Software Development in Relation Algebra with Ampersand

Stef Joosten[1,2](✉)

[1] Open Universiteit Nederland, Postbus 2960, 6401 DL Heerlen, The Netherlands
stef.joosten@ou.nl
[2] Ordina NV, Nieuwegein, The Netherlands

Abstract. Relation Algebra can be used as a programming language for building information systems. This paper presents a case study to demonstrate this principle. We have developed a database-application for legal reasoning as a case study, of which a small part is discussed in this paper to illustrate the mechanisms of programming in Relation Algebra. Beside being declarative, relation algebra comes with attractive promises for developing big software. The compiler that was used for this case study, Ampersand, is the result of an open source project. Ampersand has been tried and tested in practice and is available as free open source software.

Keywords: Relation algebra · Software development · Legal reasoning · Information systems design · Ampersand · MirrorMe · Big software

1 Introduction

This paper investigates how relation algebra can be used as a programming language for information systems. A compiler, Ampersand [19], is used to compile concepts, relations and rules into a working database-application. Ampersand is a syntactically sugared version of heterogeneous relation algebra [23]. We present a case study to demonstrate programming in relation algebra and its impact on the software development process. The case study takes the reader by the hand in the thinking process of a software developer who programs with relation algebra.

The use of relation algebra as a programming language is not new. It stands in the tradition of relation algebra [17], logic programming [16], database application programming [5], and formal specification of software. Ampersand uses these ideas to compile relation algebra to working software, an idea which is also found in RelView [?] by Berghammer (Univ. of Kiel). Some differences with earlier programming languages are discussed in Sect. 6, after presenting the case study.

The user may regard Ampersand as a programming language, especially suited for designing the back-end of information systems. The axioms of Tarski can be used to manipulate expressions in a way that preserves meaning [32]. This makes Ampersand a declarative language.

Our case study shows an argument assistant for legal professionals, which was built as innovation project at Ordina. The purpose of this argument assistant is to support legal professionals in constructing a legal brief[1]. The challenge is to create a program that consists of relation algebra as much as possible. In doing so, we hope to learn more about software development in relation algebra.

Section 2 introduces Ampersand and its computational semantics. Section 3 introduces a theory of legal reasoning, which was developed for argument assistance. Section 4 discusses the programming mechanism in the application, and Sect. 5 visualizes that mechanism. Section 6 reflects on software development in Ampersand. It also provides an overview of the use of Ampersand in practice and an outlook to its further development.

2 Ampersand

In this section we explain the basics of Ampersand. The reader is expected to have sufficient background in relation algebra, in order to understand the remainder of this paper.

The core of an Ampersand-script is a tuple $\langle \mathbb{H}, \mathbb{R}, \mathbb{C}, \mathfrak{T} \rangle$, which consists of a set of rules $\mathbb{H}$, relations $\mathbb{R}$, concepts $\mathbb{C}$, and a type function $\mathfrak{T}$. Ampersand-scripts are interpreted by the compiler as an information system. The rules constitute a theory in heterogeneous relation algebra. They constrain a body of data that resides in a database. The Ampersand-compiler generates a database from relations in the script. A database-application[2] assists users to keep rules satisfied throughout the lifetime of the database. It is also generated by Ampersand.

A rule is an equality between two terms. Terms are built from relations. Ampersand interprets every relation as a finite set of pairs, which are stored in the database. The phase in which Ampersand takes a script, and turns it into a database, is what we will refer to as *compile-time*. The phase in which a user interacts with the database, is what we will refer to as *run-time*. At run-time, Ampersand can decide which rules are satisfied by querying the database. The compiler generates all software needed to maintain rules at run-time. If a rule is not satisfied as a result of data that has changed, that change is reverted (rolled back) to maintain a state in which all rules are satisfied. Changes to the database are not specified by the software developer, but generated by the compiler. Rules in Ampersand are maintained rather than executed directly.

Atoms are values that have no internal structure, meant to represent data elements in a database. From a business perspective, atoms are used to represent concrete items of the world, such as `Peter`, `1`, or `the king of France`. By

[1] A *brief* is a document that is meant to summarize a lawsuit for the judge and counterparty. It provides legal reasons for claims in a lawsuit based on regulations, precedents, and other legally acceptable sources. It shows how the reasoning applies to facts from the case.

[2] Ampersand generates an application that consists of a relational database and interface components. Currently this application runs server-side on a PHP/MySQL platform and on a web-browser on the client-side.

convention throughout the remainder of this paper, variables a, b, and c are used to represent *atoms*. The set of all atoms is called $\mathbb{A}$. Each atom is an instance of a *concept*.

Concepts (from set $\mathbb{C}$) are names we use to classify atoms in a meaningful way. For example, you might choose to classify `Peter` as a person, and `074238991` as a telephone number. We will use variables A, B, C, D to represent concepts. The term $\mathbb{I}_A$ represents the *identity relation* of concept A. The expression $a \in A$ means that atom a is an instance of concept A. In the syntax of Ampersand, concepts form a separate syntactic category, allowing a parser to recognize them as concepts. Ampersand also features specialization. Specialization is needed to allow statements such as: "An orange is a fruit that ....". Specialization is not relevant for the remainder of this paper.

