Journal of Universal Computer Science, vol. 15, no. 1 (2009), 355-397
submitted: 26/6/08, accepted: 1/12/08, appeared: 1/9/09 © *J.UCS*

Dynamic Data Warehouse Design with Abstract State Machines

Jane Zhao, Klaus-Dieter Schewe
(Information Science Research Centre, Palmerston North, New Zealand
janeqzhao@hotmail.com, kdschewe@acm.org)

Henning Koehler
(University of Queensland, Brisbane, Australia
henning@itee.uq.edu.au)

Abstract: On-line analytical processing (OLAP) systems deal with analytical tasks that support decision making. As these tasks do not depend on the latest updates by transactions, it is assumed that the data required by OLAP systems are kept in a data warehouse, which separates the input from operational databases from the outputs to OLAP. However, user requirements for OLAP systems change over time. Data warehouses and OLAP systems thus are rather dynamic and the design process is continuous. In order to easily incorporate new requirements and at the same time ensure the quality of the system design, we suggest to apply the Abstract State Machine (ASM) based development method. This assumes we capture the basic user requirements in a ground model and then apply stepwise refinements to the ground model for every design decisions or further new requirements. In this article, we show that a systematical approach which is tailored for data warehouse design with a set of formal refinement rules can simplify the work in dynamic data warehouse design and at the same time improves the quality of the system.

Key Words: Abstract State Machine, On-Line Analytical Processing, Data Warehouse, Refinement
Category: D.2.10, H.4

1 Introduction

On-line analytical processing (OLAP) systems deal with analytical tasks that support decision making. As these tasks do not depend on the latest updates by transactions, it is assumed that the data required by OLAP systems is kept in a data warehouse, which separates the input from operational databases from the output to OLAP. However, business requirements for OLAP systems change over time and such changes are frequent. Thus data warehouses and OLAP systems are dynamic, and the design process is continuous.

The issue of dynamic data warehouse design has attracted lots of attention, but most of the work focused on the issue of materialised view selection. In [Theodoratos and Sellis, 1999], the data warehouse is seen as a set of materialised views over the source database. The problem of incorporating new requirements becomes looking for additional views to materialise to answer a new set of queries. Similarly in [Gupta et al., 1995], they look at how a re-defined user query can be computed more efficiently using the already materialised result of the original query. The work by [Kotidis and Roussopoulos, 2001; Lawrence and Rau-Chaplin, 2006] have assumed a set of materialised view over a data warehouse. They both worked at how to reselect the set of materialised views to better serve the changed queries.

We tackle the dynamics more completely, by schema evolution as in [Blaschka et al., 1999], and by view integration as in [Bouzeghoub and Kedad, 2000]. Blaschka et al.'s [Blaschka et al., 1999] approach is adapted to the design of multidimensional databases so they focus on the dimension evolutions through some algebraic operations whereas our approach is meant for the commonly used relational databases. Bouzeghoub et al.'s [Bouzeghoub and Kedad, 2000] method finds a set of views for the data warehouse rather than data warehouse schemas in our case. In addition, we look at the impact on the optimisation issue too. In that, our approach differs from the work in [Theodoratos and Sellis, 1999] by considering the case of adding new views, if it is more beneficial even if the new queries can be rewritten by the existing views.

In short, our design approach aims at simplifying the design work by a method that guides the application of the refinement rules, and improving the quality of the design which results from our approach being grounded in the general method of Abstract State Machines (ASMs, [Börger and Stärk, 2003]) as outlined in [Schewe and Zhao, 2005a]. ASMs have already proven their usefulness in many application areas, such as hardware and software architecture, databases, software engineering, etc. [Börger and Glässer, 1995; Barnett et al., 2001; Prinz and Thalheim, 2003; Gurevich et al., 1997]. Furthermore, the ASM method explicitly supports our view of systems development to start with an initial specification called *ground model* [Börger, 2003a] that is then subject to *refinements* [Börger, 2003b]. So quality criteria such as the satisfaction of constraints can first be verified for the ground model, while the refinements are defined in a way that preserves already proven quality statements. This is similar to the approach taken in [Schewe, 1997], which contains a refinement-based approach to the development of data-intensive systems using a variant of the B method [Abrial, 1996].

The general idea for the ground model is to employ three interrelated ASMs, one for underlying operational databases, one for the data warehouse, and one for dialogue types [Lewerenz et al., 1999] that can be used to integrate views on the data warehouse with OLAP functionality [Thomson, 2002]. For the data warehouse level the model of multi-dimensional databases [Gyssens and Lakshmanan, 1996], which are particular relational databases, can be adopted. Then a large portion of the refinement work has to deal with view integration as predicted one of the major challenges in this area in [Widom, 1995]. The work in

[Ma et al., 2005] discusses a rule-based approach for this task without reference to any formal method.

Our work started from [Zhao and Schewe, 2004] which shows the development of an ASM ground model for data warehouses and OLAP system based on the fundamental idea of separating input from operational databases from output to OLAP systems. According to this idea we obtain a model of three interconnected ASMs, one for the operational database(s), one for the data warehouse, and one for the OLAP system.

This idea was extended in [Schewe and Zhao, 2005b] focusing on cost-efficient distribution of data warehouses. The work in [Schewe and Zhao, 2005b] exploits fragmentation techniques from [Özsu and Valduriez, 1999] and the recombination of fragments, but still remains on rather informal grounds. In [Zhao, 2005], we started to formalize the distribution process and showed how the three-layered specification was extended to distributed data warehouses.

The ASM modelling language was extended in [Schewe and Zhao, 2007] by types. Our rationale for introducing typed ASMs is to obtain an application-specific version of ASMs. Though the quality-assurance aspect is an important argument for a formal development method, there is still an aversion against using a rigid formal approach, in particular in areas, in which informal development is still quite dominant such as OLAP systems. So providing a version of ASMs that are easier to use due to the adoption of more familiar terminology will be a necessary step towards the simplification of systems development and the desired quality improvement.

In this paper, we extend our work in [Koehler et al., 2007] which deals with dynamic data warehouse and OLAP design. In specific, we have looked at the case of incorporating OLAP functionalities, such as the applications in business statistics. Furthermore, we present our general design approach which is tailored to the design of dynamic data warehouse and OLAP system design.

The rest of paper is organized as follows: in Section 2 we introduce the general idea of the ASM method, and ASM with types (TASM). In Section 3, we present a ground model in TASM using a grocery store as an example. In Section 4 we discuss our refinement approach tailored for dynamic data warehouse design. In Section 5, we describe the view integration technique through introducing the general idea of schema transformation and the formal definition of a subset of refinement rules. In Section 6, we show how we deal with the issue of dynamic data warehouse and OLAP system design using some examples. In Section 7, we demonstrate how we incorporate changes of OLAP functionality using examples in business statistics. We conclude with a brief summary in Section 8.

2 Systems Development with Abstract State Machines

Abstract State Machines (ASMs, [Börger and Stärk, 2003]) have been developed as means for high-level system design and analysis. The general idea is to provide a through-going uniform formalism with clear mathematical semantics without dropping into the pitfall of the "formal methods straight-jacket". That is, at all stages of system development we use the same formalism, the ASMs, which is flexible enough to capture requirements at a rather vague level and at the same time permits almost executable systems specifications. Thus, the ASM formalism is precise, concise, abstract and complete, yet simple and easy to handle, as only basic mathematics is used.

The systems development method itself just presumes to start with the definition of a *ground model ASM* (or several linked ASMs), while all further system development is done by refining the ASMs using quite a general notion of refinement. So basically the systems development process with ASMs is a refinement-validation-cycle. That is a given ASM is refined and the result is validated against the requirements. Validation may range from critical inspections to the usage of test cases and evaluation of executable ASMs as prototypes. This basic development process may be enriched by rigorous manual or mechanized formal verification techniques. However, the general philosophy is to design first and to postpone rigorous verification to a stage, when requirements have be almost consolidated. In the remainder of this article we will emphasize only the specification of ground model ASMs and suitable refinements (for details see [Börger and Stärk, 2003]).

2.1 Simple ASMs

As explained so far, we expect to define for each stage of systems development a collection $M_1, \dots, M_n$ of ASMs. Each ASM M_i consists of a *header* and a *body.* The header of an ASM consists of its name, an import- and export-interface, and a signature. Thus, a basic ASM can be written in the form

$\mathtt{ASM}\ M$
$\mathtt{IMPORT}\ M_1(r_{11}, \dots, r_{1n_1}), \dots, M_k(r_{k1}, \dots, r_{kn_k})$
$\mathtt{EXPORT}\ q_1, \dots, q_\ell$
$\mathtt{SIGNATURE} \dots$

Here r_{ij} are the names of functions and rules imported from the ASM M_i defined elsewhere. These functions and rules will be defined in the body of M_i — not in the body of M — and only used in M. This is only possible for those functions and rules that have explicitly been exported. So only the functions and rules $q_1, \dots, q_\ell$ can be imported and used by ASMs other than M. As in standard modular programming languages this mechanism of import- and export-interface permits ASMs to be developed rather independently from each other leaving the definition of particular functions and rules to “elsewhere”.

The *signature* of an ASM is a finite list of function names $f_1, \dots, f_m$, each of which is associated with a non-negative integer ar_i, the *arity* of the function f_i. In ASMs each such function is interpreted as a total function $f_i : \mathcal{U}^{ar_i} \to \mathcal{U} \cup \{\perp\}$ with a not further specified set $\mathcal{U}$ called *super-universe* and a special symbol $\perp \notin \mathcal{U}$. As usual, f_i can be interpreted as a partial function $\mathcal{U}^{ar_i} \nrightarrow \mathcal{U}$ with domain $dom(f_i) = \{\mathbf{x} \in \mathcal{U}^{ar_i} \mid f_i(\mathbf{x}) \neq \perp\}$.

The functions defined for an ASM including the static and derived functions, define the set of *states* of the ASM.

In addition, functions can be *dynamic* or not. Only dynamic functions can be updated, either by and only by the ASM, in which case we get a *controlled* function, by the environment, in which case we get a *monitored* function, or by neither the ASM nor the environment, in which case we get a *derived* function. In particular, a dynamic function of arity 0 is a variable, whereas a static function of arity 0 is a constant.

2.2 States and Transitions

If f_i is a function of arity ar_i and we have $f(x_1, \dots, x_{ar_i}) = v$, we call the pair $\ell = (f, \mathbf{x})$ with $\mathbf{x} = (x_1, \dots, x_{ar_i})$ a *location* and v its *value*. Thus, each *state* of an ASM may be considered as a set of location/value pairs.

If the function is dynamic, the values of its locations may be updated. Thus, states can be updated, which can be done by an *update set*, i.e. a set Δ of pairs (ℓ, v), where ℓ is a location and v is a value. Of course, only *consistent* update sets can be taken into account, i.e. we must have

$$(\ell, v_1) \in \Delta \wedge (\ell, v_2) \in \Delta \Rightarrow v_1 = v_2.$$

Each consistent update set Δ defines *state transitions* in the obvious way. If we have $f(x_1, \dots, x_{ar_i}) = v$ in a given state s and $((f, (x_1, \dots, x_{ar_i})), v') \in \Delta$, then in the successor state s' we will get $f(x_1, \dots, x_{ar_i}) = v'$.

In ASMs consistent update sets can be obtained from *update rules*, which can be defined by the following language:

- the skip rule `skip` indicates no change;
- the update rule $f(t_1, \dots, t_n)$:= t with an n-ary function f and terms $t_1, \dots, t_n, t$ indicates that the value of the location determined by f and the terms $t_1, \dots, t_n$ will be updated to the value of term t;
- the sequence rule r_1 `seq` ... `seq` r_n indicates that the rules $r_1, \dots, r_n$ will be executed sequentially;

- the block rule r_1 `par` $\dots$ `par` r_n indicates that the rules $r_1, \dots, r_n$ will be executed in parallel;
- the conditional rule

$$\texttt{if } \varphi_1 \texttt{ then } r_1 \texttt{ elsif } \varphi_2 \dots \texttt{ then } r_n \texttt{ endif}$$

  has the usual meaning that r_1 is executed, if φ_1 evaluates to true, otherwise r_2 is executed, if φ_2 evaluates to true, etc.;
- the let rule `let` $x = t$ `in` r means to assign to the variable x the value defined by the term t and to use this x in the rule r;
- the forall rule `forall` x `with` φ `do` r `enddo` indicates the parallel execution of r for all values of x satisfying φ;
- the choice rule `choose` x `with` φ `do` r `enddo` indicates the execution of r for one value of x satisfying φ;
- the call rule $r(t_1, \dots, t_n)$ indicates the execution of rule r with parameters $t_1, \dots, t_n$ (call by name).

