Ampersand
Applying Relation Algebra in Practice

Gerard Michels[1], Sebastiaan Joosten[1], Jaap van der Woude[1], and Stef Joosten[1,2]

[1] Open Universiteit Nederland,
Postbus 2960,
6401 DL HEERLEN
[2] Ordina NV
Nieuwegein
{gerard.michels,jaap.vanderwoude,stef.joosten}@ou.nl

Abstract. Relation algebra can be used to specify information systems and business processes. It was used in practice in two large IT projects in the Dutch government. But which are the features that make relation algebra practical? This paper discusses these features and motivates them from an information system designer's point of view. The resulting language, Ampersand [1], is a syntactically sugared version of relation algebra. It is a typed language, which is supported by a compiler. The design approach, also called Ampersand, uses software tools that compile Ampersand scripts into functional specifications. This makes Ampersand interesting as an application of relation algebra in the industrial practice. The purpose of this paper is to define Ampersand and motivate its features from a practical perspective.

This work is part of the research programme of the Information Systems & Business Processes (IS&BP) department of the Open University.

Keywords: heterogeneous relation algebra – domain-specific feedback – specification language – business rules – requirements engineering – type system – rule based design.

1 Introduction

Ampersand is a simple requirements specification language with relational semantics. It is a syntactically sugared version of relation algebra. It has been designed for students and practitioners with minimal mathematical background, who use it for designing business processes. In the sequel, we shall call these users 'requirements engineers', because Ampersand enables them to design by eliciting requirements. The purpose of this paper is to describe the features of

[1] It is named after the ampersand symbol (&), which means "and". The name hints at the desire to have it all: getting the best from both business and IT, achieving results from theory and practice alike, and realising the desired results effectively and more efficiently than ever before.

H. de Swart (Ed.): RAMICS 2011, LNCS 6663, pp. 280–293, 2011.

Ampersand that make it practical. Ampersand is currently used in teaching at the Open University of the Netherlands (OUNL) and it has already been used to design several large scale IT-systems in the Dutch government[2].

A challenge in designing Ampersand was to obtain a useful requirements specification language that is faithful to Tarski's axioms. This sets Ampersand apart from Codd's relational model [3], which has been criticized for being unfaithful to its mathematical origins [4]. In this paper Ampersand is defined with semantics in a homogeneous relation algebra which is faithful to Tarski's axioms. But only relations with a type are considered useful. Ampersand with semantics in a heterogeneous relation algebra is discussed in [15]. Generalization (subtyping) is exploited in that paper to embed the relations appropriately such that the composition and union operations are total and the Tarski axioms are valid to a large extent.

Ampersand features rules, relations and concepts. It has a type system that blocks compilation as long as a script contains type incorrect expressions. In essence, a specification in Ampersand is a set of rules and a set of relation symbol declarations with a concept-based type. Each rule is an expression (a relation term) in a relation algebra that must be kept true throughout time. Thus, each relation that is a rule represents an invariant requirement of the business. A requirements engineer takes responsibility for a correspondence between requirements (in natural language) and rules (i.e., constraints in relation algebra).

Ampersand features a compiler that produces requirements specifications in natural language in the form of a PDF-document. A type system ensures the absence of relation terms that are predictably nonsensical to the end user. Relation terms that are type incorrect (i.e., make no sense) are reported back to the requirements engineer as a type error. This feedback explains why an erroneous relation term makes no sense within the chosen context. The feedback generator is implemented as an attribute grammar of untyped relation terms with a type function as one of its attributes. This is described in section 4. This implementation allows for future extensions with new attributes to analyze and report in even more detail.

The relevance of relation algebra in practice has surprised the authors. Formalizing requirements results in compact scripts that allow substantial pieces of the design process to be automated. Students at the OUNL now use an online tool in which they can study the impact of their Ampersand scripts, analyse them and generate requirements specifications from them. For the future we plan to implement editing of rule models within the tool. This will give students a working prototype of their design, which is expected to improve their learning curve.

The work presented here advances the state-of-the-art of designing business processes and the IT-systems that support them. The innovation Ampersand

[2] In 2007 a new information system to support the Dutch Immigration Service in managing all immigration requests was designed. In 2010 a platform independent design for all courts of administrative law in the Netherlands was generated.

brings is to generate design artifacts, such as class diagrams, directly from requirements. The idea to use algebra for knowledge representation [1] lies at the heart of our research. Ideas from heterogeneous relation algebra [13,7] were borrowed to construct a type system. Furthermore, the existence of a well studied body of knowledge called relation algebra [10] eliminated the need to invent a new (possibly poorer) language. Ampersand has the distinct advantage of tapping into an existing, well studied body of knowledge. Besides, relation algebras are being used (e.g. [2]) and taught (e.g. [5]).

