Deriving Functional Specifications from Business Requirements with Ampersand

STEF JOOSTEN
Open University of the Netherlands
Ordina NV

How do we know that a functional specification (of some information system) satisfies the requirements put forward by the business? This paper proposes a method, Ampersand, as an answer to this problem. It also proposes an accompanying tool, ADL, that generates data models, service catalogues and their specifications, to assist requirements engineers in producing a sound functional specification. Ampersand improves the life of requirements engineers in two ways: it gives them the means to prove business requirements consistent and complete and it gives them several design artifacts to facilitate their discussions with software designers. The generated functional specification features platform independence and provable compliance with the rules of the business.

In a case study, a set of requirements is presented together with the formal specification generated from it. This example, featuring an order-delivery-invoice process, complies fully with the business rules.

Ampersand has been tried in various practical situations with paying customers and classroom experience has shown that this approach can be taught to business analysts. This evidence supports the feasibility from a practitioner's point of view.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.10 [**Software Engineering**]: Design Methodologies; I.1.4 [**Symbolic and Algebraic Manipulation**]: Applications

General Terms: Design, Languages

Additional Key Words and Phrases: Automated design, business processes, business rules, information system design, rule based design, software engineering, requirements engineering.

1. INTRODUCTION

Requirements describe the users intentions expectations and needs, but also delimit the information system needed to realize these intentions. One of the challenges of requirements engineering is to how to unify the informal process of capturing the needs of users with the formal process of specifying an information system. The first requires communicative skills and informal techniques, the latter requires formal techniques to prescribe the desired properties of an information system and its context. This paper proposes to translate business requirements into a functional specification by means of a compilation process. That addresses the challenge in

bottomstuff

two ways. One the user side, business requirements can be kept entirely in natural language. On the technical side, the functional specification can be entirely formal. The compiler guarantees compliance between the two.

The contribution of this article consists of a method, Ampersand, and an accompanying tool, ADL, both intended for requirements engineers. From a given set of formally defined business requirements, ADL generates a functional specification consisting of a data model, a service catalogue, a formal specification of the services, and a function point analysis. New to this approach is the use of relation algebra for requirements elicitation. Each requirement is represented as a rule in relation algebra, which serves as an invariant to be satisfied by the business. We call these rules 'business rule' to indicate that it is owned and motivated by the business. The one-to-one correspondence of a business requirement with a business rule offers traceability. ADL translates business rules back into the natural language, providing adequate feedback to the requirements engineer for validating his work. ADL also translates the entire set of requirements to design artifacts that are required in the subsequent software engineering process.

In this article we consider every activity that precedes software design to be in the scope of requirements engineering. A distinction is made between descriptive and prescriptive activities [Greenspan et al. 1994]. This article uses the word *requirement* in a descriptive sense, to describe an explicit or implicit need of users. On the other hand, the word *specification* is used to prescribe properties of a system to be built. The phrase *functional* specification denotes a prescription of desired functionality, and a *non-functional* specification prescribes any other desired quality, such as a performance property, a security property, or an efficiency property.

Many attempts have been made to formalize functional specifications, such as Z [Spivey 1992], CSP [Roscoe et al. 1997], LOTOS [ISO 1987], VDM [Jones 1986], Larch [Guttag and Horning 1993], but the number of attempts to formalize requirements are fewer and less known [Borgida et al. 1985; **?**; **?**]. Still there are a number of good reasons to formalize requirements too. These reasons form the rationale for developing Ampersand:

—A formally denoted requirement is inherently falsifiable. An informal requirement is not necessarily falsifiable, making it unsure whether such a requirement can be used objectively as a criterion of satisfaction. A formal statement that has to be kept true during the lifetime of the information system, provides requirements engineers with an objective satisfaction criterion.

—Requirements are compositional. A specification is not. Adding and dropping requirements during the requirements elicitation must converge to a final specification. This specification must satisfy all requirements (i.e. it is complete) and contain no contradictions (i.e. it is consistent). In order to preserve compositionality, it makes sense to derive specifications from requirements.

—The specification must be traceable, meaning that every element can be traced back to the requirement to motivate its existence.

—Communication with users is error-prone, due to the use of natural language. A formalized requirements language is a powerful instrument for a requirements engineer, for the purpose of unveiling flawed use of natural language, arbitrating consensus where necessary, and facilitating discussions about requirements.

—A formal requirement language is better suited for software support.

To use relation algebra as a requirements language to functional specifications, which is yields stronger compliance guarantees during requirements elicitation. This allows requirements engineers to translate business requirements directly to functional specifications. The effect is that they can guarantee the compliance of the functional specification with business requirements, and the produce a number of design artifacts that software designers require. Khedri and Bourguiba [Khedri and Bourguiba 2004] have addressed this problem, motivated by their observation that one of the shortcomings of mainstream design methods is that their processes are based on crafty procedures and not on rigorous procedures founded on mathematics. They used relation algebra to express state transitions, using a graph clustering technique to obtain architectural components. However, the authors have recognised the drawback of using clustering techniques, being that most graph partitioning problems are shown to be NP-complete or NP-hard. They propose further empirical studies to establish what is an acceptable complexity for large software systems. Ampersand differs from Khedri's approach, because it uses relation algebra to describe business invariants. Khedri and Bourguiba use relation algebra to relate state transitions.