Instead of `seq` we simply use ; and instead of `par` we write $\|$. The idea is that the rules of an ASM are evaluated in parallel. If the resulting update set is consistent, we obtain a state transition. Then a *run* of an ASM is a finite or infinite sequence of states $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$ such that each s_{i+1} is the successor state of s_i with respect to the update set Δ_i that is defined by evaluating the rules of the ASM in state s_i.

We omit the formal details of the definition of update sets from these rules. These can be found in [Börger and Stärk, 2003].

The definition of rules by expressions $r(x_1, \dots, x_n) = r'$ makes up the body of an ASM. In addition, we assume to be given an *initial state* and that one of these rules is declared as the *main rule*. This rule must not have parameters.

2.3 Notion of Refinement

The general notion of *refinement* relates two ASMs M and M^*. In principle, as the semantics of ASMs is defined by its runs, we would need a correspondence between such runs, i.e.

- a correspondence between the states s of M and the states s^* of M^*, and
- a correspondence between the runs of M and M^* involving states s and s^*, respectively.

However, in contrast to many formal methods the notion of refinement in ASMs does not require all states to be taken into account. We only request to have a correspondence between "states of interest".

Formally, let S and S^* be the sets of states of ASMs M and M^*, respectively. A *correspondence of states* between M and M^* is a one-one binary relation $\equiv \subseteq S \times S^*$ such that $s_0 \equiv s_0^*$ holds for the initial states s_0 and s_0^* of M and M^*, respectively. In particular, we must have

$$s \equiv s_1^* \wedge s \equiv s_2^* \Rightarrow s_1^* = s_2^*$$

and

$$s_1 \equiv s^* \wedge s_2 \equiv s^* \Rightarrow s_1 = s_2.$$

Then we say that M^* is a *correct refinement* of M iff for each run $s_0^* \to s_1^* \to \dots$ of M^* there is a run $s_0 \to s_1 \to \dots$ of M and there are index sequences $0 = i_0 < i_1 < \dots$ and $0 = j_0 < j_1 < \dots$ such that $s_{i_x} \equiv s_{j_x}^*$ holds for all x. We say M^* is a *complete refinement* of M iff M is a correct refinement of M^*.

The focus of ASM refinement method [Börger, 2003b] is to support the usage of refinements for correctly reflect and explicitly document an intended design decision, adding more details to a more abstract design description, e.g. for making an abstract program executable, for improving a program by additional features or by restricting it through precise boundary conditions which exclude certain undesired behaviors.

The general scheme of ASM refinement is illustrated by a commutative diagram in Figure 1. The scheme describes a (m, n)-refinement which includes cases, such as $(m, 0)$-refinement with $m > 0$, where some of the abstract operations have been eliminated for the purpose of, e.g. optimisation, and $(0, n)$-refinement with $n > 0$, where the concrete machine has longer run segments than the abstract one due to, for example adding new features. The size of m and n is determined dynamically by the states of interest. For refinement correctness proof, it is shown in [Schellhorn, 2001] that every (m, n)-refinement with $n > 1$ can be reduced to $(m, 1)$-refinement.

There are three types of ASM refinement pattern derived from the refinement scheme which include the conservative extension, procedural refinement, and data refinement.

Conservative extension is a purely incremental refinement which is typically used to introduce new behavior in a modular fashion, like exception handling, robustness features, etc.

Procedural refinement, also called submachine refinement, consists in replacing in a given machine one machine by another (usually more complex) machine.

Data refinements are given by $(1, 1)$-refinements where abstract states and rules are mapped to concrete ones in such a way that the effect of each concrete

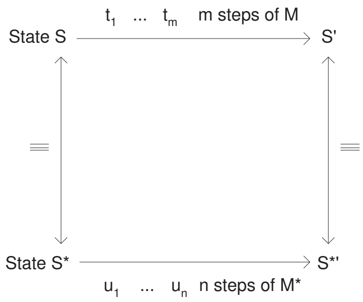

Figure 1: The ASM refinement scheme

operation on concrete data types is the same as the effect of the corresponding abstract operation on abstract data types.

2.4 Typed ASM

When designing data intensive systems such as data warehouses, we need to model the data and the operations accurately. To make this task easier, we have extended the ASMs with types in [Link et al., 2006]. We started with a *type system*

$$t = b \mid \{t\} \mid a : t \mid t_1 \times \cdots \times t_n \mid t_1 \oplus \cdots \oplus t_n \mid \mathbb{1}$$

Here b represents a not further specified collection of base types such as *label*, *int*, *date*, etc. $\{\cdot\}$ is a set-type constructor, $a : t$ is a type constructor with a of type *label*, which is introduced as attributes used in join operations. The base type *label* is used for referencing tuples. We require attribute names (i.e. the a in "$a : t$") to be of type *label* so we can store them (e.g. for storing FDs function dependencies). $\times$ and $\oplus$ are constructors for tuple and union types. $\mathbb{1}$ is a trivial type. With each type t we associate a *domain* $dom(t)$ in the usual way, i.e. we have

- $dom(\{t\}) = \{x \subseteq dom(t) \mid |x| < \infty\}$,
- $dom(a : t) = dom(t)$,
- $dom(t_1 \times \cdots \times t_n) = dom(t_1) \times \cdots \times dom(t_n)$,
- $dom(t_1 \oplus \cdots \oplus t_n) = \coprod_{i=1}^{n} dom(t_i) = \bigcup_{i=1}^{n} \{i\} \times dom(t_i)(disjointunion)$,
- $dom(\mathbb{1}) = \{\mathbf{1}\}$

For this type systems we obtain the usual notation of subtyping, defined by the smallest partial order $\leq$ on types satisfying

- $t \leq \mathbb{1}$ for all types t;
- if $t \leq t'$ holds, then also $\{t\} \leq \{t'\}$;
- if $t \leq t'$ holds, then also $a : t \leq a : t'$;
- if $t_{i_j} \leq t'_{i_j}$ hold for $j = 1, \dots, k$, then $t_1 \times \cdots \times t_n \leq t'_{i_1} \times \cdots \times t'_{i_k}$ for $1 \leq i_1 < \cdots < i_k \leq n$;
- if $t_i \leq t'_i$ hold for $i = 1, \dots, n$, then $t_1 \oplus \cdots \oplus t_n \leq t'_1 \oplus \cdots \oplus t'_n$.

We say that t is a *subtype* of t' iff $t \leq t'$ holds. Obviously, subtyping $t \leq t'$ induces a canonical projection mapping $\pi^t_{t'} : dom(t) \to dom(t')$.

The *signature* of a TASM is defined analogously to the signature of an "ordinary" ASM, i.e. by a finite list of function names $f_1, \dots, f_m$. However, in a TASM each function f_i now has a *kind* $t_i \to t'_i$ involving two types t_i and t'_i. We interpret each such function by a total function $f_i : dom(t_i) \to dom(t'_i)$. Note that using $t'_i = t''_i \oplus \mathbb{1}$ we can cover also partial functions.

The functions of a TASM including the dynamic and static functions, define the set of states of the TASM. More precisely, each pair $\ell = (f_i, x)$ with $x \in dom(t_i)$ defines a *location* with $v = f_i(x)$ as its *value*. Thus, each *state* of a TASM may be considered as a set of location/value pairs.

We call a function R of kind $t \to \{\mathbb{1}\}$ a *relation*. This generalises the standard notion of relation, in which case we would further require that t is a tuple type $a_1 : t_1 \times \cdots \times a_n : t_n$. In particular, as $\{\mathbb{1}\}$ can be considered as a truth value type, we may identify R with a subset of $dom(t)$, i.e. $R \simeq \{x \in dom(t) \mid R(x) \neq \emptyset\}$. In this spirit we also write $x \in R$ instead of $R(x) \neq \emptyset$, and $x \notin R$ instead of $R(x) = \emptyset$.

If we translate a TASM $\mathfrak{M}$ into an ASM $\Phi(\mathfrak{M})$, we have the following theorem:

Theorem 1. *For each TASM $\mathfrak{M}$ there is an equivalent ASM $\Phi(\mathfrak{M})$.*

Of course, this theorem also follows immediately from the main results on the expressiveness of ASMs in [Blass and Gurevich, 2003; Gurevich, 2000]. A constructive proof was given in [Schewe and Zhao, 2007].

2.5 Data Refinement in TASM

To apply ASM refinement method in the design of data warehouses and OLAP systems, we first clarify what are the states of interest in the general definition of refinement. For this assume that names of functions, rules, etc. are completely different for $\mathfrak{M}$ and $\mathfrak{M}^*$. Then consider formulae $\mathcal{A}$ that can be interpreted by pairs of states (s, s^*) for $\mathfrak{M}$ and $\mathfrak{M}^*$, respectively. Such formulae will be called *abstraction predicates*. Furthermore, let the rules of $\mathfrak{M}$ and $\mathfrak{M}^*$, respectively, be partitioned into "main" and "auxiliary" rules such that there is a correspondence $\triangleright$ between main rules r of $\mathfrak{M}$ and main rules r^* of $\mathfrak{M}^*$. Finally, let s_0, s_0^* be the initial states of $\mathfrak{M}$ and $\mathfrak{M}^*$, respectively.

Definition 2. A TASM $\mathfrak{M}^*$ is called a *(data) refinement* of a TASM $\mathfrak{M}$ iff there is an *abstraction predicate* $\mathcal{A}$ with $(s_0, s_0^*) \models \mathcal{A}$ and there exists a correspondence between main rule r of $\mathfrak{M}$ and main rule r^* of $\mathfrak{M}^*$ such that for all states $s, \overline{s}$ of $\mathfrak{M}$, where $\overline{s}$ is the successor state of s with respect to the update set Δ_r defined by the main rule r, there are states $s^*, \overline{s}^*$ of $\mathfrak{M}^*$ with $(s, s^*) \models \mathcal{A}$, $(\overline{s}, \overline{s}^*) \models \mathcal{A}$, and $\overline{s}^*$ is the successor state of s_m^* with respect to the update set Δ_{r^*} defined by the main rule r^*.

While Definition 2 gives a proof obligation for refinements in general, it still permits too much latitude for data-intensive applications. In this context we must assume that some of the controlled functions in the signature are meant to be persistent. For these we adopt the notion of *schema*, which is a subset of the signature consisting only of relations. Then the first additional condition should be that in initial states these relations are empty.

The second additional requirement is that the schema $\mathcal{S}^*$ of the refining TASM $\mathfrak{M}^*$ should dominate the schema $\mathcal{S}$ of TASM $\mathfrak{M}$. For this we need a notion of computable query. For a state s of a TASM $\mathfrak{M}$ let $s(\mathcal{S})$ define its restriction to the schema. We first define isomorphisms starting from bijections $\iota_b : dom(b) \rightarrow dom(b)$ for all base types b. This can be extended to bijections ι_t for any type t as follows:

$$\iota_{t_1 \times \cdots \times t_n}(x_1, \ldots, x_n) = (\iota_{t_1}(x_1), \ldots, \iota_{t_n}(x_n))$$
$$\iota_{t_1 \oplus \cdots \oplus t_n}(i, x_i) = (i, \iota_{t_i}(x_i))$$
$$\iota_{\{t\}}(\{x_1, \ldots, x_k\}) = \{\iota_t(x_1), \ldots, \iota_t(x_k)\}$$

Then ι is an *isomorphism* of $\mathcal{S}$ iff for all states s, the permuted state $\iota(s)$, and all $R : t \rightarrow \{\mathbb{1}\}$ in $\mathcal{S}$ we have $R(x) \neq \emptyset$ in s iff $R(\iota_t(x)) \neq \emptyset$ in $\iota(s)$. A query $f : \mathcal{S} \rightarrow \mathcal{S}^*$ is *computable* iff f is a computable function that is invariant under isomorphisms, i.e. we have $f \circ \iota = \iota \circ f$.

Definition 3. A data refinement $\mathfrak{M}^*$ of a TASM $\mathfrak{M}$ with abstraction predicate $\mathcal{A}$ is called a *strong data refinement* iff the following holds:

1. $\mathfrak{M}$ has a schema $\mathcal{S} = \{R_1, \ldots, R_n\}$ such that in the initial state s_0 of $\mathfrak{M}$ we have $R_i(x) = \emptyset$ for all $x \in dom(t_i)$ and all $i = 1, \ldots, n$.
2. $\mathfrak{M}^*$ has a schema $\mathcal{S}^* = \{R_1^*, \ldots, R_m^*\}$ such that in the initial state s_0^* of $\mathfrak{M}^*$ we have $R_i^*(x) = \emptyset$ for all $x \in dom(t_i^*)$ and all $i = 1, \ldots, m$.
3. There exist computable queries $f : \mathcal{S} \rightarrow \mathcal{S}^*$ and $g : \mathcal{S}^* \rightarrow \mathcal{S}$ such that for each pair (s, s^*) of states with $(s, s^*) \models \mathcal{A}$ we have $g(f(s(\mathcal{S}))) = s(\mathcal{S})$.

