Dynamic Data Warehouse Design as a Refinement in ASM-based Approach

Henning Koehler **Klaus-Dieter Schewe** **Jane Zhao**

Massey University, Information Science Research Center, Department of Information Systems
Private Bag 11222, Palmerston North, New Zealand
Email: [h.koehler|k.d.schewe|j.zhao]@massey.ac.nz

Abstract

On-line analytical processing (OLAP) systems deal with analytical tasks in businesses. As these tasks do not depend on the latest updates by transactions, it is assumed that the data used in OLAP systems are kept in a data warehouse, which separates the input from operational databases from the outputs to OLAP. Typical OLAP queries are data intensive, and thus time consuming. In order to speed up the query computation, it is a common practice to materialize some of the computations as views based on a set of queries given. In general we wish to optimize query time under a given maintenance constraints. However, OLAP queries are not static and may change over time. Thus designing data warehouse is an ongoing task. This process is also called dynamic or incremental design. In this paper, we approach this issue as a refinement step in our Abstract State Machine (ASM) based data warehouse design, and support it by a set of standard refinement rules.

1 Introduction

Data Warehouses are data-intensive systems that are used for analytical tasks in businesses such as analyzing sales/profits statistics, cost/benefit relation statistics, customer preferences statistics, etc. Those queries, also referred as OLAP queries, are resource intensive. In order to speed up the performance, it is common in the data warehouse to store some of the query results or some intermediate results as materialized views. However it is not feasible to store all the query results due to maintenance constraints. Thus selection of views to materialize is an important task in any data warehouse design.

As business requirements in analysis change over time, view selection becomes an ongoing issue. Theodoratos et al approaches this issue as dynamic data warehouse design in [10, 9]. They see the data warehouse as a set of materialised views over the base database, without underlying star schema. Their approach reserves the existing materialised view set, and does not materialise new views which can be computed from the existing ones, even though this might be beneficial for query performance.

The goal of this article is to incorporate the dynamic design as a refinement step in our ASM-based data warehouse design framework, and present a set of standard refinement rules with application heuristics from benefit analysis. Although our view selection algorithm is meant for further selection of materialised views, it can also be used in incremental view selection from the very beginning.

In [13] we started developing 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 [8] focusing on cost-efficient distribution of data warehouses. This last idea exploits fragmentation techniques from [6] and the recombination of fragments, but still remains on rather informal grounds. In [11], we started to formalize the distribution process and showed how the three-layered specification was extended to distributed data warehouses.

The rest of paper is organized as: in Section 2 we introduce the general idea of the ASM method, and TASM, an extension of ASM with types. In Section 3, we present a ground model in TASM of a data warehouse application. In Section 4 we discuss our approach for dynamic data warehouse design using refinements. We conclude with a brief summary in Section 5.

2 Systems Development with Abstract State Machines

Abstract State Machines (ASMs, [1]) 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

Copyright ©2007, Australian Computer Society, Inc. This paper appeared at Fourth Asia-Pacific Conference on Conceptual Modelling (APCCM 2007), Ballarat, Victoria, Australia, January 2007. Conferences in Research and Practice in Information Technology, Vol. 67. John F. Roddick and Annika Hinze, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

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 [1]).

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

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

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 $f_i : \mathcal{U}^{ar_i} \to \mathcal{U} \cup \{\bot\}$ with a not further specified set $\mathcal{U}$ called *super-universe* and a special symbol $\bot \notin \mathcal{U}$. As usual, f_i can be interpreted as a partial function $\mathcal{U}^{ar_i} \nrightarrow \mathcal{U}$ with domain $dom(f_i) = \{\vec{x} \in \mathcal{U}^{ar_i} \mid f_i(\vec{x}) \neq \bot\}$.

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 $\ell = (f, \vec{x})$ with $\vec{x} = (x_1, \dots, x_{ar_i})$ 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:

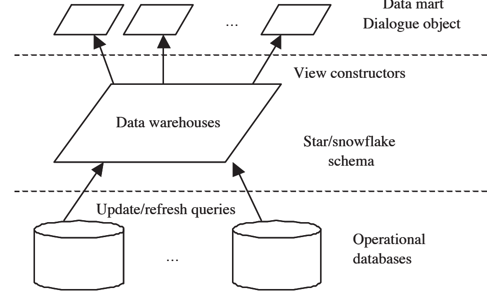

Figure 1: The general architecture of a data warehouse and OLAP

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

  `if` φ_1 `then` r_1 `elsif` $\varphi_2 \dots$ `then` r_n `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 $s_0 \to s_1 \to s_2 \to \dots$ 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 [1].