Others have pursued the idea of integrating business requirements with specifications (e.g. [Kazhamiakin et al. 2004; Wan-Kadir and Loucopoulos 2004]) but translating relation algebra to functional specifications has not been done before.

This paper argues that Ampersand 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 2003] 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]). Practice, however, still requires 'the business' to translate requirements into rules, because available tools require rules to be specified in terms of actions, based on conditions and/or events. close one-to-one correspondence of a business requirement, being an expression in natural language, and a business rule, which formalizes the requirement into a precise and falsifiable statement.

This paper takes it one step further. Ampersand was designed to address end users in natural language only, reserving formal representations exclusively for requirements engineers. Thus, a requirements engineer can involve his audience deeper into the requirements elicitation process, because the conversation is conducted in natural language only. The use of natural language allows a requirements engineer excell in the informal process working with end users to consent over the requirements. At the same time the precise transcription to business rules represents everything that is needed to get a formal, platform independent functional specification in a style similar to [Spivey 1992; Jones 1986; Warmer and Kleppe 1998]. The direct correspondence between the two is needed to support the correctness of the requirement engineer's interpretation. As the tool generates useful design artifacts, it also helps to automate the design of information systems and business processes one step further.

This paper 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 are also Event-Condition-Action (ECA) rules of active databases, such as [Dayal et al. 1988; Loucopoulos et al. 1991; Paton and Diaz 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 relation 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 is built up as follows. Section 2 shows how business requirements are represented and how a functional specification is derived from them. Section 3 describes the method used to generate the functional specification. Section 4 describes results of this research and effects of this work outside academia. Section 5 discusses several issues raised by this approach, such as the process control mechanism that can be derived from business rules.

2. REQUIREMENTS

This section introduces and defines ampersand, taking requirements elicitation as a starting point. The purpose is to have the right interaction with stakeholders to define the right requirements and represent them completely and unambiguously. This section starts in section 2.1 with an example in which requirements are represented according to Ampersand. The next section defines a language to discuss business rules (section 2.2). Section 2.3 then presents the same example in the formal language. The specification generated by ADL is presented in 2.4.

2.1 Example: an order process

This example illustrates a multi-step process involving orders, deliveries, and invoices. In subsequent sections this example is formalized and the resulting functional specification is discussed.

The order process is defined by an understanding between providers and clients that is made explicit by the following ten statements:

(1) Ultimately all orders issued to a provider must be accepted by that very provider. The provider will be signalled of orders waiting to be accepted.

(2) A provider can accept only orders issued to himself.

(3) No delivery is made to a client without an order accepted by the provider.

(4) Ultimately, each order accepted must be delivered by the provider who has accepted that order. The provider will be signalled of orders waiting to be delivered.

(5) All deliveries are made to the client who ordered the delivery.

(6) A client accepts invoices for delivered orders only.

(7) There must be a delivery for every invoice sent.

(8) For each delivery made, the provider must send an invoice. The provider will be signalled when invoices can be sent.

(9) Accept payments only for invoices sent.

(10) Each invoice sent to a client must be paid. Both the client and the provider will be signalled for payments due.

Ampersand looks at these statements 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 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.

In order to support this order process, a process engine is required that signals parties for action. Signalling is achieved by deriving signals from rule violations. Rules that are to be maintained by people may be violated temporarily, allowing a person the time to take an action. For instance: if a new order comes in, a person must accept (or reject) that order and that may take some time. This means that rule 1 is violated, but only for the time it takes to accept the order. Computers are supposed to do everything instantly, so rules maintained by the computer may never be violated. This means that the computer must either react to the event in order to prevent a rule from being violated, or else it must block the transaction and produce an appropriate message. So whenever an event occurs (e.g. arrival of a new order), the process engine must check whether this event might cause violation of any requirement and subsequently signal the appropriate party. In this way, the information technology can maintain all business rules.

By way of example, a typical scenario illustrates how the process might proceed in practice:

(1) The client creates a new order. This generates a signal for the provider. The signal is a logical consequence of rule 1, which is (temporarily) violated by an order that has been issued by a client, but is not accepted (yet).

(2) The provider must accept the order, which generates a signal to deliver the order. The signal is a logical consequence of rule 4, which is (temporarily) violated by an order that has been accepted, but is not (yet) delivered.

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

(4) The provider sends an invoice, which creates a signal to the client that an invoice is due for payment. The signal is a logical consequence of rule 10, which is (temporarily) violated by an invoice that has not been paid (yet).

(5) The client pays the invoice, and no more signals are raised. This means that the order has been processed completely and the case is closed.

2.2 Language

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 [De Morgan 1883; Peirce 1883; 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 language of Ampersand are introduced by means of definitions taken from [Maddux 2006]. The presentation in this paper assumes that the reader is familiar with relation algebras.