Relations (from set $\mathbb{R}$) are used in information systems to store facts. A *fact* is a statement that is true in a business context. Facts are stored and kept as data in a computer. As data changes over time, so do the contents of these relations. In this paper relations are represented by variables r, s, and d. We represent the declaration of a relation r by $nm_{\langle A,B\rangle}$, in which nm is a name and A and B are concepts. We call A the source concept and B the target concept of the relation. The term $\mathbb{V}_{[A\times B]}$ represents the *universal relation* over concepts A and B.

The meaning of relations in Ampersand is defined by an interpretation function $\mathfrak{I}$. It maps each relation to a set of facts. Furthermore, it is a run-time requirement that the pairs in r are contained in its type:

$$\langle a,b\rangle \in \mathfrak{I}(nm_{\langle A,B\rangle}) \Rightarrow\ a \in A \wedge b \in B \tag{1}$$

Terms are used to combine relations using operators. The set of terms is called $\mathbb{T}$. It is defined by:

Definition 1 (terms).
The set of terms, $\mathbb{T}$, is the smallest set that satisfies, for all $r, s \in \mathbb{T}$, $d \in \mathbb{R}$ and $A, B \in \mathbb{C}$:

$$d \in \mathbb{T} \qquad (\textit{every relation is a term}) \tag{2}$$
$$(r \cap s) \in \mathbb{T} \qquad (\textit{intersection}) \tag{3}$$
$$(r - s) \in \mathbb{T} \qquad (\textit{difference}) \tag{4}$$
$$(r; s) \in \mathbb{T} \qquad (\textit{composition}) \tag{5}$$
$$r^{\smile} \in \mathbb{T} \qquad (\textit{converse}) \tag{6}$$
$$\mathbb{I}_A \in \mathbb{T} \qquad (\textit{identity}) \tag{7}$$
$$\mathbb{V}_{[A\times B]} \in \mathbb{T} \qquad (\textit{full set}) \tag{8}$$

Throughout the remainder of this paper, terms are represented by variables r, s, d, and t. The *type* of a term r is a pair of concepts given by $\mathfrak{T}(r)$. $\mathfrak{T}$ is a partial function that maps terms to types. If term r has a type, this term is called *type correct*. The Ampersand compiler requires all terms to be type correct, or else it will not generate any code. The type function and the restrictions it suffers are discussed in [14]. However, for the remainder of this paper this is irrelevant.

The meaning of terms in Ampersand is an extension of interpretation function $\mathfrak{I}$. Let A and B be finite sets of atoms, then $\mathfrak{I}$ maps each term to the set of pairs for which that term stands.

Definition 2 (interpretation of terms).
For every $A, B \in \mathbb{C}$ and $r, s \in \mathbb{T}$

$$\mathfrak{I}(r) = \{\langle a,b\rangle |\ a\ r\ b\} \tag{9}$$
$$\mathfrak{I}(r \cap s) = \{\langle a,b\rangle |\ \langle a,b\rangle \in \mathfrak{I}(r)\ \ \textit{and}\ \ \langle a,b\rangle \in \mathfrak{I}(s)\} \tag{10}$$
$$\mathfrak{I}(r - s) = \{\langle a,b\rangle |\ \langle a,b\rangle \in \mathfrak{I}(r)\ \ \textit{and}\ \langle a,b\rangle \notin \mathfrak{I}(s)\} \tag{11}$$
$$\mathfrak{I}(r;s) = \{\langle a,c\rangle |\ \ \textit{for some}\ b,\ \langle a,b\rangle \in \mathfrak{I}(r)\ \ \textit{and}\ \langle b,c\rangle \in \mathfrak{I}(s)\} \tag{12}$$
$$\mathfrak{I}(r^{\smile}) = \{\langle b,a\rangle |\ \langle a,b\rangle \in \mathfrak{I}(r)\} \tag{13}$$
$$\mathfrak{I}(\mathbb{I}_A) = \{\langle a,a\rangle |\ a \in A\} \tag{14}$$
$$\mathfrak{I}(\mathbb{V}_{[A\times B]}) = \{\langle a,b\rangle |\ a \in A, b \in B\} \tag{15}$$

Ampersand has more operators than the ones introduced in Definition 2: the complement (prefix unary $-$), Kleene closure operators (postfix $^+$ and *), left- and right residuals (infix \ and/), relational addition (infix †), and product (infix $\times$). These are all expressible in the definitions above, so we have limited this exposition to the operators introduced above.

The complement operator is defined by means of the binary difference operator (Eq. 4).

$$\mathfrak{T}(r) = \langle A, B\rangle\ \Rightarrow\ \overline{r}\ =\ \mathbb{V}_{[A\times B]} - r \tag{16}$$

This definition is elaborated in [32].