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 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 [4]. We started with a *type system*

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

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. $1\!\!1$ is a trivial type. With each type t we associate a *domain* $dom(t)$ in the usual way, i.e. we have $dom(\{t\}) = \{x \subseteq dom(t) \mid |x| < \infty\}$, $dom(a : t) = dom(t)$, $dom(t_1 \times \cdots \times t_n) = dom(t_1) \times \cdots \times dom(t_n)$, $dom(t_1 \oplus \cdots \oplus t_n) = \coprod_{i=1}^{n} dom(t_i)$ (disjoint union), and $dom(1\!\!1) = \{\mathbf{1}\}$.

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

- $t \leq 1\!\!1$ 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 $j = 1, \ldots, k$, 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 $i = 1, \ldots, n$, 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 $f_1, \ldots, f_m$. 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 $t'_i = t''_i \oplus 1\!\!1$ 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 $t \to \{1\!\!1\}$ 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 $\{1\!\!1\}$ 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 $R(x) = \emptyset$.

3 The Ground Model

Based on the fundamental idea of separating input from operational databases from output to OLAP systems, we consider a three-tier model from Figure 1. At the bottom tier, there is the operational database model. Its major function is to extract data from operational database according to the data warehouse requests. These can be the incremental updates. At the middle tier, there is the data warehouse model, which propagates updates from operational database to the materialized views, and support OLAP queries by extracting data from data warehouse upon OLAP requests. At the top tier, there is the OLAP system, which sends query requests to data warehouse and supports typical OLAP functions such as roll-up, drill-down, slice and dice, etc. Applying the ASMs-based design method, we first construct a ground model for the data warehouse. Based on the three-tier architecture, we construct three linked ASMs, the DB-ASM, the DW-ASM, and the OLAP-ASM. In the following, we will use a grocery store chain data warehouse as an example to explain the construction of the ground model in details.

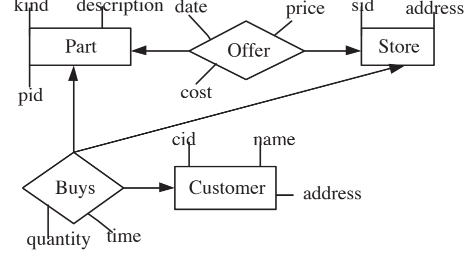

Figure 2: The operational database schema

3.1 The Operational Database ASMs

The operational database model is modelled as to extract data for the data warehouse upon request. The data warehouse in fact presents a consolidated view of the operational database through aggregating the transaction data in operational database. How the aggregation is done is determined by the data warehouse model. Therefore the data extraction rules are defined in data warehouse model. As the source data are in operational database, the extraction rules are executed in database model upon the system requests. We assume for simplicity that all data sources are relational, thus the ASMs signatures would just describe the relation schemata.

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 2. That is, in a relational representation we would have five relation schemata Store, Part, Customer_DB, Buys and Offer. We use a simple star schema for the data warehouse as illustrated by the HERM diagram in Figure 3, which results in five relation schemata Shop, Product, Customer_DW, Purchase and Time, that are imported from 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
`SIGNATURE`
  Store= $s : label \times sid : string \times address : string \to \{1\!\!1\}$,
  Part= $p : label \times kind : string \times description : string \to \{1\!\!1\}$,
  Customer_DB= $c : label \times cid :$

$string \times name : string \times address : string \rightarrow \{\mathbb{1}\}$,
Buys= $b : label \times time : string \times quantity : real \times s : label \times c : label \times p : label \rightarrow \{\mathbb{1}\}$,
Offer= $o : label \times date : string \times price : real \times cost : real \times p : label \times s : label \rightarrow \{\mathbb{1}\}$
BODY
main = if selected(extract) then
extract_purchase;
extract_customer;
extract_shop;
extract_product;
extract_time
endif

3.2 The Data Warehouse ASMs

For the data warehouse ASMs model we follow the same line of abstraction as for the model of operational databases, i.e. the signature for DW-ASM have functions that correspond to the relation schemata in the data warehouse. In our data warehouse model, the detailed data are kept in the star schema such that any OLAP query is defined over the star schema. For the performance concern, we use the strategy that each OLAP query is rewritten over the materialized views in data warehouse. If not possible, then the OLAP query result will be created as the materialized view and stored in the data warehouse. For each data warehouse schema we define one rule for data extraction. In addition, DW-ASM imports the database schema functions from DB-ASM and the OLAP view functions from OLAP-ASM for referencing purpose. On the other hand it exports the data warehouse schema functions and the respective data extraction rules. Similarly, we define the view creation rules in OLAP-ASM, and import them to DW-ASM. As the basic function of the data warehouse DW-ASM are to support OLAP queries and to maintain the data up to date, we define two group of rules, one for data extraction and one for refresh. The refresh rules are for materialized views on DW-ASM. In addition, we model the system request using one monitored function V, which is set to be the selected OLAP view.

