Key words requirements engineering – software engineering – automated design – relation algebra – business rules – information system design – rule based design

Requirements Engineering Journal manuscript No.
(will be inserted by the editor)

Ampersand

a method for functional requirements specification

anonymized

Address(es) of author(s) should be given

submitted: August 11th, 2009

Abstract Is it possible to derive functional specifications directly from requirements by automated means? An affirmative answer to this question is given by the Ampersand method, a new approach to requirements specification. Ampersand is characterized by an automated translation of requirements into functional specifications and into a functional prototype. It features compliance between requirements and functional specifications, because the translation is exclusively carried out by means of semantics preserving transformations. Ampersand also features early exposure of users to the consequences of their functional requirements. The method to derive a functional specification from requirements is embodied by a compiler, called ADL. The same compiler also produces functional prototypes, i.e. working software. Each prototype exhibits the appopriate behaviour needed to keep the functional requirements satisfied throughout time. This helps requirements engineers to predict consequences of specific requirements.

1 Introduction

To write correct functional specifications can be quite a challenge for requirements engineers. They must accommodate the needs of many stakeholders, such as users, software engineers, patrons and auditors. The concerns of these stakeholders may conflict, however. For example, users may wish to add, change, and remove requirements as insight progresses, whereas software engineers may want to freeze requirements in order to get systems built. Also, stakeholders must be able to convince themselves that their requirements are well represented. In order to build systems, software engineers must get a complete and consistent representation of all functional requirements. On top of all that, stakeholders want functional specifications fast so they can take the impact of their requirements into consideration. In a situation where the attention of the requirements engineer is drawn towards the concerns of stakeholders, where time is pressing, and stakes are contradictory, how much effort can a requirements engineer spend on the correctness and completeness of the functional specification? These demands illustrate the challenge that requirements engineers face.

The general problem addressed by this paper is how to represent functional requirements from the perspective of a requirements engineer, in order to obtain correct functional specifications from them Specifically, this paper proposes an incremental approach to requirements specification, Ampersand[1]. Ampersand produces a correct functional specification together with a functional prototype, by means of a compiler called ADL. By formalizing functional requirements in the language ADL, a requirements engineer can evaluate consequences already during the requirement elicitation phase.

The guarantor of a correct functional specification is the requirements engineer. In order to tempt him to specify functional requirements in a formal manner, Ampersand rewards him with functional prototypes and correct functional specifications. However, if this is to work in practice, the use of formalism must be restricted to requirements engineers only. The practical work conducted with Ampersand (section 6.2) has proven this. With an ADL-compiler at hand, learning the formalism becomes much more like learning a programming language. For the same purpose, ADL has been kept as minimal as possible. It contains a faithful implementation of relation algebra, without any exceptions or extensions. Knowing that mathematics is not everyone's favorite subject, we have spent substantial effort on the learning material.

Ampersand allows requirements engineers to address their stakeholders exclusively in natural language. Stakeholders on the user side, such as patrons, users, and auditors, need not be confronted with any formalism at

[1] The name Ampersand comes from the ampersand symbol, &, which stands for *and*. It is an appeal to requirements engineers who want to have it all: business requirements *and* information technology, theory *and* practice, process control *and* IT-services

all. Patrons may benefit by getting assurance that their functional specifications are correct. This eliminates a well known risk factor in IT-projects. Users may benefit by getting the opportunity to play with functional prototypes. Auditors may benefit by the assurance that the functional specifications are provably compliant with requirements.

Ampersand has been tried on various occasions, the most ambitious of which was the design the information provisioning for the Dutch Immigration Office (IND). Thus, it has already established its value for the requirements engineering practice. These efforts have also led to a course at the Open University of the Netherlands, which is currently taken by approximately 100 students per year.

This paper proceeds with a brief overview of the background of this research. Ampersand is introduced by means of an example in section 3. Section 4 then discusses the method itself. Technical details of the method are provided in section 5, which describes the derivation of functional specifications and provides the semantics of the software defined by that functional specification. Results obtained so far with Ampersand are described in section 6. That section also discusses some of the issues raised by Ampersand: Can business analysts learn to specify requirements in ADL? What are the implications for the software engineering process? Which limitations are encountered in the use of Ampersand? The paper concludes in section 8 that requirements are not only necessary, but in fact sufficient to define a fully functional service layer for information systems. It also concludes that the compositionality of Ampersand yields interesting perspectives to improve the design of large information systems.

2 Background

There is a widespread awareness of the problems caused and cost incurred by informally specifying software [25, 18]. The need to represent requirements formally (established for instance in [36]) and the availability of numerous formal specification techniques, such as [50, 26, 56], underscore the need for more precise specifications. Ampersand employs a formal technique. Requirements engineers are expected to create a concrete set of requirements in a formal language. All efforts in our research have been directed towards making the formalism as simple as possible, producing tools to make life easier, and to spend effort in teaching methods. In this respect, Ampersand is different from approaches that elicit formal notations from natural languages, as in [38] or in business rule management systems such as Haley.

So what is new? Although many have attempted to formalize functional specifications, proposals to formalize functional requirements are fewer and less well known [5, 17, 20]. Ampersand differs from these methods, because it uses relation algebra as specification language. Other methods use logic (e.g. logic programming [33] or description logic [2]) or ontology (e.g. OWL [11]). Being executable, ADL differs also from formal specification languages that are non-executable such as Z [50], CSP [45], LOTOS [24], VDM [26], and Larch [19]. Relation algebra itself has been used before, e.g. by Khedri and Bourguiba [31], but for a different purpose. They relate state transitions for calculating architectural components. They too observed that mainstream design methods are based on crafty procedures and not on rigorous procedures founded on mathematics.

What is the basic idea behind Ampersand? If Ampersand represents functional requirements as rules, and functional requirements are under jurisdiction of the business [32], the central idea behind the method must be business rules [40]. Among others, Chris Date advocates a more declarative approach to information system design [8], and puts business rules in the center of the design process. A similar plea is heard coming from the business rules community, which argues strongly in favour of declarative rules [53]. Business rules are rules of, by, and for business people (art. 9 of the Business Rule Manifesto [53]). Recently, business rules are increasingly advocated to serve as functional requirements (e.g. [55, 46]).

In Ampersand, a business rule represents a condition in the business that should be true at all times or should be made true if it happens to be false. A business rule is much like a law or an agreement between people. Consider for instance the following business rule: “Any permit application is decided upon by an employee, who is authorized to handle applications from county in which the applicant lives.”. As long as there is no application, all is well. Suppose a permit application arrives. Now this rule is no longer true, because there is an application, but no one has decided upon that application. In order to make it true, someone has to make a decision. That someone must be an employee, authorized to handle applications from the applicant’s county. The rule does not say who should do it, nor what that person should do in order to take a decision. It is merely a condition that must be kept true at all times.

Ampersand adopts the view that each rule is to be maintained at all times, either by human intervention or by automated activity. By representing business rules in relation algebra, maintaining a rule (i.e. keeping it true at all times) can be supported by a computer. If a situation occurs in which the rule is not satisfied, the computer can signal that situation to persons (or computers) who can fix it. In this way, appropriate action is triggered. This is how information technology can satisfy all functional requirements at all times. The concreteness of a rule implies that every corresponding functional requirement is SMART (specific, measurable, attainable, realisable, traceable [37]). Thus Ampersand helps re-

quirements engineers to make functional requirements SMART.

The ADL-language, embedded in the tool ADL, is based on relation algebra [35]. This is a branch of mathematics that has been studied extensively for over a century [10,43,48] and has inspired the paradigm of relational databases, SQL [7] included. Date, however, has criticized the database community, and SQL in particular, for being unfaithful to relation algebra [8]. Relation algebra has also inspired other things, such as Prolog [6], a programming language that uses Horn-clauses (a restricted class of relation algebra expressions) as a means to write computer programs. Ampersand's choice was inspired by observing that relation algebra comes surprisingly far in representing functional requirements. It inspired us to represent numerous problems in this manner, just to see where practice poses its limitations. The cases reported later in this paper demonstrate that business processes and the information systems supporting them can be specified in this way.

The following reasons motivate our choice to use relation algebra:

1. Each requirement in natural language corresponds to one rule in relation algebra. This helps the requirements engineer to focus, because he can occupy himself with one rule at a time.
2. Semantics are well defined, which is necessary for deriving results by means of a computer. The semantics of the relation algebra used by Ampersand are defined in section 5.
3. Reasoning in relation algebra is point free. This makes it ideally suited for equational reasoning, which is convenient for automatically proving compliance between rules and functional specifications.
4. Relation algebra has strong ties with natural language, which is necessary to facilitate dialogues with various stakeholders.
5. The operational semantics are understandable, which is necessary to use functional prototypes for sharpening the requirements.
6. Relation algebra is non-procedural, which is common in mathematics but not in the business rules community. This property is helps to preserve semantics of functional requirements in the ADL compiler.

Relation algebra has also inspired research in functional dependencies in the seventies and eighties, which led to the development of active databases [9,34,42,57]. This development has expanded the relational paradigm with Event-Condition-Action (ECA) rules. Today, action rules dominate the scene of the business rule community. Relation algebra, however, does not specify action. It specifies conditions that must remain true at all times. Such rules are called *invariant*. Ampersand, being a syntactically sugared version of relation algebra, also specifies rules of the business by invariants rather than action rules. Like business rules [53], a rule in Ampersand represents a state of affairs that must invariably remain true at all times. This makes business rules essentially similar to relation algebra rules (i.e. Ampersand rules) Both specify the business in a non-procedural way (article 4 of the business rule manifesto [53]). The ADL-compiler derives action rules from invariants in order to make the (procedural) software for the functional prototype. Since a limited number of invariants gives rise to a large number of action rules, the two types of rules cannot be treated as the same thing.