A *rule* is a pair of terms $r, s \in \mathbb{T}$ with $\mathfrak{T}(r) = \mathfrak{T}(s)$, which is syntactically recognizable as a rule.

$$\text{RULE } r = s$$

This means $\mathfrak{I}(r) = \mathfrak{I}(s)$. In practice, many rules are written as:

$$\text{RULE } r \subseteq s$$

This is a shorthand for

$$\text{RULE } r \cap s = r$$

We have enhanced the type function $\mathfrak{T}$ and the interpretation function $\mathfrak{I}$ to cover rules as well. If $\mathfrak{T}(r) = \mathfrak{T}(s)$ and $\mathfrak{T}(s) = \langle A, B\rangle$:

$$\mathfrak{T}(\text{RULE } r = s) = \langle A, B\rangle \tag{17}$$
$$\mathfrak{T}(\text{RULE } r \subseteq s) = \langle A, B\rangle \tag{18}$$
$$\mathfrak{I}(\text{RULE } r = s) = \mathfrak{I}(\mathbb{V}_{[A\times B]} - ((s - r) \cup (r - s))) \tag{19}$$
$$\mathfrak{I}(\text{RULE } r \subseteq s) = \mathfrak{I}(\mathbb{V}_{[A\times B]} - (r - s)) \tag{20}$$

We call rule r *satisfied* when $\mathfrak{I}(\text{RULE } r = s) = \mathfrak{I}(\mathbb{V}_{[A \times B]})$. As the population of relations used in r changes with time, the satisfaction of the rule changes accordingly. A software developer, who conceives these rules, must consider how to keep each one of them satisfied. We call a rule *violated* if it is not satisfied. The set $\mathfrak{I}((s - r) \cup (r - s))$ is called the *violation set* of RULE $r = s$. To *resolve* violations means to change the contents of relations such that the rule is satisfied[3]. Each pair in the violation set of a rule is called a violation of that rule.

The software developer must define how to resolve violations when they occur. She does so by inserting and/or deleting pairs in appropriately chosen relations. Whatever choice she makes, she must ensure that her code yields data that satisfies the rules. When we say: "rule r specifies this action" we mean that satisfaction of rule r is the goal of any action specified by rule r.

3 Conceptual Analysis

As a case study, an argument assistance system, MirrorMe, was built. We have chosen to implement the ideas of Toulmin [27], because his book "The uses of Arguments" is still one of the most influential works in the area of legal argument management[4]. Toulmin is regarded as the first scholar in modern history to come up with a usable theory of argumentation. Recent work typically draws on Toulmin, so his doctrine offers a good starting point. Toulmin's ideas have been implemented before, for example in a tool called ArguMed [28].

Software systems that support legal arguments have been around for many decades. Verheij [29] distinguishes between argument assistance systems and automated reasoning systems. Automated reasoning has never gained wide acceptance for making legal decisions, because lawyers and judges alike feel that human judgment should be at the core of any legal decision. Attempts to apply mathematical logic to legal judgments have had limited impact for similar reasons [21]. Legal reasoning differs from logic reasoning because of the human judgments that are involved. The literature on legal reasoning [15] makes it quite clear why mathematical logic alone does not suffice. Argument assistance systems [30] have been more successful, because they respect the professional freedom of legal professionals to construct their own line of argumentation. Such systems offer help in many different ways. They can help by looking up legal references, jurisdiction from the past, scholarly works etc. They can also help to construct and validate arguments by keeping arguments and evidence organized. They can store, disclose and share legal evidence.

In our case study we use logic to reason about the correctness of a program. The argumentation principles of Toulmin are *implemented in* logic rather than *replaced by* logic. The structure of MirrorMe was designed by conceptually analysing ideas of Toulmin, such as claim, warrant, argument, and rebuttal. They appear in MirrorMe as relations. The Ampersand compiler generates a

[3] To *restore invariance* is sometimes used as a synonym to resolving violations. Consequently, a rule is sometimes called *invariant*.

[4] Thanks to Elleke Baecke for pointing us towards this source.

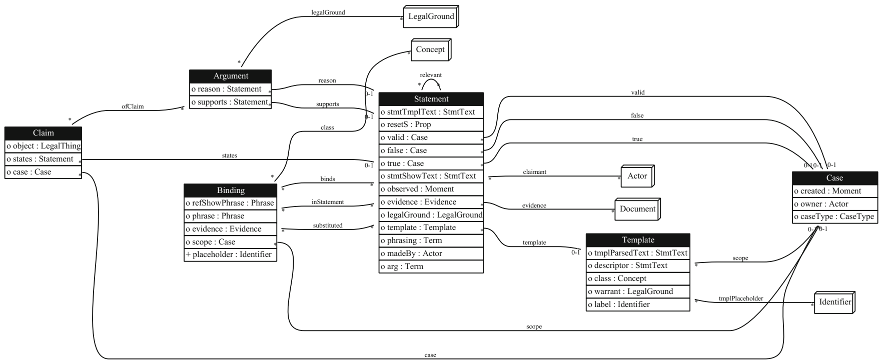

Fig. 1. Conceptual data model

conceptual data model to help the software developer to oversee all relations. Even though our case study yields a model that is a bit too large for this paper, Fig. 1 gives a good impression of what it looks like.