Again we use the same grocery example. As shown in Figure 3, the data warehouse is represented by a relational database schema with five relation schemata Shop, Product, Customer_DW, Purchase and Time. In this simple example, we take the total sale by shop, month and year, called V_Msales, as one of the OLAP views, which is also kept as a materialized view in DW-ASM, suffixed with *dw*.

ASM DW-ASM
IMPORT
DB-ASM(Store, Part, Customer_DB, Buys, Offer),
OLAP-ASM(V_Msales, *create_V_Msales*)
EXPORT
Shop, Product, Customer_DW, Time, Purchase,
extract_purchase, *extract_customer*, *extract_shop*, *extract_product*, *extract_time*
SIGNATURE
V (monitored),
Shop= $sh : label \times sid : string \times name : string \times town : string \times region : string \times state : string \times phone : string \rightarrow \{\mathbb{1}\}$,
Product= $pr : label \times pid : string \times category : string \times desc : string \rightarrow \{\mathbb{1}\}$,
Customer_DW= $cs : label \times cid : string \times name : string \times address : string \rightarrow \{\mathbb{1}\}$,
Time= $t : label \times time : string \times day : string \times week : string \times month : string \times quarter : string \times year : string \rightarrow \{\mathbb{1}\}$,
Purchase= $pch : label \times money - sale : real \times qty : real \times profit : real \times sh : label \times pr : label \times cs : label \times t : label \rightarrow \{\mathbb{1}\}$,
V_Msales_dw= $m_{dw} : label \times shop : string \times region : string \times st : string \times month : string \times qtr : string \times year : string \times msale : real \rightarrow \{\mathbb{1}\}$
BODY
main =
if selected(refresh) then
refresh_V_Msale_dw
elsif selected(create-view) then
create_view(V)
endif
create_view(V) = case V of
V_sales: *create_V_Msales*
endcase
extract_purchase =
forall i, p, s, t, p', c with
$\exists q.\text{Buys}(i, p, q, t) \neq \emptyset \wedge$
$\exists n, a.\text{Customer_DB}(i, n, a) \neq \emptyset \wedge$
$\exists k, d.\text{Part}(p, k, d) \neq \emptyset \wedge$
$\exists a'.\text{Store}(s, a') \neq \emptyset \wedge$
$\exists d.(\text{Offer}(p, s, p', c, d) \neq \emptyset \wedge$
$date(t) = d)$
do let
$Q = sum(q \mid \text{Buys}(i, p, q, t) \neq \emptyset)$,
$S = Q * p'$, $P = Q * (p' - c)$
in $\text{Purchase}(i, p, s, t, Q, S, P) := \{1\}$
enddo
extract_customer = ...
refresh_V_Msale_dw = *create_V_Msales*
...

3.3 The OLAP ASMs

The top-level ASM dealing with OLAP is a bit more complicated, as it realizes the idea of using dialogue objects for this purposes. The general idea from [7] 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, or the users may create new dialogue objects without closing the opened ones, or they may close some of the dialogue objects, or quit when they finish their work with the system.

The OLAP model defines a set of business queries that the data warehouse need to support, and a set of OLAP operations, such as, roll-up, drill-down, slice and dice, etc. In order to focus on the discussion of dynamic design, we abstract it to a set of OLAP views, i.e. the OLAP queries, and the data extraction rules for the OLAP views. The example OLAP view is the total sale by shop, month and year, called V_Msales.

