# Strategic term rewriting and its application to a VDM-SL to SQL conversion

T.L. Alves, P.F. Silva, J. Visser and J.N. Oliveira

Dep. Informática, Universidade do Minho,
Campus de Gualtar, 4700-320 Braga, Portugal,

**Abstract.** We constructed a tool, called VooDooM, which converts datatypes in VDM-SL into SQL relational data models. The conversion involves transformation of algebraic types to maps and products, and pointer introduction.
The conversion is specified as a theory of refinement by calculation. The implementation technology is strategic term rewriting in Haskell, as supported by the Strafunski bundle. Due to these choices of theory and technology, the road from theory to practise is straightforward.
**Keywords:** Strategic term rewriting, program calculation, VDM, SQL.

## 1 Introduction

The information system community is indebted to Codd for his pioneering work on the foundations of the relational data model [9]. Since then, relational database theory has been thoroughly studied [24, 34, 12]. At the heart of this we find *normalization*, a theory whereby efficient collections of (relational) files are derived from the original design, which can be encoded in a data-processing language such as SQL [16].

Functional dependency theory and normalization deviate from standard model-oriented formal specification and reification techniques [17, 10]. In the latter, designs start from abstract models which are abstract enough to dispense with normalization. Does one arrive at similar database designs by using data reification techniques?

References [29–32] address a formal calculus which has been put forward as an alternative to standard normalization theory, by framing database design into the wider area of data refinement [17]. Data models, such as described by E-R diagrams, for instance, are turned into systems of equations involving set-theoretic notions such as finite mappings, sets, and sequences. Integrity constraints and business rules are identified with abstraction invariants [25] and datatype invariants [17], respectively, whose structural synthesis (analysis) by calculation is at the core of the calculus.

The main purpose of this paper is to describe the design of a *database schema calculator* which, inspired by [32], infers SQL relational meta-data from abstract data models specified in the ISO standard VDM-SL formal modelling notation [10]. The calculus is implemented using Haskell-based strategic term rewriting [23], and embedded in a full fledged source code processing tool following a grammar-centered approach to language tool development [18]. This database calculator, named VooDooM, is being used in the *Information Knowledge Fusion* ($\Sigma$!2235) project, to generate the database of a knowledge representation management system.

### 1.1 Related work

Most work on formal methods in relational database design is concerned with formal models of relational data. This interest dates back to (at least) [6], where a formalization of a relational data model is given using the VDM notation.

The formal specification and design of a program implementing simple update operations on a binary relational database called NDB is described in [38]. This single level description of NDB is the starting point of [11], where a case study in the modular structuring of this "flat" specification is presented. The authors present a second specification which makes use of an $n$-ary relation module, and a third one which uses an $n$-ary relation module with type and normalization constraints. They demonstrate the reusability of their modules, and also outline specifications of an $n$-ary relational database with normalization constraints, and an $n$-ary relational database with a two-level type hierarchy and no normalization constraints. However, their emphasis is on the modularization techniques adopted to organize VDM specifications into modules.

Samson and Wakelin [33] present a comprehensive survey about the use of algebraic methods to specify databases. They compare a number of approaches according to the features covered and enumerate some features not normally covered by such methods.

Barros [5] describes an extension to the traditional database design aimed at formalizing the development of (relational) database applications. A general method for the specification of relational database applications using Z is presented. A prototype is built to support the method. It provides for editing facilities and is targeted at the DBPL database management system.

The purpose of Baluta [4] is to rigorously specify the basic features of the relational data model version 2 (RM/V2) as defined by Codd [8], using the Z language.

More recently, Necco [26] exploits aspects of *data processing* which are functional in nature and can take advantage of recent developments in the area of *generic functional programming* and calculi. Generic Haskell is used to animate a generic model of a subset of the standard relational database calculus, written in the style of model-oriented formal specification.

## 2 Strategic term rewriting

In traditional term rewriting, one can distinguish the rewriting *equations* of a particular term rewriting system (TRS) from the *strategy* that is used to apply these equations to an input term. Most commonly, term rewriting environments have a fixed rewriting strategy, such as the leftmost-innermost strategy. In some rewriting environments, for instance those where the equations may be governed by conditions and may be stratified into default and regular equations, more sophisticated strategies may be employed. But in any case, these strategies are fixed, i.e. hard-wired into the environment.

By contrast, strategic term rewriting generalizes the traditional term rewriting paradigm by making rewriting strategies programmable, just as the equations are. Among the first rewriting environments to offer such programmable rewriting strategies are Stratego [35] and the Rewriting Calculus [7]. Such environments offer a small set of basic strategy combinators, which can be combined with each other and with rewriting equations to construct term rewriting systems with arbitrarily complex strategies.

**Combinators**

| | | |
|---|---|---|
| $s ::=$ | $\mathsf{id}$ | Identity strategy |
| $\mid$ | $\mathsf{fail}$ | Failure strategy |
| $\mid$ | $\mathsf{seq}(s, s)$ | Sequential composition |
| $\mid$ | $\mathsf{choice}(s, s)$ | Left-biased choice |
| $\mid$ | $\mathsf{all}(s)$ | All immediate components |
| $\mid$ | $\mathsf{one}(s)$ | One immediate component |
| $\mid$ | $\mathsf{adhoc}(s, a)$ | Type-based dispatch |