To convey the flavour of software development in relation algebra, it suffices to discuss a tiny part of the whole program. Therefore, we shall discuss the part of the conceptual model that is used in the sequel. The context in which a user creates arguments and reasons about them is a legal case. For this reason, validity, falsehood and truth of statements are related to the case. A *statement* is a phrase in natural language that can be either true or false. An example is "`The employee, John Brown, is entitled to 50 Euros`"[5]. In MirrorMe, a user can define a template, such as "`The employee, [emp], is entitled to [increase]`". The strings "`[emp]`" and "`[increase]`" are called placeholders. The reason for using placeholders is that legal rules are stated in general terms, e.g. "`Every employee is entitled to an increase in salary`". The user of MirrorMe will pick a legal text (from a source he trusts) and substitute parts of that text by placeholders. When the facts are known, the argument can be completed by substituting placeholders by actual phrases.

It is precisely this substitution process that we have chosen to describe in this paper as a case study in programming with relation algebra.

4 Programming in Relation Algebra

At this point we have reached the core of this case study. We focus on the substitution of placeholders when their values change. By focusing on this tiny detail, we can discuss the mechanics "under the hood" of the application generated by Ampersand.

[5] This type of statement is typical for cases. It is valid only in the case where this particular John Brown is known. In other cases, where John Brown is unknown, this statement is meaningless.

First we show how the computer solves the issue by looking at an excerpt of a log file (Fig. 2). It shows an alternating sequence of the computer (ExecEngine) mentioning a rule, followed by insert or delete actions to satisfy that rule. Then we zoom in further, one rule at a time, to explain precisely what each rule looks like and how programming is done. We then discuss the same flow of events by means of a graph (Fig. 3) in which these rules and actions are nodes. This graph serves as an event flow diagram to illustrate the process behind Fig. 2.

```
ExecEngine run started
ExecEngine satisfying rule 'signal phrase update'
InsPair(resetS,Statement,Stat623,Statement,Stat623)
ExecEngine satisfying rule 'flush substitutions'
DelPair(substituted,Binding,Bind625,Statement,Stat623)
ExecEngine satisfying rule 'reset statement text'
InsPair(stmtShowText,Statement,Stat623,StmtText,
        The employee, [emp], is entitled to [increase].)
ExecEngine satisfying rule 'done initializing'
DelPair(resetS,Statement,Stat623,Statement,Stat623)
ExecEngine satisfying rule 'substitute'
InsPair(stmtShowText,Statement,Stat623,StmtText,
        The employee, James, is entitled to [increase].)
InsPair(substituted,Binding,Bind625,Statement,Stat623)
ExecEngine satisfying rule 'fill shownPhrase'
InsPair(shownPhrase,Binding,Bind625,Phrase,James)
ExecEngine run completed
```

Fig. 2. Log file of a substitution

The log file of Fig. 2 has been taken from a computer that carries out the procedure to satisfy all rules. The machine will only act on rules that are violated. The first rule to be violated is “signal phrase update”, in which the computer signals that something or someone has made a change in relation *phrase*. The procedure ends by doing the necessary substitutions, ensuring that all statements have the actual phrase of a placeholder in their text.

Let us now study the actual rules to see how these rules cause the right actions to take place in the correct order. We will follow the log file from Fig. 2 in reverse direction, reasoning backwards from the result.

Whether a placeholder has been edited can be observed by comparing its new phrase to the shown phrase[6]. The phrase of a placeholder is kept in the relation *phrase*. The shown phrase is kept in the relation *shownPhrase*. The sole purpose for having the relation *shownPhrase* is to detect a change in *phrase*. Let us introduce *differB* to represent the bindings with an updated phrase:

[6] Note that we use the notion “the phrase of a placeholder” to indicate a pair from $\mathit{phrase}^{\smile};\mathit{placeholder}$.

$$\mathit{differB} = \mathbb{I}_{Binding} \cap \mathit{shownPhrase}; \overline{\mathbb{I}_{Phrase}}; \mathit{phrase}^{\smile}$$

When the phrase of a placeholder changes, that phrase must be updated in every statement in which the placeholder was used. That update action is specified in Sect. 4.2. Section 4.1 specifies how *shownPhrase* is made equal to *phrase*, after all necessary substitutions are done.

4.1 Rule: Fill shownPhrase

$$\text{RULE } (\mathbb{I}_{Binding} \cap \mathit{substituted}/\mathit{inStatement}); \mathit{phrase} \ \subseteq \ \mathit{shownPhrase}$$

This rule says that for each binding that has been substituted in every statement it is used in, the *phrase* must be equal to the *shownPhrase*. When violated, it is satisfied by inserting all violations into *shownPhrase*.