ASM OLAP-ASM
IMPORT
DW-ASM(Shop, Product, Customer_DW, Time, Purchase)
EXPORT V_Msales, create_V_Msales
SIGNATURE
V_Msales= $m : label \times sh : string \times region : string \times st : string \times month : string \times qtr : string \times year :$

$string \times msale : real \rightarrow \{\mathbb{1}\}$

```
BODY
  main =...
  create_V_Msales =
    forall s, r, st, m, q, y with
    ∃n, t', ph.Shop(s, n, t', r, st, ph) ≠ ∅
    ∧∃t.Time(t, m, q, y) ≠ ∅
    do let S = sum(s' | ∃c, p, t, q', p'.
        Purchase(c, p, s, t, q', s', p') ≠ ∅)
      in V_Msales(s, r, st, m, q, y, S) := {1}
    enddo
  ...
```

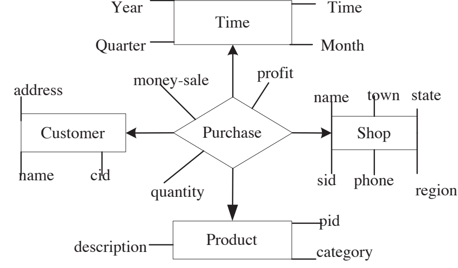

Figure 3: The data warehouse star schema

4 Dynamic Design as Refinement

The set of OLAP queries is not static, but changes over time. When a new query needs to be supported, or an existing query is obsolete, we need to concern about the followings:

- add more data from the operational database to the data warehouse;
- rewrite the query using the existing materialized views;
- materialize more views to data warehouse;
- integrate materialized views after adding new views;
- remove redundant materialized views from the data warehouse.

Since our focus here is to model the dynamic design as refinement, we incorporate a simple view selection algorithm based on a simple cost model into a set of refinement rules and apply view integration for the change of data warehouse schema, for which we have started our investigation in [5] and [3].

4.1 The Cost Model

For simplicity, we follow a similar approach in [2] to calculate the cost for query evaluation and view maintenance, in specific we consider the size of a view as its maintenance cost and also as a query evaluation cost if the view is used to compute the query, as shown below.

The evaluation cost of query q by computing q from the materialized view v is modelled as:

$qcost(q) = s(v)$
where $s(v)$ is the size of the view.

Similarly, the cost of maintaining a view v is modelled as:

$mcost(v) = s(v)$

We compute the benefit of materialising and using view v_1 instead of view v_2 to compute query q with frequency f as follows:

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

In this the terms $s(v_i) \times f$ represent the total query cost for q, while $s(v_i)$ represents the maintenance cost for v_i. Therefore, a zero benefit shows that v_1 and v_2 have the same cost, a negative benefit shows that v_1 costs more than v_2, otherwise v_2 costs more than v_1.

When maintaining v_1 is not an additional cost, the benefit of using existing materialized view v_1 to compute query q with frequency f over creating a new view v for query q is:

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

where $s(v)$ is the size of view from query q, v_1 is selected from the view from existing materialized view set to compute q.

It is clear that we have a simplified cost model, however, a more complex one can be plugged in easily.

4.2 A View Selection Algorithm

In the following we present a simple selection algorithm for determining if the new query is to be materialized or it is rewritten over the existing materialized views.

Definition 1 A view(query) v_1 is called to be *finer* than v_2, denoted as $v_1 \succ v_2$, if v_2 is computable from v_1 by aggregating operations.

For an example, v_1 of *sales by day* is *finer* than v_2 of *sales by month*, since we get the monthly sales by summing up the daily sales for the month.

For a given new query q with frequency f, we create a view v from q. In the following algorithm, with input of a given materialised view set MV, and an existing OLAP query set V_{OLAP}, we output:

- whether v gets added to MV;
- if yes, a set of existing queries $Q \in V_{OLAP}$ to be recomputed from v
- if not, a materialised view $v_k \in MV$ to be used for computing q

The algorithm:

- choose all the views $v_i \in MV$ such that $v_i \succ v$; That is, we select all the existing materialized views that v can be computed from.
- compute the benefits b_i of using v_i over the view v as follows:
  $b_i = (s(v) - s(v_i)) \times f + s(v)$
- choose v_k from above such that b_k is maximal;
- choose all the views $v_j \in V_{OLAP}$, the set of OLAP views representing the existing queries, with frequency f_j, such that $v \succ v_j$;
- for each such v_j let $v'_j \in MV$ be the materialized view used to compute v_j; compute the benefit of using v over the view v'_j as follows:
  $b(v, v'_j) = ((s(v'_j) \times f_j + s(v'_j)) - (s(v) \times f_j + s(v))$

- compute the sum $b_{sum} = \Sigma(b(v, v'_j))$, for all $v_j \in V_{OLAP}$ with $b(v, v'_j) > 0$;
- if $b_k \leq 0 \vee b_{sum} > b_k$, add the view v to V, add those $v_j \in V_{OLAP}$ to Q for which $b(v, v'_j) > 0$;
- if $b_k \geq b_{sum}$ set v_k to be the view for computing q.