Ampersand represents rules in an algebra $\langle \mathcal{R}, \cup, \bar{}, ;, ^\smile, \mathbb{I} \rangle$, in which $\mathcal{R}$ is a set of relations, $\cup$ and ; are binary operators of type $\mathcal{R} \times \mathcal{R} \to \mathcal{R}$, $\bar{}$ and $^\smile$ are unary operators of type $\mathcal{R} \to \mathcal{R}$ and $\mathbb{I}$ represents an identity relation, and satisfying the following axioms for all relations $r, s, t \in \mathcal{R}$:

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

To enrich the expressive power for users, Ampersand uses other operators as well. These are the binary operators $\cap$, $\dagger$, $\vdash$, and $\equiv$, all of which are of type $\mathcal{R} \times \mathcal{R} \to \mathcal{R}$. They are defined by:

$$r \cap s = \overline{\overline{s} \cup \overline{r}} \tag{11}$$
$$r \dagger s = \overline{\overline{s};\overline{r}} \tag{12}$$
$$r \vdash s = \overline{r} \cup s \tag{13}$$
$$r \equiv s = (r \vdash s) \cap (s \vdash r) \tag{14}$$

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'.

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 $b \in B$. 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 $r : A \rightarrow B$

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 I gives the names and types of all operators. Not all

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 I. Operators and their types

expressions that can be written are type correct. An expression must satisfy the condition mentioned in table I, otherwise it is undefined. A compiler for a language to be used for this purpose must therefore contain a type checker to ensure that all expressions satisfy the condtions mentioned in table I.

The complete list of notations with meanings attached to them is given in table II. If x, y, and z are used in an unbound manner, universal quantification (i.e. "for all x, y, and z") is assumed.

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 $x\ s\ y$.
intersection	$r \cap s$	$x(r \cap s)y$ means that $x\ r\ y$ and $x\ s\ y$.
composition	$r; s$	$x(r; s)y$ means that there is a z such that $x\ r\ z$ and $z\ s\ y$.
r. addition	$r \dagger s$	$x(r \dagger s)y$ means that for all z, $x\ r\ z$ or $z\ s\ y$ is true.
identity	$\mathbb{I}$	equals. Also: $a\ \mathbb{I}\ b$ means $a = b$.

Table II. Operators and their semantics

2.3 Formalization of the example

The following relations formalize the requirements of section 2.1:

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

Besides, two multiplicity requirements are made: for every order o there is a delivery d such that $o = \mathit{of}(d)$ and for every delivery d there is a invoice i such that $d = \mathit{delivery}(i)$.

In this particular example, all relations happen to be functions. In practical, more typical, situations nonfunctional relations occur as well. The meaning of each relation is given in the following table.

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

Formalization of each rule in relation algebra yields one business rule for each requirement:

(1) $addressedTo \vdash accepted$

(2) $accepted \vdash addressedTo$

(3) $provider \vdash of; accepted$

(4) $of; accepted \vdash provider$

(5) $of; from \vdash deliveredTo$

(6) $delivery^{\smile}; to \vdash deliveredTo$

(7) $sentBy = delivery; provider$

(8) $provider = delivery^{\smile}; from$

(9) $paidBy \vdash to$

(10) $to \vdash paidBy$

2.4 Functional Specification

The business rules from section 2.3 have been passed through the functional specification generator, yielding the results that are presented in this section. It consists of a data structure, a service catalogue, a function point analysis, and a formal specification for every service.

A data analysis of the formalized rules yields a class diagram, which is shown in figure 1. The algorithm to derive the class diagram from the relational specifica-

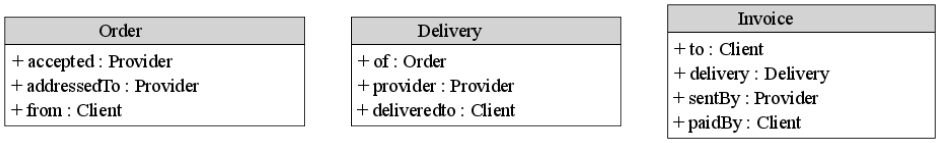

Fig. 1. Data structure

tion performs an ontological analysis on the rules and relations. It uses multiplicity properties to reason about existency dependencies between concepts, with class definitions as a result. Due to the simplicity of this particular example only classes

were generated. More complicated examples also produce associations, generalizations, and multiplicity properties.