**Notation**

| | | |
|---|---|---|
| $d$ | ... | data |
| $c$ | ... | data constructors |
| $\overline{d}$ | ... | data with failure "$\uparrow$" |
| $a$ | ... | type-specific actions |
| $s$ | ... | strategies |
| $a@d$ | ... | application of $a$ to $d$ |
| $s@d$ | ... | application of $s$ to $d$ |
| $d \Rightarrow \overline{d}$ | ... | big-step semantics |
| $a : t$ | ... | type handled by $a$ |
| $d : t$ | ... | type of a datum $d$ |
| $[d]$ | ... | indivisible data |
| $c(d_1 \cdots d_n)$ | ... | compound data |

**Meaning**

| | |
|---|---|
| $\mathsf{id}@d$ | $\Rightarrow d$ |
| $\mathsf{fail}@d$ | $\Rightarrow \uparrow$ |
| $\mathsf{seq}(s, s')@d$ | $\Rightarrow \overline{d}$ if $s@d \Rightarrow d' \wedge s'@d' \Rightarrow \overline{d}$ |
| $\mathsf{seq}(s, s')@d$ | $\Rightarrow \uparrow$ if $s@d \Rightarrow \uparrow$ |
| $\mathsf{choice}(s_1, s_2)@d$ | $\Rightarrow d'$ if $s_1@d \Rightarrow d'$ |
| $\mathsf{choice}(s_1, s_2)@d$ | $\Rightarrow \overline{d}$ if $s_1@d \Rightarrow \uparrow \wedge s_2@d \Rightarrow \overline{d}$ |
| $\mathsf{all}(s)@[d]$ | $\Rightarrow [d]$ |
| $\mathsf{all}(s)@c(d_1 \cdots d_n)$ | $\Rightarrow c(d_1' \cdots d_n')$ if $s@d_1 \Rightarrow d_1', \ldots, s@d_n \Rightarrow d_n'$ |
| $\mathsf{all}(s)@c(d_1 \cdots d_n)$ | $\Rightarrow \uparrow$ if $\exists i.\ s@d_i \Rightarrow \uparrow$ |
| $\mathsf{one}(s)@[d]$ | $\Rightarrow \uparrow$ |
| $\mathsf{one}(s)@c(d_1 \cdots d_n)$ | $\Rightarrow c(\cdots d_i' \cdots)$ if $\exists i.\ s@d_1 \Rightarrow \uparrow \wedge \cdots \wedge s@d_{i-1} \Rightarrow \uparrow \wedge s@d_i \Rightarrow d_i'$ |
| $\mathsf{one}(s)@c(d_1 \cdots d_n)$ | $\Rightarrow \uparrow$ if $s@d_1 \Rightarrow \uparrow, \ldots, s@d_n \Rightarrow \uparrow$ |
| $\mathsf{adhoc}(s, a)@d$ | $\Rightarrow a@d$ if $a : t$ and $d : t$ |
| $\mathsf{adhoc}(s, a)@d$ | $\Rightarrow s@d$ if $a : t \wedge d : t' \wedge t \neq t'$ |

**Identities**

[unit] $s \equiv \mathsf{seq}(\mathsf{id}, s) \equiv \mathsf{seq}(s, \mathsf{id}) \equiv \mathsf{choice}(\mathsf{fail}, s) \equiv \mathsf{choice}(s, \mathsf{fail})$

[zero] $\mathsf{fail} \equiv \mathsf{seq}(\mathsf{fail}, s) \equiv \mathsf{seq}(s, \mathsf{fail}) \equiv \mathsf{one}(\mathsf{fail})$

[skip] $\mathsf{id} \equiv \mathsf{choice}(\mathsf{id}, s) \equiv \mathsf{all}(\mathsf{id})$

[nested type dispatch]

$\mathsf{adhoc}(\mathsf{adhoc}(s, a), a') \equiv \mathsf{adhoc}(s, a')$ if $a : t \wedge a' : t$

$\mathsf{adhoc}(\mathsf{adhoc}(s, a), a') \equiv \mathsf{adhoc}(\mathsf{adhoc}(s, a'), a)$ if $a : t \wedge a' : t' \wedge t \neq t'$

$\mathsf{adhoc}(\mathsf{adhoc}(\mathsf{fail}, a), a') \equiv \mathsf{choice}(\mathsf{adhoc}(\mathsf{fail}, a), \mathsf{adhoc}(\mathsf{fail}, a'))$ if $a : t \wedge a' : t' \wedge t \neq t'$

**Fig. 1.** Specification of a guideline set of basic strategy combinators.

Figure 1 shows a set of such basic strategy combinators, along with their operational semantics.

Using the basic strategy combinators, more elaborate ones can easily be constructed. Consider for instance the following definitions:

$$
\begin{aligned}
&\mathit{try}(s) &&= \mathsf{choice}(s, \mathsf{id})\\
&\mathit{repeat}(s) &&= \mathit{try}(\mathsf{seq}(s, \mathit{repeat}(s)))\\
&\mathit{full\_topdown}(s) &&= \mathsf{seq}(s, \mathsf{all}(\mathit{full\_topdown}(s)))\\
&\mathit{innermost}(s) &&= \mathsf{seq}(\mathsf{all}(\mathit{innermost}(s)), \mathit{try}(\mathsf{seq}(s, \mathit{innermost}(s))))
\end{aligned}
$$

The $try$ combinator takes a potentially failing strategy as argument, and attempts to apply it. When failure occurs, the identity strategy id is used to recover. The $repeat$ combinator repeatedly applies its argument strategy, until it fails. The $full\_topdown$ combinator applies its argument once to every node in a term, in pre-order. Finally, the $innermost$ strategy applies its argument in left-most innermost fashion to a term, until it is not applicable anywhere anymore, i.e. until a fixpoint is reached.