4.3 Notion of Refinement

The general notion of *refinement* relates two ASM $\mathfrak{M}$ and $\mathfrak{M}^*$. In principle, as the semantics of ASMs is defined by its runs, we would need a correspondence between such runs. 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 ASM $\mathfrak{M}$ and $\mathfrak{M}^*$, respectively. A *correspondence of states* between $\mathfrak{M}$ and $\mathfrak{M}^*$ 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 $\mathfrak{M}$ and $\mathfrak{M}^*$, respectively. In particular, we must have

$$\begin{array}{ll} s \equiv s_1^* \wedge s \equiv s_2^* \Rightarrow s_1^* = s_2^* & \text{and} \\ s_1 \equiv s^* \wedge s_2 \equiv s^* \Rightarrow s_1 = s_2. & \end{array}$$

Then we say that an ASM $\mathfrak{M}^*$ is a correct *refinement* of ASM $\mathfrak{M}$ iff for each run $s_0^* \rightarrow s_1^* \rightarrow \dots$ of $\mathfrak{M}^*$ there is a run $s_0 \rightarrow s_1 \rightarrow \dots$ of $\mathfrak{M}$ 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. $\mathfrak{M}^*$ is called a complete *refinement* of ASM $\mathfrak{M}$ iff $\mathfrak{M}$ is a correct refinement of $\mathfrak{M}^*$.

In the application of ASMs for data warehouses and OLAP systems, we first clarify what are the states of interest in this definition. 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 $\rhd$ between main rules r of $\mathfrak{M}$ and r^* of $\mathfrak{M}^*$. Finally, take initial states s_0, s_0^* for $\mathfrak{M}$ and $\mathfrak{M}^*$, respectively.

Definition 2 An ASM $\mathfrak{M}^*$ is called a *data refinement* of an ASM $\mathfrak{M}$ iff there is an *abstraction predicate* $\mathcal{A}$ with $(s_0, s_0^*) \models \mathcal{A}$ and there exists a correspondence between main rules r of $\mathfrak{M}$ and 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 request is that the refined schema $\mathcal{S}^*$ of ASM $\mathfrak{M}^*$ should dominate the schema $\mathcal{S}$ of ASM $\mathfrak{M}$. For this we need a notion of computable query. For a state s of a ASM $\mathfrak{M}$ 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:

$$\begin{array}{l} \iota_{t_1 \times \dots \times t_n}(x_1, \dots, x_n) = (\iota_{t_1}(x_1), \dots, \iota_{t_n}(x_n)) \\ \iota_{t_1 \oplus \dots \oplus t_n}(i, x_i) = (i, \iota_{t_i}(x_i)) \\ \iota_{\{t\}}(\{x_1, \dots, x_k\}) = \{\iota_t(x_1), \dots, \iota_t(x_k)\} \end{array}$$

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.

$$\iota_{S^*} \circ f \circ (\iota_S)^{-1} = f$$

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

1. $\mathfrak{M}$ has a schema $\mathcal{S} = \{R_1, \dots, R_n\}$ 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 $i = 1, \dots, n$.
2. $\mathfrak{M}^*$ has a schema $\mathcal{S}^* = \{R_1^*, \dots, R_m^*\}$ 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 $i = 1, \dots, m$.
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 $f(s(\mathcal{S})) = s^*$, and $g(f(s(\mathcal{S}))) = s(\mathcal{S})$.

4.4 The Formal Rules

Applying the notion of refinement defined above, we present in the following a set of standard rules to support the dynamic design as refinement in our ASMs-based framework. In specific, we intend to set up rules of the form:

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

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 to the size of the refinement rules and the space limits of the paper, here we only introduce a subset of the refinement rules as follows, more details can be found in [5, 3]:

1. ***schema extension**, add a new type $\mathcal{R}$. In such case we obtain a dominant schemata.*

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

   Since simply adding a new type causes no changes on the existing views, the corresponding abstraction predicate $\mathcal{A}$ is defined as *true*. The corresponding computable queries f and g only need to keep the types other than $\mathcal{R}$ unchanged.

2. ***schema extension**, add a new attribute A to the type $\mathcal{R}$. If A does not become a key attribute, we obtain a dominant schema.*

$$\frac{\mathfrak{M} \rhd \mathcal{R} = \ell : label \times t \to \{\mathbb{1}\}}{\mathfrak{M}^* \ \rhd \mathcal{R}^* = \ell^* : label \times A : t_a \times t \to \{\mathbb{1}\}} \varphi$$

Then the side condition φ is

$$key(\mathcal{R}) \to A,$$

where $\to$ denotes a functional dependency.