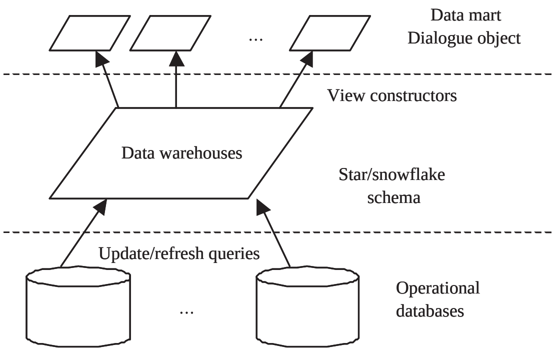

Figure 2: The general architecture of a data warehouse and OLAP

Definition 3 provides a stronger proof obligation for refinements in the application area we are interested in. Furthermore, this notion of strong data refinement heavily depends on the presence of types.

3 The Ground Model

The basic idea of data warehousing is to separate the source data from the target data which are used by OLAP systems ([Widom, 1995]). From this viewpoint, the data source, the data warehouse and the OLAP system can be easily modelled, at rather a high abstraction level, a ground model in ASM method, by a three-tier model as shown in Figure 2. The bottom tier represents the source data which are usually the operational databases. At the middle tier, data from the operational database are extracted, and then cleaned, integrated, transformed, and stored, which represents the data warehouse. At the top tier, OLAP functions, in particular, a set of OLAP queries/views(we treat them equally) are defined, functions of opening or closing datamarts, copies of the OLAP views, roll-up or drill-down on opened datamarts, etc are constructed.

In the following we assume a centralized data warehouse for a grocery store chain. In the operational database, we have a single operational database with five relation schemata as illustrated by the HERM diagram in Figure 3, a start schema for the data warehouse as shown in Figure 4, and the total sales analysis by shop, month and year as an example for the basic requirement in OLAP.

3.1 The Operational Database ASM

The operational database ASM models the source data and a set of functions that are needed in supporting the data warehouse. Assuming for simplicity that

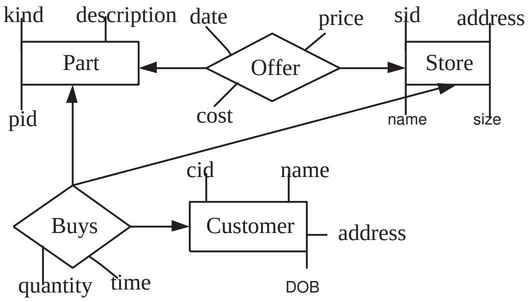

Figure 3: The operational database schema

all data sources are relational, the ASMs signatures would just describe the relation schemata. For example, a n-nary relation R will be modelled as a boolean function $R(n)$. Hence, we will define a set of boolean functions for modelling the database relations. Further, we introduce a universe **request** to model the data warehouse requests, such as data extraction for S_1, for instance. We use an external function **req**:*request* for representing the current request to be served. We have to assume that there is a mechanism built in the system to handle the synchronisation problem in the data warehouse, such as opening a maintenance window for data extraction or refresh. Furthermore, we need a function **r-type**:*request*$\rightarrow$ { *extract, refresh, open-datamart*, ... }, where the type list can be extended as required. We handle the request of data extraction from data warehouse module, in a similar way as to handling a database query request from other application systems, such that the details of data extraction are defined in the DW ASM.

`ASM` DB-ASM
`IMPORT`
  DW-ASM(Shop, Product, Customer_DW, Time,
  Purchase, *extract_purchase*, *extract_customer*, *extract_shop*,
  extract_product, *extract_time*)
`EXPORT`
  Store, Part, Customer_DB, Buys, Offer, *main*,r-type, req
`SIGNATURE`
  Store:$sid \times name \times size \times address \rightarrow \{1\!\!1\}$,
  Part:$pid \times kind \times descriptin \rightarrow \{1\!\!1\}$,
  Customer_DB:$cid \times name \times dob \times address \rightarrow \{1\!\!1\}$,
  Buys:$time \times cid \times sid \times pid \times quantity \rightarrow \{1\!\!1\}$,
  Offer:$pid \times sid \times date \times price \times cost \rightarrow \{1\!\!1\}$,

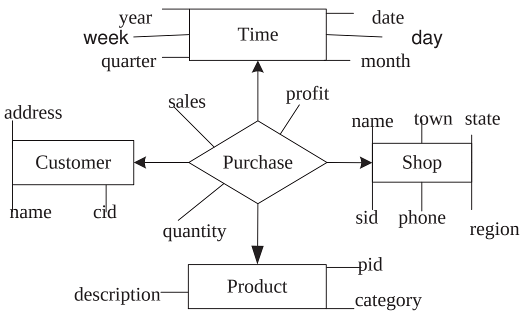

Figure 4: The data warehouse star schema

r-type:*request* → {*extract, refresh, open-datamart*, ...},
req:*request* (monitored)
`BODY`
main = `if` type(req)=*extract* `then`
extract_purchase, *extract_customer*, *extract_shop*,
extract_product, *extract_time*
`endif`

3.2 The Data Warehouse ASM

For the data warehouse ASMs we follow the same line of abstraction as for the operational databases, i.e. using boolean functions to model the data warehouse relation schemata. We use a simple star schema for the data warehouse as illustrated by the HERM diagram in Figure 4, which results in five relation schemata Shop, Product, Customer_DW, Purchase and Time. We define one transition rule for each data warehouse relation. When the rules are called, the data warehouse relations get refreshed. The data warehouse ASM is interlinked with the OLAP ASM by importing the functions defined for each OLAP queries/views in OLAP ASM, and the creation rules for the views.

`ASM` DW-ASM

`IMPORT`
DB-ASM(Store, Part, Customer_DB, Buys, Offer, req, r-type)),
OLAP-ASM(View_sales, DM-View_sales, the-datamart, the-matching-view, *create_View_sales*)

`EXPORT`
Shop, Product, Customer_DW, Time, Purchase

$extract\_purchase$, $extract\_customer$, $extract\_shop$,
$extract\_product$, $extract\_time$, $main$

`SIGNATURE`

Shop:$sid \times name \times town \times region \times state \times phone \rightarrow \{\mathbb{1}\}$,
Product:$pid \times category \times description \rightarrow \{\mathbb{1}\}$,
Customer_DW:$cid \times name \times address \rightarrow \{\mathbb{1}\}$,
Time:$date \times day \times week \times month \times quarter \times year \rightarrow \{\mathbb{1}\}$,
Purchase:$cid \times sid \times pid \times date \times qty \times sale \times profit \rightarrow \{\mathbb{1}\}$

`BODY`

$main$ =`if` type(req)=*open-datamart* `then`
open_datamart(the-datamart(req)) `endif`

$extract\_purchase$ = `forall` i, p, s, d, p', c `with`
$\exists t.(i, p, s, t, p', c) \in \pi_{cid,pid,sid,time,price,cost}$
(Buys $\bowtie$ Customer_DB $\bowtie$ Part $\bowtie$ Store $\bowtie$ Offer) $\wedge\, t.date = d$
`do let` $Q = src[0, \pi_q, +](\{(t, q) \mid (i, s, p, t, q) \in Buys \wedge$
$t.date = d\})$, $S = Q * p'$, $P = Q * (p' - c)$
`in` Purchase$(i, p, s, d, Q, S, P) := 1$ `enddo`

$extract\_shop$ = `forall` s, n, a `with`
$\exists s'.(s, n, s', a) \in$ Store
`do let` $t = a.town, r = a.region, st = a.state, ph = a.phone$
`in` Shop$(s, n, t, r, st, ph) := 1$ `enddo`

$extract\_customer$ = `forall` i, n, a `with`
$(i, n, a) \in$ Customer_DB
`do let` $i' = i, n' = n, a' = a$
`in` Customer_DW$(i', n', a') := 1$ `enddo`

$extract\_product$ = `forall` p, k, d `with`
$(p, k, d) \in$ Part
`do let` $p' = p, c = k, d' = d$
`in` Product$(p', c, d') := 1$ `enddo`

$extract\_time$ = `forall` t `with`
$\exists\, c, p, s, q.(c, p, s, q, t) \in$ Buys
`do` if Time$(t.date, t.day, t.week, t.quarter, t.month, t.year) = \perp$
`then` Time$(t.date, t.day, t.week, t.quarter, t.month, t.year) := 1$
`enddo`

$open\_datamart(dm)$ = `case` the-matching-view(dm) `of`
V_sales : *create_V_sales*;
`forall` s, r, st, m, q, y, S, P `with`
$(s, r, st, m, q, y, S, P) \in$ V_sales `do`
DM-V_sales$(dm, s, r, st, m, q, y, S, P) := 1$ `enddo`
`endcase`

3.3 The OLAP ASM

The top-level ASM dealing with OLAP is a bit more complicated, as it realises the idea of using dialogue objects for this purposes. The general idea from [Schewe and Schewe, 2000] is that each user has a collection of open dialogue objects, i.e. OLAP queries for our purposes here. At any time we may get new users by the operation “login”, or the users may create new dialogue objects by “open”, without closing the opened ones, or they may close some of the dialogue objects, or quit when they finish their work with the system. Of course, the major function of OLAP is to open a view for the corresponding OLAP query, and allow the user to perform further operations, such as roll-up and drill-down over the opened datamarts. We will leave out the modelling of the OLAP operations such as roll-up or drill-down in the follow-up refinements.

In the ground model of OLAP ASM, we define the universe **user** to model the user of the system, the universe **datamart** to model the opened datamarts, the universe **view** to model the views for the OLAP queries, and the universe **operation** to model the OLAP operations issued. Over the universes, we define the function **o-type**:**operation** $\rightarrow \{login, open, close, quit\}$, the function **owner**: **datamart** $\rightarrow$ **user**, the function **issuer**: **operation** $\rightarrow$ **user**, the function **the-datamart**:**operation** $\rightarrow$ **datamart**, which returns the datamart over which a close operation is performed, the function **the-view**:**operation** $\rightarrow$ **view**, which gives the view over which a open operation is performed, and the function **the-matching-view**:**datamart** $\rightarrow$ **view** which gives the matching view of the datamart. We use an external function **op**:operation to represent the current operation processed, **registered**:**user** for the logged on users, and a set of functions $\mathbf{V}_i$ $(i = 1, \ldots, n)$ to define the views for the respective OLAP queries available in the system. With each of the views we define a general function **DM-V**$_i$ to model the datamarts that are opened over view $\mathbf{V}_i$.

Figure 5 illustrates this processing of the main rule of OLAP-ASM.

`ASM` OLAP-ASM

`IMPORT` DW-ASM(Shop, Product, Customer_DW, Time, Purchase)

`EXPORT` V_sales, DM-V_sales, *create_V_sales*, *main*,
the-datamart, the-matching-view

`SIGNATURE`

V_sales:$sid \times region \times state \times month \times quarter \times year \times$
$sales \times profit \rightarrow \{1\!\!1\}$,

DM-V_sales:$dm \times sid \times region \times state \times month \times quarter \times$
$year \times sales \times profit \rightarrow \{1\!\!1\}$,

type: $op \rightarrow \{open, close, quit\}$,

owner: $datamart \rightarrow user$,

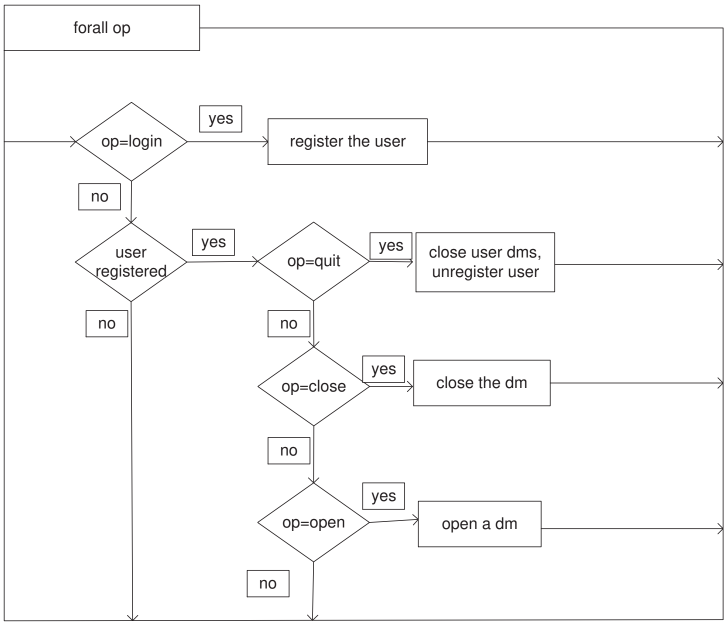

Figure 5: The main rule in OLAP-ASM

issuer: $op \to user$,
the-datamart: $op \to datamart$,
the-view: $op \to view$,
the-matching-view: $datamart \to view$