The following 36 service definitions are generated from the rules:

```
(1) newOrder( In   a1 : Provider    ;
              In   a2 : Provider    ;
              In    f : Client      ;
              Out obj : OrderHandle )
(2) getOrder( In   x : OrderHandle ;
              Out a1 : Provider    ;
              Out a2 : Provider    ;
              Out  f : Client      )
(3) delOrder( In x : OrderHandle )
(4) updOrder( In  x : OrderHandle ;
              In a1 : Provider    ;
              In a2 : Provider    ;
              In  f : Client      )
(5) newDelivery( In    o : OrderHandle    ;
                 Out obj : DeliveryHandle )
(6) getDelivery( In  x : DeliveryHandle ;
                 Out o : OrderHandle    ;
                 Out p : Provider       ;
                 Out d : Client         )
(7) delDelivery( In x : DeliveryHandle )
(8) updDelivery( In x : DeliveryHandle ;
                 In o : OrderHandle    ;
                 In p : Provider       ;
                 In d : Client         )
(9) newInvoice( In    d : DeliveryHandle ;
                In    p : Client         ;
                Out obj : InvoiceHandle  )
(10) getInvoice( In  x : InvoiceHandle  ;
                 Out t : Client         ;
                 Out d : DeliveryHandle ;
                 Out s : Provider       ;
                 Out p : Client         )
(11) delInvoice( In x : InvoiceHandle )
(12) updInvoice( In x : InvoiceHandle  ;
                 In t : Client         ;
                 In d : DeliveryHandle ;
                 In s : Provider       ;
                 In p : Client         )
(13) newProvider( In  provider : Provider ;
                  Out      obj : Provider )
(14) delProvider( In x : Provider )
(15) newClient( In  client : Client ;
                Out    obj : Client )
(16) delClient( In x : Client )
```

```
(17) assocOrder( In o : OrderHandle ;
                 In p : Provider    )
(18) memberOrder( In s : OrderHandle ;
                  In t : Provider    )
(19) removeOrder( In o : OrderHandle ;
                  In p : Provider    )
(20) providersOrder( In        o : OrderHandle ;
                     Out handles : {Provider}  )
(21) ordersOrder( In        p : Provider      ;
                  Out handles : {OrderHandle} )
(22) assocDeliver( In p : Provider    ;
                   In o : OrderHandle )
(23) memberDeliver( In s : Provider    ;
                    In t : OrderHandle )
(24) removeDeliver( In p : Provider    ;
                    In o : OrderHandle )
(25) ordersDeliver( In        p : Provider      ;
                    Out handles : {OrderHandle} )
(26) providersDeliver( In        o : OrderHandle ;
                       Out handles : {Provider}  )
(27) assocPending( In p : Provider       ;
                   In d : DeliveryHandle )
(28) memberPending( In s : Provider       ;
                    In t : DeliveryHandle )
(29) removePending( In p : Provider       ;
                    In d : DeliveryHandle )
(30) deliveriesPending( In        p : Provider         ;
                        Out handles : {DeliveryHandle} )
(31) providersPending( In        d : DeliveryHandle ;
                       Out handles : {Provider}     )
(32) assocPayable( In i : InvoiceHandle ;
                   In c : Client        )
(33) memberPayable( In s : InvoiceHandle ;
                    In t : Client        )
(34) removePayable( In i : InvoiceHandle ;
                    In c : Client        )
(35) clientsPayable( In        i : InvoiceHandle ;
                     Out handles : {Client}      )
(36) invoicesPayable( In        c : Client          ;
                      Out handles : {InvoiceHandle} )
```

The results of a global function point analysis conformant to IFPUG principles are given in table III. This yields a total of 150 function points.

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

	function points
Order	22
Delivery	22
Invoice	22
Other Classes	16
Associations	68

Table III. Function Point Analysis

[Floyd 1967; Hoare 1969]. 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 paricular service.

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

When called, this service behaves as follows[1]:

{Pre: `True`}
`updOrder(x,a1,a2,f)`
{Post: `x.accepted` = `a1` and
`x.addressedTo` = `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) $\forall o : Order : o.accepted = o.addressedTo$

(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) $\forall i : Invoice : i.to = idelivery.of.from$

This terminates the formal specification of the service called *updOrder*.

The document that comprises the functional specification is generated in the form of LaTeX source code.

This section introduced a language to express rules, relations, and concepts. This was followed by an example, large enough to be nontrivial and small enough to fit in this paper. Its formalization was given in section 2.3. The functional specification, which has been generated from these rules, was presented in section 2.4. The following section describes how a functional specification is derived from business rules.

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

3. AMPERSAND

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, defining clauses, synthesizing a data model, synthesizing a service catalogue, synthesizing a function point analysis, and synthesizing a document. Ampersand is characterized by the use of a relation algebra for representing business rules. Another characteristic is that the method can be automated. A third characteristic is compliance of the services defined in the functional specification to the business requirements that are input of the method.

Ampersand was developed to help aligning business requirements with information technology. The name 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.

This section explains the way Ampersand derives functional specifications. Each step of Ampersand is discussed in a separate section.

3.1 Compiling

A compiler must translate a language in which a business analyst specifies business rules. That language may be any language that satisfies the axioms in section 2.2. 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 compiler was built in the functional programming language Haskell, using Swierstra's parser combinator package[2] [Swierstra et al. 1999; Swierstra and Duponcheel 1996]. 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.

Since ADL specifies rules without actions, it can be considered a purely declarative language.

The compiler yields a data structure in Haskell, which represents the parse tree.

3.2 Type checking

Type checking must be done to ensure that every expression has a unique type according to table I. Users themselves define which concepts are in the isa-relation (e.g. Affidavit isa Document). The isa-relation as understood in the concept algebra is the reflexive, transitive closure of the isa-pairs defined by users. Antisymmetry of the isa-relation must be verified by the type-ckecker.

A type checker has been built using the AG-preprocessor by Dijkstra and Swierstra [Dijkstra and Swierstra 2005]. The rules of table I are checked by attribute propagation, which basically makes use of the essential properties of Haskell's lazy evaluation to compute its result. 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.

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

Once a specification has passed the type checker, formula manipulators kick in to do the hard work. The remaining sections are written under the assumption that the set of rules is type correct.

3.3 Defining 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.

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 element of $\overline{r}$ 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[3].