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

$$\forall x(\exists a.\mathcal{V}^*(a, x) = 1 \Leftrightarrow \mathcal{V}(x) = 1)$$

The corresponding computable queries f and g can be obtained as

$$\mathcal{R}^* := \{(a, x) \mid (x) \in \mathcal{R}\}, \text{ for a constant } a,$$

and

$$\mathcal{R} := \{(x) \mid \exists a.(x, a) \in \mathcal{R}^*\},$$

respectively.

3. ***Type Integration**: Merge two types $\mathcal{R}_1$ and $\mathcal{R}_2$ into one type $\mathcal{R}^*$.*

$$\frac{\begin{array}{ll}\mathfrak{M} \rhd & \mathcal{R}_1 = \ell_1 : label \times K_1 : t_{k1} \times X : t_x \\ & \quad \times t_1 \to \{\mathbb{1}\} \\ & \mathcal{R}_2 = \ell_2 : label \times K_2 : t_{k2} \times Y : t_y \\ & \quad \times t_2 \to \{\mathbb{1}\}\end{array}}{\begin{array}{ll}\mathfrak{M}^* \rhd & \mathcal{R}^* = \ell^* : label \times K_1 : t_{k1} \times X : t_x \\ & \quad \times t_1 \times t_2 \to \{\mathbb{1}\}\end{array}} \varphi$$

Then the side condition φ is as

$$K_1 = K_2 \wedge \exists h.\mathcal{R}_1[K_1 \cup X] = h(\mathcal{R}_2[K_2 \cup Y]),$$

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

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

$$\begin{array}{c}\forall k_1, x, x_1, x_2(\mathcal{R}^*(k_1, x, x_1, x_2) = 1 \\ \Leftrightarrow \mathcal{R}_1(k_1, x, x_1) = 1 \wedge \mathcal{R}_2(k_1, h^{-1}(x), x_2) = 1)\end{array}$$

The corresponding computable queries f and g can be obtained as

$$\begin{array}{c}\mathcal{R}^* := \{(k_1, x, x_1, x_2) \mid (k_1, x, x_1) \in \mathcal{R}_1 \\ \wedge (k_1, h^{-1}(x), x_2) \in \mathcal{R}_2\}\end{array}$$

and

$$\begin{array}{c}\mathcal{R}_1 := \{(k_1, x, x_1) \mid \exists x_2.(k_1, x, x_1, x_2) \in \mathcal{R}^*\} \| \\ \mathcal{R}_2 := \{(k_1, h^{-1}(x), x_2) \mid \exists x_1.(k_1, x, x_1, x_2) \in \mathcal{R}^*\}\end{array}$$

4. ***Type Integration**: Replace type $\mathcal{R}_2$ if type $\mathcal{R}_1$ contains part of $\mathcal{R}_2$.*

$$\frac{\begin{array}{ll}\mathfrak{M} \rhd & \mathcal{R}_1 = \ell_1 : label \times K_1 : t_{k1} \times X : t_x \\ & \quad \times t_1 \to \{\mathbb{1}\} \\ & \mathcal{R}_2 = r_2 : label \times K_2 : t_{k2} \times Y : t_y \\ & \quad \times t_2 \to \{\mathbb{1}\}\end{array}}{\begin{array}{ll}\mathfrak{M}^* \rhd & \mathcal{R}_1^* = r_1^* : label \times K_1 : t_{k1} \times X : t_x \\ & \quad \times t_1 \to \{\mathbb{1}\} \\ & \mathcal{R}_2^* = r_2^* : label \times t_2 \\ & \quad \times r_1^* : label \to \{\mathbb{1}\}\end{array}} \varphi$$

Then the side condition φ is

$$K_1 = K_2 \wedge \exists h.\mathcal{R}_2[K_2 \cup X] \subset h(\mathcal{R}_1[K_1 \cup Y]),$$

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

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

$$\begin{array}{c}\forall k_1, x, x_1, x_2(\exists l.(\mathcal{R}_1^*(l, k_1, x, x_1) = 1 \\ \wedge \mathcal{R}_2^*(x_2, l) = 1) \\ \Leftrightarrow \mathcal{R}_1(k_1, x, x_1) = 1 \wedge \mathcal{R}_2(k_1, h^{-1}(x), x_2) = 1)\end{array}$$