4.2 Rule: Substitute

Let us now look into the process of substituting placeholders by phrases. The relation *tmplParsedText* contains the original text, provided by a user. Placeholders are specified by enclosing them in brackets, e.g. "`The employee, [emp], is entitled to [increase]`". The text in which a placeholder has been substituted by a phrase, e.g. "`The employee, John Brown, is entitled to 50 Euros`", is kept in relation *stmtShowText*. Each substitution that has been done in a statement corresponds to a binding-statement pair in the relation *substituted*. This relation keeps track of all substitutions. After a placeholder has been substituted, it no longer occurs in *stmtShowText*. This poses a problem if we want to substitute the new phrase in *stmtShowText*. For the placeholder that defines the place in the text where to substitute, is no longer in that text. Therefore, substitutions must be done in the original text of the statement. All placeholders in that text must then be substituted again. So, the text in *stmtShowText* must first be reset to the original text from *tmplParsedText*. To keep track of substitutions correctly, all corresponding binding-statement pairs must be removed from the relation *substituted*. Only after resetting is done, the substitutions can be put back in place with the new phrases filled in.

We define a relation *resetS* to register the statements that are being reset. In statements that are not being reset, $\mathbb{I}_{Statement} - \mathit{resetS}$, substitutions can take place. All placeholders that have a binding with a phrase can be substituted. The following rule specifies the action of substituting placeholders.

$$\begin{aligned}&\text{RULE}\\ &\quad (\mathbb{I}_{Binding} \cap \mathit{phrase}; \mathit{phrase}^{\smile}); \mathit{inStatement}; (\mathbb{I}_{Statement} - \mathit{resetS})\\ &\subseteq\\ &\quad \mathit{substituted}\end{aligned}$$

Violations of this rule are binding-statement pairs, of which the binding has a phrase and the statement is not being reset. Hence, this rule can be satisfied by inserting every violation into the relation substituted.

4.3 Rule: Done Initializing

Resetting a statement is done when two conditions are met. First, every statement that is (still) being reset may have no bindings in the relation *substituted*. Second, the text in *stmtShowText* corresponds to the text in *tmplParsedText*. So the rule that specifies the action is:

$$\begin{array}{l}\text{RULE}\\ \quad (\mathit{resetS} - \mathit{inStatement}^{\smile}; \mathit{substituted}) \cap \\ \quad \mathit{template}; \mathit{tmplParsedText}; \mathit{stmtShowText}^{\smile}\\ \subseteq \\ \quad \overline{\mathit{resetS}}\end{array}$$

Violations of this rule are statements that are no longer being reset, but are still in the relation *resetS*. The appropriate action is to remove them from *resetS*.

4.4 Rule: Reset Statement Text

To satisfy one condition from Sect. 4.3, the text in *stmtShowText* must be made equal to the original text in the template. The action is specified by the following rule:

$$\text{RULE } \mathit{resetS}; \mathit{template}; \mathit{descriptor} \ \subseteq \ \mathit{stmtShowText}$$

Violations of this rule are descriptors of templates that belong to statements that are being reset. These violations can be resolved by inserting them in *stmtShowText*.

4.5 Rule: Flush Substitutions

To satisfy the other condition from Sect. 4.3, the following rule specifies the action to be taken:

$$\text{RULE } \mathbb{V}_{[\mathit{Binding} \times \mathit{Binding}]}; \mathit{inStatement}; \mathit{resetS} \ \subseteq \ \overline{\mathit{substituted}}$$

Every binding in a statement that is being reset needs to be removed from the relation *substituted*. The software developer can implement this by deleting all violations of this rule from the relation *substituted*.

4.6 Rule: Signal Phrase Update

When the phrase in a binding is edited and the new phrase differs from the shown phrase, this signals that substitutions must be flushed (see Sect. 4.5), that the statement text must be reset to the original text (see Sect. 4.4), that the reset-state must be revoked (see Sect. 4.3), that substitution must take place (see Sect. 4.2), and finally that the phrase detection is switched off again (see Sect. 4.1). The initial condition occurs if a binding has been used (substituted) in a statement, and the binding satisfies *differB*. The following rule specifies the action that resetting a statement can start:

$$\text{RULE } \mathbb{I}_{Statement} \cap \mathit{inStatement}^{\smile}; \mathit{differB}; \mathit{substituted} \ \subseteq \ \mathit{resetS}$$

Violations of this rule are statements of which a substitution must be re-done. The software developer can have these violations added to *resetS* to satisfy this rule. In doing so, the chain of events is triggered that ends when all rules are satisfied.

5 Programming in the Small

Let us take a closer look at the programming process. Working in relation algebra, a software developer must think about satisfying constraints. She considers how violations arise from changing content of relations. And she thinks about insert and delete actions to restore these violations. In our case study, we have reasoned with the event types: Ins $\langle \mathit{relation} \rangle$ and Del $\langle \mathit{relation} \rangle$[7]. Table 1 shows which event types may violate which rules. Recall Sect. 4, where the process of substitution started by updating the value of a placeholder. This meant doing an insert after a delete on the relation *phrase*, causing the updated phrase to appear in *differB*. That was signaled by the rule "signal phrase update". In general, the software developer must decide how to resolve violations that occur as a result of some event. This can be done by choosing among the duals of the event types from Table 1. By systematically swapping Ins for Del and vice-versa, the table shows by which type of events violations can be restored. To ensure progress, the software developer will pick a different relation for restoring than the relation that causes the violation. The software developer can draw a graph that contains all information from Table 1 and the dual information. Figure 3 shows part of that graph. A circle represents a rule, a rectangle represents an event type, and the arrows connect them. The software developer can work her way through the

Table 1. By which type of events can rules be violated?

