

Refinements in Typed Abstract State Machines

Sebastian Link, Klaus-Dieter Schewe, and Jane Zhao

Massey University, Department of Information Systems &
Information Science Research Centre
Private Bag 11 222, Palmerston North, New Zealand
{s.link|k.d.schewe|j.zhao}@massey.ac.nz

Abstract. While Abstract State Machines (ASMs) provide a general purpose development method, it is advantageous to provide extensions that ease their use in particular application areas. This paper focuses on such extensions for the benefit of a “refinement calculus” in the area of data warehouses and on-line analytical processing (OLAP). We show that providing typed ASMs helps to exploit the existing logical formalisms used in data-intensive areas to define a ground model and refinement rules. We also note that the extensions do not increase the expressiveness of ASMs, as each typed ASM will be equivalent to an “ordinary” one.

1 Introduction

The research reported in this article is part of a project that aims at providing sound and complete refinement rules for the development of distributed data warehouses and on-line analytical processing (OLAP) systems [21]. The rationale is that such a “refinement calculus” would simplify the development task and at the same time increase the quality of the resulting systems. As outlined in [14] we base our work on Abstract State Machines (ASMs, [5]), as they have been already proven their usefulness in many application areas. Furthermore, the ASM method explicitly supports our view of systems development to start with an initial specification called *ground model* [3] that is then subject to *refinements* [4]. This is similar to the approach taken in [12], which contains a refinement-based approach to the development of data-intensive systems using a variant of the B method [1].

The general idea for the data warehouse ground model is to employ three interrelated ASMs, one for underlying operational databases, one for the data warehouse, and one for dialogue types [10] that can be used to integrate views on the data warehouse with OLAP functionality [18]. For the data warehouse level the model of multi-dimensional databases [8], 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 [19]. The work in [11] discusses a rule-based approach for this task without reference to any formal method.

However, both the ground model and the refinement rules tend to become difficult, as well-known database features such as declarative query expressions,

views, bulk updates, etc. are not well supported. Therefore, we believe it is a good idea to incorporate such features into the ASM method to make it easier to use and hence also more acceptable, when applied in an area such as OLAP. This leads us to typed ASMs (TASMs), for which we focus on the incorporation of relations and bulk update operations. There are other approaches to introducing typed versions of ASMs, e.g. [6] following similar ideas, but with different focus, which lead to using different type systems. The work in [20] presents a very general approach to combine ASMs with type theory.

In Section 2 we give a brief introduction on ASMs method, in Section 3 we introduce TASMs and define their semantics by runs following the usual approach used for ASMs. Then in Section 4 we show how TASMs can be used to define a simple ground model for data warehouses. Finally, in Section 5 we discuss refinement rules for TASMs.

2 Abstract State Machines in a Nutshell

Abstract State Machines (ASMs, [5]) 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”.

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 mechanised formal verification techniques. However, the general philosophy is to design first and to postpone rigorous verification to a stage, when requirements have been almost consolidated.

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

```

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

```

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_ℓ 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 $f_i : \mathcal{U}^{ar_i} \rightarrow \mathcal{U} \cup \{\perp\}$ 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 $\mathcal{U}^{ar_i} \dashrightarrow \mathcal{U}$ 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 none of both, 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** ... **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 $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$ 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 [5].

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.

3 Typed Abstract State Machines

The 3-tier architecture for data warehouses and OLAP systems in [21,22] basically requires relations on the database and the data warehouse tiers, while the OLAP tier requires sets of complex values. Therefore, following the line of research in [13] we start with a *type system*

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

Here b represents a not further specified collection of base types such as *CARD*, *INT*, *DATE*, etc. $\{\cdot\}$ is a set-type constructor, $a : t$ is a type constructor with label a , which is introduced as attributes used in join operations. \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 $dom(\{t\}) = \{x \subseteq dom(t) \mid |x| < \infty\}$, $dom(a : t) = \{a : v \mid v \in dom(t)\}$, $dom(t_1 \times \dots \times t_n) = dom(t_1) \times \dots \times dom(t_n)$, $dom(t_1 \oplus \dots \oplus t_n) = \coprod_{i=1}^n dom(t_i)$ (disjoint union), and $dom(\mathbb{1}) = \{\mathbb{1}\}$.

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 $t_1 \times \dots \times t_n \leq t'_{i_1} \times \dots \times t'_{i_k}$ for $1 \leq i_1 < \dots < i_k \leq n$;
- if $t_i \leq t'_i$ hold for $i = 1, \dots, n$, then $t_1 \oplus \dots \oplus t_n \leq t'_1 \oplus \dots \oplus t'_n$.

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

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* $t_i \rightarrow t'_i$ involving two types t_i and t'_i . We interpret each such function by a total function $f_i : \text{dom}(t_i) \rightarrow \text{dom}(t'_i)$. Note that using $t'_i = t''_i \oplus \mathbb{1}$ we can cover also partial functions. 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 none of both, in which case we get a *derived* function.