This research is relevant for requirements engineers and students alike, who specify business processes by means of constraints. They can rely on the feedback from their design tool. This audience consists mostly of students and practitioners in business information systems. The attention given to feedback is justified by the difficulty of learning how to formalize business rules in Ampersand.

The paper starts in section 2 with an informal introduction on requirements and the way an information system satisfies them. Then the syntax and semantics of Ampersand are defined. Next the type function is introduced. The Ampersand compiler does type checking and type deduction simultaneously. It has been implemented as an Haskell attribute grammar. The feedback system implementation is described and demonstrated.

2 Language for Requirements

Requirements engineers need a language that is shared by all stakeholders. For example, let us assume we are working for an auction house in Dendermonde. The sentence "Peter presides over auction room 3" makes sense in this context, if there is a person called Peter, there is an auction room 3, and the auction house is familiar with the idea that a person presides over an auction room. In this context the sentence "Vehicle 06-GNL-3 presides over auction room 3" does not make sense, because vehicles never preside over auction rooms. That is non-sense. The type system is an instrument by which the requirements engineer can exclude a significant class of nonsense. The declaration *presides* : `Person` $\sim$ `AuctionRoom` introduces a (heterogeneous) relation with the name *presides*, in which only pairs of persons and auction rooms reside. By specifying relations, a requirements engineer introduces the basic sentences of a language. More complex sentences are made by means of the operators of relation algebra. If stakeholders agree to use that language to express requirements, we call this language *shared*.

Ampersand sees an information system as a collection of data that represents facts in a given context. If, for example, Peter presides over auction room 3, one expects a tuple $\langle$`'Peter'`, `'auction room 3'`$\rangle$ in a relation *presides* to represent this fact. Some time later that relation might contain the tuple $\langle$`'Sue-Ellen'`, `'auction room 3'`$\rangle$. In an information system, data content changes as facts in the given context change. Rules constrain that data content. For example, the auction house might require that a bid on a lot at an auction may be placed only by persons who are registered as a bidder for that particular auction. Using relations to represent parts of that sentence, a requirements

engineer assembles relation terms and declares them as logical expressions with the keyword RULE.

$$bid : \texttt{Person} \sim \texttt{Lot} \tag{1}$$

$$at : \texttt{Lot} \sim \texttt{Auction} \tag{2}$$

$$registeredfor : \texttt{Person} \sim \texttt{Auction} \tag{3}$$

$$\texttt{RULE}\ bid \fatsemi at \Rightarrow registeredfor \tag{4}$$

This fragment assembles relations into a rule by means of operators from relation algebra, in this example ⨾ (compose), and ⇒ (implication). The full set of operators is introduced in definition 3. In this manner, a requirements engineer formalizes all requirements that need to be maintained by an information system. He can do this one rule at a time, adding requirements and nuance to the specification incrementally. In the fragment above the requirements engineer may add nuance by adding a rule covering that *at* is at least a partial function, which is likely to be assumed by the reader of the original requirement.

An information system contains all relations and their data. The task of the information system is to keep all rules satisfied. If a bid is placed by an unregistered bidder, the computer might simply block the transaction, explaining that you must be registered in order to place a bid. However, it might also signal this event to a registrar, to register the new bidder. Or, it might automatically look up this person, conclude that he is a subscribed member and register him automatically. For the purpose of this paper, we restrict the large number of options to two: One option is to block with an error message while the data remain in the old state (rollback). The other option is to proceed with one or more violations. In that case signals are raised, which persist as long as the violation persists. This mechanism allows an information system to control business processes, i.e., collaborations between people and computers. Blocking rules enforce consistent data throughout the business process. Signal rules trigger people to take some action in order to restore the situation that all rules are satisfied. Such rules may be satisfied with some delay, because people need some time to act.

Summarizing, Ampersand is a language built on binary relations, providing a natural model for data and business rules. This differs from the idea to use relations as a model for nondeterministic programs, which became common in the eighties [12]. There are other relational languages for constraint-based modelling, like Alloy [9] or Z [14]. Semantically, Ampersand, Z, and Alloy bear great resemblance [8]. All three are declarative specification languages with a type system. Underlying Alloy, one will find relations and atoms in a similar way Ampersand has. Both Alloy and Ampersand do automatic analysis, albeit in different ways. Ampersand uses notations of relation algebra, where Alloy and Z are more into set-notations. More significant differences are found in what Ampersand does with a specification. Unlike Alloy and Z, Ampersand generates data structures and code for a database that can hold every model of an Ampersand script. Alloy is a model checker, whereas Ampersand derives an implementation. Alloy tries to find a model, whereas Ampersand verifies any particular model at runtime

and reports violations. Alloy supports consistency checks at design time, as the user can look for counterexamples on specific assertions. Ampersand checks for consistency at runtime, and does so automatically.

