AMMBR: A Method to Maintain Business Rules

Stef Joosten

May 2, 2007

Abstract

Ideally, an information infrastructure supports an organization by maintaining rules that are specific to the business. Such rules are known as *business rules* [**?**]. This paper proposes a method to define such an infrastructure. It allows organizations to build information systems that comply to their business rules in a provable way. The method is implemented in a tool that generates functional specifications. The tool is an instrument for designers (information architects) who define processes and information systems.

This paper is meant for readers who are interested in the mathematical background of deriving software from business rules. It uses relation algebra as a vehicle for processing business rules.

keywords: compliance, business rule, relation algebra, software generation, provable software, process control.

1 Introduction

In order to define an information system to support an organization, we look at organizations as systems that maintain rules. An ideal information infrastructure supports employees and other stakeholders to maintain the rules of the business. Yet rules must be concrete if they are to be used by computers. The method is therefore restricted to those rules of which violation can be established unambiguously and objectively. To *maintain* a rule means to prevent or correct all violations, either by human or automated actions. Temporary violation of a rule is only acceptable for rules that are maintained by humans, but the infrastructure must signal these violations and make them available to those people

The problem addressed in this paper is how to define software services to maintain all rules that apply in a given context. This paper proposes a method, AMMBR (pronounce: Amber), to derive software services from a set of rules and to assemble these services into an information system (i.e. a system of people and computers) that maintains these rules. AMMBR consists of the following steps:

represent rules Business rules are represented in relation algebra [**?**]. Requirement elicitation and reuse of existing patterns produce the desired set of rules. This is the only step in AMMBR that requires human effort. The following steps are all purely algorithmic, meaning that a computer can do the work.

determine clauses Once the set of rules is checked and found type consistent, it is reduced to clauses. A clause is a rule that cannot be broken in smaller pieces without losing meaning.

derive code fragments From the clauses, code fragments are derived for violation detection and for automated actions.

assemble services For every role, a set of services is defined that allows a user (who has that role) to change the state of the system while maintaining all rules.

In AMMBR, human involvement is required only in representing rules. So, there is a tool that generates the specification of a service layer directly from the business rules. This tool has been built in the form of a compiler, which produces functional specifications. It compiles from a language called ADL, which is relation algebra made suitable for compilation by a computer. The functional specification produced by this tool defines a service layer that maintains the rules defined in the ADL source code. The tool provides physical proof that an information system can be defined up to its service layer by means of rules only.

Using experimental, earlier versions of AMMBR and ADL, information systems have been defined for banks, insurance companies and governmental organizations.

The contribution of this paper is that business rules can define software, thus providing a piece of the bridge to span the abyss between information technology and business. From a computing perspective this is a step towards automating the design of information systems. From a business perspective this provides mathematical certainty that business rules are being maintained.

2 Preliminaries

Deze sectie is zwaar in aanbouw. Ik heb hier 'eventjes' een aantal regels bijeengezocht die verderop in bewijzen gebruikt worden. Nog aanvullen en tekst van maken (de hele sectie is nog niet voor review geschikt; doorlezen bij sectie 3).

This section defines the formalism in which rules are described. The first two laws (1 and 2) state that an identity relation may be removed if it stands left or right of a semicolon (the composition operator). Equation 3 states that the semicolon is an associative operator. This means that it does not matter how brackets are placed

in an expression with two or more semicolons. In the sequel brackets are omitted in situations with associative operators.

$$
\begin{aligned}
r;\mathbb{I} &= r && (1)\\
\mathbb{I};r &= r && (2)\\
(p;q);r &= p;(q;r) \;=\; p;q;r && (3)\\
(p \dagger q) \dagger r &= p \dagger (q \dagger r) \;=\; p \dagger q \dagger r && (4)
\end{aligned}
$$

The following laws show how the conversion operator behaves.

$$
\begin{aligned}
r^{\smile\smile} &= r && (5)\\
(r \cup q)^{\smile} &= r^{\smile} \cup q^{\smile} && (6)\\
(r \cap q)^{\smile} &= r^{\smile} \cap q^{\smile} && (7)\\
(r;s)^{\smile} &= s^{\smile};r^{\smile} && (8)\\
(r \dagger s)^{\smile} &= s^{\smile} \dagger r^{\smile} && (9)
\end{aligned}
$$