3 Example

In order to demonstrate how requirements are transformed into an information system, this section shows a retail process that is defined by ten functional requirements. Although this example has been kept simple for the sake of brevity, it does involve providers and clients who exchange orders, deliveries, and invoices. The requirements are presented in section 3.1. In Ampersand, requirements are expressed in natural-language sentences for the business audience. Formalization is done separately by the requirements engineer, which is discussed in section 3.2. From the formalized requirements, the ADL compiler generates a fully operational functional prototype, which is shown in section 3.3. The demonstration consists of a walkthrough of steps interlaced with screenshots. Section 3.4 finally discusses the functional specification produced by the ADL compiler. That concludes this example, after which section 4 will proceed to discuss the method, i.e. the steps a requirements engineer takes to produce requirements.

3.1 Requirements of the Retail Process

The retail process in this example consists of orders, deliveries and invoices. Orders are routinely accepted by the provider who receives the order. The provider will subsequently deliver and send an invoice. Finally, the client will pay. The stakeholders in this example are providers and clients.

The actual business process is defined as an agreement between providers and clients. For this particular example, the process consists of the following functional requirements

1. Accepting an order is always done by the provider to whom the order was addressed.
2. Some orders may be awaiting acceptance. These orders are signalled to the appropriate provider.
3. Deliveries are made by the provider who has accepted an order.
4. Ultimately, each order accepted must be delivered by the provider who has accepted that order. The provider will be signalled of orders waiting to be delivered.

5. Deliveries are made to the client who ordered the delivery.
6. A client accepts invoices for delivered orders only.
7. There must be a delivery for every invoice sent.
8. For each delivery made, the provider must send an invoice. The provider must be signalled when invoices can be sent.
9. The client shall pay only for invoices sent to him.
10. Each invoice sent to a client must be paid. Both the client and the provider must be signalled for payments due.

Each of these requirements can be interpreted as a business rule, i.e. a rule that all parties have agreed to live by.

3.2 Formalization

Ampersand considers statements 1 thru 10 not only as an agreement between parties, but uses them as functional requirements for an information system as well. That is: these statements are to be maintained at all times, so they are the business rules that govern this particular process. To *maintain* a rule means to ensure that the rule is satisfied at all times. If for some reason a rule is (temporarily) not satisfied, its violation can be signalled to human participants. That signal is then interpreted as a prompt for intervention, upon which a human actor is supposed to react. Once truth has been restored, for instance as a consequence of human intervention, the signal stops. If a rule is not signalled to human participants for intervention, the system issues an error rather than wait for the reaction that never comes. In that way, all rules are being maintained: either by humans (through the signalling process) or by the computer.

The requirements engineer is responsible for formalizating functional requirements. The ADL compiler will then generate a prototype and a functional specification. In order to formalize the requirements from section 3.1, the requirements engineer defines ten relations[2]. The notations used in the following specification follow the conventions of relation algebra and are defined in section 5.

$$
\begin{aligned}
\mathit{of} &: \mathit{Delivery} \to \mathit{Order} \\
\mathit{provider} &: \mathit{Delivery} \to \mathit{Provider} \\
\mathit{accepted} &: \mathit{Order} \to \mathit{Provider} \\
\mathit{issuedTo} &: \mathit{Order} \to \mathit{Provider} \\
\mathit{deliveredTo} &: \mathit{Delivery} \to \mathit{Client} \\
\mathit{from} &: \mathit{Order} \to \mathit{Client} \\
\mathit{sentTo} &: \mathit{Invoice} \to \mathit{Client} \\
\mathit{delivery} &: \mathit{Invoice} \to \mathit{Delivery} \\
\mathit{sentBy} &: \mathit{Invoice} \to \mathit{Provider} \\
\mathit{paidBy} &: \mathit{Invoice} \to \mathit{Client}
\end{aligned}
$$

[2] In this particular example, all relations happen to be functions. In other situations nonfunctional relations occur as well.

The requirements engineer also defines the meaning in natural language of each relation, as given in table 1.

expression	meaning
$o = \mathit{of}(d)$	Delivery d corresponds to order o.
$p = \mathit{provider}(d)$	Provider p has delivered delivery d.
$p = \mathit{accepted}(o)$	Provider p has accepted order o.
$p = \mathit{issuedTo}(o)$	Order o is addressed to provider p.
$c = \mathit{deliveredTo}(d)$	Delivery d has been delivered to client c.
$c = \mathit{from}(o)$	Client c has issued order o.
$c = \mathit{sentTo}(i)$	Invoice i is addressed to client c.
$d = \mathit{delivery}(i)$	Invoice i covers delivery d.
$p = \mathit{sentBy}(i)$	Provider p has sent invoice i.
$c = \mathit{paidBy}(i)$	Client c has paid invoice i.

Table 1 Natural language meaning of the relations

In this example, the requirements engineer has described two multiplicity requirements:

- for every order there is a corresponding delivery (in the relation *of*).
- for every delivery there is a corresponding invoice (in the relation *delivery*).

Formalization[3] of the requirements yields one business rule (in relation algebra) for each requirement:

1. $\mathit{accepted} \vdash \mathit{issuedTo}$
   Accepting an order is always done by the provider to whom the order was addressed.
2. $\mathit{issuedTo} \vdash \mathit{accepted}$
   Orders that are awaiting acceptance are signalled to the provider to whom the order was addressed.
3. $\mathit{accepted}; \mathit{of}^{\smile} \vdash \mathit{provided}$
   If a provider has accepted an order, then that provider is the one to make the delivery.
4. $\mathit{of}^{\smile}; \mathit{accepted} \vdash \mathit{provider}$
   Ultimately, each order accepted must be delivered by the provider who has accepted that order. The provider will be signalled of orders waiting to be delivered.
5. $\mathit{of}; \mathit{from} \vdash \mathit{deliveredTo}$
   Deliveries are made to the client who ordered the delivery.
6. $\mathit{delivery}^{\smile}; \mathit{sentTo} \vdash \mathit{deliveredTo}$
   A client accepts invoices for delivered orders only.
7. $\mathit{sentBy} = \mathit{delivery}; \mathit{of}; \mathit{issuedTo}$
   An invoice must sent by the provider to whom an order was issued.
8. $\mathit{provider} = \mathit{delivery}^{\smile}; \mathit{from}$
   For each delivery made, the provider must send an invoice.
9. $\mathit{paidBy} \vdash \mathit{sentTo}$
   Accept payments only for invoices sent.

[3] The mathematical meaning of the relational operators is defined formally in section 5.

10. $sentTo \vdash paidBy$
    Each invoice sent to a client must be paid.

The requirements engineer is responsible for the correspondence between each business rule in natural language (section 3.1) and the corresponding formalization. He is trained to formulate rules in natural language such that there is a one to one correspondence between each business rule and the corresponding formal rule. Also, ADL offers some support in that it can translate from relation algebra to natural language. Besides, the requirements engineer assigns roles to rules. If a rule is linked to a role, that role will receive signals whenever that rule is violated. Thus, the person fulfilling that role can take appropriate measure to resolve the signal. If a rule is not linke to a role, a computer will take charge of maintaining that rule. In that case, violations of the rule are not tolerated and will result in error messages from the computer.

In this example: rules 1, 4, and 8 are to be maintained by the provider, rule 10 is maintained by the customer. The remaining rules, 2, 3, 5, 6, 7, and 9 are maintained by the computer.

3.3 Demonstration

A requirements engineer can generate a functional prototype by processing the rules through an ADL compiler. By implementing these rules in the service layer (service bus) of an enterprise, the rules are being maintained throughout the organization. Thus, all rules can be respected regardless of the user interface systems on top or the software systems connected to that service bus. In this way, Ampersand specifies a so called compliant services layer (CSL). Figure 1 shows a screenshot of the CSL generated for the retail process from section 3.2). This screen shows the complete state of the entire retail process for all roles in a single screen. In reality, every role will only see his or her own part. That involves covering the CSL by a appropriate user interface for each role. Needless to say that portal technology is an attractive means to achieve this.

The demonstration shows the CSL as an interactive screen, with entities and signals. The ADL compiler has generated three entities (orders, deliveries, and invoices) and the appropriate functionality to create, update and delete individual instances of those entities. Also, it has generated four signals (order, deliver, pending, and payable) to prompt individuals for actions to be taken. The data shown in figure 1 represents three completed process instances and four empty signals. The four signals being empty means that nobody is prompted, so there is no work to be done according to the system.