3 Ampersand

This section defines the core of Ampersand. Its abstract syntax is defined in section 3.1. The semantics of relation terms are given in section 3.2. Ampersand chooses an interpretation that allows the representation of requirements and their satisfaction by means of data. A type function is introduced which is partially defined. All expressions with an undefined type are deemed incorrect and therefore rejected by the type checker. The type system is discussed in section 3.3.

3.1 Syntax

The Ampersand syntax consists of constant symbols for (business) concepts, (business) elements, relations and relation operators. Relation terms can be constructed with relations and relation operators.

Let $\mathbb{C}$ be a set of concept symbols. A concept is represented syntactically by an alphanumeric character string starting with an upper case character. In this paper, we use A, B, and C as concept variables.

Let $\mathbb{U}$ be a set of atom symbols. An atom is represented syntactically by an ASCII string within single quotes. All atoms are an element of a concept, e.g. `'Peter'` is an element of `Person`. We use a, b, and c as atom variables.

Let $\mathbb{D}$ be a set of relation symbols. A relation symbol is represented syntactically by an alphanumeric character string starting with a lower case character. For every $A, B \in \mathbb{C}$, there are special relation symbols, I_A and $V_{A\times B}$. We use r, s, and t as relation variables.

Let $\neg$, $^{\sqcup}$, $\sqcup$, and $\fatsemi$ be relation operators of arity 1, 1, 2, and 2 respectively. The binary relation operators $\sqcap$,$\Rightarrow$ and $\equiv$ are cosmetic and only defined on the interpretation function (see definition 3).

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

Definition 1 (relation terms)
$\mathbb{R}$ *is recursively defined by*

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

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

Definition 2 (statements)
An Ampersand design of context $\mathfrak{C}$ *is a user-defined collection of statements where*

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

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

An Ampersand script is a user-defined collection of relation declarations, relation elements and rule statements. It describes a (business) context $\mathfrak{C}$, in compliance with the OMG standard *Semantics of Business Vocabulary and Business Rules (SBVR)*. The Ampersand compiler contains a parser, which extracts *RUL*, *REL*, and *POP* from an Ampersand script that describes $\mathfrak{C}$.

3.2 Semantics

The previous section defines the syntactic structure of Ampersand. This section introduces an interpretation $\mathfrak{I}(R)$ that defines the semantics of a relation term R. This function interprets relation terms based on *POP* and a relation algebra [10] $\langle \mathbb{R}, \cup, \overline{\ }, ;, {}^{\smile}, \mathbb{I} \rangle$ where $\mathbb{R} \subseteq \mathcal{P}(\mathbb{U})$. All relation symbols used in a relation term are either declared by the user in *REL* or I_A or $V_{A\times B}$. So, the relation algebra on $\mathbb{R}$ is configured by the user through *REL*. The interpretation of all relation symbols in $\mathbb{D}$ is completely user-defined through *POP*. Thus, given some *REL* and some *POP*, $\mathfrak{I}(R)$ determines whether some relation holds between two elements.

Definition 3 (interpretation function)
Given some $\mathfrak{C}$, *the interpretation function of relation terms is defined by*

$$\begin{aligned}
\textit{relation} \quad & \mathfrak{I}(r) = \{\langle a, b\rangle \mid a\ r\ b \in \text{POP}\} && (5)\\
\textit{identity} \quad & \mathfrak{I}(I_A) = \{\langle a, a\rangle \mid a \in A\} && (6)\\
\textit{universal} \quad & \mathfrak{I}(V_{A\times B}) = \{\langle a, b\rangle \mid a \in A,\ b \in B\} && (7)\\
\textit{complement} \quad & \mathfrak{I}(\neg R) = \overline{\mathfrak{I}(R)} && (8)\\
\textit{converse} \quad & \mathfrak{I}(R^{\sqcup}) = \mathfrak{I}(R)^{\smile} && (9)\\
\textit{union} \quad & \mathfrak{I}(R \sqcup S) = \mathfrak{I}(R) \cup \mathfrak{I}(S) && (10)\\
\textit{composition} \quad & \mathfrak{I}(R \fatsemi S) = \mathfrak{I}(R); \mathfrak{I}(S) && (11)
\end{aligned}$$