These laws are useful to show, for instance, that the conversion of an injective relation is univalent. Let r be injective.

$$
\begin{array}{ll}
 & r;r^{\smile} \vdash \mathbb{I}\\
\Leftrightarrow & \qquad\qquad \{r = r^{\smile\smile} \text{ by law 5}\}\\
 & r^{\smile\smile};r^{\smile} \vdash \mathbb{I}
\end{array}
$$

The last line in this derivation says that $r^{\smile}$ is univalent (definition 23). So the derivation proves that

$$\mathit{injective}(r) \;\Leftrightarrow\; \mathit{univalent}(r^{\smile}) \qquad (10)$$

Another useful law is the following:

$$p \vdash r \;=\; \overline{p} \cup r \qquad (11)$$

It shows how to remove a subset operator ($\vdash$) from any equation. Of similar use is a law to replace an equality by two subsets.

$$(p = r) \;=\; p \vdash r \cup r \vdash p \qquad (12)$$

The following law shows how the union operator can be removed from compositions of relations.

$$p;(q \cup r);s \;=\; p;q;s \cup p;r;s \qquad (13)$$

It would have been nice if the same distribution property holds for the intersect operator, $\cap$, but this is not always the case. This property holds only if p is univalent and s is injective:

$$p;(q \cap r);s \;=\; p;q;s \cap p;r;s \tag{14}$$

Since the identity relation $\mathbb{I}$ is both injective and univalent, this law can also be replaced by the following two laws:

$$\begin{array}{rcl} \mathit{injective}(s) & \Rightarrow & (q \cap r);s \;=\; q;s \cap r;s \qquad (15) \\ \mathit{univalent}(p) & \Rightarrow & p;(q \cap r) \;=\; p;q \cap p;r \qquad (16) \end{array}$$

Another way to formulate these laws in an unconditional way is:

$$p;(q \cap r);s \;\vdash\; p;q;s \cap p;r;s \tag{17}$$

Two other useful laws move relations across the $\vdash$ symbol. However, this law is conditional too. If f is a function (that is: a univalent and total relation) then

$$\begin{array}{rcl} r \vdash s;f^{\smile} & \Leftrightarrow & r;f \vdash s \qquad (18) \\ r \vdash f;s & \Leftrightarrow & f^{\smile};r \vdash s \qquad (19) \end{array}$$

The following equivalences are attributed to the famous mathematician De Morgan:

$$\begin{array}{rcl} \overline{r} \cup \overline{s} & = & \overline{r \cap s} \\ \overline{r} \cap \overline{s} & = & \overline{r \cup s} \end{array} \tag{20}$$

De Morgan's equivalences turn out to be among the most frequently used laws, because the allow you to 'move around' the complement operator. The operators ; and † satisfy similar laws:

$$\begin{array}{rcl} \overline{r};\overline{s} & = & \overline{r \dagger s} \\ \overline{r} \dagger \overline{s} & = & \overline{r;s} \end{array} \tag{21}$$

Equations 20 and 21 are known as De Morgan's laws.

Less well known, but equally important are the following equivalences, which De Morgan called "Theorem K".

$$\begin{array}{rcl} & p;q \;\vdash\; r & \\ = & & \\ & p^{\smile};\overline{r} \;\vdash\; \overline{q} & \qquad (22) \\ = & & \\ & \overline{r};q^{\smile} \;\vdash\; \overline{p} & \end{array}$$

Well known properties of relations are the so called 'multiplicity properties', or *multiplicities*: univalent, total, surjective, and injective.

- **univalent**
  A relation $r : A \times B$ is *univalent* if each element of A corresponds to at most one element of B. This property is defined by:

  $$r^{\smile}; r \ \vdash \ \mathbb{I}_B \tag{23}$$

  The definition says that $a \ r \ b$ and $a \ r \ b'$ imply $b = b'$ for all a, b, and b'. To denote that relation r is univalent, we write $univalent(r)$.