By way of demonstrating the functional prototype, let us walk through a typical scenario step by step:

1. The client creates a new order by pressing the "Create new" button in the window "Order" and filling in the fields "issuedTo" and "from". Figure 2 shows the corresponding screen shot, just after Joosten[4] has created an order for Balmark's. This generates a signal called "order" for Balmark's, who is the provider. This follows logically from rule 1. By means of this signal, Balmark's can know that a new order has arrived. Under the hood of the rule engine, issuing an order has temporarily violated rule 1. The violation consists of an order that exists without having been accepted. It is this violation that is used as a signal to prompt Balmark's.
2. Balmark's accepts the order, resulting in the situation of figure 3. Notice that the signal "Order" is no longer up, but instead, the delivery signal has been raised. That signal prompts Balmark's to deliver the order. The signal has been derived by the ADL compiler as a logical consequence of rule 4.
3. Notice too, that the delivery of order 31617 is already expected in the table of deliveries (figure 3), even though a delivery record does not exist yet. The reason is that the requirements engineer defined the relation *of* as a surjective relation, so the compiler knows that this delivery "is coming". The situation after delivering Joosten's order is given in figure 4. We now see that the signal "pending" prompts Balmark's to send out an invoice, which is a logical consequence of rule 8.
4. The provider sends an invoice, which creates a signal to the client that an invoice is due for payment. Figure 5 now shows the table of delivery records and invoice records, and the new contents of the signals. As a logical consequence of rule 10, Joosten is prompted for payment.
5. Once Joosten has paid the invoice, no more signals are raised. As shown in figure 6 all signals are empty. This means that the order has been processed completely and the case is closed. In case somebody other than Joosten would have paid the invoice, the system would refuse to change the state of the database and produce an error message instead. This happens because rule 9 is being violated, which has no role assigned to it. Therefore the computer will maintain rule 9, simply by refusing to execute the transaction.

This demonstration showed the execution of (one instance of) the retail process defined in section 3.1. The way it works can be understood by realising that rules are maintained by signalling all possible violations, and subsequently dealing with them or preventing them to affect the data stored in the system. In this way, the entire set of data is kept free of violations for all rules maintained by the system.

The functional prototype, however is not the real reason for a requirments engineer to use Ampersand. The purpose is to obtain a correct functional specification. That is discussed in the following section.

[4] a Dutch name, pronounced "Yoasten"

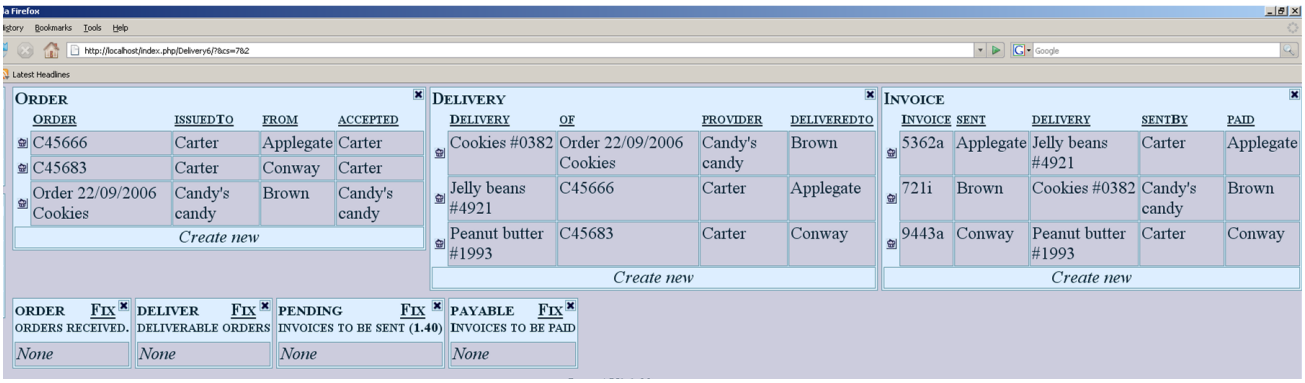

Fig. 1 Functional Prototype demonstration

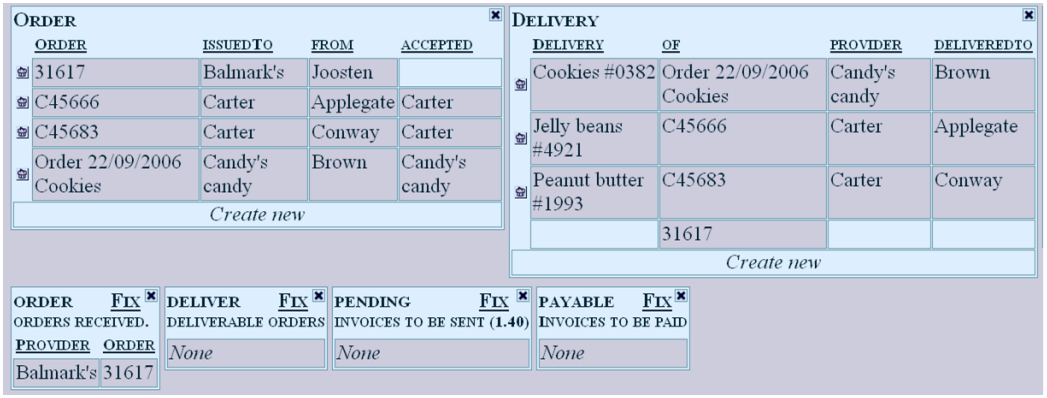

Fig. 2 Joosten has issued an order

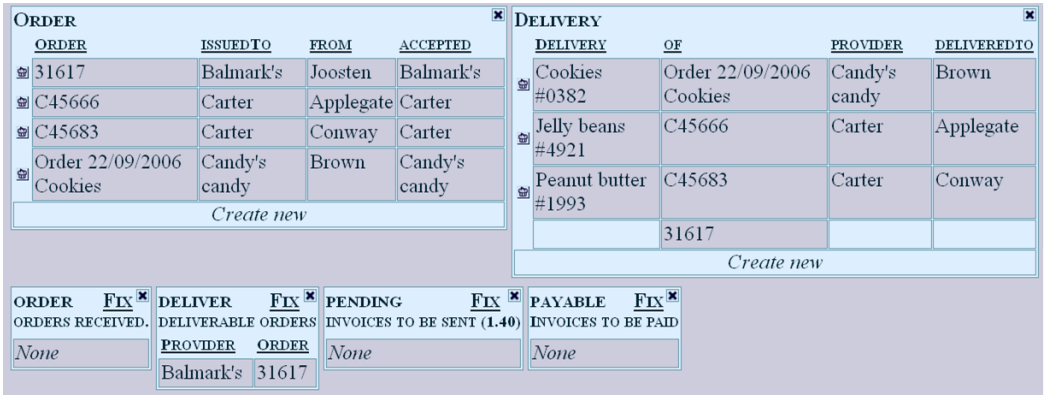

Fig. 3 Balmark's has accepted Joosten's order

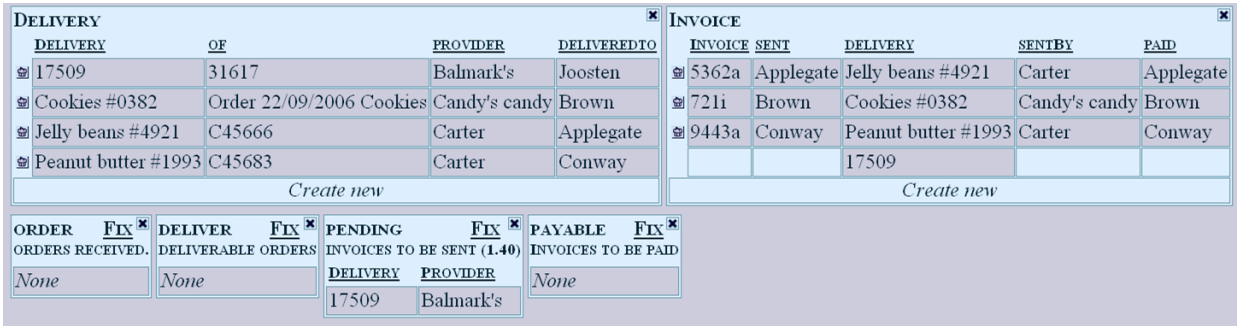

Fig. 4 Balmark's has delivered Joosten's order

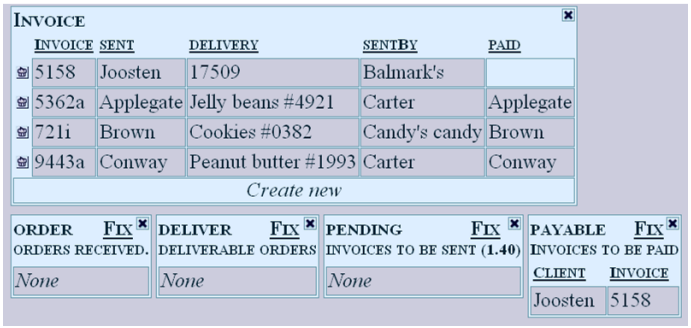

Fig. 5 Balmark's has sent Joosten an invoice

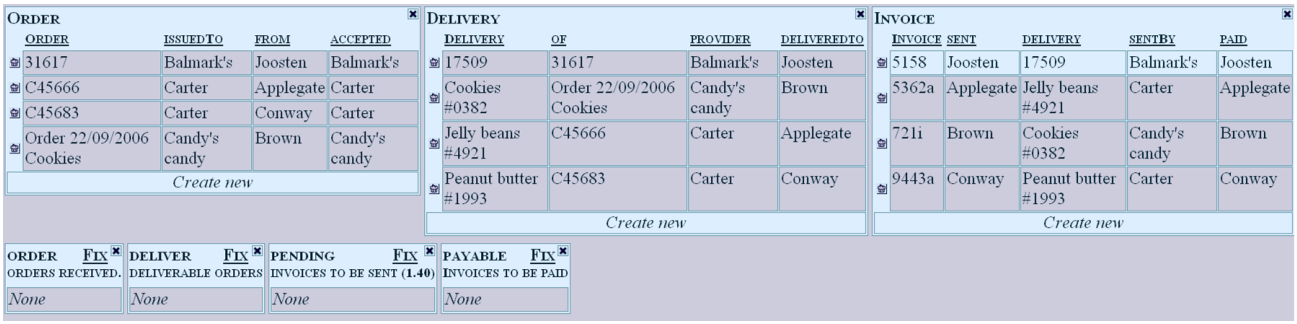

Fig. 6 Joosten has paid his invoice

3.4 Functional Specification

The purpose of Ampersand is to generate a functional specification for an information system that supports the requirements of the business. After having formalized those requirements, the ADL compiler generates a document that contains the functional specification of a service layer. This service layer is intended for use within a service oriented architecture. For brevity's sake, we present a small selection of samples taken from that document.