The challenge of combining strategic term rewriting with strong typing was first met by the Haskell-based Strafunski bundle [23], which we will use in this paper, and the Java-based JJTraveler framework [36, 19]. A formal semantics of typed strategic programming was constructed subsequently [20]. Further generalizations were provided in the Haskell context [22, 21].

Strategic term rewriting has several benefits over traditional term rewriting. The most important benefits derive from the fact that many applications require rewrite equations that together do not form a confluent and terminating TRS. A program refactoring system, for instance, may require equations both for "extract method" and for "inline method". A document processing system may include equations that change mark-up only inside the context of certain document tags. In a traditional term rewriting environment, the only option to obtain sufficient control over when and where equations are applied, is to switch to so-called 'functional style'. This means that every rewrite rule $t \mapsto \ldots s \ldots$ is reformulated to include function symbols to control rewriting: $f(t) \mapsto \ldots g(s) \ldots$. This way, the rewriting strategy in fact becomes explicit in the additional function symbols, but is thoroughly entangled with the rewrite equations. In strategic programming, the rewrite equations can stay as they are, the strategy can be specified separately, and both equations and strategies can be used and reused in different combinations to obtain different TRSs. So, apart from full control over when and where equations are applied, strategic rewriting enhances separation of concerns, reusability, and understandability.

In this paper, we will rely on strategic term rewriting to cleanly separate the individual conversion rules from the strategy of applying them to the abstract syntax terms. We will use the Strafunski bundle as strategic term rewriting environment in which to implement the conversion tool.

## 3 Database design by calculation

The calculation method which underlies our VDM-SL to SQL conversion tool finds its roots in a "data refinement by calculation" strategy which originated in [29, 30] and has been focussed towards relational database design more recently [31, 32]. Reference [28] describes its application to reverse engineering legacy databases.

### 3.1 Abstraction and representation

The calculus consists of inequations of the form $A \leq B$ (read: "*data type B implements, or refines data type A*") which abbreviates the fact that there is a surjective, possibly partial function $A \xleftarrow{F} B$ (the *abstraction relation*) and an injective, total relation

$A \xrightarrow{R} B$ (the *representation relation*) such that

$$F \cdot R = id_A \tag{1}$$

where $id_A$ is the identity function on datatype $A$. ($F$ is traditionally referred to as a *retrieve* function [17].) Since the equality $R = S$ of two relations $R$ and $S$ is bi-inclusion $R \subseteq S \wedge S \subseteq R$, we have two readings of equation (1): $id_A \subseteq F \cdot R$, which ensures that every inhabitant of the abstract datatype $A$ gets represented at $B$-level; and $F \cdot R \subseteq id_A$, which prevents "confusion" in the representation process:

$$\langle \forall\, b \in B, a \in A \;:\; b\, R\, a :\; \langle \forall\, a' \in A \;:\; a'\, F\, b :\; a' = a \rangle \rangle$$

("Never forget whom you are representing".)

Below we will present a series of particular $\leq$-equations which together specify a data model refinement calculus. The types of the refinement relations will be mapped onto rewrite rules in the implementation.

## 3.2 Preorder

It can be shown that $\leq$ is a preorder, reflexivity meaning that any datatype represents itself ($R = F = id$) and transitivity meaning that $\leq$-steps can be chained by sequentially composing abstractions and representations:

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"] & \wedge & B \arrow[r, bend left, "S"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "G"] & \Rightarrow & A \arrow[r, bend left, "S \cdot R"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "F \cdot G"] \end{tikzcd} \tag{2}$$

This suggests that one may *calculate* implementations from specifications

$$Spec = X \leq X' \leq X'' \leq \cdots \leq Imp$$

by adding implementation *details* in a controlled manner. This also makes sense wherever the representation of a parameter of a datatype needs to be promoted to the overall parametric datatype by *structural data refinement*:

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"] & \Rightarrow & \mathsf{F}\, A \arrow[r, bend left, "\mathsf{F}\, R"] \arrow[r, phantom, "\leq"] & \mathsf{F}\, B \arrow[l, bend left, "\mathsf{F}\, F"] \end{tikzcd} \tag{3}$$

where $\mathsf{F}$ is such a parametric type, e.g. `set of` $A$ in VDM-SL notation. (Technically, $\mathsf{F}$ is named a *relator* [3].) This is valid also for parametric types of higher arity, such as those of standard VDM-SL:

- binary product types $A \times B$ and $n$-ary ones $\prod_{i=0}^{n} A_i$, which can be specified in VDM-SL as (nested) tuples or via record types, (semantically equivalent modulo selectors). E.g. `A*B` or `compose AB of a: A b: B end`, respectively.
- sum types $A + B$, which in VDM-SL are specified by writing `A | B` for suitably specified (disjoint) $A$ and $B$, extensible to finitary sums $\sum_{i=0}^{n} A_i$.
- finite mappings $A \rightharpoonup B$, written `map` $A$ `to` $B$ in VDM-SL, in which case the abstraction of the domain datatype is required to be injective (otherwise the outcome may not be a mapping).