- **total**
  A relation $r : A \times B$ is *total* if each element of A corresponds to at least one element of B. This property is defined by:

  $$\mathbb{I}_A \ \vdash \ r; r^{\smile} \tag{24}$$

  The definition says that for all $a \in A$ there exists $b \in B$ such that $a \ r \ b$. To denote that relation r is univalent, we write $univalent(r)$.

- **function**
  A relation is a *function* if it is both univalent and total. That is: a relation $r : A \times B$ is a function if every element of A corresponds to precisely one element of B. If r is univalent but not total then it is called a *partial function*.

- **injective**
  A relation $r : A \times B$ is *injective* if each element of B corresponds to at most one element of A. This property is defined by:

  $$r; r^{\smile} \ \vdash \ \mathbb{I}_A \tag{25}$$

  The definition says that $b \ r \ a$ and $b \ r \ a'$ imply $a = a'$ for all a, a', and b. Notice that the property injective is defined conversely to the property univalent. To denote that relation r is injective, we write $injective(r)$.

- **surjective**
  A relation $r : A \times B$ is *surjective* if each element of B corresponds to at least one element of A. This property is defined by:

  $$\mathbb{I}_B \ \vdash \ r^{\smile}; r \tag{26}$$

  The definition says that for all $b \in B$ there exists $a \in A$ such that $a \ r \ b$.

3 Clauses

This section explains:

1. how to derive clauses from a set of rules, by a normalization procedure.

2. how to transform clauses into left identity clauses and right identity clauses, if possible.

3. how to derive multiplicity properties from clauses thus obtained.

A *rule* is an expression e that must be maintained. To *maintain* rule r means to ensure that $V \vdash r$ at all times, where V is the complete relation (i.e. everything is an element of V). In order to discuss maintaining rules, we must introduce some vocabulary. As long as $V \vdash r$, we say that r is *satisfied*, or simply that it *holds*. If r is not V, we say that *invariance is broken*, or rule r is *not satisfied*, or it *does not hold*. Since $r \cup \overline{r} = V$, 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 $V \vdash r$.

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 (conjuncts) r_i are separated by the conjunction operator $\cap$. Maintaining all rules (every r_i) in context C means to ensure that $R_C = V$ at all times[1].

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

$$r_0 \cap r_1 \cap ... \cap r_n \;=\; e_0 \cap e_1 \cap ... \cap e_m \tag{27}$$

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

$$e_i = e_j \;\Rightarrow\; i = j \tag{28}$$

So each e_i thus defined is a clause. Since this definition calls for 'as many unique expressions as possible', no clause e can be written as a conjunction. (If an a and b would exist such that $e = a \cap b$, then we could have had one clause more. This contradicts the definition of clause.)

3.1 Normalization

In order to transform a set of rules into clauses, a normalization procedure is performed. This procedure transforms a set of rules by meaning preserving transformations. Consequently, the result has the appropriate form and still has the exact same meaning. All normalization rules in this section have the form $a \rightarrow b$, and they all satisfy $a = b$ (which means they are meaning preserving).

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

1. Move all $^\smile$ operators inward, using the following transformations:

$$\begin{array}{rcl}(r \cup s)^\smile & \to & r^\smile \cup s^\smile \\ (r \cap s)^\smile & \to & r^\smile \cap s^\smile \\ (r; s)^\smile & \to & s^\smile; r^\smile \\ (r \dagger s)^\smile & \to & s^\smile \dagger r^\smile \\ \overline{r}^\smile & \to & \overline{r^\smile}\end{array}$$

2. Move all ; operators inwards, as much as possible, using the following transformations:

$$\begin{array}{rcll}q; (r \cup s) & \to & q; r \ \cup \ q; s & \\ (q \cup r); s & \to & q; s \ \cup \ r; s & \\ q; (r \cap s) & \to & q; r \ \cap \ q; s & \text{if } q \text{ is a function} \\ (q \cap r); s & \to & q; s \ \cap \ r; s & \text{if } s^\smile \text{ is a function}\end{array}$$