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:

$$r_0 \cap r_1 \cap ... \cap r_n \;=\; e_0 \cap e_1 \cap ... \cap e_m \tag{24}$$

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

$$e_i = e_j \;\Rightarrow\; i = j \tag{25}$$

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 had a conjunctive normal form, each of its conjuncts is a clause in disjunctive normal form.

3.4 Synthesizing a data model

Every concept used in the business rules is interpreted as a UML class [Rumbaugh et al. 1999]. The data model is obtained by the observation that all functions can serve as an attribute of a class. Many laws exist in relation algebra that allow derivation of the multiplicity properties: univalence, totality, surjectivity,

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

and injectivity. Examples of such laws are:

$$univalent(r) \wedge univalent(s) \Rightarrow univalent(r;s) \quad (26)$$
$$surjective(s) \wedge surjective(r) \Rightarrow surjective(r;s) \quad (27)$$
$$total(s) \wedge total(r) \Rightarrow total(r;s) \quad (28)$$
$$injective(s) \wedge injective(r) \Rightarrow injective(r;s) \quad (29)$$
$$univalent(r) \vee univalent(s) \Rightarrow univalent(r \cap s) \quad (30)$$
$$injective(r) \vee injective(s) \Rightarrow injective(r \cap s) \quad (31)$$
$$total(r) \wedge total(s) \Rightarrow total(r \cup s) \quad (32)$$
$$univalent(r \cup s) \Rightarrow univalent(r) \wedge univalent(s) \quad (33)$$
$$injective(r \cup s) \Rightarrow injective(r) \wedge injective(s) \quad (34)$$
$$surjective(r \cup s) \Rightarrow surjective(r) \wedge surjective(s) \quad (35)$$
$$total(r \cup s) \Rightarrow total(r) \wedge total(s) \quad (36)$$

With these, and other laws, as many multiplicity properties as possible are derived. This is done with a closure algorithm, which terminates because the number of derivable multiplicity properties (4 times the number of relations) is finite.

Attributes are those relations r or $r^\smile$ that are both univalent and total (i.e. functions) This means that a surjective and injective relation will be used as attribute as well, because its converse ($r^\smile$) is a function. All remaining relations are made into associations.

Suppose that for a particular concept A, a number of functions $f_0 ... f_n$ satisfy

$$\mathbb{I}_A = \bigcap_i f_i; f_i^\smile \quad (37)$$

If this set is minimal, i.e. removing one of the functions would no longer satisfy equation 37, these functions constitute a *key* of concept A. Keys are determined to make use of the property that an instance of A is uniquely identified by key attributes. Notice that A may have multiple keys, when different sets of functions satisfy the requirement for being a key.

Generalizations are derived from the isa-relation which is derived from the specification.

A data modeling application has been built in Haskell. It produces not only class diagrams, but entity-relationship[Chen 1976] diagrams as well. The drawing applications `dot` and `neato` of the GraphViz package [Koutsofios and North 1993] have been used to create the graphics.

3.5 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, so different services can be given to different programmers for implementation. 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.

In practice, it is not necessary to build every service from the specification. If the applications that will use the service layer are known, one needs to build only those services that are actually used.

3.6 Synthesizing a function point analysis

Function points are a standardized way of determining the complexity of software, fostered by IFPUG (http://www.ifpug.org/). The function points determined by the analysis follows the IFPUG guidelines, and has been validated independently by experienced function point experts from two different IT organizations.

Although complexity estimates remain a subjective matter, function points are probably the best thing there is in objectively estimating the complexity of an information system.

4. RESULTS

To support the finding that a functional specification can be derived from business rules, three results are presented as evidence. First there is a generator that produces functional specifications, which proves that Ampersand can be automated. Secondly, Ampersand has been used in practice for various purposes, which proves that it is a relevant method. Thirdly, Ampersand has been taught successfully, which proves that ordinary professionals can learn what is needed to use the method. Each result is discussed in a separate section.

4.1 A generator for functional specifications

The generator mentioned in section 3, ADL, is a program that can be used on 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 LaTeX based .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 generator was developed in an evolutionary way, in a period that started in 1996 and still continues. The current version of the generator is being used in practice both in education (Open University of the Netherlands) and in industry (Ordina and TNO-ICT).

4.2 Results in practice

Various research projects and projects in business have supplied a significant amount of evidence that supports the power of business rules. These projects have been conducted in various locations. Research at the Open University of the Netherlands (OUNL) has focused on two things: developing the method, developing the course, and building a violation detector. Research at Ordina and TNO Informatie- en

Communicatie Technologie has been conducted to establish the usability of ADL-rules for representing genuine business rules in genuine enterprises. Collaboration with the university of Eindhoven has produced a first design of a rule base. Experiments conducted 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 [Dijkman et al. 2001], the predecessor of Ampersand, was conceived in 1996, resulting in a conceptual model called WorkPAD [Joosten 1996] and conceptual studies such as [Joosten and Purao 2002]. 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 [Joosten et.al. 2002], 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 [van Beek 2000], 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 metamodeling [Joosten and Purao 2002; Dijkman and Joosten 2002], 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 [Dommelen and Joosten 1999], 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 [Joosten et al. 2003]. 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) [IEEE: Architecture Working Group of the Software Engineering Committee 2000] in 2004. Several inconsistencies were found in the last (draft) RBAC standard [ANSI/INCITS 359: Information Technology 2004].