The corresponding computable queries f and g can be obtained as

$$\begin{array}{c}\mathcal{R}_1^* := \mathcal{R}_1 \| \\ \mathcal{R}_2^* := \{(x_2, l) \mid \exists l_1, x_1, x.(k_1, h^{-1}(x), x_2) \in \mathcal{R}_2 \\ \wedge (l, k_1, x, x_1) \in \mathcal{R}\}\end{array}$$

and

$$\begin{array}{c}\mathcal{R}_1 := \mathcal{R}^* \| \\ \mathcal{R}_2 := \{(k_1, y, x_2) \mid \exists x_1, l.((k_1, h(y), x_1) \in \mathcal{R}_1^* \\ \wedge (x_2, l) \in \mathcal{R}_2^*)\}\end{array}$$

Considering a new query q is to be supported, we first perform the view selection, and then apply the refinement rules according to the selection result.

Example 1 In this simple example, we show how the view selection works. Let us 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: $v_1 = 20000$, its frequency $f_1 = 30$;
View $\mathcal{V}_2$, the total sale by state and month, its average number of tuples: $v_2 = 3000$, its frequency $f_2 = 10$;
And $\mathcal{V}_1$ is materialized, $\mathcal{V}_2$ is rewritten from $\mathcal{V}_1$.

Now the user requests for a new OLAP query, view $\mathcal{V}$, the total sale by region and day, its average number of tuples: $v = 18000$, its frequency $f = 5$.
It is easy to see in such a simple example, that we have, $\mathcal{V}_1 \succ \mathcal{V}$, that means, we can rewrite the new view from $\mathcal{V}_1$, but we need find out if it is an optimal solution to do so. We compute the benefit of rewriting $\mathcal{V}$ from $\mathcal{V}_1$:

$$\begin{aligned} b_1 &= (v - v_1) \times f + v \\ &= (18000 - 20000) \times 5 + 18000 \\ &= 8000 \end{aligned}$$

b_1 is positive, that means, it is better to rewrite $\mathcal{V}$ from $\mathcal{V}_1$. However, we shall also consider if materialize $\mathcal{V}$ will be beneficial for other views that are rewritten from $\mathcal{V}_1$.

Since we have, $\mathcal{V} \succ \mathcal{V}_2$, so we compute the benefit of writing $\mathcal{V}_2$ from $\mathcal{V}$:

$$\begin{aligned} b =& (v_1 \times f_2 + v_1) - (v \times f_2 + v) \\ &= (20000 \times 10 + 20000) - (18000 \times 10 + 18000) \\ &= 22000 \end{aligned}$$

Now we have $b > b_1$, that means, we shall materialize the new view $\mathcal{V}$ and rewrite $\mathcal{V}_2$ from $\mathcal{V}$.

Example 2 In this example we show how we carry out dynamic design as refinements. Assume the money sales V_Msales in the ground model is in US$, and now the store manager wants to see it in EURO and also additional information, such as profit. This is a simple example but it represents one type of typical changes.

It is clear that we can not rewrite the new view from the existing ones, so we shall materialize the new view, and perform the refinement steps as follows:

1. We first apply the *schema extension* rule, *rule #1*, to add the OLAP view V_Msales_EURO to the OLAP-ASM:

$$\begin{aligned} &\text{OLAP-ASM}^* \rhd V_Msales_EURO \\ &\quad = m_1 : label \times shop : string \times \\ &\quad region : string \times st : string \times \\ &\quad month : string \times qtr : string \times \\ &\quad year : string \times profit : real \times \\ &\quad msale_euro : real \to \{\mathbb{1}\} \end{aligned}$$

2. Then we apply *rule #1* again to add the same OLAP view to the DW-ASM:

$$\begin{aligned} &\text{DW-ASM}^* \rhd V_Msales_EURO_dw \\ &\quad = m_1 : label \times shop : string \times \\ &\quad region : string \times st : string \times \\ &\quad month : string \times qtr : string \times \\ &\quad year : string \times profit : real \times \\ &\quad msale_euro : real \to \{\mathbb{1}\} \end{aligned}$$