3. Distribute $\cup$ over $\cap$, using the following transformations:

$$\begin{array}{rcl}q \cup (r \cap s) & \to & (q \cup r) \ \cap \ (q \cup s) \\ (q \cap r) \cup s & \to & (q \cup s) \ \cap \ (r \cup s)\end{array}$$

4. Move all complement operators inward across $\cap$ and $\cup$, using the following transformations:

$$\begin{array}{rcl}\overline{r \cup s} & \to & \overline{r} \ \cap \ \overline{s} \\ \overline{r \cap s} & \to & \overline{r} \ \cup \ \overline{s}\end{array}$$

5. Remove identical clauses and terms:

$$\begin{array}{rcl}r \cup r & \to & r \\ r \cap r & \to & r\end{array}$$

6. Remove opposite clauses and terms:

$$\begin{array}{rcl}r \cup \overline{r} & \to & V \\ r \cap \overline{r} & \to & \overline{V}\end{array}$$

7. Absorb redundant terms:

$$\begin{array}{rcl}r \cap (\overline{r} \cup s) & \to & r \cap s \\ r \cup (\overline{r} \cap s) & \to & r \cup s \\ r \cap (r \cup s) & \to & r \\ r \cup (r \cap s) & \to & r\end{array}$$

8. Remove redundant clauses and terms:

$$\begin{array}{rcl}r \cup V & \to & V \\ r \cup \overline{V} & \to & r \\ r \cap V & \to & r \\ r \cap \overline{V} & \to & \overline{V}\end{array}$$

9. Remove redundant brackets:

$$\begin{array}{rcl} q \cap (r \cap s) & \rightarrow & q \cap r \cap s \\ (q \cap r) \cap s & \rightarrow & q \cap r \cap s \\ q \cup (r \cup s) & \rightarrow & q \cup r \cup s \\ (q \cup r) \cup s & \rightarrow & q \cup r \cup s \\ q; (r; s) & \rightarrow & q; r; s \\ (q; r); s & \rightarrow & q; r; s \\ q \dagger (r \dagger s) & \rightarrow & q \dagger r \dagger s \\ (q \dagger r) \dagger s & \rightarrow & q \dagger r \dagger s \end{array}$$

The normalization procedure continues to apply these transformations until none of them are applicable. If the normalization procedure reduces one or more clauses to $\overline{V}$, there is an inconsistency. Note that this renders the entire set of rules inconsistent (i.e. equivalent to $\overline{V}$). Any clause that is reduced to V is a tautology and is consequently removed by the normalization procedure.

Without proof, we claim that:

- The normalization procedure terminates.
- The result is a conjunction of clauses.
- Each clause can be written as a disjunction of (zero or more) positive and negative expressions:

$$\overline{a_1} \cup \overline{a_2} \cup ... \cup \overline{a_k} \ \cup \ c_1 \cup c_2 \cup ... \cup c_l \tag{29}$$

Expressions that are separated by disjunctions are called *disjuncts* or *terms*. A term is called *negative* if it is an expression enclosed by a complement operator. A *positive term* is any term that is not negative. Each expression a_i in equation 29 is called an *antecedent* of the clause and each expression c_j is called a *consequent* of the clause.

3.2 Left- and right identity clauses

The clauses that are produced by the normalization procedure can be distinguished further into three categories:

1. A clause with only one negative term, which is equal to $\mathbb{I}$ is called a *left identity clause*.
2. A clause with only one positive term, which is equal to $\mathbb{I}$ is called a *right identity clause*.

3. All other clauses.

Let $\overline{a} \cup c$ be a clause and let a be of the form

$$f_1; f_2; ...; f_n; g_1; g_2; ...; g_m$$

in which every g_i is a function and every $f_j{}^{\smile}$ is a function too. In that case, $\overline{a} \cup c$ is a left identity clause, which we write as:

$$\overline{\mathbb{I}} \cup (f_1; f_2; ...; f_n)^{\smile}; c; (g_1; g_2; ...; g_m)^{\smile}$$

Let $\overline{a} \cup c$ be a clause and let c be of the form

$$t \;=\; f_1; f_2; ...; f_n; g_1; g_2; ...; g_m$$

in which every f_i is a function and every $g_j{}^{\smile}$ is a function too. In that case, $\overline{a} \cup c$ is a right identity clause, which we write as:

$$\overline{(f_1; f_2; ...; f_n)^{\smile}; c; (g_1; g_2; ...; g_m)^{\smile}} \;\cup\; \mathbb{I}$$

Note that c in the left identity clause and c in the right identity clause may consist of several terms. Therefore one clause $\overline{a} \cup c$ may yield multiple (or no) right identity clauses and multiple (or no) left identity clauses.