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[4]

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

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 [Barends 2003]. 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. 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 [Baardman and Joosten 2005]. The entire design of the process architecture [Ordina and Rabobank 2003] 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 [Fowler 1997] 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.

4.3 Teaching results

Writing business rules from business requirements has been taught twice: 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.

5. DISCUSSION

The Ampersand method described in section 3 and the evidence presented in section 4 support the claim that a functional specification can be derived from business requirements.

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

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

The following sections discuss each of these issues separately.

5.1 Rule engines

In a comparison of available methodologies for representing business rules, Herbst et.al. [Herbst et al. 1994] 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[5], Corticon[6], and Fair-Isaac[7] 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, 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 [Ross 2003] 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.

[5]ILOG. Business rule components. http://www.ilog.com/products/rules/, August 2001.
[6]Pedram Abrari and Mark Allen, Business rules user interface for development of adaptable enterprise applications, US Patent 7,020,869, March 28, 2006
[7]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.

5.2 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. 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 simplified design process. From their perspective, the design process is depicted in figure 2. The main task

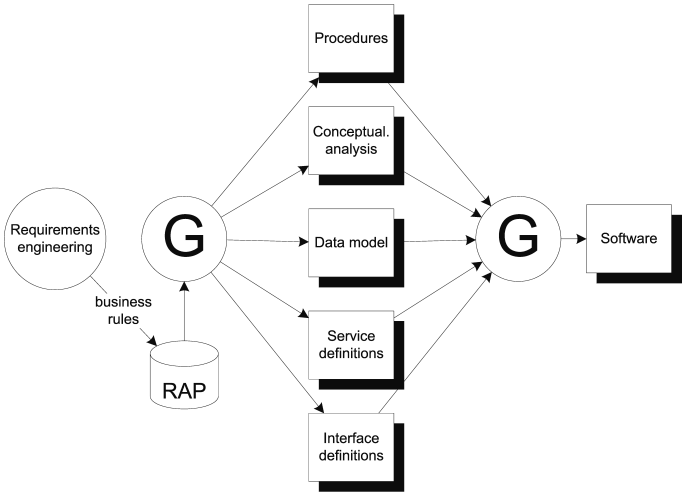

Fig. 2. Design process for rule based process management

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 functional specification generator 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 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 3. At the focus of attention is the dialogue between a problem owner and a requirements

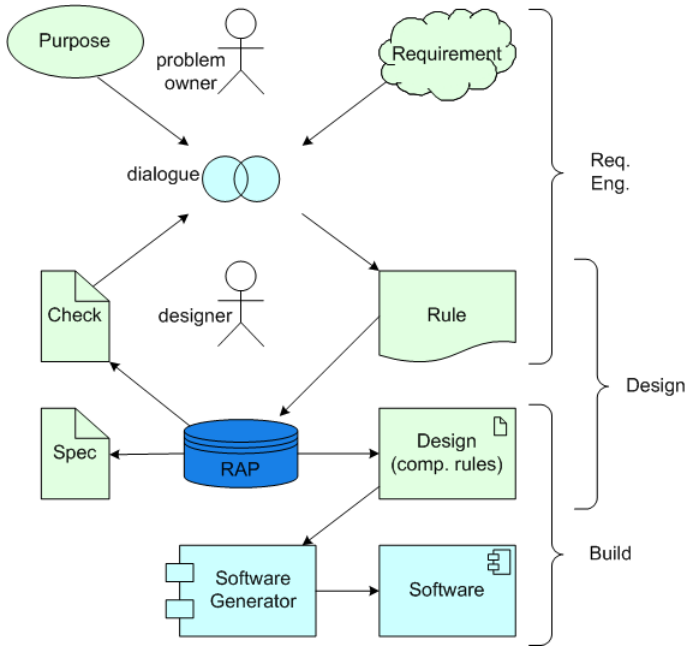

Fig. 3. Design process for rule based process management

engineer. The former decides which requirements he wants and the latter makes sure they are captured accurately and completely. The requirements engineer helps the problem owner to make requirements explicit. Ownership of the requirements remains in the business. The requirements engineer can tell with the help of his tools whether the requirements are sufficiently complete and concrete to make a buildable specification. The requirements engineer sticks to the principle of one-requirement-one-rule, enabling him to explain the correspondence of the specification to the business.

5.3 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) [Shewhart 1939]. Figure 4 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.

6. CONCLUSION

The main conclusion of this research is that business rules are sufficient to define business processes. This corroborates the claim of the Business Rules Group [Ross 2003] that rules are a first-class citizen of the requirements world (article 1.1 of the manifesto).