## 3.3 Conversion laws

It is often the case that the abstraction (resp. representation) relation is a (total) function, in which case it is an *injection* (resp. *surjection*). As an example of this we present law

$$\begin{tikzcd} A^\star \arrow[r, bend left, "seq2index"] \arrow[r, phantom, "\leq"] & I\!N \rightharpoonup A \arrow[l, bend left, "list"] \end{tikzcd} \tag{4}$$

which indexes a finite sequence, for instance,

$$seq2index([a,b,a]) = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto a\}$$
$$list(\{11 \mapsto a, 12 \mapsto b, 33 \mapsto a\}) = [a,b,a]$$

A more structural law is

$$\begin{tikzcd} A \rightharpoonup (B+C) \arrow[r, bend left, "uncojoin"] \arrow[r, phantom, "\leq"] & (A \rightharpoonup B) \times (A \rightharpoonup C) \arrow[l, bend left, "cojoin"] \end{tikzcd} \tag{5}$$

whereby *mappings of sums* are represented as *products of mappings*. (Definitions for $cojoin$ and $uncojoin$ are easy to guess.) In a situation where the abstraction is also a representation and vice-versa we have an isomorphism $A \cong B$, a special case of the $\leq$-law which works in both directions. For example, the abstraction/representation pair of the following isomorphism

$$\begin{tikzcd} A \times (B+C) \arrow[r, bend left, "distr"] \arrow[r, phantom, "\cong"] & (A \times B) + (A \times C) \arrow[l, bend left, "undistr"] \end{tikzcd} \tag{6}$$

(product distributes through sum) is well-known from set-theory.

The VDM-SL finite mapping *dom* function witnesses a very useful isomorphism between finite sets and partial finite mappings,

$$\begin{tikzcd} 2^A \arrow[r, bend left, "set2fm"] \arrow[r, phantom, "\cong"] & A \rightharpoonup 1 \arrow[l, bend left, "dom"] \end{tikzcd} \tag{7}$$

which expresses the equivalence between data models `set of A` and `map A to nil`. (The inhabitants of $A \rightharpoonup 1$, often called *right-conditions* [13], obey a number of interesting properties.) Another basic isomorphism tells us how "singleton" finite mappings disguise "pointers" (guess *opt-intro* and *opt-elim*):

$$\begin{tikzcd} A+1 \arrow[r, bend left, "opt\text{-}intro"] \arrow[r, phantom, "\cong"] & 1 \rightharpoonup A \arrow[l, bend left, "opt\text{-}elim"] \end{tikzcd} \tag{8}$$

The following isomorphism law

$$\begin{tikzcd}(B+C) \rightharpoonup A \arrow[r, bend left, "unpeither"] & \cong & (B \rightharpoonup A) \times (C \rightharpoonup A) \arrow[ll, bend left, "peither"]\end{tikzcd} \tag{9}$$

is a companion of (5).

Two important $\leq$-rules from [32] are still missing from our catalog: the representation function of one of these,

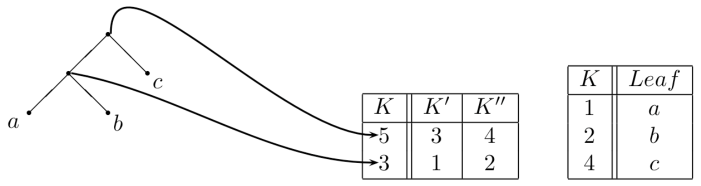
$$\begin{tikzcd}A \rightharpoonup (B \times (C \rightharpoonup D)) \arrow[rr, bend left, "unnjoin"] & \leq & (A \rightharpoonup B) \times (A \times C \rightharpoonup D) \arrow[ll, bend left, "njoin"]\end{tikzcd} \tag{10}$$

enables us to infer *composite keys* out of nested finite mappings. (See [30, 31] concerning abstraction $njoin$ and representation $unnjoin$.) In the abstraction direction (from right to left) it merges two tables which share a common (sub)key.

The other rule missing has to do with datatype "derecursivation". Suppose we are given a recursive datatype definition $\mu\mathsf{F} \cong \mathsf{F}\,\mu\mathsf{F}$ where $\mathsf{F}$ is polynomial [3, 30]. Then any "tree" in $\mu\mathsf{F}$ can be represented by a "heap" and a "pointer" to it,

$$\begin{tikzcd}\mu\mathsf{F} \arrow[rr, bend left, "rec\text{-}elim"] & \leq & (K \rightharpoonup \mathsf{F}\,K) \times K \arrow[ll, bend left, "rec\text{-}intro"]\end{tikzcd} \tag{11}$$

for $K$ a data type of *"heap addresses"*, *keys* or *"pointers"*, such that $K \cong I\!N$. For example, the binary tree on the left-hand side of (12) below will be represented — via (11) followed by (5) — by address $5$ pointing at the tables on the right-hand side:



| $K$ | $K'$ | $K''$ |
|---|---|---|
| 5 | 3 | 4 |
| 3 | 1 | 2 |

| $K$ | $Leaf$ |
|---|---|
| 1 | $a$ |
| 2 | $b$ |
| 4 | $c$ |

(12)

See [30–32] for several important details we have to skip at this point about this *generic* data representation technique, in particular in what concerns the complex abstraction invariant imposed by (11), which requires "well-founded heaps".

## 3.4 Normal form

A pattern common to equations (4, 5, 7, 8, 9, 10 and 11) is that right-hand-sides do not involve functors other than product ($\times$) and finite mapping ($\rightharpoonup$). It so happens that these are exactly the functors admissible in the following abstract model

$$DB = \prod_{i=1}^{n} \left( \prod_{j=0}^{n_i} K_j \rightharpoonup \prod_{k=0}^{m_i} D_k \right) \tag{13}$$

