance numerical calculations are not possible. Attributes can be set to customize data access like who may use an interface and which relations in an interface can be altered with that interface.

We demonstrate how to configure an interface by an example from RAP. The following interface is used to generate a web page for students to view a context.

```
-1- INTERFACE "CONTEXT" FOR Student:I[Context]
-2- BOX ["name" : ctxnm
-3-     ,"PATTERNs" : ctxpats
-4-     ,"concepts" : ctxcs
-5-     ,"ISA-relations" : ctxpats;ptgns
-6-     ,"relations" : ctxpats;ptdcs
-7-     ,"RULEs" : ctxpats;ptrls]
```

The text elements in double quotes are labels. The text after a colon is an ASCII-encoded relation algebraic expression. The first line defines a root

node with the identity relation of a concept *Context*. This means that the interface can be used to view or alter relations of any instance of the domain of the identity relation of *Context*. This interface is a view-only interface, because the attribute to grant access to edit certain relations in the interface is not set. The optional FOR-attribute restricts usage of this interface to a *Student* user only. Each instance of *Context* represents a context $\mathfrak{C}$, that could have been parsed from a script by the compiler. The box of line 2 to 7 connects six sibling nodes to the root by means of a composition operator. The relation *ctxnm* : `Context`→`ContextName` holds the relation of user-friendly names for a $\mathfrak{C}$. The relation *ctxpats* : `Context` ∼ `Pattern` holds the relation to patterns in the $\mathfrak{C}$. Patterns are syntactic containers for rules, declarations and ISA-relations i.e. *ptrls* : `Pattern` ∼ `Rule`, *ptdcs* : `Pattern` ∼ `Declaration`, and *ptgns* : `Pattern` ∼ `Gen`. The relation *ctxcs* : `Context` ∼ `Concept` holds the relation to concepts in the $\mathfrak{C}$.

4.4 Configure Layout

The layout of a web page generated from an interface is configured in Cascading Style Sheets (CSS). These web pages contain labels and textual data.

Little efforts were needed to customize the default style sheets to suit RAP. We experienced that solutions to improve the user experience and comfort are more difficult to implement in a generated page than in a hand coded page. But when such a solution is implemented on a good design, then you can easily take advantage of it in any page.

For example, RAP needs to display non-textual data like images and handlers to execute parameterized software functions. A view on a concept is invented that allows the requirements engineer to compose non-textual HTML elements based on instances of a concept. For example instances of the concept *ConceptualDiagram* are urls to actual images, which are embedded in an HTML element to display images. Likewise, all software functions on data in RAP are encapsulated by the only manually coded web page of RAP (index.php). RAP has a concept *G* refering to the G in figure 1, which contains software functions to compose HTML links to execute software like compiler functions. The Ampersand code below configures a view on an instance of G.

```
VIEW G:
 G(PRIMHTML
    "<a href='../../index.php?operation="
  ,operation           ,PRIMHTML "&file="
  ,applyto;filepath  ,applyto;filename
  ,PRIMHTML "'>"      ,functionname
  ,PRIMHTML "</a>")
```

PRIMHTML becomes actual HTML on the generated web page. Functional relations with the instance of G as a source fill the parameters of the url. The relation *operation* : `G`→`Operation` relates instances of G to an operation number. The relation *applyto* : `G`→`Script` relates instances of G to the script to which the operation of G must be applied. The relations *filepath* : `Script`→`FilePath` and *filepath* : `Script`→`FilePath` relate the script to a location on a file system. The relation *functionname* : `G`→`String` relates instances of G to a textual element to click on in order to execute a software function.

4.5 Manage Measurements for Research

The most prominent advantage of using RAP for measurements in our experience is the documentation and qualitity of input data. The quality of data in RAP is high, because data is embedded in a formally defined information structure. The information structure of data has a clear and compliant relation with business requirements, such that the purpose and meaning of data can be documented and verified easily.

Measurements do not need to be part of the script of RAP to use RAP as a data source for measurements. This makes it easy to adapt to new measurements, because no development efforts are required. We do have added a univalent relation for each of our measurements for studies to the script of RAP. Adding a univalent relation is a small effort with hardly any impact on RAP. A small advantage is gained because measurements can be documented and managed in a structured way. We have experienced control over our measurements in their implementatation and use.

Measurements have been implemented as extensions on the compiler and in spreadsheets.

5 CONCLUSIONS

In this paper we have presented an application of rule-based design to obtain a development environment for RAP.