(the interpretation of the mentioned cosmetic relation operators)

$$\begin{aligned}
\textit{intersection} \quad & \mathfrak{I}(R \sqcap S) = \mathfrak{I}(\neg(\neg R \sqcup \neg S)) && (12)\\
\textit{implication} \quad & \mathfrak{I}(R \Rightarrow S) = \mathfrak{I}(\neg R \sqcup S) && (13)\\
\textit{equivalence} \quad & \mathfrak{I}(R \equiv S) = \mathfrak{I}((R \Rightarrow S) \sqcap (S \Rightarrow R)) && (14)
\end{aligned}$$

Section 2 informally described that relations need to have a type. A relation term $R \in \mathbb{R}$ with a type $\mathfrak{T}(R)$ has an interpretation. If it has no type, it is said to have a *type error* or to be *(semantically) incorrect*. An end user might call a relation term without a type *nonsense*. Relation terms with a type error are rejected with a proper feedback message.

By a relation declaration $r : A \sim B$ the user declares the existence of a relation symbol r with a type denoted $\mathfrak{T}(r) = A \sim B$. We use X, Y as type variables. By Definition 2, given some $r : A \sim B$, the user can only define a relation element $a\ r\ b \in POP$ if $a \in A$ and $b \in B$. $\mathfrak{T}(R)$ is inspired on Hattensperger's typing function for heterogeneous relation algebra [7].

Definition 4 (typing function)
Given some $\mathfrak{C}$, the partial typing function of relation terms is defined by

$$\mathfrak{T}(r) = A \sim B \quad , \text{if } r : A \sim B \in \text{REL} \tag{15}$$
$$\mathfrak{T}(I_A) = A \sim A \quad , \text{if } A \in \mathbb{C} \tag{16}$$
$$\mathfrak{T}(V_{A\times B}) = A \sim B \quad , \text{if } A, B \in \mathbb{C} \tag{17}$$
$$\mathfrak{T}(\neg R) = \mathfrak{T}(R) \quad , \text{if } \mathfrak{T}(R) \text{ is defined} \tag{18}$$
$$\mathfrak{T}(R^{\sqcup}) = B \sim A \quad , \text{if } \mathfrak{T}(R) = A \sim B \tag{19}$$
$$\mathfrak{T}(R \sqcup S) = \mathfrak{T}(R) \quad , \text{if } \mathfrak{T}(R) = \mathfrak{T}(S) \tag{20}$$
$$\mathfrak{T}(R \fatsemi S) = A \sim C \quad , \text{if } \mathfrak{T}(R) = A \sim B, \mathfrak{T}(S) = B \sim C \tag{21}$$

3.3 Type System

Only type correct expressions are allowed in Ampersand scripts. Ampersand allows overloading of relation symbols. This means that relation term names need not be unique. Overloading is necessary for practical reasons only, in order to give users more freedom to choose names. In particular, requirements engineers may choose short uniform names, such as *aggr*, *has*, *in*, to represent different relations for different types. The type system deduces all possible types for any given relation term name. If there are multiple possibilities, a type error is given. The script writer can disambiguate any relation term name by adding type information explicitly.

Ampersand also allows that two different concepts overload an atom symbol, i.e., an $a \in A$ has an interpretation different from $a \in B$. Atoms are only interpreted in the context of typed relation terms, preventing an ambiguous identity of atoms. In practice, business concepts may be overlapping, e.g., `'Peter'` the `Person` has the same practical identity as `'Peter'` the `Auctioneer`. In the companion paper, Van der Woude and Joosten [15] embed generalization of concepts in Ampersand. They introduce supertype relations (ϵ), called embeddings, to define total extensions of the partial (heterogeneous) relational operators except for the complement. These extensions give requirements engineers a type-controlled freedom to express themselves in relations on sub- or superconcepts. The use of negation in business rules with generalization requires further investigation.

The requirements engineer declares the supertype relation between subconcept A and superconcept B as $\epsilon : A \sim B \in REL$.

Altogether, the problem that the type system must solve is twofold. The type system must deduce one relation term from a relation term name, and check the type of relation terms simultaneously. We use R', S', T' as variables to denote relation term names.

The type function $\mathfrak{T}'(R')$ is based on the partial typing function $\mathfrak{T}(R)$. The type system examines $\mathfrak{T}'(R')$ to mark the name R' as bound to a relation, ambiguous, undeclared or undefined.

- If some r is not declared with a type in REL, then r is said to be *undeclared*.
- If some R' can only refer to one type correct relation term R, then R' is said to *bind to* R.
- If some R' does not refer to any type correct relation term, then R' is said to be *undefined*.
- If some R' can refer to more than one type correct relation term, i.e., an *alternative*, then R' is said to be *ambiguous*.