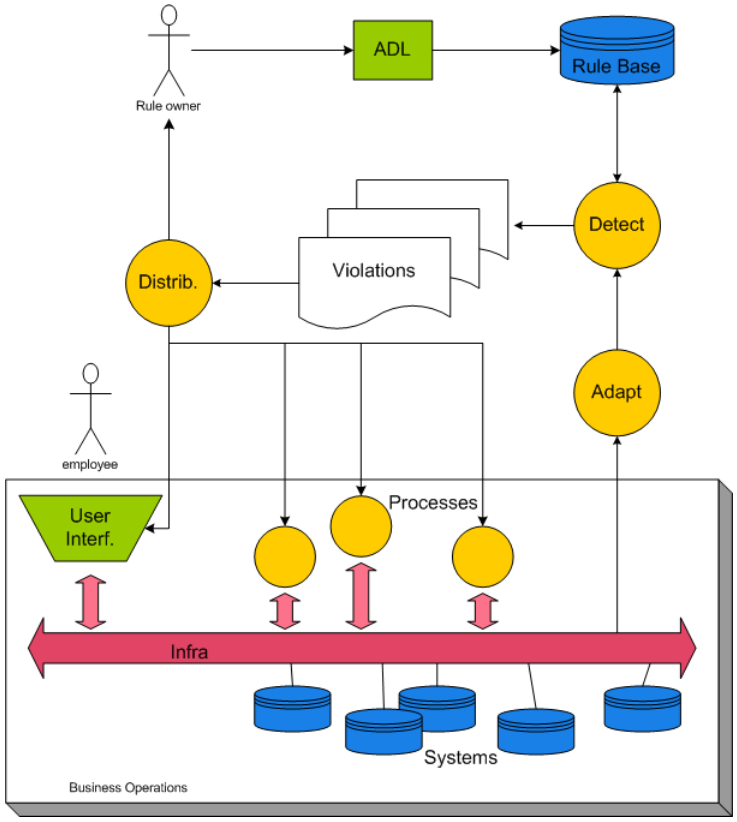

Fig. 4. Principle of rule based process management

This paper makes it possible to make functional specifications directly from business requirements. The most significant quality of such functional specifications is the mathematically guaranteed compliance with the requirements of the business. A useful side effect of this piece of design automation is that design takes less time. Estimates made at Ordina predict substantial savings in effort and turnaround time, but these predictions need more corroboration in practice before being published.

Further research is required in the following areas:

—**Design automation**
Further design automation is foreseen in process modeling. Section 5.3 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 5.3 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.

7. 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 they must remain anonymous 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. Also, many thanks are due to TNO-ICT in Groningen (the Netherlands), especially to Rieks Joosten, for being the first user of Ampersand. Finally, I also wish to thank Martin Zomer and Bert Paping for validating the function point count and Sebastian Joosten for his contributions to this research.

REFERENCES

ANSI/INCITS 359: Information Technology. 2004. *Role Based Access Control Document Number: ANSI/INCITS 359-2004.* InterNational Committee for Information Technology Standards (formerly NCITS).

Baardman, E. and Joosten, S. 2005. Procesgericht systeemontwerp. *Informatie 47,* 1 (Jan.), 50–55.

Barends, R. November 26, 2003. Activeren van de administratieve organisatie. Research report, Bank MeesPierson and Open University of the Netherlands.

Berners-Lee, T., Hendler, J., and Lassila, O. 2001. The semantic web. *Scientific American 284,* 5, 1–7.

Borgida, A., Greenspan, S. J., and Mylopoulos, J. 1985. Knowledge reperesentation as the basis for requirements specification (reprint). In *Wissensbasierte Systeme*, W. Brauer and B. Radig, Eds. Informatik-Fachberichte, vol. 112. Springer, 152–169.

Chen, P. 1976. The entity relationship model - toward a unified view of data. *ACM Transactions on Database Systems 1,* 1, 9–36.

Clocksin, W. F. and Mellish, C. S. 1981. *Programming in PROLOG*. Springer-Verlag, Berlin, New York.

Date, C. J. 2000. *What not how: the business rules approach to application development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Dayal, U., Buchmann, A. P., and McCarthy, D. R. 1988. 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*. Springer-Verlag New York, Inc., New York, NY, USA, 129–143.

De Morgan, A. 1883. On the syllogism: Iv, and on the logic of relations. *Transactions of the Cambridge Philosophical Society 10,* read in 1860, reprinted in 1966, 331–358.

Dijkman, R. M., Ferreira Pires, L., and Joosten, S. M. 2001. 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*, A. Evans, Ed. Lecture Notes in Informatics, vol. 7. FIZ, Karlsruhe.

Dijkman, R. M. and Joosten, S. M. 2002. An algorithm to derive use case diagrams from business process models. In *Proceedings IASTED-SEA 2002* (November 4-6).

Dijkstra, A. and Swierstra, S. D. 2005. Typing haskell with an attribute grammar. In *Advanced Functional Programming*, V. Vene and T. Uustalu, Eds. Lecture Notes in Computer Science, vol. 3622. Springer, Berlin, 1–72.

Dommelen, W. v. and Joosten, S. 1999. Vergelijkend onderzoek hulpmiddelen beheersing bedrijfsprocessen. Tech. rep., Anaxagoras and EDP Audit Pool.

Floyd, R. W. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, J. T. Schwartz, Ed. Proceedings of Symposia in Applied Mathematics, vol. 19. American Mathematical Society, Providence, Rhode Island, 19–32.

Fowler, M. 1997. *Analysis Patterns - Reusable Object Models*. Addison-Wesley, Menlo Park.