For every service, a formal specification is generated that specifies precisely what each service must do in order to maintain all business rules. The formal specification is provided in terms of pre- and postconditions as introduced by Floyd and Hoare [15,22]. For brevity's sake, this article shows the full specification of one service only: `updOrder`. ADL generates the complete specification for all services. Each service specification that is generated consists of the service heading, the pre- and postcondition semantics, and the invariants that must be maintained by this particular service.

```
updOrder( In  x : OrderHandle ;
          In a1 : Provider    ;
          In a2 : Provider    ;
          In  f : Client      )
```

When called, this service behaves as follows[5]:

```
      {Pre: True}
updOrder(x,a1,a2,f)
      {Post: x.accepted = a1 and
             x.issuedTo = a2 and
             x.from = f             }
```

Besides, the service call `updOrder(x,a1,a2,f)` must ensure truth of the following conditions after the call has finished, provided that condition is true when the service is called:

1. $\forall o : Order : \ o.accepted \ = \ o.issuedTo$
2. $\forall d : Delivery : \ d.provider \ = \ d.of.accepted$
3. $\forall d : Delivery : \ d.of.accepted \ = \ d.provider$
4. $\forall d : Delivery : \ d.of.from \ = \ d.deliveredTo$
5. $\forall i : Invoice : \ i.sentTo \ = \ idelivery.of.from$

The document that comprises the functional specification can be generated in the form of LaTeX source code, RTF or MS-word format. It can therefore be used inside a larger document describing the information system.

This example has demonstrated how a requirements engineer can define a retail process and the underlying service layer. We have presented the prototype software and the functional specification generated by the ADL compiler. The contribution of Ampersand is that these artifacts are generated directly from the (formalized) requirements. In the following chapter we discuss the steps of the Ampersand method, showing the central role of the requirements engineer and the simplification of the software process by automating design activities.

[5] The notation is pre-postcondition, also known as Floyd-Hoare notation

4 Method

Ampersand is a method for defining software services and controlling business processes, comprising the steps of compiling a set of business rules, checking that set for type consistency and completeness, synthesizing a service catalogue, synthesizing a functional prototype, and synthesizing a document called "functional specification". Ampersand is characterized by the use of a relation algebra for representing business rules. Another characteristic is that the method is automated. A third characteristic is compliance of the services defined in the functional specification to the business requirements that are input of the method.

The ADL compiler was developed in an evolutionary way. The current version of the compiler is being used in practice both in education (Open University of the Netherlands) and in industry (Ordina and TNO-ICT).

This section explains the way Ampersand derives functional specifications. Each step of Ampersand is discussed separately.

4.1 Compiling

A compiler translates business rules, written in any language that satisfies the axioms in section 5. An instance of such a language has been implemented. This language is called ADL (A Description Language), and it compiles a plain ASCII text file. This particular embodiment of an ADL compiler was built in the functional programming language Haskell, using Swierstra's parser combinator package[6] [51,52]. Besides relation algebra to describe rules, ADL features design patterns, which are basically a set of rules, contexts in which rules are applied, and a signalling construct to define which rules are to be signalled to people and which are to be maintained by a computer.

4.2 Type checking

Type checking is done on all rules put together, to ensure that the entire set of rules has a semantically consistent interpretation and that the compiler can produce meaningful output. A type checker must check the rules imposed by heterogeneous relation algebra [47], which are specified in table 3 in section 5. It must also check the classification of concepts (e.g. Affidavit isa Document), which has a tree structure (an antisymmetric structure which is the reflexive, transitive closure of all specializations defined by users, see section 5). The reflexive, transitive closure must be computed by the type checker and the antisymmetry must be checked.

An instance of such a type checker has been built in Haskell, using the rules of table 3. Antisymmetry of the isa-relation is established by means of a Warshall-algorithm, which is also used to compute the reflexive, transitive closure of isa-pairs.

Any specification that passes the type checker can be compiled into a functional specification and a functional prototype. That is: the type checker is a filter that passes only specifications that have a semantically sound interpretation. The remaining sections are written under the assumption that the set of rules is type correct.

4.3 Synthesizing a service catalogue

Services are defined for every class that has attributes and for all relations that are not an attribute. For any class with attributes, four services are defined: a create, get, remove and update service. Besides, if that class has keys, a get-service for each key is defined as well. For all relations that are not used as attribute, insert and delete services are defined.

Each service is specified formally, as described in section 3.4. Every service specification is self contained, so the work to implement these services can be distributed over a team of programmers to implement. If the service satisfies its pre- and postcondition specification and if it maintains the invariants specified, then that service maintains all business rules. A faithful implementation of every service will therefore yield a service layer that is compliant to those business rules. Since the service layer maintains all rules at all times, this layer is called the Compliant Service Layer (CSL). The benefit of having a CSL is that there is a mathematical guarantee that the requirements are maintained.

In practice, it is not always necessary to build every service from the specification. Nevertheless, also partial implementations are compliant, because every single service maintains all rules in the applicable context. So, depending on the application(s) on top, a service layer must implement only those services that are actually used.

4.4 Synthesizing a functional prototype

A working prototype that maintains requirements specified in ADL has been demonstrated in section 3.3. This prototype consists of a model, a view and a controller. The model is built as a database (MySQL) which has been generated from the relations and multiplicities specified in ADL. The functionality has been generated in terms of SQL queries, covered by a thin layer of PHP-services. The controller is implemented in PHP. It regulates the dialogue in the view, while calling the database services as needed. The view has been implemented in HTML, which is generated from PHP. Thus, the prototype can be run from anywhere on the internet. A trial installation has been built on Linux with an Apache

[6] http://www.cs.uu.nl/wiki/HUT/WebHome

server, and another one on a Windows-XP computer, also running Apache.

The functionality of the functional prototype consists of CRUD-functionality for all generated entities, insert-delete functionality for all independent relations, signalling functionality, and checking for errors. Since the structure of the services and the structure of the user interface are both derived from one single ADL-model, This structure has been implemented in PHP by means of PHP-tables.

A limitation of this architecture is that rules cannot be changed on the fly. Instead, one has to recompile the (adapted) set of rules and reinstall. Future research is anticipated to create a bootstrapping system, in which rules can be removed and added on the fly.

4.5 Synthesizing a functional specification

The ADL compiler can be used on the internet (URL: `http://www.sig-cc.org/Adl`). This way, it can be used without download-and-install. It translates ASCII-files ADL-syntax to either

- a set of HTML-pages, producing online documentation of the design;
- a .PDF document containing the functional specification;

The use of LaTeX enables the requirements engineer to introduce pieces of explanations in the generated text, or to restructure the specification to suit specific needs.

The functional specification has the following text structure:

1. The first chapter is an introduction, that specifies the purpose of the document and introduces the remainder of the text.
2. The second chapter enumerates all requirements in natural language. Only this chapter needs approval of the patron, because the remainder of the specification covers these requirements precisely (no more, no less).
3. The third chapter is a conceptual analysis, in which the business rules are formalized in a traceable manner. It is used by the requirements engineer to ensure that the formal representation of a business rule corresponds to the informal representation in natural language.
4. The fourth chapter contains a data model.
5. The fifth and subsequent chapters contain the service catalogue. Each chapter covers one design pattern, defining all services of that particular pattern formally.

This section introduced a compiler, type checker, service catalogue, functional prototype generator, and functional specification generator as a sequence of steps that are used to automate the design of information systems and business processes. These steps constitute the Ampersand method, which is an automated method.

5 Theory

What is under the hood of an Ampersand engine? The ADL language and the ADL compiler are based on a nontrivial heterogeneous relation algebra. In this section we present the full algebraic definition. The message is that any language that satisfies the axioms of this section is regarded as a faithful implementation of ADL. This way of defining the language has been chosen to allow maximum room for syntactically sugaring the language without comprimising the method. The definition draws on existing theory [35], which has been subject of study for over a century. This brings the benefit of a well conceived set of operators with well known properties.

The presentation may deviate somewhat from conventional ways to present relation algebra. The reason is that Ampersand uses one particular interpretation of heterogeneous algebras (i.e. an allegory of sets), whereas literature on algebras abstracts away from interpretations wherever possible.

5.1 Operators

Ampersand expresses rules in terms of relations and operators to connect these relations. A binary relation $r : A \times B$ is a subset of the cartesian product $A \times B$, in which A and B are concepts. In English: a relation $r : A \times B$ is a set of pairs $\langle a, b\rangle$ with $a \in A$ and $b \in B$.