Definition 5 (type function)
Given some $\mathfrak{C}$, the type function of relation term names is defined by

$$\mathfrak{T}'(r) = \{A \sim B \mid (r : A \sim B) \in \text{REL } \} \tag{22}$$
$$\mathfrak{T}'(I_A) = \{A \sim A\} \tag{23}$$
$$\mathfrak{T}'(V_{A\times B}) = \{A \sim B\} \tag{24}$$
$$\mathfrak{T}'(R'_X) = \mathfrak{T}'(R') \cap \{X\} \tag{25}$$
$$\mathfrak{T}'(\neg R') = \mathfrak{T}'(R') \tag{26}$$
$$\mathfrak{T}'(R'^{\sqcup}) = \{B \sim A | A \sim B \in \mathfrak{T}'(R')\} \tag{27}$$
$$\mathfrak{T}'(R' \sqcup S') = \mathfrak{T}'(R') \cap \mathfrak{T}'(S') \tag{28}$$
$$\mathfrak{T}'(R' \fatsemi S') = \{A \sim C \mid \sharp\{B \mid A \sim B \in \mathfrak{T}'(R'), B \sim C \in \mathfrak{T}'(S')\} = 1\} \tag{29}$$

Rule 25 enables the requirements engineer to disambiguate an ambiguous relation term name with type information. Rule 29 ensures that if $R' \fatsemi S'$ yields more than one alternative with type $A \sim C$, then $A \sim C \notin \mathfrak{T}'(R' \fatsemi S')$.

This type system is not monotone, i.e., a well typed program can become ill-typed by adding rules for different, independent relations. When this happens to a requirements engineer, he will notice that the introduction of a new relation forces him to add more type information to other parts in his script.

4 Feedback System

The type system is embedded in the feedback system of the Ampersand compiler. The feedback system must give an error message if and only if a relation term name cannot be bound to a relation term. The error messages must be concise, i.e., correct and kept short, precise and relevant. The quality of the feedback deserves attention, because it lets students focus on learning rule-based design.

The feedback system is implemented in Haskell [11] based on an attribute grammar [6] Haskell library developed at Utrecht University. The feedback system checks all relation term names R' used in any statement in RUL or POP, given the conceptual structure of $\mathfrak{C}$ represented by REL. REL is an inherited attribute called *context structure*. Within the scope of a statement, given the context structure, R' is bound, ambiguous, undeclared or undefined.

A bound R' implies the existence of one suitable alternative for any subname S' of R'. The reader may verify this surjective function from subname S' to bound R' in rule 22-29. However, $\mathfrak{T}'(S')$ may yield more than one alternative. The feedback system uses two attributes to determine the type of any S' within the scope of R'. $\mathfrak{T}'(S')$ is a synthesized attribute called *pre-type*. Some $X \in \mathfrak{T}'(S')$ is an inherited attribute called *automatic type directive*. The automatic type directive is based on Definition 5.

For example, consider two statements a rule and a relation element, both yielding relation *rel1*:

$$rel1 \sqcup rel2 \in RUL$$

$$\texttt{'atom1'}\ rel1\ \texttt{'atom2'} \in POP$$

where

$$REL = \{rel1 : \texttt{Cpt1} \sim \texttt{Cpt2}, rel1 : \texttt{Cpt1} \sim \texttt{Cpt3}, rel2 : \texttt{Cpt1} \sim \texttt{Cpt2}\}$$

The rule binds to a typed relation term, although $\mathfrak{T}'(rel1)$ yields more than one alternative:

$$\begin{aligned}
\mathfrak{T}'(rel1) &= \{\texttt{Cpt1} \sim \texttt{Cpt2},\ \texttt{Cpt1} \sim \texttt{Cpt3}\} && \text{(rule 22, ambiguous)}\\
\mathfrak{T}'(rel2) &= \{\texttt{Cpt1} \sim \texttt{Cpt2}\} && \text{(rule 22, bound)}\\
\mathfrak{T}'(rel1 \sqcup rel2) &= \mathfrak{T}'(rel1) \cap \mathfrak{T}'(rel2) && \text{(rule 28)}\\
&= \{\texttt{Cpt1} \sim \texttt{Cpt2},\ \texttt{Cpt1} \sim \texttt{Cpt3}\} \cap \{\texttt{Cpt1} \sim \texttt{Cpt2}\}\\
&= \{\texttt{Cpt1} \sim \texttt{Cpt2}\} && \text{(bound)}\\
&= \mathfrak{T}'((rel1 \sqcup rel2)_{\texttt{Cpt1}\sim\texttt{Cpt2}}) && \text{(auto)}\\
&= \mathfrak{T}'((rel1_{\texttt{Cpt1}\sim\texttt{Cpt2}} \sqcup rel2_{\texttt{Cpt1}\sim\texttt{Cpt2}})_{\texttt{Cpt1}\sim\texttt{Cpt2}}) && \text{(auto)}\\
\mathfrak{T}'(rel1_{\texttt{Cpt1}\sim\texttt{Cpt2}}) &= \{\texttt{Cpt1} \sim \texttt{Cpt2}\} && \text{(rule 22, bound)}\\
\mathfrak{T}'(rel2_{\texttt{Cpt1}\sim\texttt{Cpt2}}) &= \{\texttt{Cpt1} \sim \texttt{Cpt2}\} && \text{(rule 22, bound)}\\
& && \text{(done)}
\end{aligned}$$