Green, P. and Rosemann, M. 2000. Integrated process modeling: an ontological evaluation. *Information Systems 25,* 2, 73–87.

Greenspan, S., Mylopoulos, J., and Borgida, A. 1994. On formal requirements modeling languages: Rml revisited. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, 135–147.

Gruber, T. R. 1993. A translation approach to portable ontology specifications. *Knowledge Acquisition 5,* 2, 199–220.

Guttag, J. V. and Horning, J. J. 1993. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA.

Herbst, H., Knolmayer, G., Myrach, T., and Schlesinger, M. 1994. 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*. Elsevier Science Inc., New York, NY, USA, 29–46.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12,* 10 (October), 576–580 and 583.

IEEE: Architecture Working Group of the Software Engineering Committee. 2000. *Standard 1471-2000: Recommended Practice for Architectural Description of Software Intensive Systems*. IEEE Standards Department.

ISO. 1987. 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. First edition.

Jones, C. B. 1986. *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

Joosten, R., Knobbe, J.-W., Lenoir, P., Schaafsma, H., and Kleinhuis, G. 2003. Specifications for the RGE security architecture. Tech. Rep. Deliverable D5.2 Project TSIT 1021, TNO Telecom and Philips Research, The Netherlands. Aug.

Joosten, S. 1996. Workpad - a conceptual framework for workflow process analysis and design. Unpublished.

Joosten, S. and Purao, S. R. 2002. A rigorous approach for mapping workflows to object-oriented is models. *Journal of Database Management 13*, 1–19.

Joosten et.al., S. 2002. *Praktijkboek voor Procesarchitecten*, 1st ed. Kon. van Gorcum, Assen.

Kazhamiakin, R., Pistore, M., and Roveri, M. 2004. A framework for integrating business processes and business requirements. In *EDOC '04: Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04)*. IEEE Computer Society, Washington, DC, USA, 9–20.

Khedri, R. and Bourguiba, I. 2004. Formal derivation of functional architectural design. In *2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM'04)*. IEEE Computer Society Press, Washington, DC, USA, 356–365.

Koutsofios, E. and North, S. C. 1993. *Drawing graphs with dot.* AT&T Bell Laboratories, Murray Hill, NJ.

Loucopoulos, P., McBrien, P., Schumaker, F., Theodoulidis, B., Kopanas, V., and Wangler, B. 1991. Integrating Database Technology, Rule-based Systems and Temporal Reasoning for Effective Software: the TEMPORA paradigm. *Journal of Information Systems 1*, 129–152.

Maddux, R. 2006. *Relation Algebras.* Studies in Logic and the Foundations of Mathematics, vol. 150. Elsevier Science.

Ordina and Rabobank. 2003. Procesarchitectuur van het servicecentrum financieren. Tech. rep., Ordina and Rabobank. Oct. presented at NK-architectuur 2004, www.cibit.nl.

Paton, N. and Diaz, O. 1999. Active Database Systems. *ACM Computing Surveys 1,* 31, 63–103.

Peirce, C. S. 1883. Note b: the logic of relatives. In *Studies in Logic by Members of the Johns Hopkins University (Boston)*, C. Peirce, Ed. Little, Brown & Co.

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.

Roscoe, A. W., Hoare, C. A. R., and Bird, R. 1997. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA.

Ross, R. G. 2003. *Principles of the Business Rules Approach*, 1 ed. Addison-Wesley, Reading, Massachussetts.

Rumbaugh, J., Jakobson, I., and Booch, G. 1999. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, Mass.

Scheer, A.-W. 1998. *Aris-Business Process Frameworks.* Springer-Verlag New York, Inc.

Schröder, F. W. K. E. first published in Leipzig, 1895. Algebra und logik der relative. In *Vorlesungen über die Algebra der Logik (exacte Logik)*. Chelsea.

Shewhart, W. A. 1988 (originally published in 1939). *Statistical Method From the Viewpoint of Quality Control.* Dover Publications, New York.

Spivey, J. 1992. *The Z Notation: A reference manual*, 2nd ed. International Series in Computer Science. Prentice Hall, New York.

Swierstra, S. D., Azero Alocer, P. R., and Saraiava, J. 1999. Designing and implementing combinator languages. In *Advanced Functional Programming, Third International School, AFP'98*, D. Swierstra, P. Henriques, and J. Oliveira, Eds. LNCS, vol. 1608. Springer-Verlag, 150–206.

Swierstra, S. D. and Duponcheel, L. 1996. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming*, J. Launchbury, E. Meijer, and T. Sheard, Eds. LNCS-Tutorial, vol. 1129. Springer-Verlag, 184–207.

van Beek, J. 2000. Generation workflow - how staffware workflow models can be generated from protos business models. M.S. thesis, University of Twente.

Wan-Kadir, W. M. N. and Loucopoulos, P. 2004. Relating evolving business rules to software design. *Journal of Systems Architecture 50,* 7, 367–382.

Warmer, J. and Kleppe, A. 1998. *The object constraint language: precise modeling with UML.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

WIDOM, J. 1996. The starburst active database rule system. *IEEE Transactions on Knowledge and Data Engineering 8,* 4, 583–595.

Received May 2007; revised Month Year; accepted Month Year.