Rules are represented in an algebra $\langle \mathcal{R}, \cup, \overline{\ }, ;, \breve{\ }, \mathbb{I}\rangle$, in which $\mathcal{R}$ is a set of relations, $\cup$ and ; are binary operators of type $\mathcal{R} \times \mathcal{R} \to \mathcal{R}$, $\overline{\ }$ and $\breve{\ }$ are unary operators of type $\mathcal{R} \to \mathcal{R}$, $\mathbb{I}$ represents an identity relation, and the algebra satisfies the following axioms for all relations $r, s, t \in \mathcal{R}$:

$$r \cup s = s \cup r \tag{1}$$
$$(r \cup s) \cup t = r \cup (s \cup t) \tag{2}$$
$$\overline{\overline{x} \cup \overline{y}} \cup \overline{\overline{x} \cup y} = x \tag{3}$$
$$(r; s); t = r; (s; t) \tag{4}$$
$$(r \cup s); t = r; t \ \cup \ s; t \tag{5}$$
$$r; \mathbb{I} = r \tag{6}$$
$$r^{\breve{}\breve{}} = r \tag{7}$$
$$(r \cup s)^{\breve{}} = r^{\breve{}} \cup s^{\breve{}} \tag{8}$$
$$(r; s)^{\breve{}} = s^{\breve{}}; r^{\breve{}} \tag{9}$$
$$r^{\breve{}}; \overline{r; y} \cup \overline{s} = \overline{s} \tag{10}$$

The full relation is denoted $\mathbb{V}$ and the empty relation is $\overline{\mathbb{V}}$. To facilitate understanding of the operators, table 2 explains these operators in more accessible terms. This table uses x, y, and z in an unbound manner, meaning that universal quantification (i.e. "for all x, y, and z") is assumed.

To enrich the expressive power for users, Ampersand uses other operators as well. These are the binary operators $\cap$, † both of which are of type $\mathcal{R} \times \mathcal{R} \to \mathcal{R}$. They

name	notation	meaning
fact	$x\ r\ y$	there is a pair between x and y in relation r. Also: the pair $\langle x,y\rangle$ is an element of r, ($\langle x,y\rangle \in r$).
declaration	$r : A\times B$	There is a relation r with type $[A,B]$.
declaration	$f : A\rightarrow B$	There is a function f with type $[A,B]$. (A function is a relation)
implication	$r \vdash s$	if $x\ r\ y$ then $x\ s\ y$. Alternatively: $x\ r\ y$ implies $x\ s\ y$. Alternatively: r is included in s or s includes r.
equality	$r \equiv s$	$x\ r\ y$ is equal to $x\ s\ y$
complement	$\overline{r}$	all pairs not in r. Also: all pairs $\langle a,b\rangle$ for which $x\ r\ y$ is not true.
conversion	$r^{\smile}$	all pairs $\langle y,x\rangle$ for which $x\ r\ y$.
union	$r \cup s$	$x(r \cup s)y$ means that $x\ r\ y$ or $x\ s\ y$.
composition	$r;s$	$x(r;s)y$ means that there is a z such that $x\ r\ z$ and $z\ s\ y$.
identity	$\mathbb{I}$	equals. Also: $a\ \mathbb{I}\ b$ means $a = b$.

Table 2 Operators and their semantics

are defined by:

$$r \cap s = \overline{\overline{s} \cup \overline{r}} \tag{11}$$

$$r \dagger s = \overline{\overline{s};\overline{r}} \tag{12}$$

5.2 Clauses

A clause is the smallest rule that can be maintained by itself, so the set of rules produced by the type checker is transformed to a set of clauses. In this subsection we define clauses and what it means to maintain clauses.

A *rule* is an expression e that must be maintained. To *maintain* rule r means to ensure that $\mathbb{V} \vdash r$ at all times. In order to discuss maintaining rules, we must introduce some vocabulary. As long as $\mathbb{V} \vdash r$, we say that r is *satisfied*, or simply that it *holds*. If r is not $\mathbb{V}$, we say that rule r is *not satisfied*, or *invariance is broken*, or it *does not hold*. Since $r \cup \overline{r} = \mathbb{V}$, the missing pairs are determined by $\overline{r}$. Each pair in $\overline{r}$ is called a *violation* of r. To *restore invariance* of rule r means to change the contents of r in such a way that $\mathbb{V} \vdash r$.

Let r_0, r_1, ..., r_n be the rules that apply in context C. Let R_C be defined by $r_0 \cap r_1 \cap ... \cap r_n$. Expression R_C is called the *conjunction* of r_0, r_1, ..., r_n, because all expressions r_i are separated by the conjunction operator $\cap$. Each r_i is called a *conjunct*. Maintaining all rules (every r_i) in context C means to ensure that $R_C = \mathbb{V}$ at all times[7].

A *clause* is the smallest expression that is still a rule (i.e. are equal to $\mathbb{V}$). Let e_0, e_1, ..., e_m be unique expressions such that:

$$r_0 \cap r_1 \cap ... \cap r_n \ = \ e_0 \cap e_1 \cap ... \cap e_m \tag{13}$$

with m the largest possible number and 'unique' meaning that for every i and j

$$e_i = e_j \ \Rightarrow \ i = j \tag{14}$$

So each e_i thus defined is a clause. Clauses are determined by a normalization procedure that brings all rules in conjunctive normal form, and substituting the set of rules by the conjuncts thus obtained. Since every rule has a conjunctive normal form, each of its conjuncts is a clause in disjunctive normal form.

[7] Informally: R_C is kept true at all times

5.3 Concepts

For representing relations, Ampersand uses an algebra of concepts $\langle \mathcal{C}, \sqsubseteq, \top, \bot\rangle$, in which $\mathcal{C}$ is a set of Concepts, $\sqsubseteq$: $\mathcal{C} \times \mathcal{C}$, and $\top \in \mathcal{C}$ and $\bot \in \mathcal{C}$, satisfying the following axioms for all classes $A, B, C \in \mathcal{C}$:

$$A \sqsubseteq \top \tag{15}$$

$$\bot \sqsubseteq A \tag{16}$$

$$A \sqsubseteq A \tag{17}$$

$$A \sqsubseteq B \wedge B \sqsubseteq A \Rightarrow A = B \tag{18}$$

$$A \sqsubseteq B \wedge B \sqsubseteq C \Rightarrow A \sqsubseteq C \tag{19}$$

Concept $\bot$ is called 'anything' and $\top$ is called 'nothing'. Predicate $\sqsubseteq$ is called *isa*. It lets the user specify things such as 'an affidavit is a document'.

If $r : A\times B$ is a relation, A is called the *source* of the relation, and B the *target* of the relation. A relation that is a function (i.e. a univalent and total relation) is denoted as $r : A\rightarrow B$

5.4 Types

For the purpose of defining types, operator $\sqcap : \mathcal{C}\times\mathcal{C} \rightarrow \mathcal{C}$ and predicate $\diamond : \mathcal{C} \times \mathcal{C}$ are defined:

$$A \diamond B \Leftrightarrow A \sqsubseteq B \vee B \sqsubseteq A \tag{20}$$

$$A \sqsubseteq B \Rightarrow A \sqcap B \ = \ B \tag{21}$$

$$B \sqsubseteq A \Rightarrow A \sqcap B \ = \ A \tag{22}$$

$$\neg(A \diamond B) \Rightarrow A \sqcap B \ = \ \top \tag{23}$$

Predicate $\diamond$ defines whether two types are compatible and operator $\sqcap$ (the least upper bound) gives the most specific of two compatible types. Note that the conditions at the left of equations 21 through 23 are complete,

so $\sqcap$ is defined in all cases. Also, in case of overlapping conditions $A \sqsubseteq B \wedge B \sqsubseteq A$, axiom 19 says that $A = B$. causing $A \sqcap B$ to be unique. Ampersand is restricted to concepts that are not $\bot$ nor $\top$, but the two are needed to signal type errors.

Any language with operators that satisfy axioms 1 through 19 is a suitable language for Ampersand. Table 3 gives the names and types of all operators. Not all expressions that can be written are type correct. An expression must satisfy the condition mentioned in table 3, otherwise it is undefined. A compiler for a language to be used for this purpose must therefore contain a type checker to ensure that all expressions satisfy the conditions mentioned in table 3.

Let $r : A \times B$ and $s : P \times Q$

name	notation	condition	type
relation	r		$[A, B]$
implication	$r \vdash s$	$A \diamond P \wedge B \diamond Q$	$[A \sqcap P, B \sqcap Q]$
equality	$r \equiv s$	$A \diamond P \wedge B \diamond Q$	$[A \sqcap P, B \sqcap Q]$
complement	$\overline{r}$		$[A, B]$
conversion	$r^{\smile}$		$[A, B]$
union	$r \cup s$	$A \diamond P \wedge B \diamond Q$	$[A \sqcap P, B \sqcap Q]$
intersection	$r \cap s$	$A \diamond P \wedge B \diamond Q$	$[A \sqcap P, B \sqcap Q]$
composition	$r; s$	$B \diamond C$	$[A, C]$
relative addition	$r \dagger s$	$B \diamond C$	$[A, C]$
identity	$\mathbb{I}_A$		$[A, A]$

Table 3 Operators and their types

After having shown an example in section 3, explaining the method in section 4, and presenting the semantics of Ampersand in the current section, the next section discusses the practical results that have been achieved with Ampersand in practice.