Rule	Event types
signal phrase update	Ins *inStatement*, Ins *differB*, Ins *substituted*, Del *resetS*
flush substitutions	Ins *inStatement*, Ins *resetS*, Del *substituted*
reset statement text	Ins *resetS*, Ins *template*, Ins *descriptor*, Del *stmtShowText*
done initializing	Ins *resetS*, Del *inStatement*, Del *substituted*, Ins *template*, Ins *tmplParsedText*, Ins *stmtShowText*
substitute	Ins *phrase*, Ins *inStatement*, Del *resetS*, Del *substituted*
fill shownPhrase	Ins *substituted*, Del *inStatement*, Del *shownPhrase*, Ins *phrase*

[7] Ins and Del are called each others duals.

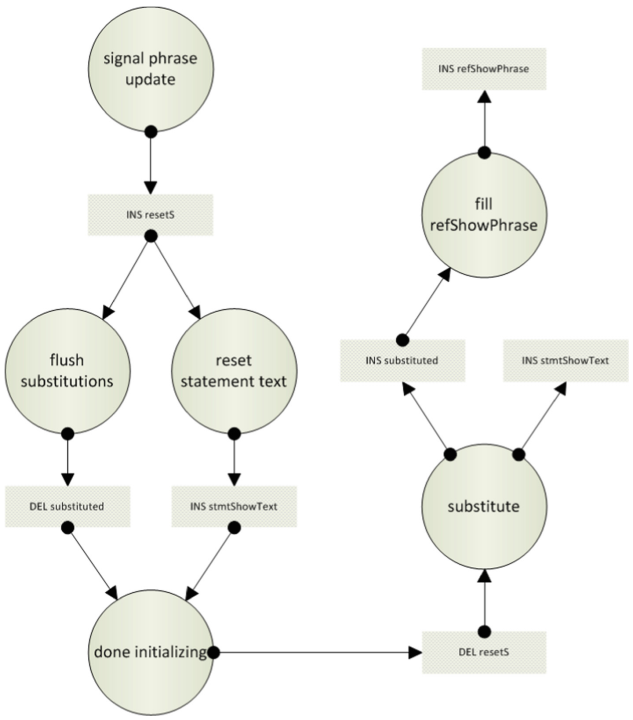

Fig. 3. Event flow

graph, to write code to restore invariants for every event type that might cause violations. Figure 3 shows just that part of the graph that corresponds with the case study in this article.

6 Reflection

Ampersand is built on the belief that software development should be automated. It uses rules to represent requirements, in the firm belief that consistent requirements are essential to the software development process [4]. Being fully aware of the fate of formal methods in computer science, Ampersand is founded on the belief that software development must be done more formally. So Ampersand is building further on the foundations laid by formal specification methods such as Z [24] and Alloy [?]. In contrast however with such methods, Ampersand is equipped with a software generator that generates an information system. Thus, specifying in Ampersand is developing software at the same time. Ampersand complements the RelView approach [?], which also generates software but is stronger in specifying complex computational problems. In retrospect we see that programming in relation algebra yields an unconventional programming experience. This experience consists of inventing rules and choosing event types for resolving violations, as illustrated by the case study. To relate this experience to programming as we know it, Sect. 6.1 makes a comparison with established programming languages (prior art). Section 6.2 summarizes the contributions to software development claimed by Ampersand. Section 6.3 summarizes the use of Ampersand in practice and Sect. 6.4 gives an outlook on research that is required in the near future.

6.1 Comparison

This section compares Ampersand with existing programming paradigms by mentioning the most important differences and similarities.

The first to compare with is the imperative programming paradigm, known from popular languages such as Java [7] and C++ [25]. Our case study cannot be called imperative, because the notion of control flow in imperative languages is very different. In imperative languages, the control flow is defined by the software developer. In this case study, the control flow emerges as a result of changes in relation content as illustrated by Fig. 3.

The case study also differs from the logic programming paradigm of Prolog [16] and of all rule engines that can be considered to be an offspring of Prolog. A difference lies in the way rules are interpreted. In logic programming, a program consists of Horn-clauses and a resolution-proof is constructed on runtime. The software developer works with notions such as backtracking (backward chaining) and unification, which are absent in Ampersand. Ampersand is not restricted to Horn-clauses; any relation-algebraic equation over relations can be used as a rule.

The case study also differs from functional programming, of which Haskell and Scala are prominent representants. A core idea in functional programming is to evaluate a program as a function from input to output [2]. A functional program consists of function definitions, that are evaluated by a term- or graph-rewriter, using various strategies such as lazy or eager evaluation. In contrast, one might interpret our case study as a relaxation of the constraint that everything is a function. In relation algebra, everything is a relation and a function is a restricted form of a relation. A similarity to functional programming is the declarative style, because substitution of equal terms without changing the semantics is a property we see in both worlds.