The relation element is ambiguous, because $\mathfrak{T}'(rel1)$ yields more than one alternative:

$$\begin{aligned}
\mathfrak{T}'(rel1) &= \{\texttt{Cpt1} \sim \texttt{Cpt2},\ \texttt{Cpt1} \sim \texttt{Cpt3}\} && \text{(rule 22, ambiguous)}\\
& && \text{(done)}
\end{aligned}$$

The requirements engineer should have specified a type directive

$$\texttt{'atom1'}\ rel1_{\texttt{Cpt1}\sim\texttt{Cpt2}}\ \texttt{'atom2'} \in POP$$

or

$$\texttt{'atom1'}\ \mathit{rel1}_{\texttt{Cpt1}\sim\texttt{Cpt3}}\ \texttt{'atom2'} \in \mathit{POP}$$

Detailed information can be obtained through synthesized attributes of the attribute grammar. Three synthesized attributes are defined in our implementation. One attribute holds either $\mathfrak{T}(R)$ if R' is bound to R and $\mathfrak{T}(R)$ is defined, or an error message as described in Section 4.1 otherwise. Another attribute holds a fully typed relation term name $R_{\mathfrak{T}(R)}$ if the previous attribute holds $\mathfrak{T}(R)$, and is undefined otherwise. The third attribute holds an extensive LaTeX report for complete detail on type errors or binding relation term names. Such a report contains equational traces like presented for the examples in this section. This report is generated by the Ampersand compiler.

4.1 Error Messages

If a relation term has an error, then an error message must be composed. We have designed templates for error messages which relate to the type system rule at which an error first occurs. The error messages are defined short but complete and specific. The templates are presented in the same order of precedence as the operators they relate to. If subterms of a relation term have error messages, then the error message of the relation term is the union of all the error messages of subterms.

relation undeclared If $\mathfrak{T}'(r)$ yields no types, then there is no relation declaration for r.

$$\textit{relation undeclared: } r$$

relation undefined Requirements engineers may use the unique name of some R e.g. ${R'}_X$. This may cause different kinds of errors which are checked in the same chronological order as described here.

If $X = A \sim B$ and A or B does not occur in the type signature of any relation declaration, then there is probably a typo in the concept name.

$$\textit{unknown concept: } A, \text{ or}$$

$$\textit{unknown concept: } B, \text{ or}$$

$$\textit{unknown concepts: } A \textit{ and } B$$

If $X \notin \mathfrak{T}'(R')$, then there is no R such that $\mathfrak{T}(R) = X$. If $R' = r$ then r_X is undeclared.

$$\textit{relation undeclared: } r_X$$

In all other cases:

$$\textit{relation undefined: } r_X$$

$$\textit{possible types are: } \mathfrak{T}'(R')$$

incompatible/ambiguous composition Let $\mathfrak{T}'(R')$ and $\mathfrak{T}'(S')$ yield alternatives. If $\mathfrak{T}'(R' \fatsemi S')$ yields no types, then there is no alternative for $R' \fatsemi S'$. In case $\sharp\{B \mid A \sim B \in \mathfrak{T}'(R'), B \sim C \in \mathfrak{T}'(S')\} = 0$, then R' and S' are incompatible for composition.

$$\textit{incompatible composition: } R' \fatsemi S'$$

$$\textit{possible types of } R' \textit{: } \mathfrak{T}'(R')$$

$$\textit{possible types of } S' \textit{: } \mathfrak{T}'(S')$$

In case $\sharp\{B \mid A \sim B \in \mathfrak{T}'(R'), B \sim C \in \mathfrak{T}'(S')\} > 1$, then the composition of R' and S' is ambiguous.

$$\textit{ambiguous composition: } R' \fatsemi S'$$

$$\textit{possible types of } R' \textit{: } \{A \sim B \mid A \sim B \in \mathfrak{T}'(R'), B \sim C \in \mathfrak{T}'(S')\}$$

$$\textit{possible types of } S' \textit{: } \{B \sim C \mid A \sim B \in \mathfrak{T}'(R'), B \sim C \in \mathfrak{T}'(S')\}$$

incompatible comparison Let $\mathfrak{T}'(R')$ and $\mathfrak{T}'(S')$ yield alternatives. If $\mathfrak{T}'(R' \sqcup S')$ yields no types, then R' and S' are incompatible for comparison.