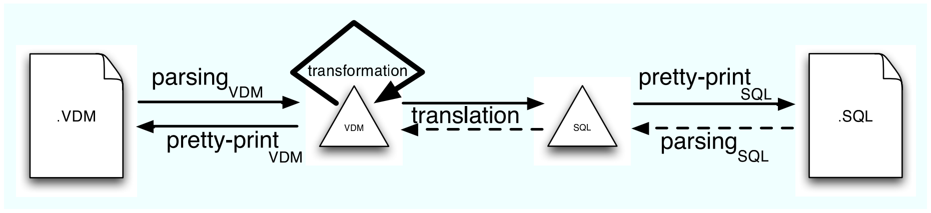


**Fig. 2.** Overall architecture of the VooDooM tool.

of a relational database, whereby every $db \in DB$ is a collection of $n$ relational tables (index $i = 1, n$) each of which is a mapping from a tuple of *keys* (index $j$) to a tuple of relevant *data* (index $k$). Wherever $m_i = 0$ we have $\prod_{k=0}^{0} D_k \cong 1$, meaning — via (7) — that we have a *finite set* of tuples in $\prod_{j=0}^{n_i} K_j$. (These are called *entity relationships* in the standard terminology.) Wherever $n_i = 0$ we are in presence of a singleton relational table. Last but not least, all $K_j$ and $D_k$ are "atomic" types, otherwise $db$ would fail first normal form (1NF) compliance [24].

To derive such normal forms, the above calculation laws can be used in combination with appropriate laws for commutativity and associativity of tuples, and laws for introduction and elimination of empty tuples. To avoid these additional bookkeeping laws, we can generalize law (10) to:

$$A \rightharpoonup (\prod_i B_i \times \prod_j (C_j \rightharpoonup D_j)) \quad \underset{g\text{-}unnjoin}{\overset{g\text{-}njoin}{\leq}} \quad (A \rightharpoonup \prod_i B_i) \times \prod_j (A \times C_j \rightharpoonup D_j) \tag{14}$$

In the implementation, we will make use of this generalization.

Thus, with this collection of calculation rules we are able to unravel (polynomial) recursive datatypes and decompose complex/nested mappings or sequences into tuples of simpler mappings, leading to models in relational normal form (13). In the upcoming section we will show how a term rewriting system can be constructed and implemented that performs such unraveling in a deterministic and confluent manner.

## 4 Design and Implementation of the VooDooM tool

This section describes the implementation of the VooDooM tool, which uses strategic term rewriting to apply the refinement laws described above to VDM-SL source code. The overall architecture of the tool is shown in Figure 2. The architecture mirrors the phases needed to tackle the problem:

1. Recognize a specification file written in VDM-SL and convert it to a format that can be used for processing: abstract syntax tree (AST);
2. Apply transformations to the AST to convert the input model into its relational equivalent; and
3. Output the transformed specification either as VDM-SL or to SQL concrete syntax.

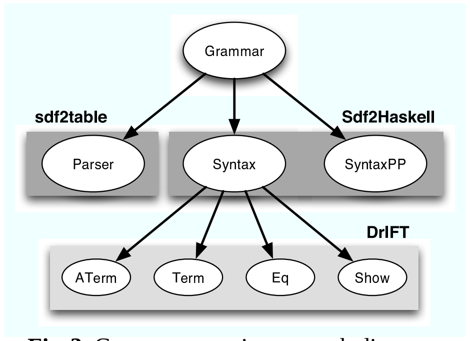


**Fig. 3.** Grammar-centric approach diagram.

To handle each of these steps, the following modules were developed:

**VDM-SL and SQL front-ends** Deal with the language issues, namely parsing, pretty-printing and abstract representation.

**Transformation engine** Receives a VDM-SL AST representing the original specification and applies the calculation laws in order to compute a relational model that refines it (also a VDM-SL AST).

**VDM-SL to SQL translator** Maps a relational model in VDM-SL AST format to an equivalent SQL AST.

In the upcoming sections we will describe the implementation of these modules in more detail. We will use the following specification of a tiny bank account management system (BAMS) as example input[1]:

```
types
  BAMS      = map AccId to Account;
  Account  :: H: set of AccHolder
              B: Amount;
  AccId     = seq of char;
  AccHolder = seq of char;
  Amount    = int
```

### 4.1 Technology

We followed a grammar-centered approach to language tool development, where various kinds of functionality are automatically generated from concrete syntax definitions of the languages involved [18]. In particular, we relied on the Haskell-based Strafunski [23] bundle to generate parsers, pretty-printers, and support libraries for abstract syntax representation and traversal from SDF grammars of VDM-SL and SQL. Figure 3 illustrates this approach. SDF is the formalism in which both grammars are expressed. Parse tables are automatically generated by the *sdf2table* tool from the SDF software bundle, which corresponds to the *Parser* ellipses of Figure 3. The AST Haskell data

[1] The intermediate steps will be presented in concrete VDM-SL syntax for clarity, although the tool actually uses ASTs.

**Table 1.** Catalog of rewriting rules. These rules are based on the various equations and inequations of the calculational data refinement theory presented in Section 3.

| Function | Rewrite rule | Law |
|---|---|---|
| *seq2index* | $A^\star \Rightarrow I\!N \rightharpoonup A$ | (4) |
| *unconjoin* | $A \rightharpoonup (B + C) \Rightarrow (A \rightharpoonup B) \times (A \rightharpoonup C)$ | (5) |
| *distr* | $A \times (B + C) \Rightarrow (A \times B) + (A \times C)$ | (6) |
| *set2fm* | $2^A \Rightarrow A \rightharpoonup 1$ | (7) |
| *opt-elim* | $A + 1 \Rightarrow 1 \rightharpoonup A$ | (8) |
| *unpeither* | $(B + C) \rightharpoonup A \Rightarrow (B \rightharpoonup A) \times (C \rightharpoonup A)$ | (9) |
| *unnjoin* | $A \rightharpoonup (B \times (C \rightharpoonup D)) \Rightarrow (A \rightharpoonup B) \times (A \times C \rightharpoonup D)$ | (10) |
| *rec-elim* | $\mu\mathsf{F} \Rightarrow (K \rightharpoonup \mathsf{F}\,K) \times K$ | (11) |

type definition and the pretty-printer are generated with the *Sdf2Haskell* tool from Strafunski. They correspond to the *Syntax* and *SyntaxPP* ellipses from the picture.