`BODY`

$main$ = if o-type(op) = *login* **then** $LOGIN$
  elsif if registered(issuer(op))=1 **then**
    if o-type(op) = *open* **then** $OPEN$
  elsif o-type(op) = *close* **then** $CLOSE$
  elsif o-type(op) = *quit* **then** $QUIT$
endif

$LOGIN$ =
  registered(issuer(op)):=1

$OPEN$ =
  import dm
    datamart(dm) := 1
    owner(dm) := issuer(op)
    the-matching-view(dm) := the-view(op)

`end-import`;
request(*open-datamart*,*dm*):= 1

$CLOSE$ =
owner(the-datamart(op)) := $\perp$
datamart(the-datamart(op)) := $\perp$
close_datamart(the-datamart(op))

$QUIT$ =
`let` $usr = issuer(op)$ `in`
`forall` dm `with` owner(dm) = usr
`do` *close_datamart*(dm)
datamart(dm) := $\perp$ owner(dm) := $\perp$ `enddo`
registered(usr) := $\perp$

close_datamart(dm) = `case` the-matching-view(dm) `of`
V_sales : `forall` s, r, st, m, q, y, S, P `with`
(the-datamart(op), s, r, st, m, q, y, S, P) $\in$ DM-V_sales `do`
DM-V_sales(the-datamart(op), s, r, st, m, q, y, S, P) :=
$\perp$ `enddo`
`enddo` `endcase`

create_V_sales = `forall` s, r, st, m, q, y `with`
$\exists n, t, ph.(s, n, t, r, st, ph) \in$ Shop$\wedge$
$\exists d, d', w.(d, d', w, m, q, y) \in$ Time
`do let` $S = src[0, \pi_{s'}, +]$
$(\{(i, s, p, d, s') \mid \exists q', p'.$
$(i, s, p, d, q', s', p') \in$ Purchase $\wedge$
$d.month = m \wedge d.year = y\})$
$P = src[0, \pi_{p'}, +]$
$(\{(i, s, p, d, p') \mid \exists q', s'.$
$(i, s, p, d, q', s', p') \in$ Purchase $\wedge$
$d.month = m \wedge d.year = y\})$
`in` V_sales(s, r, st, m, q, y, S, P) := 1
`enddo`

4 The Refinement Approach

The ASM method assumes that we first set up a ground model. In particular, we have assumed separate ASMs for the database, the data warehouse and the OLAP. Each of these ASMs uses separate *controlled functions* to model states of the system by logical structures and *rules* expressing transitions between these states. The ASMs are then linked together via queries that are expressed by these transitions. The ground model above captures only the basic requirements, further refinements are required for incorporating features such as systems optimisation, implementation, and new OLAP requirements. Our refinement approach is developed based on the 3-tier model and the ASM method, the former provides the logical structure of the system, the latter provides the step-by-step refinement approach. In the following, we classify refinements into three categories: requirements capture, system optimisation, and system implementation. For each category we present a set of refinement rules, in a rather abstract manner, that are ultimately decomposed and formalised into a set of concrete rules in a later development stage, e.g. the formal rules for view integration, which will be discussed in Section 5.

4.1 Requirements Capture

Like most software systems, data warehouse design begins with the requirements from the user end, i.e. the OLAP system. We build data warehouse schemas based on what OLAP needs, for example, the set of analysis queries or reports. As this is not a one-off process due to the dynamic nature of business analysis, it is not uncommon that we may need to deal with new OLAP requirements regularly after the data warehouse has been implemented. The new requirements may result in change in the data warehouse schemas. We tackle this problem with the schema integration technique, i.e. we integrate the new set of data schema from the new requirements with the existing data warehouse schema, so we get an integrated data store and at the same time we maintain a data warehouse with little redundancy.

Using the data warehouse/OLAP ASM ground model as a basis, we handle new OLAP requirement, such as adding new OLAP functions, as follows: in the OLAP ASM, define the new OLAP functions; in the data warehouse ASM, to support the new OLAP functions, define the data and the extraction rule which are needed but not yet present; in the database ASM, incorporate the changes corresponding to the changes from the data warehouse. In fact, we are propagating the changes from the OLAP tier down through the data warehouse tier further to the operational database tier.

The refinements for requirement capturing are classified under conservative extension or incremental refinement in ASM method. That means, the existing functions will be preserved when new features are added in the refinement. A tailored refinement process for systematically capturing new requirements in OLAP is described below.

1. *Add a new rule to the OLAP ASM*: This is used to model an additional OLAP function by adding a new rule name and the definition of the rule for the new function to the OLAP ASM.

It is presumed that the newly added function is not present in the OLAP ASM before. The new function will work under a condition to be included in OLAP machine such that the old machine has no effect under. In such case, adding a new rule preserves the existing functions from the old machine.

2. *Add a new controlled function to the OLAP ASM*: This is used to model a view that is needed for the support of any new OLAP function, provided the existing view definitions are not yet sufficient.

3. *Add new controlled function(s) to the DW ASM*: This is used to model the schema that is needed in supporting the new OLAP function.

4. *Integrate controlled functions on the DW ASM*: This is used whenever the schema is extended. As a consequence, the view creation rules on the OLAP ASM must be changed accordingly.

   The integration process aims to preserving the information by the notion of schema equivalence and dominance when two schemas are integrated. This step relates a set of schema transformation rules.

5. *Add additional controlled functions to the DW ASM*: This is a consequence of the view integration, when new schema should be added to data warehouse after the integration.

6. *Change the rules on DW ASM*: These rules are defined for extracting data for the data warehouse refresh. Thus each change to a data warehouse schema, the corresponding extraction rule should be adapted.

7. *Change the rules on DB ASM*: These rules are used in data extraction upon data warehouse refresh request. Any changes, either new addition or updates, to data extraction rules should be reflected in the related rules in DB ASM.

8. *Change the functions/rules on OLAP ASM*: This is used to change the functions or rules that are affected in this refinement process, such as rules that make reference to the schemas which are changed during the integration, or rules that process the newly added OLAP functions.

4.2 Optimisation

Some refinements are used to optimise the performance of the system. These refinement rules are applied to reorganise the specification independently from the user requirements simply for optimisation reasons. Refinements for system optimisation can be classified under procedural refinement in ASM method.

We consider some typical optimisation steps in data warehousing:

– To materialise the OLAP views: That is, to compute the OLAP queries in advance and store them as views in the data warehouse. When the queries are called, they can be answered by the stored views instantly from the data warehouse without waiting for computation of the queries. This will speed up the system performance particularly as business analysis is usually data intensive, but it also result in the issue of view maintenance.

– To update the data warehouse incrementally: That is, not to recompute the queries from scratch, as the case in our ground model but only propagate the changes to the data warehouse.

Again a tailored refinement process for systematically incorporating the above two optimisation steps is specified as follows:

10. *Incorporate view materialisation* :
    (a) *Add new controlled function in DW ASM* : This is used to add the OLAP views to the data warehouse as the materialised views.

        For a more effective approach in view selection we can adopt some selection process or algorithm.
    (b) *Integrate materialised views in DW ASM* : This is used to reduce redundancy that may have occurred after more views are materialised. A set of transformation rules can be applied.
    (c) *Add new rules in DW ASM* : This is used to define the transition rules to maintain the materialised views up to date with the data warehouse changes. These rules are called after each refreshing of the data warehouse.
    (d) *Change the rules in DW ASM* : These rules are for opening datamart for the OLAP ASM. After the view materialisation or view integration, these rules need to be adapted too.
11. *Incorporate incremental updates* :
    (a) *Add monitored functions in DB ASM* : This is used to define relations to store updates of the source relations, called delta files.
    (b) *Add controlled functions in DW ASM* : This is used to define relations for store computed changes for data warehouse relations.
    (c) *Add rules in DW ASM* : This is used to define the rules for computing the changes from source relations and propagating changes into data warehouse relations.
    (d) *Replace rules in DB ASM* : This is used to replace the refresh rules with the rules for incremental updates.

4.3 Implementation

The final group of the refinements in our discussion is the system implementation refinements. Our refinements for implementation are not just limited to those dealing with the refinements which moves the specification to codes but also those refinements which realise high level design decisions such as data distribution, the focus of the refinement process in the following. This group of refinements can be classified under the procedural or data refinement in ASM method.

12. *Apply implementation refinements*: These refinement rules apply to the ASMs on all three levels and consist of realising design decisions for moving the ASMs closer to their implementation while preserving the semantics of runs. It is not intended to discuss this further since the topic of moving specification to codes has been thoroughly discussed in [Schewe, 1997].

13. *Distribution design*:

    (a) *Replicate the data warehouse and the OLAP ASMs*: For each node in the network assume the same copy of the data warehouse ASM and the OLAP ASM.

    (b) *Remove controlled functions and rules in local OLAP ASMs*: If the needed OLAP functionality is different at different network nodes, then these rules will simply reduce the corresponding OLAP ASM.

    (c) *Fragment controlled functions in local data warehouse ASMs*: This rules will reorganise and reduce a local data warehouse ASM, if the corresponding OLAP ASM does not need all of the replicated data warehouse. The refresh rules are then adapted accordingly.

    (d) *Recombine fragments in local data warehouse ASMs*: This rules will reorganise a local data warehouse ASM according to query cost considerations. The refresh rules are then adapted accordingly.

    (e) *adapt the view creation rules accordingly in local OLAP ASM*: This is used when fragmentation is implemented. The OLAP views are created over the fragments at the local data warehouse.

5 Refinement Rules

Our intention is to set up rules of the form

$$\frac{\mathfrak{M} \rhd aFunc, \ldots, aRule, \ldots}{\mathfrak{M}^* \rhd newFunc, \ldots, newRule = \ldots}\,\varphi$$

That is, we indicate under some side conditions φ, which parts of the machine $\mathfrak{M}$ will be replaced by new functions and rules in a refining machine $\mathfrak{M}^*$. Furthermore, the specification has to indicate, which relations belong to the schema, and the correspondence between main rules.

Due the space constraint, we will introduce a subset of the rules which are used in the examples.

5.1 Schema Extension

This group of rules deal with the schema extensions. This either concerns new attributes, new types, new subtypes or the simplification of hierarchies. These rules are needed in step 1 of our method.

Rule 1 *Add a new type $\mathcal{R}$. In such case we obtain a dominant schema.*

$$\mathfrak{M}^* \ \rhd \mathcal{R} = \ell : label \times t$$

With a new type added we get a dominant schema. The corresponding abstraction predicate $\mathcal{A}$ is simply defined as *true*. The corresponding computable queries f and g can be defined as identity function.

Rule 2 *Add a new attribute A to the type R, i.e. $attr(R_{new}) = attr(R) \cup \{A\}$. In addition, the new attribute may be used to extend the key, i.e. we may have $key(R_{new}) = key(R) \cup \{A\}$.*

$$\frac{\mathfrak{M} \rhd R = t \to \{\mathbb{1}\}}{\mathfrak{M}^* \ \rhd R_{new} = A : t_a \times t \to \{\mathbb{1}\}}$$

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall x.(\exists a.R_{new}(a, x) = 1 \Leftrightarrow R(x) = 1)$$

The corresponding computable queries f and g:

$$R_{new} := \{(a, x) \mid (x) \in R\}, \text{ for a constant } a$$

and

$$R := \{(x) \mid \exists a.(x, a) \in R_{new}\}$$

respectively.

Adding a new attribute A by *Rule 2* does not change the cardinality of the type R, i.e. $card(R) = card(R_{new})$. This rule always results in a dominant schema.

The next two rules allow to introduce a new subtype via selection or projection on non-key-attributes. In both cases we have schema equivalence.

Rule 3 *For a type R introduce a new relationship type R'_{new} with $comp(R'_{new})$ $= \{r : R\} = key(R'_{new})$ and add a constraint $R'_{new} = \sigma_\varphi(R)$ for some selection formula φ.*

$$\frac{\mathfrak{M} \rhd R = r : id \times t \to \{\mathbb{1}\}}{\begin{array}{l}\mathfrak{M}^* \rhd R = r : id \times t \to \{\mathbb{1}\} \\ \qquad R'_{new} = r : ref \to \{\mathbb{1}\}\end{array}}$$

with a constraint:

$$R'_{new} = \sigma_\psi(R)$$

for some selection formula ψ.

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall r.(R'_{new}(r) = 1 \Leftrightarrow \exists x.\psi(x) = 1 \wedge R(r, x) = 1)$$

The corresponding computable queries f:

$$R'_{new} := \{(r) \mid \exists x.(\psi(x) = 1 \wedge (r, x) \in R)\}$$

and g an identity function, respectively.