$$\textit{incompatible comparison: } R' \sqcup S'$$

$$\textit{possible types of } R' \textit{: } \mathfrak{T}'(R')$$

$$\textit{possible types of } S' \textit{: } \mathfrak{T}'(S')$$

ambiguous type If R' is ambiguous within the scope of a statement, then the ambiguity is reported as an error.

$$\textit{ambiguous relation: } R'$$

$$\textit{possible types: } \mathfrak{T}'(R')$$

4.2 Demonstration

Let us demonstrate a typical constraint that some relation is contained within another relation.

$$(\mathit{rel1} \sqcap \mathit{rel2} \Rightarrow \mathit{rel0}) \in \mathit{RUL}$$

where

$$\begin{aligned} \mathit{REL} &= \{\mathit{rel1} : \texttt{Cpt1} \sim \texttt{Cpt2}, \mathit{rel2} : \texttt{Cpt3} \sim \texttt{Cpt4}\} \\ R \sqcap S &= \neg(\neg R \sqcup \neg S) \\ R \Rightarrow S &= \neg R \sqcup S \end{aligned}$$

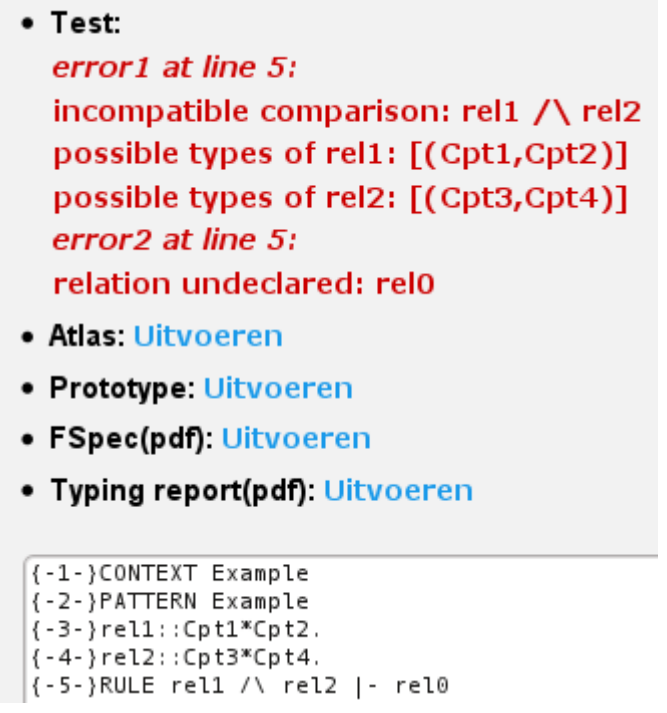

Fig. 1. compiler screen snippet with type error (Dutch)

The rule contains two errors. Both will be mentioned in the error message as depicted in Figure 1.

These messages provide the requirements engineer with relevant and sufficient information which they understand. For complete detail, the requirements engineer requests a LaTeXreport containing a trace like:

$$
\begin{aligned}
\mathfrak{T}'(\mathit{rel1}) &= \{\texttt{Cpt1} \sim \texttt{Cpt2}\} && \text{(rule 22, bound)}\\
\mathfrak{T}'(\neg \mathit{rel1}) &= \mathfrak{T}'(\mathit{rel1}) && \text{(rule 26)}\\
&= \{\texttt{Cpt1} \sim \texttt{Cpt2}\} && \text{(bound)}\\
\mathfrak{T}'(\mathit{rel2}) &= \{\texttt{Cpt3} \sim \texttt{Cpt4}\} && \text{(rule 22, bound)}\\
\mathfrak{T}'(\neg \mathit{rel2}) &= \mathfrak{T}'(\mathit{rel2}) && \text{(rule 26)}\\
&= \{\texttt{Cpt3} \sim \texttt{Cpt4}\} && \text{(bound)}\\
\mathfrak{T}'(\neg \mathit{rel1} \sqcup \neg \mathit{rel2}) &= \mathfrak{T}'(\neg \mathit{rel1}) \cap \mathfrak{T}'(\neg \mathit{rel2}) && \text{(rule 28)}\\
&= \{\} && \text{(undefined)}\\
\mathfrak{T}'(\mathit{rel0}) &= \{\} && \text{(rule 22, undeclared)}\\
& && \text{(done)}
\end{aligned}
$$