A difference with database programming is found in the type of algebra that is used. Relational databases are founded on relational algebra [5]. They are typically programmed in SQL. In contrast with database programming, Ampersand implements heterogeneous relation algebra [23]. A software developer working with relational algebra sees n-ary tables, while Ampersand is restricted to binary relations. The comparison between relation*al* algebra and relation algebra might relate to comparing relational databases with graph databases [31], although no literature was found to corroborate this.

In the tradition of formal specification, there are many relational approaches, such as Z [24], CSP [22], LOTOS [9], VDM [11]. Where formal specification techniques typically analyse and diagnose specifications, Ampersand actually synthesizes (generates) information systems.

If Ampersand represents a programming style at all, we might call it "a relation-algebraic style of programming". That style would be characterized by a programmer who specifies constraints and a computer trying to satisfy these constraints by resolving violations.

6.2 Contribution

Contributions of Ampersand to the software development process are:

- Ampersand has the usual benefits of a declarative language: This means that terms can be manipulated by Tarski's axioms without changing their semantics [32]. It also means that the order in which rules are written has no consequence for their semantics.
- Heterogeneous relation algebra has a straightforward interpretation in natural language [12]. We have used that to formalize business requirements without exposing business stakeholders to any formal notation.
- Heterogeneous relation algebra in Ampersand is statically typed [14]. There is much evidence for significantly lower software maintenance cost due to static typing as opposed to dynamic typing [6,20].
- Heterogeneous relation algebra is well studied. As a consequence, many tools that are readily available in the public domain can be put to good use. For executives of large organizations it can be reassuring that the formalism is free of childhood diseases.
- Relation algebra facilitates composing software from reusable components, because a program consists of rules. Since the union of sets of rules is a set of rules, compositionality comes from the union operator. In practice, when components are brought together in larger assemblies, hardly any adjustments have to be made[8].

Ampersand also has disadvantages. It appears to be difficult to learn for large groups of software professionals. Research [18] shows that this is largely due to deficits in prerequisite knowledge, especially skills in discrete mathematics. Also, programming appears to be difficult in practice.

6.3 Ampersand in Practice

Ampersand has been used in practice both in education (Open University of the Netherlands) and in industry (Ordina and TNO-ICT). For example, Ordina designed a proof-of-concept in 2007 of the INDIGO-system. This design was based on Ampersand, to obtain correct, detailed results in the least amount of time. Today INDIGO is in use as the core information system of the Dutch immigration authority, IND. More recently, Ampersand was used to design an

[8] This is (unsubstantiated) experience collected from projects we have done with Ampersand.

information system called DTV for the Dutch food authority, NVWA. A prototype of DTV was built in Ampersand and was used as a model to build the actual system. TNO-ICT, a major Dutch industrial research laboratory, is using Ampersand for research purposes. For example, TNO-ICT did a study of international standardizations efforts such as RBAC (Role Based Access Control) in 2003 and architecture (IEEE 1471-2000) [8] in 2004. Several inconsistencies were found in the last (draft) RBAC standard [1]. TNO-ICT has also used the technique in conceiving several patents[9]. At the Open University of the Netherlands, Ampersand is being taught in a course called Rule Based Design [12]. In this course, students use a platform called RAP, which has been built in Ampersand [18]. RAP has been the first Ampersand-application that has run in production.

6.4 Further Research

Further research on this topic is required to bring relation algebra still closer to the community of practitioners. Further use of relation algebra can be made by incorporating a model checker, such as the Alloy analyser [?], to detect inconsistent rules. An exciting new development is Amperspiegel [13], which brings notational flexibility at the fingertips of the user. Developments in the Ampersand-compiler are going towards a rule-repository (written in Ampersand itself). This will make collaborative information systems development in Ampersand easier, because the repository can assist in automating the software development process further. Other research is needed towards a comprehensive theory of information systems. Currently, there is no theory (in the mathematical meaning of the word) for information systems. In the Ampersand project, a sub-project called "Formal Ampersand" is being conducted to achieve this goal.

References

1. ANSI, INCITS 359: Information Technology: Role Based Access Control Document Number: ANSI/INCITS 359–2004. InterNational Committee for Information Technology Standards (formerly NCITS) (2004)
2. Backus, J.: Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. Commun. ACM **21**(8), 613–641 (1978). doi:10.1145/359576.359579
3. Berghammer, R., Ehler, H., Zierer, H.: Towards an algebraic specification of code generation. Sci. Comput. Program. **11**(1), 45–63 (1988). doi:10.1016/0167-6423(88)90064-0
4. Boehm, B.W.: Software Engineering Economics. Advances in Computing Science and Technology. Prentice Hall PTR, Upper Saddle River (1981)
5. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970). http://doi.acm.org/10.1145/362384.362685