Rule 4 *For a type R and attributes $A_1, \ldots, A_n \in attr(R)$ such that there are no $B_i \in key(R)$ with $A_i \geq B_i$ (for projection on non-key-attributes) introduce a new relationship type R'_{new} with $comp(R'_{new}) = \{r : R\} = key(R'_{new})$ and $attr(R'_{new}) = \{A_1, \ldots, A_n\}$, and add a constraint $R'_{new} = \pi_{A_1,\ldots,A_n}(R)$.*

$$\frac{\mathfrak{M} \rhd R = r : id \times A_1 : t_1 \times \ldots A_n : t_n \times t \to \{\mathbb{1}\}}{\begin{array}{l}\mathfrak{M}^* \rhd R = r : id \times A_1 : t_1 \times \ldots A_n : t_n \times t \to \{\mathbb{1}\} \\ \qquad R'_{new} = r : ref \times A_1 : t_1 \times \ldots A_n : t_n \to \{\mathbb{1}\}\end{array}} \varphi$$

with the side condition φ:

$$\forall B_i \in key(R) \text{ we have } A_i \ngeq B_i.$$

The constraint:

$$key(R'_{new}) = \{r : ref\}$$

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall x_1, \ldots, x_n, r.((R'_{new}(r, x_1, \ldots, x_n) = 1) \Leftrightarrow \exists x.R(r, x_1, \ldots, x_n, x) = 1)$$

The corresponding computable queries f:

$$R'_{new} := \{(r, x_1, \ldots x_n) \mid \exists x.(r, x_1, \ldots x_n, x) \in R\}$$

and g an identity function, respectively.

Rule 5 *Replace types R, $R_1, \dots, R_n$ with $comp(R_i) = \{r_i : R\} = key(R_i)$ and $card(R_i, R) = (1,1)$ $(i = 1, \dots, n)$ by a new type R_{new} with $comp(R_{new}) = comp(R)$, $attr(R_{new}) = attr(R) \cup \bigcup_{i=1}^{n} attr(R_i)$ and $key(R_{new}) = key(R)$.*

$$\frac{\begin{array}{l}\mathfrak{M} \rhd R = r : id \times t \to \{\mathbb{1}\} \\ \quad R_1 = r : ref \times t_1 \to \{\mathbb{1}\} \\ \quad \dots \\ \quad R_n = r : ref \times t_n \to \{\mathbb{1}\}\end{array}}{\mathfrak{M}^* \rhd R_{new} = r : id \times t \times t_1 \cdots \times t_n \to \{\mathbb{1}\}} \varphi$$

with the side condition φ:

$$comp(R_i) = \{r_i : R\} = key(R_i) \wedge card(R, R_i) = (1,1)(i = 1, \dots, n)$$

The constraint:

$$key(R_{new}) = key(R)$$

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall x_1, \dots, x_n, r, x.(R'_{new}(r, x, x_1, \dots, x_n) = 1 \Leftrightarrow$$
$$(R(r, x) = 1 \wedge R_1(r, x_1) = 1 \wedge \dots \wedge R_n(r, x_n) = 1))$$

The corresponding computable queries f and g:

$$R_{new} := \{(r, x, x_1, \dots x_n) \mid (r, x) \in R \wedge (r, x_1) \in R_1 \wedge \dots \wedge (r, x_n) \in R_n\}$$

and

$$R := \{(r, x) \mid \exists x_1, \dots x_n.((r, x, x_1, \dots, x_n) \in R_{new})\} \|$$
$$R_1 := \{(r, x_1) \mid \exists x, x_2, \dots, x_n.(r, x, x_1, \dots, x_n) \in R_{new}\} \| \dots \|$$
$$R_n := \{(r, x_n) \mid \exists x, x_1, \dots, x_{n-1}.(r, x, x_1, \dots, x_n) \in R_{new}\}$$

respectively.

5.2 Type Integration

This group of rules deals with the integration of types in step 4 of our method. *Rule 6* considers the equality case, *Rule 7* considers the containment case, and *Rule 8* covers the overlap case. Note that these transformation rules cover the core of the approaches in [Koh and Chen, 1994; Spaccapietra and Parent, 1994; Larson et al., 1989].

Rule 6 *If R_1 and R_2 are types with $key(R_1) = key(R_2)$ and we have the constraint $R_1[key(R_1) \cup X] = f(R_2[key(R_2) \cup Y])$ for some $X \subseteq comp(R_1) \cup attr(R_1)$, $Y \subseteq comp(R_2) \cup attr(R_2)$ and a bijective mapping f, then replace these types by R_{new} with $comp(R_{new}) = comp(R_1) \cup (comp(R_2) - Y - key(R_2))$, $attr(R_{new}) = attr(R_1) \cup (attr(R_2) - Y - key(R_2)) \cup \{D\}$ and $key(R_{new}) = key(R_1) \cup \{D\}$ and an optional new distinguishing attribute D.*

$$\frac{\begin{array}{c}\mathfrak{M} \rhd R_1 = K_1 : t_{k_1} \times X : t_x \times t_1 \to \{\mathbb{1}\} \\ R_2 = K_2 : t_{k_2} \times Y : t_y \times t_2 \to \{\mathbb{1}\}\end{array}}{\mathfrak{M}^* \rhd R_{new} = K_1 : t_{k_1} \times X : t_x \times t_1 \times t_2 \to \{\mathbb{1}\}} \varphi$$

with the side condition φ:

$$K_1 = K_2 \wedge R_1[K_1 \cup X] = h(R_2[K_2 \cup Y]),$$

where K_1 and K_2 are the keys, h is a fixed computable bijective mapping.

The corresponding abstraction predicate $\mathcal{A}$:

$$\begin{array}{c}\forall k_1, x, x_1, x_2.(R_{new}(k_1, x, x_1, x_2) = 1 \\ \Leftrightarrow R_1(k_1, x, x_1) = 1 \wedge R_2(h^{-1}(k_1, x), x_2) = 1)\end{array}$$

The corresponding computable queries f and g:

$$R_{new} := \{(k_1, x, x_1, x_2) \mid (k_1, x, x_1) \in R_1 \wedge (h^{-1}(k_1, x), x_2) \in R_2\}$$

and

$$\begin{array}{r}R_1 := \{(k_1, x, x_1) \mid \exists x_2.(k_1, x, x_1, x_2) \in R_{new}\} \| \\ R_2 := \{(h^{-1}(k_1, x), x_2) \mid \exists x_1.(k_1, x, x_1, x_2) \in R_{new}\}\end{array}$$

respectively.

When X and Y are empty, then *Rule 6* merges two types by combining the two attribute sets.

Rule 7 *If R_1 and R_2 are types with $key(R_1) = key(R_2)$ and the constraint $R_2[key(R_2) \cup Y] \subset f(R_1[key(R_1) \cup X]$ holds for some $X \subseteq comp(R_1) \cup attr(R_1)$, $Y \subseteq comp(R_2) \cup attr(R_2)$ and a bijective mapping f, then replace R_1 by $R_{1,new}$ with $comp(R_{1,new}) = comp(R_1)$, $attr(R_{new}) = attr(R_1) \cup \{D\}$ and $key(R_{new}) = key(R_1) \cup \{D\}$ and an optional new distinguishing attribute D. Furthermore, replace R_2 by $R_{2,new}$ with $comp(R_{2,new}) = \{r_{new} : R_{1,new}\} \cup (comp(R_2) - Y - key(R_2))$, $attr(R_{2,new}) = attr(R_2) - Y - key(R_2)$ and $key(R_{2,new}) = \{r_{new} : R_{1,new}\}$.*

$$\frac{\begin{array}{c}\mathfrak{M} \rhd R_1 = r_1 : id \times K_1 : t_{k_1} \times X : t_x \times t_1 \rightarrow \{\mathbb{1}\} \\ R_2 = K_2 : t_{k_2} \times Y : t_y \times t_2 \rightarrow \{\mathbb{1}\}\end{array}}{\begin{array}{c}\mathfrak{M}^* \rhd R_1 = r_1 : id \times K_1 : t_{k_1} \times X : t_x \times t_1 \rightarrow \{\mathbb{1}\} \\ R_{2,new} = r_1 : ref \times t_2 \rightarrow \{\mathbb{1}\}\end{array}} \varphi$$

with the side condition φ:

$$K_1 = K_2 \wedge R_2[K_2 \cup X] \subseteq h(R_1[K_1 \cup Y]),$$

where K_1 and K_2 are the keys, h is a fixed computable bijective mapping.

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall k, x, x_1, x_2, r.(R_1(r, k, x, x_1) = 1 \Rightarrow$$
$$(R_2(h(k, x), x_2) = 1 \Leftrightarrow R_{2,new}(r, x_2) = 1))$$

The corresponding computable queries f and g:

$$R_{2,new} := \{(r, x_2) \mid \exists k, x.((k, x, x_2) \in R_2 \wedge \exists x_1.(r, h(k, x), x_1) \in R_1\}$$

and

$$R_2 := \{(k, x, x_2) \mid \exists r.((r, x_2) \in R_{2,new} \wedge \exists x_1.(r, h(k, x), x_1) \in R_{1,new})\}$$

respectively.

Rule 8 *If R_1 and R_2 are types with $key(R_1) = key(R_2)$ such that for $X \subseteq comp(R_1) \cup attr(R_1)$, $Y \subseteq comp(R_2) \cup attr(R_2)$ and a bijective mapping f the constraints*

$$\begin{array}{ll} R_2[key(R_2) \cup Y] \subseteq f(R_1[key(R_1) \cup X]) & , \\ R_2[key(R_2) \cup Y] \supseteq f(R_1[key(R_1) \cup X]) & \textit{and} \\ R_2[key(R_2) \cup Y] \cap f(R_1[key(R_1) \cup X]) = \emptyset & \end{array}$$

are not satisfied (the first two cases are covered by Rule 6 *and* 7*, the last one has no case for integration) then replace R_1 by $R_{1,new}$ with $comp(R_{1,new}) = \{r_{1,new} : R_{new}\} \cup (comp(R_1) - X - key(R_1))$, $attr(R_{1,new}) = attr(R_1) - X - key(R_1)$ and $key(R_{1,new}) = \{r_{1,new} : R_{new}\}$, replace R_2 by $R_{2,new}$ with $comp(R_{2,new}) = \{r_{new} : R_{1,new}\} \cup (comp(R_2) - Y - key(R_2))$, $attr(R_{2,new}) = attr(R_2) - Y - key(R_2)$ and $key(R_{2,new}) = \{r_{new} : R_{1,new}\}$ and introduce a new type R_{new} with $comp(R_{new}) = comp(R_1) \cap (key(R_1) \cup X)$, $attr(R_{new}) = attr(R_1) \cap (X \cup key(R_1) \cup \{D\}$, and $key(R_{new}) = key(R_1) \cup \{D\}$ and an optional new distinguishing attribute D.*

$$\frac{\begin{array}{r} \mathfrak{M} \rhd R_1 = K_1 : t_{k_1} \times X : t_x \times t_1 \to \{\mathbb{1}\} \\ R_2 = K_2 : t_{k_2} \times Y : t_y \times t_2 \to \{\mathbb{1}\} \end{array}}{\begin{array}{r} \mathfrak{M}^* \rhd R_{1,new} = r : ref \times t_1 \to \{\mathbb{1}\} \\ R_{2,new} = r : ref \times t_2 \to \{\mathbb{1}\} \\ R_{new} = r : id \times K_1 : t_{k_1} \times X : t_x \to \{\mathbb{1}\} \end{array}} \varphi$$

with the side condition φ:

$$K_1 = K_2$$

where K_1 and K_2 are the keys.

The guideline: apply the rule when none of the following hold:

$$\begin{array}{l} R_2[key(R_2) \cup Y] \subseteq h(R_1[key(R_1) \cup X]), \\ R_2[key(R_2) \cup Y] \supseteq h(R_1[key(R_1) \cup X]) \quad \text{and} \\ R_2[key(R_2) \cup Y] \cap h(R_1[key(R_1) \cup X]) = \emptyset \end{array}$$

with a fixed computable bijective mapping h.

The corresponding abstraction predicate $\mathcal{A}$: $\forall k, x, x_1, x_2.$
$((\exists r.(R_{new}(r,k,x) = 1 \wedge R_{1,new}(r,x_1) = 1) \Leftrightarrow R_1(k,x,x_1) = 1) \wedge$
$(\exists r.(R_{new}(r,k,x) = 1 \wedge R_{2,new}(r,x_2) = 1) \Leftrightarrow R_2(h^{-1}(k,x),x_2) = 1))$

The corresponding computable queries f and g:

$$\begin{array}{r} R_{1,new} := \{(z(k), x_1) \mid \exists x.(k, x, x_1) \in R_1\| \\ R_{2,new} := \{(z(k), x_2) \mid \exists x.(h(k,x), x_2) \in R_2\| \\ R_{new} := \{(z(k), k, x) \mid \exists x_1.(k, x, x_1) \in R_1 \vee \exists x_2.(h(k,x), x_2) \in R_2\} \end{array}$$

and

$$\begin{array}{r} R_1 := \{(k, x, x_1) \mid \exists r.((r,k,x) \in R_{new} \wedge (r, x_1) \in R_{1,new})\| \\ R_2 := \{(h(k,x), x_2) \mid \exists r.((r,k,x) \in R_{new} \wedge (r, x_2) \in R_{2,new})\} \end{array}$$

respectively, for some fixed computable injective function $z : K_1 \to id$.