3.3 Multiplicity derivation

Multiplicity information is important for deriving data structures. Some multiplicities are given (by the modeler) and others can be derived from those already given. This section shows how multiplicities can be derived from given ones.

For deriving multiplicities, we use the following laws.

$$\begin{aligned}
\mathit{injective}(r) &= \mathit{univalent}(r^{\smile}) && (30)\\
\mathit{surjective}(r) &= \mathit{total}(r^{\smile}) && (31)\\
\mathit{univalent}(r) \cap \mathit{univalent}(s) &\vdash \mathit{univalent}(r; s) && (32)\\
\mathit{surjective}(s) \cap \mathit{surjective}(r) &\vdash \mathit{surjective}(r; s) && (33)\\
\mathit{total}(s) \cap \mathit{total}(r) &\vdash \mathit{total}(r; s) && (34)\\
\mathit{injective}(s) \cap \mathit{injective}(r) &\vdash \mathit{injective}(r; s) && (35)\\
\mathit{univalent}(r) \cup \mathit{univalent}(s) &\vdash \mathit{univalent}(r \cap s) && (36)\\
\mathit{injective}(r) \cup \mathit{injective}(s) &\vdash \mathit{injective}(r \cap s) && (37)\\
\mathit{total}(r) \cap \mathit{total}(s) &\vdash \mathit{total}(r \cup s) && (38)\\
\mathit{univalent}(r \cup s) &\vdash \mathit{univalent}(r) \cap \mathit{univalent}(s) && (39)\\
\mathit{injective}(r \cup s) &\vdash \mathit{injective}(r) \cap \mathit{injective}(s) && (40)\\
\mathit{surjective}(r \cup s) &\vdash \mathit{surjective}(r) \cap \mathit{surjective}(s) && (41)\\
\mathit{total}(r \cup s) &\vdash \mathit{total}(r) \cap \mathit{total}(s) && (42)
\end{aligned}$$

4 Code Fragments

In this section we derive code for computers, which is built around two basic operations: insert and delete. The insert operation can be represented in terms of pre- and postconditions[2].

$$\begin{array}{l} \quad \{\text{pre: } \mathtt{P} = p\} \\ \texttt{INSERT } \Delta \texttt{ IN P} \\ \quad \{\text{post: } \mathtt{P} = p \cup \Delta\} \end{array}$$

The same can be expressed mathematically by the function $insert_{\mathtt{P}}$:

$$insert_{\mathtt{P}}(\Delta, r) \;=\; r_{[\mathtt{P}:=\mathtt{P}\cup\Delta]} \tag{43}$$

This function inserts a set of pairs, Δ, into its argument.

The delete operation is introduced similarly:

$$\begin{array}{l} \quad \{\text{pre: } \mathtt{P} = p\} \\ \texttt{DELETE } \Delta \texttt{ FROM P} \\ \quad \{\text{post: } \mathtt{P} = p \cap \overline{\Delta}\} \end{array}$$

This too is expressed mathematically by the function $delete_{\mathtt{P}}$:

$$delete_{\mathtt{P}}(\Delta, r) \;=\; r_{[\mathtt{P}:=\mathtt{P}\cap\overline{\Delta}]} \tag{44}$$