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

6. Hanenberg, S., Kleinschmager, S., Robbes, R., Tanter, É., Stefik, A.: An empirical study on the impact of static typing on software maintainability. Empirical Softw. Eng. **19**(5), 1335–1382 (2014). doi:10.1007/s10664-013-9289-1
7. Harms, D., Fiske, B.C., Rice, J.C.: Web Site Programming With Java. McGraw-Hill, New York City (1996). http://www.incunabula.com/websitejava/index.html
8. IEEE: Architecture Working Group of the Software Engineering Committee: Standard 1471–2000: Recommended Practice for Architectural Description of Software Intensive Systems. IEEE Standards Department (2000)
9. ISO: ISO 8807: Information processing systems - open systems interconnection - LOTOS - a formal description technique based on the temporal ordering of observational behaviour. Standard, International Standards Organization, Geneva, Switzerland. 1st edn. (1987)
10. Jackson, D.: A comparison of object modelling notations: Alloy, UML and Z. Technical report (1999). http://sdg.lcs.mit.edu/publications.html
11. Jones, C.B.: Systematic Software Development Using VDM. Prentice Hall International (UK) Ltd., Hertfordshire (1986)
12. Joosten, S., Wedemeijer, L., Michels, G.: Rule Based Design. Open Universiteit, Heerlen (2013)
13. Joosten, S.J.C.: Parsing and print of an with triples. In: Pous, D., Struth, G., Hfner, P. (eds.) RAMiCS 2017, p. Under Submission. Springer International Publishing, Berlin (2017)
14. Joosten, S.M.M., Joosten, S.J.C.: Type checking by domain analysis in ampersand. In: Kahl, W., Winter, M., Oliveira, J.N. (eds.) RAMICS 2015. LNCS, vol. 9348, pp. 225–240. Springer, Cham (2015). doi:10.1007/978-3-319-24704-5_14
15. Lind, D.: Logic and Legal Reasoning. The National Judicial College Press, Beijing (2007)
16. Lloyd, J.W.: Foundations of Logic Programming. Springer, New York (1984)
17. Maddux, R.: Relation Algebras. Elsevier Science, Studies in Logic and the Foundations of Mathematics (2006)
18. Michels, G.: Development environment for rule-based prototyping. Ph.D. thesis, Open University of the Netherlands (2015)
19. Michels, G., Joosten, S., van der Woude, J., Joosten, S.: Ampersand. In: Swart, H. (ed.) RAMICS 2011. LNCS, vol. 6663, pp. 280–293. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21070-9_21
20. Petersen, P., Hanenberg, S., Robbes, R.: An empirical comparison of static and dynamic type systems on API usage in the presence of an IDE: Java vs. groovy with eclipse. In: Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014, pp. 212–222. ACM, New York (2014). doi:10.1145/2597008.2597152
21. Prakken, H.: Ai & law, logic and argument schemes. Argumentation **19**(3), 303–320 (2005). doi:10.1007/s10503-005-4418-7
22. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River (1997)
23. Schmidt, G., Hattensperger, C., Winter, M.: Heterogeneous relation algebra. In: Brink, C., Kahl, W., Schmidt, G. (eds.) Relational Methods in Computer Science, pp. 39–53. Springer, New York (1997). ISBN 3-211-82971-7
24. Spivey, J.: The Z Notation: A Reference Manual. International Series in Computer Science, 2nd edn. Prentice Hall, New York (1992)
25. Stroustrup, B.: The C++ Programming Language, 3rd edn. Addison-Wesley Professional, Boston (1997)

26. Swart, H., Berghammer, R., Rusinowska, A.: Computational social choice using relation algebra and relview. In: Berghammer, R., Jaoua, A.M., Möller, B. (eds.) RelMiCS 2009. LNCS, vol. 5827, pp. 13–28. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04639-1_2
27. Toulmin, S.E.: The Uses of Argument. Cambridge University Press, Cambridge (1958). http://www.amazon.com/exec/obidos/redirect?tag=citeulike-20&path=ASIN/0521534836
28. Verheij, B.: Automated argument assistance for lawyers. In: Proceedings of the 7th International Conference on Artificial Intelligence and Law, ICAIL 1999, pp. 43–52. ACM, New York (1999). doi:10.1145/323706.323714
29. Verheij, B.: Artificial argument assistants for defeasible argumentation. Artif. Intell. **150**(1), 291–324 (2003). doi:10.1016/S0004-3702(03)00107-3. http://www.sciencedirect.com/science/article/pii/S0004370203001073
30. Verheij, B.: Virtual Arguments. On the Design of Argument Assistants for Lawyers and Other Arguers. T.M.C. Asser Press, The Hague (2005)
31. Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., Wilkins, D.: A comparison of a graph database and a relational database: a data provenance perspective. In: Proceedings of the 48th Annual Southeast Regional Conference, ACM SE 2010, pp. 42:1–42:6. ACM, New York (2010). doi:10.1145/1900008.1900067
32. van der Woude, J., Joosten, S.: Relational heterogeneity relaxed by subtyping. In: Swart, H. (ed.) RAMICS 2011. LNCS, vol. 6663, pp. 347–361. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21070-9_25