From the abstract syntax, further components are generated in the form of Haskell Class instances, using the *DrIFT* tool. The *ATerm* instances support serialization to the ATerm format, which is used as interchange format between the generated parser and other components. The *Term* instances support generic traversal and strategic term rewriting over ASTs. The last two (*Eq* and *Show*), are not mandatory: they add comparison and printing functions to the Haskell data types.

As SQL grammar, we were able to employ a previously developed grammar. The VDM-SL grammar was developed by reconstructing the concrete syntax definition of the ISO standard [15] in SDF, as reported elsewhere [2].

## 4.2 Transformation

The transformation engine is the core module of the VooDooM tool. It is responsible for the refinement of the VDM-SL data types to a relational form, in accordance with the refinement laws presented above.

We make ample use of strategic term rewriting techniques in its implementation. The overall approach is as follows. First, we formulate individual term rewriting rules on the basis of the type signatures of the representation functions of the refinement laws. Table 1 lists these individual rules. Secondly, we use strategy combinators to compose these individual rules into a transformation engine that applies the individual rules in a way that a normal form is reached in a deterministic and confluent manner.

Before transformation begins, a single traversal is made over the AST that represents the complete VDM-SL input specification, to collect all sub-ASTs that represent data type definitions into a list. The transformation process itself operates on this collection. The transformation process is organized into the following sequential phases:

**Inlining and recursion removal** The rewrite rules for conversion operate on datatypes, not on systems of named data type definitions. To avoid needing to perform lookups of data type names during transformation, we start by inlining, i.e. replacing all data type names by their definitions. This technique leads to the loss of the top level data type names, which in some cases are useful. To overcome this problem, singleton composes are introduced before inlining those types.

Of course, this substitution process would run into cycles if we did not treat recursive definitions differently. For this reason, the recursion removal rewrite rule *rec-elim* is used in combination with inlining. After these rules have exhaustively been applied, a set of non-recursive, independent datatypes is obtained that is amenable to further transformation. Exhaustive application is realized by using the *repeat* combinator.

After the inlining step, our example specification will look as follows:

```
types
  BAMS = map compose AccId of seq of char end to
             compose Account of
           H: set of compose AccHolder of seq of char end
           B: compose Amount of int end
         end
```

Though our example does not contain recursive datatypes, the tree example of Section 3 illustrates recursion removal. More examples are given in [30, 31].

**Desugaring** We limit the language of data type definitions by removing those constructs for which we have a simple elimination rule: sets, sequences, and optionals. Sequences of characters are viewed as atomic and excluded from desugaring, because we want to map them to native SQL strings (varchar). Also, we rewrite all tuples to VDM-SL's compose construct. This desugaring step is performed by applying the rules *seq2index*, *set2fm*, and *optElim*, in a single traversal.

In the same traversal, we rewrite tuples to VDM-SL compose constructs. Alternatively, we could have desugared composes to nested tuples, but that would lead to the loss of names of composes and their fields. Of course, if all tuples are eliminated in favour of composes, this has the consequence that all calculation laws involving products should be mapped to rewrite rules involving composes. This has as additional benefit that various rules (e.g. 14) can be generalized, because composes are $n$-ary, rather than binary.

After desugaring, our example specification looks as follows:

```
types
  BAMS = map compose AccId of seq of char end to
             compose Account of
           H: map compose AccHolder of seq of char end to NIL
           B: compose Amount of int end
         end
```

This expression contains only maps and products (compose), but is not yet in relational form.

**Conversion to relational form** After having the desugared structure, further transformation rules can now be applied. At this stage, the needed rules are *unconjoin*, *unpeither*, and the generalized version of *unnjoin*. In addition, a rule for flattening nested composes is needed to bring expressions into the best form to be rewritten with that generalized rule. These rewrite rules need to be applied exhaustively throughout the AST. The *innermost* combinator is suitable for this.

After conversion, our example specification is in the relational normal form which follows:

```
types
  BAMS = compose mapAggr of
           map compose AccId of seq of char end
            to compose Amount of int end
           map compose tuple of
                    seq of char
                    seq of char
               end
            to NIL
         end
```

**Resugaring** Finally, sets are reintroduced into the expression, using the `dom` rule. Thus, any occurrence of the form `map x to NIL` is converted to `set of x`. This occurs when further simplification was not possible. This is justified, because these can be represented directly in SQL. When VDM-SL is targeted as output language, tuples are reintroduced where binary composes with anonymous fields occur.

## 4.3 SQL Translation

During transformation, an initial specification is transformed into a relational normal form. In the translation process these VDM-SL data types are converted to SQL tables and attributes.

The translation of normal forms to SQL is straightforward. The relational equivalent of a *map* is a table in which the domain of the map is the primary key. The relational counterpart of a *set* is a table with a compound primary key on all columns to guarantee uniqueness. The elements of maps and sets, which are products of elementary VDM-SL data types, are converted to SQL column attributes (that are also of elementary types).

Because basic VDM-SL and SQL data types are not compatible, a correspondence between them must be made. Table 2 shows the correspondence implemented in the VooDooM tool. The table also shows constraints to be added to the SQL data model to better preserve the semantics of some VDM-SL data types. Only Standard SQL92 [16] data types were chosen, to provide a solution that works for all SQL vendor dialects.