Rule 9 *If R and R' are types with $comp(R') \cup attr(R') = Z \subseteq comp(R) \cup attr(R)$ such that the constraint $R' = \sigma_\varphi(\pi_Z(R))$ holds for some selection condition φ, then omit R'.*

$$\frac{\begin{array}{r} \mathfrak{M} \rhd R = Z : t_z \times t \to \{\mathbb{1}\} \\ R' = Z : t_z \to \{\mathbb{1}\} \end{array}}{\mathfrak{M}^* \rhd R = Z : t_z \times t \to \{\mathbb{1}\}} \varphi$$

with side condition φ:

$$R' = \sigma_\psi(\pi_Z(R))$$

for some selection condition ψ.

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall z.(R'(z) = 1 \Leftrightarrow \exists x.R(z,x) = 1 \wedge \psi(z) = 1)$$

The corresponding computable queries g:

$$R' := \{(z) \mid \exists x.(z,x) \in R_1 \wedge \psi(z) = 1\}$$

and f an identity function.

6 Dynamic Data Warehouse Design

As performance is a critical issue in data warehousing, we will look at the impacts of the system dynamics on the set of materialised views in this section. We suggest to materialise a new view under a space constraint if it is beneficial for the overall query performance, even if the new query can be computed from the existing materialised views. In addition, we apply view integration techniques whenever new views are added.

We first present the models for query evaluation and view maintenance and for determining view selection, and then describe the view selection process. We demonstrate how the system model is refined using some examples.

6.1 Cost and Benefit Model

For simplicity, we adopt the basic idea from [Harinarayan et al., 1996] in estimation of the query evaluation cost and view maintenance cost. That is, we use the size of a view v as its maintenance cost and the query evaluation cost if the query is computed totally from v:

$$qcost(q,v) = s(v)$$

Similarly, we have the view maintenance cost:

$$mcost(v) = s(v)$$

Furthermore, we introduce the notion of benefit for computing a view v from a materialised view v_1 instead of v_2, where v is queried with frequency $f(v)$:

$$b(v, v_1, v_2) = (s(v_2) - s(v_1)) \cdot f(v)$$

When presented with a new OLAP view v, we have to decide whether to materialise it, or compute it from an existing materialised view v_2. For that we could simply compare the benefit $b(v, v, v_2)$ with the maintenance cost $mcost(v)$, and materialise v if the benefit is greater than the cost.

However, it is possible that another OLAP view v' which has not been materialised also benefits from materialising v. This happens if

1. v' could be compute from v (denoted by $v' \succ v$), and
2. $s(mv(v')) > s(v)$, where $mv(v')$ is the materialized view from which v' is currently computed

In this case we get an additional benefit of $b(v', v, mv(v'))$.

In order to decide whether or not to materialize v, we have to add up the benefits for all OLAP views, and compare this figure to the maintenance cost. The OLAP views benefitting are

$$bv(v) = \{v' \in \text{OLAP-views} \mid v' \succ v \wedge mv(v') > v\}$$

The total benefit $b(v)$ of materializing v is thus:

$$b(v) = \sum_{v' \in bv(v)} b(v', v, mv(v'))$$

Further we introduce a notion of benefit for comparing two materialised views v_1 over v_2 in computing query q of frequency f as:

$$b(v_1, v_2) = (s(v_2) - s(v_1)) \times (f + 1)$$

where both the query cost and the view maintenance cost are considered. If b is positive, it means materialising v_1 is more beneficial. The other way round otherwise.

In the case that one of the views is already materialised, thus maintaining it does not involve additional cost, the benefit of using existing materialized view v_1 to compute query q over creating a new view v for query q is composed:

$$bx(v_1) = (s(v) - s(v_1)) \times f + s(v)$$

For deciding if materialising a new view v is more favorable for other OLAP view o, the benefit is estimated:

$$bz(o) = ((s(v(o)) - s(v)) \times f(o)$$

where $v(o)$ is a materialised view which is used for computing o.

If the sum of $bz(o)$, where o is all the OLAP views which make the $bz(o)$ greater than 0, is greater than $s(v)$, we consider it is beneficial to materialise view v, and therefore we rewrite all the OLAP views o using v.

The above estimation is rather simple. However, it can be replaced by a more comprehensive one easily.

6.2 View Selection Process

The basic idea of our view selection is that under a given space constraint S, we always materialise a new view if it can not be computed from the existing materialised views, or it is more beneficial for computing other OLAP queries. Although we should be more concerned with the time for refreshing the materialised views than the storage space, our justification is that the larger the total size of the views, the longer it will take to maintain. Whenever there is a new OLAP view added, we invoke the selection process to check if the set of existing materialised views can be used to compute the OLAP view. In order to do so, we define a notion of fineness to compare two views, such that we can compute the less fine one from the finer one. We agree with [Kotidis and Roussopoulos, 2001] that it is rarely beneficial to compute a view using multiple views working with typical OLAP queries involving aggregations, and thus we do not consider this option.

Definition 4. A view(query) v_1 is said to be *finer* than v_2, denoted as $v_1 \succ v_2$, if v_2 is computable from v_1 by aggregating operations.

Example 1. If v_1 is the view of *sales by day*, and v_2 is the view of *sales by month*, then v_1 is *finer* than v_2, since we can get the monthly sales by summing up the daily sales for the month. □

Our view selection algorithm is used for determining whether a new view v is to be materialized for a query q, or whether an existing materialized view m' should be used. We proceed with finding the best candidate m' from the materialised view set mv based on $s(m')$. If not found, v will be materialised if it meets the constraint S. Otherwise, i.e. if there exists a materialised view m' from which q can be computed, we move on with calculating the total benefit $b(v)$ of materialising v. If the sum of $b(v)$ is greater than the maintenance cost $s(v)$, we materialise v and rewrite all queries which benefit. We still use m' for computing or updating the materialized view v, as this is more efficient.

Let mv be the set of materialised views, ov the set of OLAP views, and v the newly added view. We denote the frequency of OLAP view v by $f(v)$ and its size by $s(v)$. The function $mview : ov \rightarrow mv$ maps OLAP views to the corresponding materialized views from which they are calculated. Finally, the space constraint S_{max} gives the maximal space allowed for the materialised views.

The following algorithm describes how we can update ov, mv and the function $mview$ when a new OLAP view v is added.

$select(mv, ov, v(q), f(v), m') =$
  $b_{max} := 0, b_{sum} := 0$
  `forall` $m \in mv$ `do`

```
        if m ≻ v(q) then do
            b(m) := (s(v(q)) − s(m)) × f(v) + s(v(q));
            if b(m) > b_max then
                b_max = b(m), m' := m enddo
    enddo
    if b_max = 0 ∧ S' + s(v(q)) ≤ S then mv := mv ∪ v(q)
    else forall o ∈ ov do
        if v(q) ≻ o then do
            b(o) := (s(the-view(o)) − s(v)) × f(o)
            b_sum := b_sum + b(o) enddo
    enddo

    if b_sum > s(v) ∧ S' + s(v(q)) ≤ S then mv := mv ∪ v(q)
```

```
add_view(v) =
  ov := ov ∪ {v}
  m' := ⊥
  forall m ∈ mv with m ≻ v do
    if m' = ⊥ or s(m) < s(m') then
      m' := m
  enddo
  mview(v) := m'
  if s(mv) + s(v) ≤ S_max then do
    bv := {v}
    forall o ∈ ov with v ≻ o do
      if s(mview(o) > s(v)) then
        bv := bv ∪ {o},
        b := b + (s(mview(o)) − s(v)) · f(o)
    enddo
    if b > s(v) then do
      mv := mv ∪ {v}
      forall o ∈ bv do
        mview(o) := v
    enddo
  enddo
```

```
add_view(v) =
  ov := ov ∪ {v}
  mview(v) := m' with m' ∈ mv, m' ≻ v and s(m') minimal
  if s(mv) + s(v) ≤ S_max then do
```

$bv := \{v' \in ov \mid v' \succ v \wedge s(mview(v')) > s(v)\}$
$b := \sum_{v' \in bv}(s(mview(s')) - s(v)) \cdot f(v')$
if $b > s(v)$ then do
  $mv := mv \cup \{v\}$
  forall $v' \in bv$ do
    $mview(v') := v$ enddo enddo

6.3 Some Examples

Example 2. Let us look at a case for view selection. Assume that our current data warehouse has two OLAP views:

View $\mathcal{V}_1$: the total sale by shop and day, its average number of tuples: $s(\mathcal{V}_1) = 20000$, its frequency $f(\mathcal{V}_1) = 2$;

View $\mathcal{V}_2$: the total sale by state and month, its average number of tuples: $s(\mathcal{V}_2) = 5000$, its frequency $f(\mathcal{V}_2) = 0.3$;

Furthermore, $\mathcal{V}_1$ is materialized, $\mathcal{V}_2$ is rewritten from $\mathcal{V}_1$ and not materialised, and space is not a concern in this case.

Now the user requests for a new OLAP query, view $\mathcal{V}$, the total sale by region and day, its average number of tuples: $s(\mathcal{V}) = 10000$, its frequency $f(\mathcal{V}) = 0.8$.

We now need to decide whether to materialize $\mathcal{V}$, or instead compute $\mathcal{V}$ from one of the existing materialized views. For the latter case, the best (and only) materialized view to compute $\mathcal{V}$ from is $\mathcal{V}_1$, since $\mathcal{V}_1 \succ \mathcal{V}$ and $\mathcal{V}_2 \not\succ \mathcal{V}$.

The views which would benefit from materializing $\mathcal{V}$ are $\mathcal{V}$ and $\mathcal{V}_2$, so the benefit $b(\mathcal{V})$ of doing so can be computed as

$$\begin{aligned} b(\mathcal{V}) &= (s(\mathcal{V}_1) - s(\mathcal{V})) \cdot f(\mathcal{V}) + (s(\mathcal{V}_1) - s(\mathcal{V})) \cdot f(\mathcal{V}_2) \\ &= (20000 - 10000) \cdot 0.8 + (20000 - 10000) \cdot 0.3 \\ &= 11000 \end{aligned}$$

On the other hand, the update cost $s(\mathcal{V})$ for materializing $\mathcal{V}$ is only 10000. Thus we should materialize $\mathcal{V}$ and rewrite the query for $\mathcal{V}_2$ to use $\mathcal{V}$. □

It is obvious that $\mathcal{V}_1 \succ \mathcal{V}$ holds, which means we can rewrite the new view from $\mathcal{V}_1$. Our estimation of the benefit of rewriting $\mathcal{V}$ from $\mathcal{V}_1$:

$$\begin{aligned} b(\mathcal{V}_1) &= (s(\mathcal{V}) - s(\mathcal{V}_1) \times f(\mathcal{V}) + s(\mathcal{V}) \\ &= (18000 - 20000) \times 5 + 18000 \\ &= 8000 \end{aligned}$$

As we have $b_{max} = b(\mathcal{V}_1)$ positive, we should rewrite $\mathcal{V}$ from $\mathcal{V}_1$. However, we shall also consider if materialize $\mathcal{V}$ will be more beneficial for other OLAP views which are rewritten from $\mathcal{V}_1$.

It is obvious that we have $\mathcal{V} \succ \mathcal{V}_2$, so the benefit of writing $\mathcal{V}_2$ from $\mathcal{V}$ is estimated:

$$\begin{aligned} b(\mathcal{V}_2) =& (s(\mathcal{V}_1) - s(mathcalV)) \times f(mathcalV_2) \\ &= (20000 - 18000) \times 10 \\ &= 20000 \end{aligned}$$

Now we have $b_{sum} = b(o)$ and $b_{sum} > s(\mathcal{V})$, that means, we should materialise the new view $\mathcal{V}$ and rewrite $\mathcal{V}_2$ from $\mathcal{V}$. □

Example 3. Let us look at a case of dynamic data warehouse design. Assume the money figures in the ground model is in US\$, and we have OLAP query on total sales in US\$ with no figures on profit for the sake of the example. A new OLAP query is requested from the store manager for total sales in EURO and the corresponding profit.