5 Conclusions and Further Research

In this paper we have introduced Ampersand by means of an abstract syntax, semantics, and a type system, and motivated this from a practical use of specifying information systems. Ampersand allows requirements engineers to represent a situation specific language by relation declarations and express truths in it. True facts are represented by relation elements and requirements are represented by rules. Rules are terms in a syntactic calculus of relations based on Tarski's axioms. The relation terms are enriched with typing arguments to enable the filtering of rules without a type, and relation elements of undeclared relations. Ampersand allows overloading of relation and atom symbols and generalization of concepts, in order to give requirements engineers a more natural syntax. The type system filters relation terms with an undefined or ambiguous type.

Ampersand generates precise feedback on type errors. This feature keeps requirements engineers from implementing rules which will never hold. The feedback is generated by an attribute grammar on relation term names. The concreteness and relevance of the generated feedback is meant to focus students on learning rule-based design. So far, the practical experience with students is encouraging. Systematic experimentation and evaluation of the learning process is scheduled in the nearby future.

For the purpose of systematic research and learning, Ampersand development is moving from an ASCII text editor towards an Integrated Development Environment (IDE) for education. We consider this IDE as an information system for the business process of system development with Ampersand. As such it is described within Ampersand, and a prototype web-based information system is generated with the compiler.

The current information system for system development with Ampersand, called *Atlas*, is read-only. The Atlas is the prototype generated from an Ampersand script describing the Ampersand language. The compiler loads type correct Ampersand scripts, and derived data like rule violations and pictures into the database of this system. Through the generated web-interface students can explore the design of their context $\mathfrak{C}$ by clicking on related elements. Student behaviour is stored in the database and may serve as input to prove intended didactical improvements on the system.

The Atlas will become editable in controlled phases. First students will be able to edit the *POP* of their context. Certain rules, e.g., syntax rules, may be violated by changes in *POP*. These rules need to be implemented as business rules such that the generated IDE gives feedback on violations of these rules. After *POP* has successfully become editable, *REL* followed by *RUL* will become editable resulting in a rule-based educational environment to develop with Ampersand.

The use of Ampersand in practice is also encouraging. Large information system projects in the Dutch government have already been designed in Ampersand, and one has been realized.

Further research in the Ampersand project focuses on:

- Publishing the software and disclosing further results in the open source domain (ampersand.sourceforge.net).

- Refining the didactics of teaching Ampersand as a rule based design method.
- Design for large IT projects in industry.
- Generate software prototypes from Ampersand scripts.
- Automate the design of web services.

References

1. Brink, C., Schmidt, R.A.: Subsumption computed algebraically. Computers and Mathematics with Applications 23(2-5), 329–342 (1992)
2. Brink, C., Kahl, W., Schmidt, G. (eds.): Relational methods in computer science. Advances in Computing. Springer, New York (1997)
3. Codd, E.F.: A relational model of data for large shared data banks. Communications of the ACM 13(6), 377–387 (1970)
4. Date, C.J.: What not how: the business rules approach to application development. Addison-Wesley Longman Publishing Co., Inc., Boston (2000)
5. Desharnais, J.: Basics of relation algebra, `http://www2.ift.ulaval.ca/~Desharnais/Recherche/Tutoriels/TutorielRelMiCS10.pdf`
6. Dijkstra, A., Swierstra, S.D.: Typing haskell with an attribute grammar. In: Vene, V., Uustalu, T. (eds.) AFP 2004. LNCS, vol. 3622, pp. 1–72. Springer, Heidelberg (2005)
7. Hattensperger, C., Kempf, P.: Towards a formal framework for heterogeneous relation algebra. Inf. Sci. 119(3-4), 193–203 (1999)
8. Jackson, D.: A comparison of object modelling notations: Alloy, UML and Z. Tech. rep. (1999), `http://sdg.lcs.mit.edu/publications.html`
9. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
10. Maddux, R.D.: Relation Algebras. Studies in logic, vol. 150. Elsevier, Iowa (2006)
11. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press, Cambridge (2003)
12. Sanderson, J.G.: A Relational Theory of Computing. LNCS, vol. 82. Springer, New York (1980)
13. Schmidt, G., Hattensperger, C., Winter, M.: Heterogeneous Relation Algebra. In: Relational Methods in Computer Science. Advances in Computing, ch. 3, pp. 39–53. Springer, New York (1997)
14. Spivey, J.M.: The Z Notation: A reference manual, 2nd edn. International Series in Computer Science. Prentice Hall, New York (1992)
15. van der Woude, J., Joosten, S.: Relational heterogeneity relaxed by subtyping, (submitted 2011)