Let one example introduce the idea of breaking and restoring invariance. Suppose the rule $\mathtt{P} \vdash \mathtt{S} \cup \mathtt{T}$ is to be maintained by a computer, and P, S, and T are variables (memory locations) in that computer, which represent relations. We assume that before executing any code, we have $\mathtt{P} = p$, $\mathtt{S} = s$, and $\mathtt{T} = t$. The following example shows that inserting a set of pairs, Δ, in P may violate the rule. Inserting the same Δ in S restores invariance. In terms of code, we write:

$$\begin{array}{l} \quad \{\text{Assume } \mathtt{P} = p,\ \mathtt{S} = s,\ \mathtt{T} = t \text{ and } \mathtt{P} \vdash \mathtt{S} \cup \mathtt{T}\} \\ \texttt{INSERT } \Delta \texttt{ IN P} \\ \quad \{\mathtt{P} = p \cup \Delta,\ \mathtt{S} = s, \text{ and } \mathtt{T} = t, \text{ and invariance is broken: } \mathtt{P} \nvdash \mathtt{S} \cup \mathtt{T}\} \\ \texttt{INSERT } \Delta \texttt{ IN S} \\ \quad \{\text{postcondition: } \mathtt{P} = p \cup \Delta,\ \mathtt{S} = s \cup \Delta,\ \mathtt{T} = t, \text{ and } \mathtt{P} \vdash \mathtt{S} \cup \mathtt{T}\} \end{array}$$

To discuss this example, we start at the top. We may assume $\mathtt{P} \vdash \mathtt{S} \cup \mathtt{T}$ that holds. Rephrased in disjunctive form, this is equivalent to $\overline{\mathtt{P}} \cup \mathtt{S} \cup \mathtt{T}$. The first line of code inserts a set of pairs, Δ, into relation P, which means that after the first line, we have $\mathtt{P} = p \cup \Delta$. Consequently, $\overline{\mathtt{P}} = \overline{p} \cap \overline{\Delta}$, which means that some elements of Δ may

[2]this idea was introduced by Floyd and Hoare. It is well known for making assertions about computer programs and reasoning about them.

be missing from the rule, $\overline{\mathtt{P}} \cup \mathtt{S} \cup \mathtt{T}$. To neutralize that result, the missing elements must be added to either S or T. That is what happens in the second line.

To prove that rule $\mathtt{P} \vdash \mathtt{S} \cup \mathtt{T}$ is maintained, we must show that its value afterwards is equal to V, under the condition that its initial value equals V. The initial value is $\overline{p} \cup s \cup t$, which we may assume. The final value is $(p \cup \Delta) \vdash (s \cup \Delta) \cup t$, which must be proven equal to V:

$$
\begin{array}{ll}
 & (p \cup \Delta) \vdash (s \cup \Delta) \cup t \\
= & \qquad\qquad \{\text{by law 11}\} \\
 & \overline{p \cup \Delta} \ \cup \ s \ \cup \ \Delta \ \cup \ t \\
= & \qquad\qquad \{\text{De Morgan's law (20)}\} \\
 & (\overline{p} \cap \overline{\Delta}) \ \cup \ s \ \cup \ \Delta \ \cup \ t \\
= & \qquad\qquad \{\text{Distribute } \cup \text{ over } \cap\} \\
 & (\overline{p} \cup s \cup \Delta \cup t) \ \cap \ (\overline{\Delta} \cup s \cup \Delta \cup t) \\
= & \qquad\qquad \{\overline{\Delta} \cup \Delta = V\} \\
 & \overline{p} \cup s \cup \Delta \cup t \\
= & \qquad\qquad \{V \ = \ \overline{p} \cup s \cup t\} \\
 & V
\end{array}
$$

From this derivation we may conclude that invariance of $\mathtt{P} \vdash \mathtt{S} \cup \mathtt{T}$ is restored.

After this example, let us analyze how to restore invariants in all possible cases. Suppose we have clause r, and an operation t. We define: operation t *respects invariance of* r means

$$r \ \vdash \ t(r) \tag{45}$$

This definition can be understood by saying that the effect of t, which is $t(r)$, must hold under the condition that r holds.