The SQL generated for our running BAMS example is as follows:

```
CREATE TABLE table1 (            CREATE TABLE table2 (
 AccId VARCHAR (128) NOT NULL,    Attr1 VARCHAR (128) NOT NULL,
 Amount INT NOT NULL,             Attr2 VARCHAR (128) NOT NULL,
 PRIMARY KEY (AccId)              PRIMARY KEY (Attr1, Attr2)
)                                )
```

As can be seen, a composite type (the outer compose) with a map and a set (reintroduced for `map ... to NIL`) is translated to two tables in SQL. Because none of

**Table 2.** Correspondence between VDM-SL and SQL92 data types.

| VDM-SL data type | SQL data type | SQL Constraint |
|---|---|---|
| bool | SMALLINT | CHECK (.. IN (0,1)) |
| nat | INT | CHECK .. >= 0 |
| nat1 | INT | CHECK .. >= 1 |
| int | INT | |
| rat | REAL | |
| real | REAL | |
| char | CHAR ( 1 ) | |
| token | VARCHAR ( 128 ) | |
| seq of char | VARCHAR ( 128 ) | |

the compose elements have tags, they have been automatically generated as `table1` and `table2`. The fields of the inner composes have been converted to SQL attribute columns. In case of the map there are two tags: `AccId` and `Amount`. This led to the creation of two attributes with those names. The primary key of the generated table is `AccID` because it represents the domain of the map. In case of the set there are no tags, so attribute names are automatically generated: `Attr1` and `Attr2`. These two attributes together form a compound primary key, because combined they represent the domain of the set.

Thus, `table1` associates an amount to the identifier of each account in the system, while `table2` uniquely relates accounts identifiers with account holders. These two tables implement the original specification in which account identifiers are mapped to accounts, and each account has a set of account holders and an amount. The actual retrieve function that witnesses the abstraction relation between the original VDM-SL specification and this pair of SQL tables is given in [1].

## 5 Concluding remarks

A decade ago, Barros [5] referred to the derivation of database programs directly from formal specifications as an unsolved problem. By contrast, deriving the database structure was regarded as a trivial aspect. However, his specifications are Z schemata whose internal states are already close to the relational model (e.g. power-sets of products).

This is in contrast with our approach, in which the source data-model can be arbitrarily complex (as far as VDM-SL data constructors are concerned), including recursive datatypes. Our “derecursivation” law (*rec-elim*), which relationally expresses the main result of [37], bears some resemblance (at least in spirit) with “defunctionalization” [14], a technique which is used in program transformation and compilation.

On the other hand, our approach shares with [5] the view that database design should be regarded as special case of data refinement. It is orthogonal to [5] in the sense that we are not concerned with database dynamics (transactions, etc).

Another advantage of our approach is the prospect of synthesizing abstraction invariants generated by each refinement step, which is still in the *to-do* list of the project. These include abstraction / representation functions and concrete invariants. The former can be used for *data-migration* between the original VDM-SL source and the generated relational model, in a way similar to [28] and to what is done manually in [1]. The latter can be (at least in part) incorporated as SQL constraints.

Strategic term rewriting provides a realistic solution to database schema calculation when compared with previous attempts to animate the same calculus using *genetic algorithm*-based term-rewriting techniques [27].

### 5.1 Future work

We plan to extend the VooDooM tool in several ways. Firstly, in addition to the conversion of VDM-SL to SQL, we want to support the reverse process of obtaining an algebraic set of datatypes from a relational model, as already suggested by the dashed lines in the architecture overview in Figure 2.

Reversing a database to VDM-SL is not a novelty. This problem was already tackled in [28], in which the authors describe an implemented functional prototype and its application using a real world example. However that implementation has several drawbacks. The process has to be assisted manually, the initial relational model must be specified in VDM-SL, the transformation rules were coded with explicit recursion, and all traversals were hard-coded leading to a inflexibility in the implementation. With the strategic term rewriting approach, the same problem can be solved in a more pragmatic way.

Secondly, we intend to offer better support for invariants to the tool. The transformation and translation processes in both directions lack support for VDM-SL invariants. To more accurately preserve semantics, invariants should be added during the transformation process when a data type is split in two or more. However, invariants pose some difficulties when performing transformations since the data definitions which they refer are changing. Thus, invariants need also to reflect this change. When the transformations are simple rearrangements of data fields this can be easy but, since invariants can be as complicated as any function mapping the type to a boolean, the general case is not. Transforming arbitrary functionality in an automated manner is a challenging subject which would involve investigation beyond the scope of this tool. However, we intend to develop some invariant support, namely to referential integrity constraints, by providing a small subset of VDM-SL that can be mapped into SQL constraints in an automated way.

## Availability

The VooDooM tool is developed as open source software and is available from its project web page: http://voodoom.sourceforge.net/.

## Acknowledgments

The work reported in this paper has been carried out in the context of the Eureka project $\Sigma$!2235 *IKF: Information Knowledge Fusion* funded by *Agência de Inovação S.A*, and of the project *Program Understanding and Re-engineering: Calculi and Applications (PURe)* funded by *Fundação para a Ciência e a Tecnologia*, grant number POSI/ICHS/44304/2002.