3. Now we apply the *Type Integration* rule, *rule #3* at DW-ASM as follows:

$$\frac{\begin{array}{ll} \text{DW-ASM} \rhd & V_Msales_EURO_dw = \\ & m_1 : label \times shop : string \times \\ & region : string \times st : string \\ & \times month : string \times \\ & qtr : string \times year : string \\ & \times profit : real \times \\ & msale_euro : real \to \{\mathbb{1}\} \\ & \\ & V_Msales_dw = \\ & m : label \times shop : string \times \\ & region : string \times st : string \\ & \times month : string \times \\ & qtr : string \times year : string \\ & \times msale : real \to \{\mathbb{1}\} \end{array}}{\begin{array}{ll} \text{DW-ASM}^* \rhd & V_Msales_prf_dw = \\ & m^* : label \times shop : string \times \\ & region : string \times st : string \\ & \times month : string \times \\ & qtr : string \times year : string \\ & \times profit : real \times msale : \\ & real \to \{\mathbb{1}\} \end{array}} \varphi$$

Then the side condition φ is satisfied by:

- 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 $h := (id, id, id, cnv)$, where id is an identity function, and cnv is the currency conversion function from EURO to US$ of the day.

4. As a consequence of the integration, we shall replace the rule *refresh_V_Msales_dw* at the DW-ASM.

```
refresh_V_Msales_prf_dw =
    forall s, r, st, m, q, y with
    ∃n, t′, ph.Shop(s, n, t′, r, st, ph)
    ≠ ∅ ∧ ∃t.Time(t, m, q, y) ≠ ∅
    do let S = sum(s′ | ∃c, p, t, q′, p′.
        Purchase(c, p, s, t, q′, s′, p′)
          ≠ ∅);
        P = sum(p′ | ∃c, p, t, q′, s′.
        Purchase(c, p, s, t, q′, s′, p′)
          ≠ ∅)
        in V_Msales_prf_dw
          (s, r, st, m, q, y, S, P) := {1}
    enddo
```

5. Similarly we change the view creation rules at the DW-ASM.

```
create_view(V) = case V of
  V_Msales:
        create_V_Msales
  V_Msales_EURO:
        create_V_Msales_EURO
endcase
```

```
create_V_Msales =
    forall s, r, st, m, q, y, S with
    ∃P.V_Msales_prf_dw
    (s, r, st, m, q, y, S, P) ≠ ∅
    do let
    V_Msales(s, r, st, m, q, y, S) := {1}
    enddo
```

```
create_V_Msales_EURO =
    forall s, r, st, m, q, y, S, P with
    V_Msales_prf_dw
    (s, r, st, m, q, y, S, P) ≠ ∅
    do let S′ = S × cnv^−1;
        P′ = P × cnv^−1
    in V_Msales_EURO
        (s, r, st, m, q, y, S′, P′) := {1}
    enddo
```

5 Conclusion

In this article we continued our work on the refinement of an ASM ground model with dynamic design of data warehouses and OLAP applications. 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 [12, 13].

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 [5].

As OLAP queries are not static and may change over time, a development method that supports dynamic design becomes handy in data warehouse design. We have shown that this dynamic design can be described as ASM refinements. Such refinements lead to an acceptable specification of a data warehouse application. Our overall goal is to provide a handy set of sound refinement rules for the development of distributed data warehouses and OLAP systems. The rationale is that the use of ASMs allows us to verify desirable properties of the application such as consistency, and these properties are preserved by a refinement-based approach. Furthermore, heuristics such as benefit analysis used in the view selection algorithm provide further guidance to whether a refinement rule should be applied or not. We will continue our research by investigating refinement rules in data warehouse schema evolution using view integration approach, and data warehouse maintenance through incremental changes.

References

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

[2] Harinarayan, V., Rajaraman, A., and Ullman, J. D. Implementing data cubes efficiently. In *Proceedings of ACM SIGMOD '96, Montreal, June 1996*.

[3] Hui Ma, Klaus-Dieter Schewe, J. Z. View integration in data warehouse design using typed abstract state machines and strong data refinement. In *Proceedings of the Sixth International Conference on Quality Software (QSIC 2006)* (Bejing, China, 2006), IEEE Computer Society Press. to appear.

[4] Link, S., Schewe, K.-D., and Zhao, J. Refinements in typed abstract state machines. In *Perspectives of Systems Informatics – PSI 2006* (Novosibirsk, Russia, 2006). to appear.

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

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

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

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

[9] Theodoratos, D., Dalamagas, T., Simitsis, A., and Stavropoulos, M. A randomized approach for the incremental design of an evolving data warehouse. In *Proceedings of the 20th International Conference on Conceptual Modeling: Conceptual Modeling* (2001), vol. 2224 of *LNCS*, Springer-Verlag, pp. 325–338.

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

[11] Zhao, J. 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* (2005), vol. 3481 of *LNCS*, Springer-Verlag, pp. 1235–1244.

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

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