In a database both people and computer programs can take any action at an arbitrary moment. We call that an event. An *event* is an insertion or deletion of items (i.e. pairs) in a relation. If that relation is used in a clause, that clause may be affected by the event. Let $a(r)$ be the result of event a on clause r. That result might not be equal to V, i.e. there may be violations $\overline{a(r)}$ of the clause. So, in order to restore invariance, a reaction t is required that satisfies $t(a(r)) = V$. So if we propose a response t to an event a, we must prove for all possible r:

$$r \ \vdash \ t(a(r)) \tag{46}$$

A *code fragment* is a computer program that consists of an action t that is taken in response to an event a. In order to derive code fragments, we first define the possible events. An event occurs as a result of changing the contents of relations. So the relevant ones are those relations m that affect (i.e. are used in) r. Let

$mors(r)$ be the set of relations that affect r. The set of possible events consists of insert- and delete events on relations $mors(r)$. Some of these events, however, do not violate clause e. If absence of violations can be proven for every Δ, there is no need to consider the event. Therefore, rule r is affected by the following set of possible events

$$\begin{aligned} events(r) \;=\; & \{insert_m |\; m \in mors(r) \text{ and } \forall \Delta : r \;\not\vdash\; insert_m(\Delta, r)\} \\ & \cup \\ & \{delete_m |\; m \in mors(r) \text{ and } \forall \Delta : r \;\not\vdash\; delete_m(\Delta, r)\} \end{aligned}$$

Any reaction on one of these actions must restore invariance of e. This allows us to specify all code fragments f_e by the following equation:

$$\begin{aligned} f_e \;\in\; & \{\langle e,\; insert_m,\; e{:=}e \cup \overline{insert_m(\Delta, e)}\rangle | m \in mors(e) \text{ and } \overline{insert_m(\Delta, e)} \neq \emptyset\} \\ & \cup \\ & \{\langle e,\; delete_m,\; e{:=}e \cup \overline{delete_m(\Delta, e)}\rangle | m \in mors(e) \text{ and } \overline{delete_m(\Delta, e)} \neq \emptyset\} \end{aligned} \tag{47}$$

Note that there are choices involved when inserting violations $\overline{a(e)}$ in e. Any choice that satisfies equation 47 is acceptable, because the resulting code fragment will satisfy equation 46. Therefore, the designer of the code fragments is free to make his or her own choices.

By way of illustration, let us elaborate one possible choice. Suppose we restrict insertion of violations to just one of the terms of e. Let $terms(e)$ be the set of terms in e, then the possible reactions are

$$\{t{:=}t \cup \overline{a(e)} | t \in terms(e)\}$$

If actions are restricted to individual terms as well, we can enumerate possible fragments. Suppose for example, that we have a clause e

$$e \;=\; \overline{p} \cup \overline{q} \cup s \cup t$$

If compensation is restricted to terms, table 1 enumerates all possibilities.

Each of the code fragments from table 1 will maintain invariance of e. We shall prove this for one code fragment only (we have arbitrarily picked the third code

event a	action r
$insert_p(\Delta, r)$	$q := q \cap \overline{a(r)}$
$insert_p(\Delta, r)$	$s := s \cup a(r)$
$insert_p(\Delta, r)$	$t := t \cup a(r)$
$insert_q(\Delta, r)$	$p := p \cap \overline{a(r)}$
$insert_q(\Delta, r)$	$s := s \cup a(r)$
$insert_q(\Delta, r)$	$t := t \cup a(r)$
$delete_s(\Delta, r)$	$p := p \cap \overline{a(r)}$
$delete_s(\Delta, r)$	$q := q \cap \overline{a(r)}$
$delete_s(\Delta, r)$	$t := t \cup a(r)$
$delete_t(\Delta, r)$	$p := p \cap \overline{a(r)}$
$delete_t(\Delta, r)$	$q := q \cap \overline{a(r)}$
$delete_t(\Delta, r)$	$s := s \cup a(r)$