## References

1. J.J. Almeida, L.S. Barbosa, F.L. Neves, and J.N. Oliveira. Bringing Camila and SetCalc Together — the `bams.cam` and `ppd.cam` Camila Toolset demos. Technical report, DI/UM, Braga, December 1997. [45 p. doc.].
2. T. Alves and J. Visser. Development of an industrial strength grammar for VDM. Technical Report DI-PURe-05.04.29, Universidade do Minho, 2005.
3. R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J. van der Woude. Polynomial relators. In *2nd Int. Conf. Algebraic Methodology and Software Technology (AMAST'91)*, pages 303–362. Springer LNCS, 1992.
4. D. D. Baluta. A formal specification in Z of the relational data model, Version 2, of E.F. Codd. M. Sc. thesis, Concordia University, Montreal, QC, Canada, 1995.
5. R.S.M. de Barros. Deriving relational database programs from formal specifications. In Maurice Naftalin, B. Tim Denvir, and Miquel Bertran, editors, *FME '94: Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 24-18, 1994, Proceedings*, volume 873 of *Lecture Notes in Computer Science*, pages 703–723. Springer, 1994.
6. D. Bjorner and C.B. Jones. *Formal Specification and Software Development*. Series in Computer Science. Prentice-Hall International, 1982. C.A.R. Hoare, ed.
7. H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In Furio Honsell, editor, *Foundations of Software Science and Computation Structures, ETAPS'2001*, Lecture Notes in Computer Science, pages 166–180, Genova, Italy, April 2001. Springer-Verlag.
8. E. F. Codd. *Missing Information*. Addison-Wesley Publishing Company, Inc., 1990.
9. E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.
10. J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques for Software Development* . Cambridge University Press, 1st edition, 1998.
11. J.S. Fitzgerald and C.B. Jones. *Modularizing the formal description of a database system*, volume 428 of *Lecture Notes in Computer Science*. Springer, 1990.
12. H. Garcia-Molina, J. D. Ullman, and J. D. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002. ISBN: 0-13-031995-3.
13. P. Hoogendijk. *A Generic Theory of Data Types*. PhD thesis, University of Eindhoven, The Netherlands, 1997.
14. G. Hutton and J. Wright. Compiling exceptions correctly. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, volume 3125 of *Lecture Notes in Computer Science*, pages 211–227. Springer, 2004.
15. ISO. Information technology — programmming languages, their environments and system software interfaces — Vienna Development Method — specification language — part 1: Base language, Dec. 1996. (ISO/IEC 13817-1, Geneva).
16. ISO. *Information Technology – Database languages – SQL*. Reference number ISO/IEC 9075:1992(E), Nov. 1992.
17. C.B. Jones. *Software Development — A Rigorous Approach*. Series in Computer Science. Prentice-Hall International, 1980. C.A. R. Hoare.
18. M. de Jonge and J. Visser. Grammars as contracts. In *Proceedings of the Second International Conference on Generative and Component-based Software Engineering (GCSE 2000)*, volume 2177 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2000.
19. T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science, 2001. Proc. Workshop on Language Descriptions, Tools and Applications.

20. R. Lämmel. Typed Generic Traversal With Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54, 2003.
21. R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).
22. R. Lämmel and J. Visser. Strategic polymorphism requires just two combinators! Technical Report cs.PL/0212048, arXiv, December 2002.
23. R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.
24. D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.
25. C. Morgan. *Programming from Specification*. Series in Computer Science. Prentice-Hall International, 1990. C.A. R. Hoare, series editor.
26. C. Necco. Polytypic data processing. Master's thesis, Facultad de Cs. Físico Matemáticas y Naturales, University of San Luis, Argentina, 2005. (Submitted.).
27. F.L. Neves and J.N. Oliveira. ART — Um Laboratório de Reificação "Genética". In *IBERAMIA'98 — Sixth Ibero-Conference on Artificial Intelligence*, pages 201–215, Lisbon, Portugal, October 5-9 1998. (in Portuguese).
28. F.L. Neves, J.C. Silva, and J.N. Oliveira. Converting Informal Meta-data to VDM-SL: A Reverse Calculation Approach . In *VDM in Practice! A Workshop co-located with FM'99: The World Congress on Formal Methods, Toulouse, France*, September 1999.
29. J.N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2(1):1–23, April 1990.
30. J.N. Oliveira. Software reification using the SETS calculus. In Tim Denvir, Cliff B. Jones, and Roger C. Shaw, editors, *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. ISBN 0387197524, Springer-Verlag, 8–10 January 1992. (Invited paper).
31. J.N. Oliveira. Data processing by calculation, 2001. 108 pages. Lecture Notes for the 6th Estonian Winter School in Computer Science, 4-9 March 2001, Palmse, Estonia.
32. J.N. Oliveira. Calculate databases with 'simplicity', September 2004. Presentation at the IFIP WG 2.1 #59 Meeting, Nottingham, UK.
33. W.B. Samson and A.W. Wakelin. *Algebraic Specification of Databases: A Survey from a Database Perspective*. Workshops in Computing. Springer Verlag, Glasgow, 1992.
34. J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.
35. E. Visser and Z. Benaissa. A Core Language for Rewriting. In C. Kirchner and H. Kirchner, editors, *Proc. International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *ENTCS*, Pont-à-Mousson, France, September 1998. Elsevier Science.
36. J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, 2001. Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2001).
37. E. G. Wagner. All recursive types defined using products and sums can be implemented using pointers. In Clifford Bergman, Roger D. Maddux, and Don Pigozzi, editors, *Algebraic Logic and Universal Algebra in Computer Science*, volume 425 of *Lecture Notes in Computer Science*. Springer, 1990.
38. A. Walshe. *NDB: The Formal Specification and Rigorous Design of a Single-User Database System*. Prentice Hall, ISBN 0-13-116088-5, 1990.