6 Results

The contribution of Ampersand is that it allows correct functional specifications to be derived from functional requirements. In order to establish this contribution, we must show that functional specifications are correct, that this result is useful in practice, and that requirements can be formalized by the targeted community of requirements engineers. The evidence for these results is presented in the following three sections. Corectness is discussed in section 6.1. Relevance is established in section 6.2. Teaching evidence, showing that requirements engineers can do it, is presented in section 6.3

6.1 Correct functional specifications

Why can we trust a functional specification, produced by the ADL-compiler, to be correct? The evidence consists of the ADL-compiler itself, which has been constructed to transform a set of relation algebra rules into a functional prototype by using correctness preserving transformations. The functional specification is generated by software that "documents" the functional prototype. This ensures that the functional specification defines the prototype.

Work is in progress to derive correctness proofs for the prototype itself. Once that is available, correctness evidence can be presented for each individual set of requirements separately.

The fact that the functional prototype works is psychologically important for users, especially those who cannot understand the formal proofs.

6.2 Practical results

Various research projects and projects in business have supplied input to and evidence on Ampersand. These projects have been conducted in various locations. Research at the Open University (OUNL) has focused on two things: developing the method, developing a course, and building a violation detector. Research at Ordina and TNO Informatie- en Communicatie Technologie has been conducted to establish the usability of ADL-rules for representing genuine business rules in genuine enterprises. Collaboration with the university of Eindhoven has produced a first design of a rule base in 2004. Experiments conducted in various collaborative settings have provided experimental corroboration of the method and insight in the limitations and practicalities involved. This section discusses some of these projects, pointing out which evidence has been acquired by each one of them.

The CC-method [12], the predecessor of Ampersand, was conceived in 1996, based on ideas obtained during a sabbatical visit of the author to Georgia State University in the academic year 1995/96 [29]. This resulted in a conceptual model called WorkPAD [28] and conceptual studies such as [29]. The method used relational rules to analyze complex problems in a conceptual way. WorkPAD was used as the foundation of process architecture as conducted in a company called Anaxagoras, which was founded in 1997 and is now part of Ordina. Conceptual analyses, which frequently drew on the WorkPAD heritage, applied in practical situations resulted in the PAM method for business process management [30], which is now frequently used by process architects at Ordina and within her customer organizations. Another early result of the CC-method is an interesting study of van Beek [54], who used the method to provide formal evidence for tool integration problems; work that led to a major policy shift in a large IT-project of a governmental department. The early work on Ampersand has provided evidence that an important architectural tool had been found: using business rules to solve architectural issues is large projects.

For lack of funding, the CC-method has long been restricted to be used in conceptual analysis and meta-

modeling [13], although its potential for violation detection became clear as early as 1998. Metamodeling in CC was first used in a large scale, user-oriented investigation into the state of the art of BPM tools [14], which was performed for 12 governmental departments. In 2000, a violation detector was written for a large commercial bank, which proved the technology to be effective and even efficient on the scale of such a large software development project. After that, the approach started to take off.

In the meantime, TNO-ICT used Ampersand for various other purposes. For example, it was used to create consensus between groups of people with different ideas on various topics related to information security, leading to a security architecture for residential gateways [27]. Another purpose that TNO-ICT used Ampersand for was the study of international standardizations efforts such as RBAC (Role Based Access Control) in 2003 and architecture (IEEE 1471-2000) [23] in 2004. Several inconsistencies were found in the last (draft) RBAC standard [1].

The efforts at TNO-ICT have provided the evidence that Ampersand works for its intended purpose, which is to accellerate discussions about complex subjects and produce concrete results from them in practical situations. The technique has been used in conceiving several patents[8]

In 2002 research at the OUNL was launched to further this work in the direction of an educative tool. This resulted in the ADL language, which was first used in practice by a private bank [4]. The researcher described rules that govern the trade in securities. He found that business rules can very well be used to do perpetual audit, solving many problems where control over the business and tolerance in daily operations are in conflict. This work was unfortunately not followed up by that bank. At a large cooperative bank, in a large project for designing a credit management service center, debates over terminology were settled on the basis of metamodels built with Ampersand. These metamodels resulted in a noticable simplification of business processes and showed how system designs built in Rational Rose should be linked to process models [3]. The entire design of the process architecture [41] was validated in Ampersand. At the same time, Ampersand was used to define the notion of skill based distribution. This study led to the design by Ordina of a skill based insurance back-office. The same business rules that founded this design were reused in 2004 to design an insurance back office for another household brand insurer, an effective example of reuse design knowledge. This work provided useful insights about reuse of design knowledge. It also demonstrated that a collection of business rules may be used as a design pattern [16] for the purpose of reusing design knowledge. In 2006, the method was formalized and refined and described at the OUNL, yielding the current Ampersand approach. This resulted in the capability to generate functional specifications from rules. In 2007, Ampersand was used to design a brand new application architecture for the Dutch immigration service, IND. It allowed a substantial part of the design to be delivered during the tendering phase, before the project even started. It was also used to convince auditors during the tendering phase of the viability of the system.

[8] e.g. patents DE60218042D, WO2006126875, EP1727327, WO2004046848, EP1563361, NL1023394C, EP1420323, WO03007571, and NL1013450C.

6.3 Teaching results

Even if a method works, one still needs people who can do the job. Being founded in mathematics, there is a possibility that such people are difficult to find. To solve this problem, much attention has been given to teaching Ampersand to students. Writing business rules from business requirements is now being taught in the regular computer science curriculum of the Open University of the Netherlands. In order to find out how to teach this, the method was first taught in two trial courses, once in a university course and once in an industrial course. The industrial course was monitored closely by the Open University of the Netherlands, so the results could be compared. Those courses have shown that business analysts can learn how to formalize business requirements in approximately 100h (roughly 4 European Credit points).

The use of relation algebra is the challenge from an educational point of view, because mathematical techniques are not the most appealing educational goals for professionals and students with a passion for business applications.

The course consisted of an information session for potential students, 8 meetings, homework assignments in pairs, a flash quiz at the start of each meeting, a midterm essay and an examination at the end. The information session appeared to be an essential motivating issue. Motivating for students are the following factors:

- design automation gives them more time to talk to users;
- the fact that the functional specification is compliant gives them self-confidence in discussions with customers;
- compliance means that better discussions with programmers can be conducted.

In both courses, all students but one could cope with the mathematics. This one student left the course after the first meeting, mentioning insufficient prerequisite knowledge as the cause.

One of the courses was done in a classroom setting, the other course was conducted in a virtual classroom over the Internet. The results of both groups did not differ significantly.

7 Discussion

The Ampersand method described in section 4 and the evidence presented in section 6 support the claim that a functional specification can be derived from business requirements.

This claim raises a plethora of discussion issues, such as:

1. Which benefits are in it for the requirements engineer?
2. Business rules are around for some time. How do rule engines support this?
3. What is the role of modeling in information system design, when using Ampersand?
4. How does a compliant functional specification yield a compliant business process?

The following sections discuss each of these issues separately.

7.1 Benefits

Requirements engineering comprises many activities: eliciting requirements, modelling and analysing, communicating requirements, agreeing requirements, and evolving requirements [39]. From these activities, Ampersand addresses modelling and analysis. It yields a correct functional specification, meaning that the translation of formalized requirements is done automatically through semantics preserving transformations. As a consequence, requirements engineers get incremental specification and compositionality.

Correctness means that the functional specification defines software services that maintain all requirements in every applicable scope at all times. The specification, generated by ADL, contains a formal specification of every software service involved, where every software service is consistent with the requirements. Flaws are detected and diagnosed by the ADL software. It generates a functional specification only after all requirements have been defined in a formally consistent manner. As a consequence, the semantics of the functional specification corresponds provably with the requirements. On top of that, ADL generates a functional prototype. In the eyes of users, this proves the correctness quite convincingly.

Using the ADL compiler, requirements engineers can now work in an incremental manner. Incremental requirements specification means that incrementally adding requirements incurs incremental rework. This can improve current practice, where designers must scrutinize the entire design every time a requirement changes. For example, adding the requirement "*Every financial transaction over € 5000,- must be checked twice*", would normally force process modelers to scrutinize all business processes for adding extra checks for all financial transaction in every single business process. Conventional practice is even worse if a new requirement affects the data model. It may take considerable effort to assess the impact of a single change, which may be felt throughout the entire system. This phenomenon is a "scattering effect", because a single requirement change may have an impact in many different places in the system. As Ampersand generates a functional specification from requirements, it automates this scattering effect and reduces the effort of designers to this respect.. They no longer need to trace the impact and and change design models by hand. By automating the scattering effect, Ampersand has removed an important obstacle towards incremental requirements elicitation.

Compositionality means that specifications can be composed by putting components together. In data model oriented methods or object oriented methods, composing model fragments into larger models often results in making a larger model from scratch. The effort of composing models into larger models can be reduced by using sets of requirements rather than graphical models as the main building block of a large system design. Two sets of requirements can be composed by the union of both sets, whereas it takes human effort and intelligence to combine two data models into one larger model. ADL merges different sets of requirements, possibly originating from different stakeholders, into one specification. For example, the specification of a new mortgage system can be assembled by putting together requirements from a security department, from mortgage experts, from the IT-department, and subsequently from all other stakeholders.