Table 1: Code fragments for $r = \overline{p} \cup \overline{q} \cup s \cup t$

fragment): This gives

$$
\begin{array}{ll}
 & \overline{(p \cup \Delta)} \cup \overline{q} \cup s \cup (t \cup \overline{a(e)}) \\
= & \quad \{a(e) = \overline{(p \cup \Delta)} \cup \overline{q} \cup s \cup t\} \\
 & \overline{\overline{p \cup \Delta} \cup \overline{q} \cup s \cup t \cup \overline{(p \cup \Delta)} \cup \overline{q} \cup s \cup t} \\
= & \quad \{\text{De Morgan (20)}\} \\
 & \overline{p \cup \Delta} \cup \overline{q} \cup s \cup t \cup (\overline{\overline{(p \cup \Delta)} \cap \overline{\overline{q}} \cap \overline{s} \cap \overline{t}}) \\
= & \quad \{\text{remove double complement}\} \\
 & \overline{p \cup \Delta} \cup \overline{q} \cup s \cup t \cup ((p \cup \Delta) \cap q \cap \overline{s} \cap \overline{t}) \\
= & \quad \{\text{Distribute } \cap \text{ over } \cup\} \\
 & (\overline{p \cup \Delta} \cup \overline{q} \cup s \cup t \cup (p \cup \Delta)) \cap \\
 & (\overline{p \cup \Delta} \cup \overline{q} \cup s \cup t \cup q) \cap \\
 & (\overline{p \cup \Delta} \cup \overline{q} \cup s \cup t \cup \overline{s}) \cap \\
 & (\overline{p \cup \Delta} \cup \overline{q} \cup s \cup t \cup \overline{t}) \\
= & \quad \{\overline{p \cup \Delta} \cup p \cup \Delta = V \text{ and } \overline{q} \cup q = V \text{ and } s \cup \overline{s} = V \text{ and } t \cup \overline{t} = V\} \\
 & V \cap V \cap V \cap V \\
= & \\
 & V
\end{array}
$$

This derivation proves that the action of inserting an arbitrary Δ in p can be reacted upon by inserting the violations $\overline{a(e)}$ into t. Although this particular table is specific for rules of the form $\overline{p} \cup \overline{q} \cup s \cup t$, it can be generalized to clauses with any number (0 or more) negative terms and 1 or more positive terms.

A special situation occurs if a clause contains no negative terms, as in:

$$s \cup t$$

To keep this clause invariant means to keep $V \vdash s \cup t$ equal to V at all times. In a database, a representation of V might change when Δ contains new instances, i.e. instances that are not (yet) in V. Then, either s or t must be extended with $\Delta; V \cup V; \Delta$.

action a	reaction r
$\mathbb{I}_A := \mathbb{I}_A \cup \Delta$	$s := s \cup \Delta; V$
$\mathbb{I}_A := \mathbb{I}_A \cup \Delta$	$t := t \cup \Delta; V$
$\mathbb{I}_B := \mathbb{I}_B \cup \Delta$	$s := s \cup V; \Delta$
$\mathbb{I}_B := \mathbb{I}_B \cup \Delta$	$t := t \cup V; \Delta$

Table 2: Code fragments for $s \cup t$

$$
\begin{aligned}
Ins_\Delta(\overline{x}) &= Del_\Delta(x) && (48)\\
Ins_\Delta(x \cup y) &= Ins_\Delta(x) \mid Ins_\Delta(y) && (49)\\
Ins_\Delta(x \cap y) &= Ins_\Delta(x) + Ins_\Delta(y) && (50)\\
Ins_\Delta(x; y) &= \text{let } n \in target(x) \text{ (arbitrarily chosen) in} \\
&\quad Ins_{(\Delta \cap \overline{x;y});V;\mathbf{1}_n}(x) + Ins_{\mathbf{1}_n;V;(\Delta \cap \overline{x;y})}(y) && (51)\\
Ins_\Delta(m) &= m := m \cup \Delta && (52)\\
Del_\Delta(\overline{x}) &= Ins_\Delta(x) \mid Del_{(\Delta;V)}(I_{[source(x)]}) \mid Del_{(V;\Delta)}(I_{[target(x)]}) && (53)\\
Del_\Delta(x \cup y) &= Del_\Delta(x) + Del_\Delta(y) && (54)\\
Del_\Delta(x \cap y) &= Del_\Delta(x) \mid Del_\Delta(y) && (55)\\
Del_\Delta(x; y) &= Del_{\Delta;V \,\cap\, V;y^\smile}(x) \;\mid \\
&\quad Del_{x^\smile;V \,\cap\, V;\Delta}(y) && (56)\\
Del_\Delta(m) &= m := m \cap \overline{\Delta} && (57)
\end{aligned}
$$

5 Services