The functions defined for a TASM including the static and derived functions, define the set of states of the TASM. More precisely, each pair $\ell = (f_i, x)$ with $x \in \text{dom}(t_i)$ 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 $t \rightarrow \{\mathbb{1}\}$ a *relation*. This generalises the standard notion of relation, in which case we would further require that t is a tuple type of $a_1 : t_1 \times \dots \times a_n : t_n$. In particular, as $\{\mathbb{1}\}$ can be considered as a truth value type, we may identify R with a subset of $\text{dom}(t)$, i.e. $R \simeq \{x \in \text{dom}(t) \mid R(x) \neq \emptyset\}$. In this spirit we also write $x \in R$ instead of $R(x) \neq \emptyset$, and $x \notin R$ instead of $R(x) = \emptyset$.

As with ASMs we define state transitions via *update rules*, which are defined by the following language (we deviate from the syntax used in ASMs):

- the skip rule **skip**, which indicates no change;
- the assignment rule $f(\tau) := \tau'$ with a function $f : t \rightarrow t'$ and terms τ, τ' of type t and t' , respectively, which indicates that the value of the location determined by f and τ will be updated to the value of term τ' ;
- the sequence rule $r_1; \dots; r_n$, which indicates that the rules r_1, \dots, r_n will be executed sequentially;
- the block rule $r_1 \parallel \dots \parallel r_n$, which indicates that the rules r_1, \dots, r_n will be executed in parallel;
- the conditional rule $\varphi_1\{r_1\} \boxtimes \varphi_2\{r_2\} \boxtimes \dots \boxtimes \varphi_n\{r_n\}$, which 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 $\lambda x = \tau\{r\}$, which means to assign to the variable x the value defined by the term τ and to use this value in the rule r ;
- the forall rule $\mathbf{A}x \bullet \varphi\{r\}$, which indicates the parallel execution of r for all values of x satisfying φ ;

- the choice rule $@x \bullet \varphi\{r\}$ indicates the execution of r for one value of x satisfying φ ;
- the call rule $r(\tau)$ indicates the execution of rule r with parameters τ .

Each update rule r defines an update set $\Delta(r)$ in the same way as for “ordinary” ASMs [5, p.74]. Such an update set is *consistent* iff $(\ell, v_1) \in \Delta \wedge (\ell, v_2) \in \Delta \Rightarrow v_1 = v_2$ holds for all locations. Then state transitions are defined by consistent update sets. Then a *run* of a TASM 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 TASM in state s_i .

What is different in TASMs is that the terms used in the rules are typed, i.e. for each type t we obtain a set \mathbb{T}_t of terms of type t . Then also the formulae φ used in the rules change in that equational atoms $\tau_1 = \tau_2$ can only be built from terms τ_1, τ_2 that have the same type. All the rest remains unchanged.

So let us assume that for each type t we are given a set V_t of variables of type t . Then we should have $V_t \subseteq \mathbb{T}_t$ and $dom(t) \subseteq \mathbb{T}_t$, and further terms can be built as follows:

- For $\tau \in \mathbb{T}_t$ and $t \leq t'$ we get $\pi_{t'}^t(\tau) \in \mathbb{T}_{t'}$.
- For $\tau \in \mathbb{T}_{t_1 \times \dots \times t_n}$ we get $\pi_i(\tau) \in \mathbb{T}_{t_i}$.
- For $\tau_i \in \mathbb{T}_{t_i}$ for $i = 1, \dots, n$ we get $(\tau_1, \dots, \tau_n) \in \mathbb{T}_{t_1 \times \dots \times t_n}$, $\iota_i(\tau_i) \in \mathbb{T}_{t_1 \oplus \dots \oplus t_n}$, and $\{\tau_i\} \in \mathbb{T}_{\{t_i\}}$.
- For $\tau_1, \tau_2 \in \mathbb{T}_{\{t\}}$ we get $\tau_1 \cup \tau_2 \in \mathbb{T}_{\{t\}}$, $\tau_1 \cap \tau_2 \in \mathbb{T}_{\{t\}}$, and $\tau_1 - \tau_2 \in \mathbb{T}_{\{t\}}$.
- For $\tau \in \mathbb{T}_{\{t\}}$, a constant $e \in dom(t')$ and static functions $f : t \rightarrow t'$ and $g : t' \times t' \rightarrow t'$ we get $src[e, f, g](\tau) \in \mathbb{T}_{t'}$.
- For $\tau_i \in \mathbb{T}_{\{t_i\}}$ for $i = 1, 2$ we get $\tau_1 \bowtie \tau_2 \in \mathbb{T}_{\{t_1 \bowtie t_2\}}$ using the maximal common subtype $t_1 \bowtie t_2$ of t_1 and t_2 .
- For $x \in V_t$ and a formula φ we get $\mathbf{I}x.\varphi \in \mathbb{T}_t$.