Does RAP fulfill its purpose to facilitate teaching Ampersand as well as research on that subject? We have shown that the functional requirements for RAP can be formalized by means of an Ampersand script, such that RAP can be generated. Section 4 describes all the details of how RAP has been constructed. About 50 students per year have used RAP to complete a design exercise which defines 80% of the final grade of a master course on rule-based design. A first study has been accepted for publication

(Michels and Joosten, 2013), which uses analytics from RAP based on 52 students who have used RAP in the period between April 2010 and May 2011. This publication reports on six hypotheses based on those analytics in order to explore the possibilities to study student behaviour with RAP. From the above observations we conclude that:

- RAP has been generated with Ampersand;
- RAP is sufficiently practical for teaching and research.

For further research we have planned to identify didactical requirements for the exercise tool for students. Current analytics in RAP will be taken into account. The objective is to upgrade the 'sufficiently practical exercise tool' to a 'teaching exercise guide for students'.

Has the application of Ampersand brought us closer to understand how to teach Ampersand to master students of computer science and business management? With the development environment of RAP we created, we are ready to adopt RAP to new didactical insights and produce unambiguous analytics. Our first study has produced preliminary results on how to teach Ampersand. Thus, our research has progressed due to the application of Ampersand because:

- with Ampersand we have created a responsive and controlable environment for research;
- with Ampersand we could produce unambiguous analytics to show that meaningful research with RAP is feasible (Michels and Joosten, 2013).

The next step in research is to define didactical studies on how to teach Ampersand, which are based on analytics in RAP.

REFERENCES

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43.

Business Rule Solutions, LLC. (2013). RuleSpeak. Retrieved Februari 27, 2013, from http://www.rulespeak.com.

Clocksin, W. F. and Mellish, C. (1981). *Programming in Prolog*. Springer.

Date, C. J. (2000). *What Not How: the Business Rules Approach to Application Development*. Addison-Wesley Longman Publishing Co., Inc., Boston.

Dayal, U., Buchmann, A. P., and McCarthy, D. R. (1988). Rules are objects too: A knowledge model for an active, object-oriented database system. In Dittrich, K. R., editor, *OODBS*, volume 334 of *Lecture Notes in Computer Science*, pages 129–143. Springer.

Green, P. F. and Rosemann, M. (2000). Integrated process modeling: An ontological evaluation. *Inf. Syst.*, 25(2):73–87.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220.

Jackson, D. (1999). 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.

Maddux, R. (2006). *Relation Algebras*, volume 150 of *Studies in logic*. Elsevier, Iowa.

Michels, G. and Joosten, S. (2013). Progressive development and teaching with RAP. Accepted at CSERC'13.

Michels, G., Joosten, S., van der Woude, J., and Joosten, S. (2011). 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. Springer-Verlag.

Object Management Group, Inc. (2008). 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.

Paton, N. W. and Díaz, O. (1999). Active database systems. *ACM Comput. Surv.*, 31(1):63–103.

Ram, S. and Khatri, V. (2005). A comprehensive framework for modeling set-based business rules during conceptual database design. *Inf. Syst.*, 30(2):89–118.

Reisig, W. (1985). *Petri Nets: an Introduction*. Springer-Verlag New York, Inc., New York, NY, USA.

Ross, R. G. (2003a). The Business Rules Manifesto. Retrieved Februari 27, 2013, from http://www.businessrulesgroup.org/brmanifesto.htm.

Ross, R. G. (2003b). *Principles of the Business Rule Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Rumbaugh, J., Jacobson, I., and Booch, G., editors (1999). *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK.

Scheer, A.-W. W. (1998). *Aris-Business Process Frameworks*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition.

Schröder, E. (1895). *Algebra und Logik der Relative*. Vorlesungen über die Algebra der Logik (Exakte Logik) / Ernst Schröder. Teubner.

Shewhart, W. and Deming, W. (1939). *Statistical Methods from the Viewpoint of Quality Control*. Dover Books on Mathematics Series. Dover Publications, Incorporated.

van der Woude, J. and Joosten, S. (2011). 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. Springer-Verlag.

Wedemeijer, L., Joosten, S., and Michels, G. (2010). *Rule Based Design*. Open Universiteit Nederland, 1st edition.

Widom, J. (1996). The starburst active database rule system. *IEEE Trans. on Knowl. and Data Eng.*, 8(4):583–595.