It is obvious that we are not able to rewrite the new view from the existing total sales with the data on profit missing, so we extend the materialsied view set with the view for the new query. Let us define a function *cnv* for converting from US\$ to EURO. Let us recall the relevant relations in the star schema:

Shop:$sid \times name \times town \times region \times state \times phone \to \{1\!\!1\}$,
Time:$date \times day \times week \times month \times quarter \times year \to \{1\!\!1\}$,
Purchase:$cid \times sid \times pid \times date \times qty \times sale \times profit \to \{1\!\!1\}$

We proceed with the refinement steps as follows:

1. Add a new rule to the OLAP ASM:

$$\begin{aligned} &\mathit{create\_V\_sales\_euro} = \texttt{forall}\ s, r, st, m, q, y\ \texttt{with} \\ &\quad \exists n, t, ph.(s, n, t, r, st, ph) \in \text{Shop} \wedge \\ &\quad \exists d, d', w.(d, d', w, m, q, y) \in \text{Time} \\ &\quad \texttt{do let}\ S = src[0, \pi_{s'}, +](\{(i, s, p, d, s') \mid \exists q', p'. \\ &\qquad\quad (i, s, p, d, q', s', p') \in \text{Purchase} \\ &\qquad P = src[0, \pi_{p'}, +](\{(i, s, p, d, p') \mid \exists q', s'. \\ &\qquad\quad (i, s, p, d, q', s', p') \in \text{Purchase} \\ &\qquad \texttt{in}\ \text{V\_sales\_euro}(s, r, st, m, q, y, cnv(S), cnv(P)) := 1\ \texttt{enddo} \end{aligned}$$

2. Add a new controlled function to the OLAP ASM: apply the schema transformation rule 1, to add the OLAP view V_Msales_euro to the OLAP-ASM:

$$\begin{aligned} &\text{OLAP-ASM}^* \rhd \text{V\_sales\_euro} = shop \times region \times st \times \\ &\qquad month \times qtr \times year \times msale\_euro \times profit\_euro \to \{1\!\!1\} \end{aligned}$$

3. Invoke view selection. As we have indicated we need to materialise the new view, and assume space is not a concern in this case, we proceed with adding the new view to the materialised view set in the next step.

4. Apply the schema transformation rule 1 to add the new OLAP view to the DW-ASM as a materialised view:

$$\text{DW-ASM}^* \rhd \text{MV\_V\_sales\_euro} = shop \times region \times st \times month \times qtr \times year \times msale\_euro \times profit\_euro \to \{\mathbb{1}\}$$

5. Integrate controlled functions on the DW ASM: apply type integration *rule 6* to the materialised views as follows:

$$\frac{\begin{aligned}\text{DW-ASM} \rhd MV\_V\_sales\_euro &= shop \times region \times st \times month \times \\ &\quad qtr \times year \times msale\_euro \\ &\quad \times profit\_euro \to \{\mathbb{1}\} \\ MV\_V\_sales &= shop \times region \times st \times month \times \\ &\quad qtr \times year \times msale \to \{\mathbb{1}\}\end{aligned}}{\begin{aligned}\text{DW-ASM}^* \rhd MV\_V\_sales\_prf &= shop \times region \times st \times month \times qtr \\ &\quad \times year \times msale \times \\ &\quad profit\_euro \to \{\mathbb{1}\}\end{aligned}} \varphi$$

Then the side condition φ of rule 6 is satisfied as:

- both of the types have the same key: $sh \times month \times year$;
- and they map to the same tuples by that the data being originated from the same data warehouse and with no further selections;
- and the bijective mapping is defined as $h := (id, id, id, id, cnv)$, where id is an identity function for mapping the keys and other three identical attributes.

6. As a consequence of the integration, we need to replace *refresh_MV_V_sales*, which is created after view materialisation is incorporated at the DW-ASM:

refresh-MV_V_sales=
  create_V_sales;
  forall s, r, st, m, q, y, S **with**
    $(s, r, st, m, q, y, S) \in$ V_sales **do**
      MV_V_sales$(s, r, st, m, q, y, S) := 1$ **enddo**

by a new rule *refresh_MV_V_sales_prf* as follows:

refresh_MV_V_sales_prf =
  create_MV_V_sales_prf;
  forall s, r, st, m, q, y, S, P **with**
    $(s, r, st, m, q, y, S, P) \in$ V_sales **do**
      MV_V_sales_prf$(s, r, st, m, q, y, S, cnv^{-1}(P)) := 1$ **enddo**

7. Similarly we replace the view creation rules *create_V_sales* and *create_V_sales_euro* at the OLAP-ASM by the followings:

$$
\begin{array}{l}
\mathit{create\_V\_sales\_prf} \; = \\
\quad \texttt{forall}\; s, r, st, m, q, y \;\texttt{with} \\
\qquad \exists n, t, ph.(s, n, t, r, st, ph) \in \text{Shop} \wedge \\
\qquad \exists d, d', w.(d, d', w, m, q, y) \in \text{Time} \\
\qquad \texttt{do let}\; S = src[0, \pi_{s'}, +] \\
\qquad\qquad (\{(i, s, p, d, s') \mid \exists q', p'. \\
\qquad\qquad (i, s, p, d, q', s', p') \in \text{Purchase} \wedge \\
\qquad\qquad d.month = m \wedge d.year = y\}) \\
\qquad\quad P = src[0, \pi_{p'}, +] \\
\qquad\qquad (\{(i, s, p, d, p') \mid \exists q', s'. \\
\qquad\qquad (i, s, p, d, q', s', p') \in \text{Purchase} \wedge \\
\qquad\qquad d.month = m \wedge d.year = y\}) \\
\qquad \texttt{in}\; \text{V\_sales\_prf}(s, r, st, m, q, y, S, cnv^{-1}(P)) := 1 \\
\quad \texttt{enddo}
\end{array}
$$

8. Then we need to change the datamart opening rules accordingly in DW-ASM:

$$
\begin{array}{l}
\mathit{open\_datamart}(dm) = \texttt{case}\; \text{the-matching-view}(dm) \;\texttt{of} \\
\quad \text{V\_sales} : \texttt{forall}\; s, r, st, m, q, y, S, P \;\texttt{with} \\
\qquad (s, r, st, m, q, y, S, P) \in \text{MV\_V\_sales\_prf} \;\texttt{do} \\
\qquad\quad \text{DM-V\_sales}(dm, s, r, st, m, q, y, S) := 1 \;\texttt{enddo} \\
\quad \text{V\_sales\_euro} : \texttt{forall}\; s, r, st, m, q, y, S, P \;\texttt{with} \\
\qquad (s, r, st, m, q, y, S, P) \in \text{MV\_V\_sales\_prf} \;\texttt{do} \\
\qquad\quad \text{DM-V\_sales\_euro}(dm, s, r, st, m, q, y, cnv(S), P) := 1 \\
\qquad \texttt{enddo endcase}
\end{array}
$$

□

7 OLAP Functions in Business Statistics

The main undertaking of data warehousing is to support the business analysis happening in the OLAP tier. Most of such analysis requires applying business statistics for the support in decision making. Often such requirements are dynamic and arise with the change of economic and business conditions. In the following, we take single linear regression, correlation and time series analysis as the examples in the dynamic data warehouse design to demonstrate how these changes are incorporated in the data warehouse process model using our tailored refinement method.

7.1 Single Linear Regression and Correlations

Regression analysis is used primarily for the purpose of prediction. The goal is to develop a statistical model that can be used to predict the values of a dependent variable based on the values of at least one independent variable. In contrast to regression, correlation analysis is used to measure the strength of the association between numerical variables [Levine et al., 2000].

Example 4. Let us take one of the examples in [Levine et al., 2000] for the case of grocery store data warehousing. We aim at showing how the ground model is refined to support the regression analysis. Our scenario is that the director of the grocery stores is being asked to develop an approach for forecasting annual sales for all new stores. Suppose he decided to examine the relationship between the size(square footage) of a store and its annual sales, that is, to build a sample linear regression model equation as follows:

$$\hat{Y}_i = b_0 + b_1 X_i$$

$$\begin{aligned} where\ \hat{Y}_i &= \text{predicted value of } Y \text{ for observation } i \\ X_i &= \text{value of } X \text{ for observation } i \end{aligned}$$

In our example, the X is the size of a store, and the Y is the annual sales.

In order to predict the value Y, we need to compute the two coefficients, b_0(the sample Y intercept) and b_1 (the sample slope). The simplest way is to use the least-squares method, which requires a sample with details of store size and annual sales from the data warehouse. Thus we need to make the required data, store size and annual sales available for the sample selection.

Following the refinement process described in Section 4, we refine the ground model for the requirement on data in the following:

1. Add a new rule in OLAP: We first define a rule *populate_V_size_sales* for populating the required data on store size and the annual sales.

$$\begin{array}{l} \mathit{populate\_size\_sales} = \texttt{forall}\ s, \mathit{size}, y\ \texttt{with} \\ \quad (s, \mathit{size}) \in \text{Store} \wedge \\ \quad \exists d, d', w, q, m.(d, d', w, m, q, y) \in \text{Time} \wedge \\ \quad \texttt{do let}\ S = \mathit{src}[0, \pi_{s'}, +] \\ \qquad (\{(i, s, p, d, s') \mid \exists q', p'.(i, s, p, d, q', s', p') \in \text{Purchase} \wedge \\ \qquad d.\mathit{year} = y\}) \\ \quad \texttt{in}\ \text{V\_size\_sales}(s, \mathit{size}, y, S) := 1 \\ \texttt{enddo} \end{array}$$

2. Add a new controlled function to the OLAP ASM: apply the schema transformation rule 1, to add function **V_size_sales**: $(\mathit{sid} \times \mathit{size} \times \mathit{year} \times \mathit{sales})$ for supporting the above OLAP rule.

3. Add a new controlled function to DW ASM: As the store size is not available in the current data warehouse, we define a function **Store**($sid \times size$) in DW ASM.

4. Integrate controlled functions on the DW ASM: apply the schema transformation rule 6, the two functions **Shop** and **Store** are integrated to **Shop**($sid \times name \times size \times town \times region \times state \times phone$).

5. Change the view creation rules on the OLAP ASM: This is a consequence from view integration.

$$
\begin{array}{l}
\mathit{populate\_size\_sales} = \texttt{forall } s, size, y \texttt{ with} \\
\quad \exists n, t, r, st, ph.(s, n, t, r, st, ph, size) \in \text{Shop} \wedge \\
\quad \exists d, d', w, q, m.(d, d', w, m, q, y) \in \text{Time} \wedge \\
\quad \texttt{do let } S = src[0, \pi_{s'}, +] \\
\qquad (\{(i, s, p, d, s') \mid \exists q', p'.(i, s, p, d, q', s', p') \in \text{Purchase} \wedge \\
\qquad d.year = y\}) \\
\quad \texttt{in } \text{V\_size\_sales}(s, size, y, S) := 1 \texttt{ enddo}
\end{array}
$$

6. Change the rules in the DW ASM: As one of the consequences of integration, the refresh rule *extract_shop* will be changed to extract the details of store size into the relation **Shop**.

$$
\begin{array}{l}
\mathit{extract\_shop} = \texttt{forall } s, n, s', a \texttt{ with } \text{Store}(s, n, s', a) \neq \bot \\
\quad \texttt{do let } t = a.town, r = a.region, st = a.state, ph = a.phone \\
\qquad \texttt{in } \text{Shop}(s, n, s', t, r, st, ph) := 1 \texttt{ enddo}
\end{array}
$$

7. Change the rules in the OLAP ASM: In addition, all the other rules that are referring to **Shop**(which is changed in the integration), such as *create_V_sales* in the ground model OLAP ASM, will be changed to include a function f as follows:

$$newRule = oldRule \circ f$$

   where f is only applied on the relation **Shop**:

$$f(\text{Shop}) = \{(s, n, t, r, st, ph) \mid \exists s'.(s, n, s', t, r, st, ph) \in \text{Shop}\}$$

Once the newly defined view **View_size_sales** is populated, it can be used as the population for sample selection. With a sample, say last year's sales as an example, we may proceed with the regression and correlation analysis, which can be realised through applying relevant formulas over the selected sample. We will not discuss it further due to it involving only the application of statistical calculations. □

7.2 Time Series Analysis

Regression analysis provides a useful methodology for managerial decision making. Similarly, the business forecasting methods applying the concept of time series analysis are used in the process of managerial planning and control. Time series forecasting methods involve the projection of future values of a variable based entirely on the past and present observations of that variable. As an example, we may make prediction of next year annual sales for a store based on its annual sales from year 1990 up to now.

Numerous methods applying time series analysis are devised for the purpose of forecasting. In the following we discuss how our data warehouse ground model is refined to include the exponential smoothing technique as an additional OLAP function. Exponential smoothing is not just being used for smoothing (providing impression of long-term movements) but also for obtaining short term (one period into the future) forecasts [Levine et al., 2000].