The last three constructions for terms need some more explanation. Structural recursion $src[e, f, g](\tau)$ is a powerful construct for database queries [17]. It is defined as follows:

- $src[e, f, g](\tau) = e$, if $\tau = \emptyset$;
- $src[e, f, g](\tau) = f(v)$, if $\tau = \{v\}$;
- $src[e, f, g](\tau) = g(src[e, f, g](v_1), src[e, f, g](v_2))$, if $\tau = v_1 \cup v_2 \wedge v_1 \cap v_2 = \emptyset$.

In order to be uniquely defined, the function g must be associative and commutative with e as a neutral element.

We can use structural recursion to specify comprehension, which is extremely important for views. We get $\{x \in \tau \mid \varphi\} = src[\emptyset, f_\tau, \cup](\tau)$ using the static function f_τ with $f_\tau(x) = \{x\}$, if $\varphi(x)$ holds, else $f_\tau(x) = \emptyset$, which can be composed out of very simple functions [13].

For the *join* $\tau_1 \bowtie \tau_2$, using $\llbracket \cdot \rrbracket_s$ to denote the interpretation in a state s , we get $\llbracket \tau_1 \bowtie \tau_2 \rrbracket_s = \{v \in dom(t_1 \bowtie t_2) \mid \exists v_1 \in \llbracket \tau_1 \rrbracket_s, v_2 \in \llbracket \tau_2 \rrbracket_s. (\pi_{t_1}^{t_1 \bowtie t_2}(v) = v_1 \wedge \pi_{t_2}^{t_1 \bowtie t_2}(v) = v_2)\}$, which generalises the natural join from relational algebra [13].

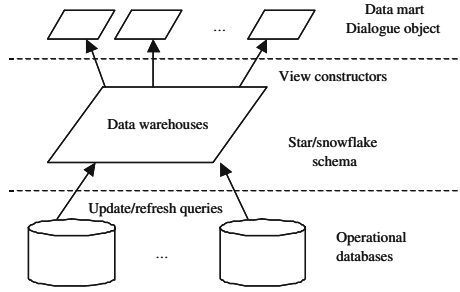


Fig. 1. The general architecture of a data warehouse and OLAP

$\mathbf{I}x.\varphi$ stands for “the unique x satisfying φ ”. If such an x does not exist, the term will be undefined, i.e. we have $\llbracket \mathbf{I}x.\varphi \rrbracket_s = v$, if $\llbracket \{x \mid \varphi\} \rrbracket_s = \{v\}$, otherwise it is undefined. If an undefined term appears in a rule, the rule defines an inconsistent update set.

For relations R of kind $t \rightarrow \{\mathbf{1}\}$ we further permit *bulk assignments*, which take one of the following forms $R := \tau$ (for replacing), $R :+_k \tau$ (for inserting), $R :-_k \tau$ (for deleting), and $R :&_k \tau$ (for updating), using each time a term τ of type $\{t\}$ and a supertype k of t . This is an old idea adopted from the database programming language Pascal/R [16]. These constructs are shortcuts for the following TASM rules:

- $R := \tau$ represents $\mathbf{A}x \bullet x \in \tau \{R(x) := \{\mathbf{1}\}\} \parallel \mathbf{A}x \bullet x \in R \wedge x \notin \tau \{R(x) := \emptyset\}$
- $R :+_k \tau$ represents $\mathbf{A}x \bullet (x \in \tau \wedge \forall y (y \in R \rightarrow \pi_k^t(x) \neq \pi_k^t(y))) \{R(x) := \{\mathbf{1}\}\}$
- $R :-_k \tau$ represents $\mathbf{A}x \bullet (x \in R \wedge \exists y (y \in \tau \wedge \pi_k^t(x) = \pi_k^t(y))) \{R(x) := \emptyset\}$
- $R :&_k \tau$ represents

$$\mathbf{A}x \bullet (x \in R \wedge \exists y (y \in \tau \wedge \pi_k^t(x) = \pi_k^t(y))) \{R(x) := \emptyset\};$$

$$\mathbf{A}x \bullet x \in \tau \wedge \forall y (y \in R \wedge y \neq x \rightarrow \pi_k^t(x) \neq \pi_k^t(y)) \{R(x) := \{\mathbf{1}\}\}$$

Up to now we have defined a typed ASM. 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 [2,7].