7.2 Rule engines

In a comparison of available methodologies for representing business rules, Herbst et.al. [21] have argued that common methods are insufficient or at least inconvenient for a complete and systematic modeling of business rules. That study remarks that rules in all methodologies can be interpreted as condition-action-rules or event-condition-action rules (collectively identified as ECA-rules). Most rule engines available in the market, such as ILOG[9], Corticon[10], and Fair-Isaac[11] employ ECA-rules. These tools are typically used in decision support situations, in which many business rules from numerous different sources apply. Examples are mortgages, life insurances, permit issuing, etc. where repetitive decisions must be made legally, transparently, traceably,

[9] ILOG. Business rule components. http://www.ilog.com/products/rules/, August 2001.

[10] Pedram Abrari and Mark Allen, Business rules user interface for development of adaptable enterprise applications, US Patent 7,020,869, March 28, 2006

[11] Serrano-Morales; Carlos A., Mellor; David J., Werner; Chris W., Marce; Jean-Luc, Lerman; Marc, Approach for re-using business rules, US Patent 7,152,053, December 19, 2006.

and timely. Rule engines are generally being deployed as a supplement to process engines. A process engine controls the business process and the rule engine computes the decision, using its rule base, process data, and data from other information systems.

From this article it may be clear that ECA-rules are incompatible with Ampersand, because it requires relation algebra instead. Ampersand uses business rules a different purpose than supporting decisions; it is about automating the design of information systems.

There is some confusion in the market about the phrase 'business rule'. It is defined by the Business Rules Community [46] as a declarative requirement, owned by the business. ECA-rules, however, are imperative in nature. This brings the tool vendor community in the awkward position of having to explain why their ECA-rules are declarative. It would be preferrable to distinguish different uses of business rules. Decision making is what most rule engines today support, and ECA-rules are a perfectly suitable way for doing that. Ampersand uses business rules for a different purpose, which is to capture business requirements in order to produce a functional specification. There are no commercial tools for Ampersand on the market yet.

7.3 Design of Information Systems

This article argues that business requirements are sufficient for a functional specification of services. The word 'sufficient' suggests that requirements engineers need not communicate with the business in any other way and that models such as class diagrams and process diagrams would be obsolete. 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 business consultant 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 business processes directly by means of business rules has consequences for requirements engineers, who will encounter a largely automated design process. From their perspective, the design process is depicted in figure 7. 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. The ADL compiler described in this article is an embodiment of that generator. These design artifacts can then be fed into another generator (an information system development environment), that produces the actual system. That second generator is typically a software development environment, of which many exist and are heavily

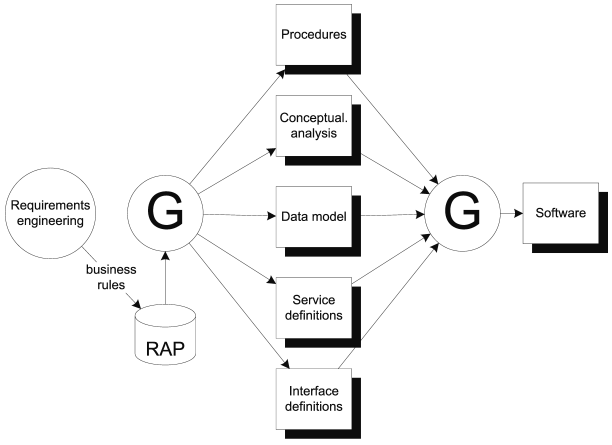

Fig. 7 Automated software process for rule based design

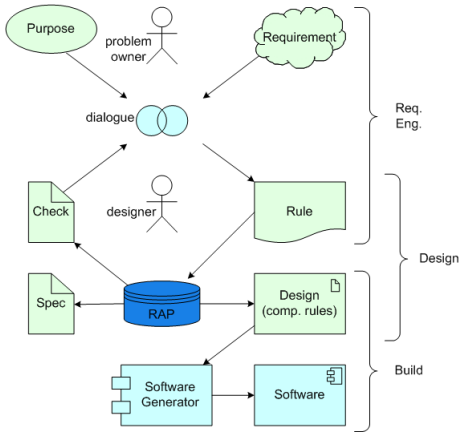

Fig. 8 Design process for rule based process management

used in the industry. Alternatively, the design can be built in the conventional way as a database application. A rule base 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 8. 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. If a requirements engineer consequently makes one rule correspond to one requirement, this helps to ensure that all functional requirements of business stakeholders are represented correctly.

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

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

Since all rules (both the ones maintained by people and the ones maintained by computers) are monitored, computers and persons together form a system that lives by the rules. This establishes compliance. Business process management (BPM) is also included, based on the assumption BPM is all about handling cases. Each case (for instance a credit approval) is governed by a set of rules. This set is called the *procedure* by which the case is handled (e.g. the credit approval procedure). Actions on that case are triggered by signals, which inform users that one of these rules is (temporarily) violated. When all rules are satisfied (i.e. no violations with respect to that case remain), the case is *closed*. This yields the controlling principle of BPM, which implements Shewhart's Plan-Do-Check-Act cycle (often attributed to Deming) [49]. Figure 9 illustrates the principle. Assume the existence of an electronic infrastructure that contains data collections, functional components, user interface components and whatever is necessary to support the work. An adapter observes the business by drawing information from any available source (e.g. a data warehouse, interaction with users, or interaction with information systems). The observations are fed to a detector, which checks them against business rules in a rule base. If rules are found to be violated, the detector signals a process engine. The process engine 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. Computer software is used to derive the actions, generating the business processs directly from the rules of the business. In comparison: workflow management derives actions from a workflow model, which models the procedure in terms of actions. Workflow models are built by modelers, who transform the rules of the business into actions and place these actions in the appropriate order to establish the desired result.

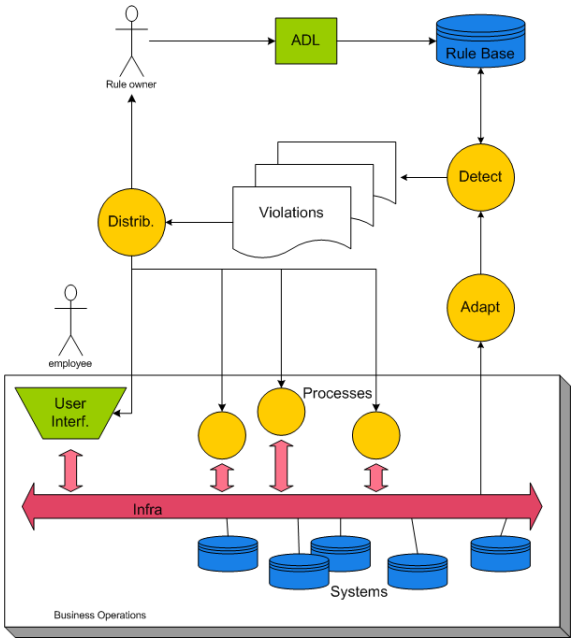

Fig. 9 Principle of rule based process management

8 Conclusion

The main conclusion of this research is that functional requirements can be transformed in correct functional specifications and that a functional prototype can be derived from them. This can be done by representing functional requirements in relation algebra. This yields a platform independent specification of a compliant service layer.

This conclusion may have implications for the design of information systems and business processes. It means that a substantial amount of design work can be automated, fewer shortcomings are to be expected in functional specifications, shorter and more predictable design times can be expected, the reliability of delivery can increase, and the IT-services specified by means of Ampersand are mathematically provable compliant with the rules of the business.

An advantage of Ampersand is that stakeholders can communicate exclusively in natural language, so they need not be exposed to any formalisms or any model. A course in Ampersand is feasible in approximately 100 hours. In the long run, Ampersand may help to reduce the risk and improve the results of large IT-projects [44].

Further research is required in the following areas:

- **Design automation**
  Further design automation is foreseen in process modeling. Section 7.4 of this article makes clear that many temporal consequences can be derived from business rules. There are concrete ideas for deriving process models as well, to further enrich the generated design artifacts.
- **A rule repository**
  In practice, the number of rules soon becomes unmanageably large. A rule repository is needed for that purpose. This repository must also control access to rules, and link each rule to its scope.
- **Process control**
  Section 7.4 argues how business processes can be controlled directly from business rules. This principle requires further research, because it might spawn a paradigm of controlling processes by process engines that work without process models.
- **Graphical business rules**
  Although business consultant can readily learn how to write relation algebra, it might be useful to have graphical representations of business rules that are more appealing than mathematical formulas. One experiment has been conducted in this area, showing that this topic is promising and requires further research.
- **Proofs**
  This article describes a functional specification generator that yields provably correct specifications. For particular applications (e.g. in banking), that proof needs to be given in writing. Users of Ampersand, who are typically business analyst, are not always capable of doing that. Research is currently on its way to derive such proofs from the same business rules. That research too must yield a generator.
- **Teaching**
  In order to make Ampersand usable, much attention is spent on teaching. The question how to educate business analysts in producing good designs in Ampersand requires rethinking of current design practices.

Acknowledgements Thanks are due to the sponsor of this research, the Open University of the Netherlands. I wish to thank Ordina and all of her customers who have contributed to this work. The fact that some of these customers cannot be mentioned by name does not diminish their contributions. The issues they raised in their projects have inspired Ampersand directly and indirectly. The RelMics community (http://www2.cs.unibw.de/Proj/relmics/html/) deserves much credit for making relation algebra accessible in practice.