The formula for exponential smoothing is defined as follows:

$$E_i = WY_i + (1 - W)E_{i-1}$$

$$\begin{aligned} \textit{where}\ E_i &= \text{value being computed in time period } i \\ E_{i-1} &= \text{value being computed in time period } i-1 \\ Y_i &= \text{observed value of the time series in period } i \\ W &= \text{subjectively assigned weight or smoothing coefficient} \\ &\quad\ \text{(where } 0 < W < 1\text{)} \\ E_1 &= Y_1 \end{aligned}$$

Example 5. In our grocery store example, let us consider the exponential smoothing (e-smoothing in short), for forecasting store's annual sales. First we refine the ground model to include the data population for annual sales by store and year. This will be a simple refinement as follows:

1. *(Add a new rule in OLAP)* we define a rule *populate-annual-sales* for populating data on the annual sales by store and year.

$$\begin{aligned} &\textit{populate-annual-sales} = \texttt{forall}\ s, y\ \texttt{with} \\ &\quad \exists n, t, r, st, ph.\text{Shop}(s, n, t, r, st, ph) \neq \perp \wedge \\ &\quad \exists d, d', w, m, q.\text{Time}(d, d', w, m, q, y) \neq \perp \\ &\quad \texttt{do let}\ S = src[0, \pi_{s'}, +](\{(i, s, p, d, s') \mid \exists q', p'. \\ &\qquad \text{Purchase}(i, s, p, d, q', s', p') \neq \perp \wedge d.year = y\}) \\ &\quad \texttt{in}\ \text{V\_annual\_esales}(s, y, S) := 1 \\ &\ \texttt{enddo} \end{aligned}$$

2. Add a new controlled function to the OLAP ASM: apply the schema transformation rule 1, to add function **V_annual_esales** ($sid \times year \times sales$) for supporting the new rule.

It would be possible to just model the OLAP function e-smooth as the open-datamart operation, for which we would need to change the open-datamart rule to incorporate the computation of e-smoothed value. Another simpler way is to keep the open-datamart unchanged and make the e-smooth function as a new operation. In the latter case, we have the refinements under system implementation as follows:

1. First, we change the operation type function **type**: **op** → {open,close,quit, e-smooth} to include the new OLAP function *e-smooth*;
2. In order to process the new type **e-smooth**, we need to change the rule *main*:

   `elseif` type(op)= *e-smooth* `then` *E-SMOOTH*

3. Then we add a new rule for the OLAP function *e-smooth*:

   E-SMOOTH = `if` the-view(op) = V_annual_sales `then`
   `import` dm
   datamart(dm) := 1, owner(dm) := issuer(op),
   the-matching-view(dm) := the-view(op)
   populate-annual-sales;
   `for all` s, y, S `with` V_annual_sales$(s, y, S) \neq \perp$ `do`
   `if` pre-esales-value(s,y) $\neq 0$ `then let`
   $E = the - weight(op) * S + (1 - the - weight(op)) *$ pre-esales-value(s, y)
   `in` DM-V_annual_sales$(dm, s, y, S, E) := 1$
   `else let` $E = S$ `in` DM-V_annual_sales$(dm, s, y, S, E) := 1$ `endif`
   `enddo`
   `end-import`

4. To support the new rule, we define two universes **evalue** to model the e-smoothed value and **weight** to model the co-efficient or weight. Over the universes, we define two ASM functions **the-weight**: **op** → **weight** for retrieving the weight from the operation, and **Pre-evalue**: **sid** × **year** → **evalue** for getting the previous year's e-smoothed value. □

8 Conclusion

In this paper we presented a formal approach to dynamic data warehouse and OLAP systems design using Abstract State Machines.

We started from a basic model that is based on the fundamental idea of separating input from operational databases from output to so-called data marts, which can be understood as views supporting particular analytical tasks. This ground model was already discussed partly in [Zhao and Ma, 2004; Zhao and Schewe, 2004].

We clarified what we want to achieve by refinements in data-intensive application areas. Strong data refinement is more restrictive with respect to changes to the signature of an ASM in order to preserve the semantics of data in accordance with schema dominance as discussed in [Ma et al., 2005]. The view integration techniques grounded in the notion of schema dominance are used to support the schema evolution in dynamic data warehouse design.

We have shown that dynamic data warehouse design can be realised by ASM refinements in our examples. With a tailored design guide and a set of correctness proved refinement rules, we aim to simplifying the design work and improving the design quality.

We will continue our research in the area of providing pragmatic guidelines for the application refinement rules, and investigate the potential of an automated approach for dynamic data warehouse design.

References

[Abrial, 1996] Abrial, J.-R. (1996). *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

[Barnett et al., 2001] Barnett, M., Campbell, C., Schulte, W., and Veanes, M. (2001). Specification, simulation and testing of com components using abstract state machines. In Moreno-Diaz, R. and Quesada-Arencibia, A., editors, *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*, volume 31, pages 266–270, Canary Islands, Spain.

[Blaschka et al., 1999] Blaschka, M., Sapia, C., and Hoefling, G. (1999). On schema evolution in multidimensional databases. In Mohania, M. and Tjoa, A. M., editors, *Data Warehousing and Knowledge Discovery – DaWaK'99*, volume 1676 of *LNCS*, pages 153–164. Springer-Verlag.

[Blass and Gurevich, 2003] Blass, A. and Gurevich, J. (2003). Abstract state machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 4(4):578–651.

[Börger, 2003a] Börger, E. (2003a). The ASM ground model method as a foundation for requirements engineering. In *Verification: Theory and Practice*, pages 145–160.

[Börger, 2003b] Börger, E. (2003b). The ASM refinement method. *Formal Aspects of Computing*, 15:237–257.

[Börger and Glässer, 1995] Börger, E. and Glässer, U. (1995). Modelling and analysis of distributed and reactive systems using evolving algebras. Technical Report BRICS- NS-95- 4, University of Aarhus.

[Börger and Stärk, 2003] Börger, E. and Stärk, R. (2003). *Abstract State Machines*. Springer-Verlag, Berlin Heidelberg New York.

[Bouzeghoub and Kedad, 2000] Bouzeghoub, M. and Kedad, Z. (2000). A logical model for data warehouse design and evolution. In *DaWaK 2000: Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery*, pages 178–188, London, UK. Springer-Verlag.

[Gupta et al., 1995] Gupta, A., Mumick, I. S., and Ross, K. A. (1995). Adapting materialized views after redefinitions. In *ACM SIGMOD*, pages 211–222.

[Gurevich et al., 1997] Gurevich, J., Sopokar, N., and Wallace, C. (1997). Formalizing database recovery. *Journal of Universal Computer Science*, 3(4):320–340.

[Gurevich, 2000] Gurevich, Y. (2000). Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111.

[Gyssens and Lakshmanan, 1996] Gyssens, M. and Lakshmanan, L. (1996). A foundation for multidimensional databases. In *Proc. 22nd VLDB Conference, Mumbai (Bombay), India*.

[Harinarayan et al., 1996] Harinarayan, V., Rajaraman, A., and Ullman, J. D. (1996). Implementing data cubes efficiently. In *Proceedings of ACM SIGMOD '96, Montreal, June 1996*, pages 205–216.

[Koehler et al., 2007] Koehler, H., Schewe, K.-D., and Zhao, J. (2007). Dynamic data warehouse design as a refinement in asm-based approach. In Roddick, J. F. and Hinze, A., editors, *Conceptual Modelling 2007 – Fourth Asia-Pacific Conference on Conceptual Modelling*, volume 67 of *CRPIT*, pages 61–69, Ballarat, Australia. Australian Computer Society.

[Koh and Chen, 1994] Koh, J. and Chen, A. (1994). Integration of heterogeneous object schemas. In Elmasri, R., Kouramajian, V., and Thalheim, B., editors, *Entity-Relationship Approach - ER'93*, volume 823 of *LNCS*, pages 297–314. Springer-Verlag.

[Kotidis and Roussopoulos, 2001] Kotidis, Y. and Roussopoulos, N. (2001). A case for dynamic view management. *ACM Trans. Database Syst.*, 26(4):388–423.

[Larson et al., 1989] Larson, J., Navathe, S. B., and Elmasri, R. (1989). A theory of attribute equivalence in databases with application to schema integration. *IEEE Transactions on Software Engineering*, 15(4):449–463.

[Lawrence and Rau-Chaplin, 2006] Lawrence, M. and Rau-Chaplin, A. (2006). Dynamic view selection for olap. In *DaWaK*, pages 33–44.

[Levine et al., 2000] Levine, D., Krehbiel, T., and Berenson, M. (2000). *Business Statistics: A First Course*. Prentice-Hall, New Jersey.

[Lewerenz et al., 1999] Lewerenz, J., Schewe, K.-D., and Thalheim, B. (1999). Modelling data warehouses and OLAP applications using dialogue objects. In Akoka, J., Bouzeghoub, M., Comyn-Wattiau, I., and Métais, E., editors, *Conceptual Modeling – ER'99*, volume 1728 of *LNCS*, pages 354–368. Springer-Verlag.

[Link et al., 2006] Link, S., Schewe, K.-D., and Zhao, J. (2006). Refinements in typed abstract state machines. In Virbitskaite, I. and Voronkov, A., editors, *Andrei Ershov Memorial Conference on Perspectives of Systems Informatics – PSI 2006*, volume 4378 of *LNCS*, pages 297–309. Springer-Verlag.

[Ma et al., 2005] Ma, H., Schewe, K.-D., Thalheim, B., and Zhao, J. (2005). View integration and cooperation in databases, data warehouses and web information systems. *Journal on Data Semantics*, IV:213–249.

[Özsu and Valduriez, 1999] Özsu, T. and Valduriez, P. (1999). *Principles of Distributed Database Systems*. Prentice-Hall.

[Prinz and Thalheim, 2003] Prinz, A. and Thalheim, B. (2003). Operational semantics of transactions. In Schewe, K.-D. and Zhou, X., editors, *Database Technologies 2003: Fourteenth Australasian Database Conference*, volume 17 of *Conferences in Research and Practice of Information Technology*, pages 169–179.

[Schellhorn, 2001] Schellhorn, G. (2001). Verification of asm refinements using generalized forward simulation. *j-jucs*, 7(11):952–979.

[Schewe, 1997] Schewe, K.-D. (1997). Specification and development of correct relational database programs. Technical report, Clausthal Technical University, Germany.

[Schewe and Schewe, 2000] Schewe, K.-D. and Schewe, B. (2000). Integrating database and dialogue design. *Knowledge and Information Systems*, 2(1):1–32.

[Schewe and Zhao, 2005a] Schewe, K.-D. and Zhao, J. (2005a). ASM ground model and refinement for data warehouses. In Beauquier, D., Börger, E., and

Slissenko, A., editors, *Proc. 12th International Workshop on Abstract State Machines – ASM 2005*, pages 369–376, Paris, France.

[Schewe and Zhao, 2005b] Schewe, K.-D. and Zhao, J. (2005b). Balancing redundancy and query costs in distributed data warehouses – an approach based on abstract state machines. In Hartmann, S. and Stumptner, M., editors, *Conceptual Modelling 2005 – Second Asia-Pacific Conference on Conceptual Modelling*, volume 43 of *CRPIT*, pages 97–105, Newcastle, Australia. Australian Computer Society.

[Schewe and Zhao, 2007] Schewe, K.-D. and Zhao, J. (2007). Typed abstract state machines for data-intensive applications. *Knowledge And Information Systems*. to appear.

[Spaccapietra and Parent, 1994] Spaccapietra, S. and Parent, C. (1994). View integration – a step forward in solving structural conflicts. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):258–274.

[Theodoratos and Sellis, 1999] Theodoratos, D. and Sellis, T. (1999). Dynamic data warehouse design. In *Data Warehousing and Knowledge Discovery – DaWaK'99*, volume 1676 of *LNCS*, pages 1–10. Springer-Verlag.

[Thomson, 2002] Thomson, E. (2002). *OLAP Solutions: Building Multidimensional Information Systems*. John Wiley & Sons.

[Widom, 1995] Widom, J. (1995). Research problems in data warehousing. In *Proceedings of the 4th International Conference on Information and Knowledge Management*. ACM.

[Zhao, 2005] Zhao, J. (2005). A formal approach to the design of distributed data warehouses. In *Computational Science and Its Applications (ICCSA 2005): International Conference, Singapore, May 9-12, 2005, Proceedings, Part II*, volume 3481 of *LNCS*, pages 1235–1244. Springer-Verlag.

[Zhao and Ma, 2004] Zhao, J. and Ma, H. (2004). Quality-assured design of on-line analytical processing systems using abstract state machines. In Ehrich, H.-D. and Schewe, K.-D., editors, *Proceedings of the Fourth International Conference on Quality Software (QSIC 2004)*, Braunschweig, Germany. IEEE Computer Society Press. to appear.

[Zhao and Schewe, 2004] Zhao, J. and Schewe, K.-D. (2004). Using abstract state machines for distributed data warehouse design. In Hartmann, S. and Roddick, J., editors, *Conceptual Modelling 2004 – First Asia-Pacific Conference on Conceptual Modelling*, volume 31 of *CRPIT*, pages 49–58, Dunedin, New Zealand. Australian Computer Society.