References

1. ANSI/INCITS 359: Information Technology. *Role Based Access Control Document Number: ANSI/INCITS 359-2004*. InterNational Committee for Information Technology Standards (formerly NCITS), Feb. 2004.
2. Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., Eds. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
3. Baardman, E., and Joosten, S. Procesgericht systeemontwerp. *Informatie 47*, 1 (Jan. 2005), 50–55.
4. Barends, R. Activeren van de administratieve organisatie. Research report, Bank MeesPierson and Open University of the Netherlands, November 26, 2003.
5. Borgida, A., Greenspan, S. J., and Mylopoulos, J. Knowledge reperesentation as the basis for requirements specification (reprint). In *Wissensbasierte Systeme* (1985), W. Brauer and B. Radig, Eds., vol. 112 of *Informatik-Fachberichte*, Springer, pp. 152–169.
6. Clocksin, W. F., and Mellish, C. S. *Programming in PROLOG*. Springer-Verlag, Berlin, New York, 1981.
7. Codd, E. F. A relational model of data for large shared data banks. *Communications of the ACM 13*, 6 (1970), 377–387.
8. Date, C. J. *What not how: the business rules approach to application development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
9. Dayal, U., Buchmann, A. P., and McCarthy, D. R. Rules are objects too: A knowledge model for an active, object-oriented databasesystem. In *Lecture notes in computer science on Advances in object-oriented database systems* (New York, NY, USA, 1988), Springer-Verlag New York, Inc., pp. 129–143.
10. De Morgan, A. On the syllogism: Iv, and on the logic of relations. *Transactions of the Cambridge Philosophical Society 10*, read in 1860, reprinted in 1966 (1883), 331–358.
11. Dean, M., and Schreiber, G. OWL web ontology language reference. W3C recommendation, W3C, February 2004.
12. Dijkman, R. M., Ferreira Pires, L., and Joosten, S. M. Calculating with concepts: a technique for the development of business process support. In *Proceedings of the UML 2001 Workshop on Practical UML - Basesd Rigorous Development Methods Countering or Integrating the eXstremists* (Karlsruhe, 2001), A. Evans, Ed., vol. 7 of *Lecture Notes in Informatics*, FIZ.
13. Dijkman, R. M., and Joosten, S. M. An algorithm to derive use case diagrams from business process models. In *Proceedings IASTED-SEA 2002* (2002).

14. Dommelen, W. v., and Joosten, S. Vergelijkend onderzoek hulpmiddelen beheersing bedrijfsprocessen. Tech. rep., Anaxagoras and EDP Audit Pool, 1999.
15. Floyd, R. W. Assigning meanings to programs. In *Mathematical Aspects of Computer Science* (Providence, Rhode Island, 1967), J. T. Schwartz, Ed., vol. 19 of *Proceedings of Symposia in Applied Mathematics*, American Mathematical Society, pp. 19–32.
16. Fowler, M. *Analysis Patterns - Reusable Object Models*. Addison-Wesley, Menlo Park, 1997.
17. Gibson, J. P. Formal requirements models: Simulation, validation and verification. Technical Report: NUIM-CS-2001-TR-02, Feb. 2001.
18. Glass, R. L. Study supports existence of software crisis; management issues appear to be prime cause. *Journal of Systems and Software 32*, 3 (1996), 183–184.
19. Guttag, J. V., and Horning, J. J. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.
20. Hausmann, J. H., Heckel, R., and Taentzer, G. Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ACM, pp. 105–115.
21. Herbst, H., Knolmayer, G., Myrach, T., and Schlesinger, M. The specification of business rules: A comparison of selected methodologies. In *Proceedings of the IFIP WG8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle* (New York, NY, USA, 1994), Elsevier Science Inc., pp. 29–46.
22. Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (October 1969), 576–580 and 583.
23. IEEE: Architecture Working Group of the Software Engineering Committee. *Standard 1471-2000: Recommended Practice for Architectural Description of Software Intensive Systems*. IEEE Standards Department, 2000.
24. ISO. ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour. Standard, International Standards Organization, Geneva, Switzerland, 15 February 1987. First edition.
25. Johnson, D. M. The systems engineer and the software crisis. *SIGSOFT Softw. Eng. Notes 21*, 2 (1996), 64–73.
26. Jones, C. B. *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1986.
27. Joosten, R., Knobbe, J.-W., Lenoir, P., Schaafsma, H., and Kleinhuis, G. Specifications for the RGE security architecture. Tech. Rep. Deliverable D5.2 Project TSIT 1021, TNO Telecom and Philips Research, The Netherlands, Aug. 2003.
28. Joosten, S. Workpad - a conceptual framework for workflow process analysis and design. Unpublished, 1996.
29. Joosten, S., and Purao, S. R. A rigorous approach for mapping workflows to object-oriented is models. *Journal of Database Management 13* (October-December 2002), 1–19.
30. Joosten et.al., S. *Praktijkboek voor Procesarchitecten*, 1st ed. Kon. van Gorcum, Assen, September 2002.
31. Khedri, R., and Bourguiba, I. Formal derivation of functional architectural design. In *2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM'04)* (Washington, DC, USA, 2004), IEEE Computer Society Press, pp. 356–365.
32. Linehan, M. H. Sbvr use cases. In *RuleML '08: Proceedings of the International Symposium on Rule Representation, Interchange and Reasoning on the Web* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 182–196.
33. Lloyd, J. W. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.
34. Loucopoulos, P., McBrien, P., Schumaker, F., Theodoulidis, B., Kopanas, V., and Wangler, B. Integrating Database Technology, Rule-based Systems and Temporal Reasoning for Effective Software: the TEMPORA paradigm. *Journal of Information Systems 1* (Feb. 1991), 129–152.
35. Maddux, R. *Relation Algebras*, vol. 150 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, 2006.
36. Maher, M. L., and de Silva Garza, A. G. Developing case-based reasoning for structural design. *IEEE Expert: Intelligent Systems and Their Applications 11*, 3 (1996), 42–52.
37. Mannion, M., and Keepence, B. Smart requirements. *SIGSOFT Softw. Eng. Notes 20*, 2 (1995), 42–47.
38. Martínez-Fernández, J. L., González, J. C., Villena, J., and Martínez, P. A preliminary approach to the automatic extraction of business rules from unrestricted text in the banking industry. In *NLDB '08: Proceedings of the 13th international conference on Natural Language and Information Systems* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 299–310.
39. Nuseibeh, B., and Easterbrook, S. Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering* (New York, NY, USA, May 2000), ACM Press, pp. 35–46.
40. OMG. Semantics of business vocabulary and business rules (sbvr). Standard, Object Management Group, Needham, Massachussetts, 2 January 2008. First edition.
41. Ordina, and Rabobank. Procesarchitectuur van het servicecentrum financieren. Tech. rep., Ordina and Rabobank, Oct. 2003. presented at NK-architectuur 2004, www.cibit.nl.
42. Paton, N., and Diaz, O. Active Database Systems. *ACM Computing Surveys 1*, 31 (1999), 63–103.
43. Peirce, C. S. Note b: the logic of relatives. In *Studies in Logic by Members of the Johns Hopkins University (Boston)* (1883), C. Peirce, Ed., Little, Brown & Co.
44. Rainer, A., and Hall, T. Identifying the causes of poor progress in software projects. In *IEEE METRICS* (2004), IEEE Computer Society, pp. 184–195.
45. Roscoe, A. W., Hoare, C. A. R., and Bird, R. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
46. Ross, R. G. *Principles of the Business Rules Approach*, 1 ed. Addison-Wesley, Reading, Massachussetts, Feb. 2003.
47. Schmidt, G., Hattensperger, C., and Winter, M. *Heterogeneous Relation Algebra*. in: Relational Methods

in Computer Science, Advances in Computing Science. Springer-Verlag, 1997, ch. 3, pp. 39–53.
48. Schröder, F. W. K. E. Algebra und logik der relative. In *Vorlesungen über die Algebra der Logik (exacte Logik)* (first published in Leipzig, 1895), Chelsea.
49. Shewhart, W. A. *Statistical Method From the Viewpoint of Quality Control*. Dover Publications, New York, 1988 (originally published in 1939).
50. Spivey, J. *The Z Notation: A reference manual*, 2nd ed. International Series in Computer Science. Prentice Hall, New York, 1992.
51. Swierstra, S. D., Azero Alocer, P. R., and Saraiava, J. Designing and implementing combinator languages. In *Advanced Functional Programming, Third International School, AFP'98* (1999), D. Swierstra, P. Henriques, and J. Oliveira, Eds., vol. 1608 of *LNCS*, Springer-Verlag, pp. 150–206.
52. Swierstra, S. D., and Duponcheel, L. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming* (1996), J. Launchbury, E. Meijer, and T. Sheard, Eds., vol. 1129 of *LNCS-Tutorial*, Springer-Verlag, pp. 184–207.
53. The Business Rules Group. Business rules manifesto – the principles of rule independence. available from http://www.BusinessRulesGroup.org, Jan. 2003.
54. van Beek, J. Generation workflow - how staffware workflow models can be generated from protos business models. Master's thesis, University of Twente, 2000.
55. Wan-Kadir, W. M. N., and Loucopoulos, P. Relating evolving business rules to software design. *Journal of Systems Architecture 50*, 7 (2004), 367–382.
56. Warmer, J., and Kleppe, A. *The object constraint language: precise modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
57. Widom, J. The starburst active database rule system. *IEEE Transactions on Knowledge and Data Engineering 8*, 4 (1996), 583–595.