4 A Data Warehouse Ground Model in TASMs

Following the basic ideas in [22], to separate the output from the input, we get the three-tier architecture of the data warehouse shown in Figure 1. At the bottom tier, we have the operational database model, which has the control of the updates to data warehouse. The updates are abstracted as data extraction

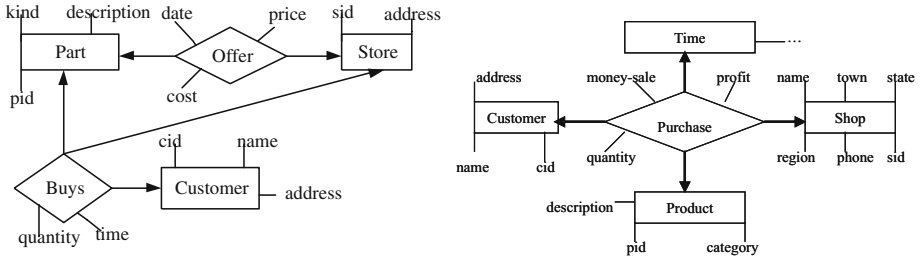


Fig. 2. The operational database and data warehouse schemata

from the operational database to maintain the freshness of the data warehouse. In the middle tier, we have the data warehouse which is modelled in star schema [9]. At the top tier, we have the data marts, which are constructed out of dialogue objects with OLAP operations. Based on this three-tier architecture, we end up with three linked ASMs as in [22], or TASM when we apply typed ASM. In the following we will use one of them as an example to show the difference between the ground model in ASM and the one in TASM.

We again use the grocery store as the example. In this case we have a single operational database with five relation schemata as illustrated in the left hand HERM diagram in Figure 2, the start schema for the data warehouse in the right hand of the figure.

We present DB-ASM (in ASM) and DB-TASM (in TASM) in the following, with our focus on the affected part, i.e. the data extraction for *Purchase*, the fact table in the data warehouse star schema:

```

ASM DB-ASM
IMPORT DW-ASM(Shop, Product, Customer, Time, Purchase)
EXPORT extract_purchase
SIGNATURE ...
BODY
  main = ...
  extract_purchase =
    forall i, p, s, t, p', c with  $\exists q. Buys(i, s, p, q, t) \neq \perp \wedge$ 
       $\exists n, a. Customer_{ab}(i, n, a) \neq \perp \wedge \exists k, d. Part(p, k, d) \neq \perp \wedge$ 
       $\exists a'. Store(s, a') \neq \perp \wedge \exists d. (Offer(p, s, p', c, d) \neq \perp \wedge Date(t) = d$ 
    do let  $Q = sum(q \mid Buys(i, s, p, q, t) \neq \perp)$ ,  $S = Q * p'$ ,
       $P = Q * (p' - c)$ 
      in  $Purchase(i, p, s, t, Q, S, P) := 1$ 
    enddo

```

```

ASM DB-TASM
IMPORT DW-TASM(Shop, Product, Customer, Time, Purchase)
EXPORT extract_purchase

```

SIGNATURE ...

BODY

 $main = \dots$ $extract_purchase = \text{Purchase} : \&\{cid \times pid \times sid \times time\}$ $\{(i, s, p, t, Q, S, P) \mid \exists pr, c. ((i, s, p, t, pr, c) \in$ $\pi_{i,s,p,t,pr,c}(\text{Buys} \bowtie \text{Customer}_{db} \bowtie \text{Part} \bowtie \text{Store} \bowtie \text{Offer} \bowtie \text{Date}(t))$ $\wedge Q = \text{src}[0, \pi_q, +](\{(i', s', p', q, t') \mid (i', s', p', q, t') \in \text{Buys} \wedge$ $i' = i \wedge s' = s \wedge p' = p \wedge t' = t\})$ $\wedge S = Q * pr \wedge P = Q * (pr - c)\}$

As shown above, we have redefined *sum* function using structural recursion constructor, and used the bulk update operation in *extract_purchase* in the typed model DB-TASM.

5 Refinement of Typed ASMs

The general notion of *refinement* in ASMs relates two ASMs \mathfrak{M} and \mathfrak{M}^* in the following way:

- a correspondence between some states s of M and some states s^* of M^* , and
- a correspondence between the runs of \mathfrak{M} and \mathfrak{M}^* involving states s and s^* , respectively.

Keeping in mind that we are looking at the application of TASM 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 \triangleright between main rules r of \mathfrak{M} and main rules r^* of \mathfrak{M}^* . Finally, take initial states s_0, s_0^* for \mathfrak{M} and \mathfrak{M}^* , respectively.

Definition 1. *A TASM \mathfrak{M}^* is called a (weak) 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, \bar{s} of \mathfrak{M} , where \bar{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^*, \bar{s}^* of \mathfrak{M}^* with $(s, s^*) \models \mathcal{A}$, $(\bar{s}, \bar{s}^*) \models \mathcal{A}$, and \bar{s}^* is the successor state of s_m^* with respect to the update set Δ_{r^*} defined by the main rule r^* .*

While Definition 1 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 $\iota_b : \text{dom}(b) \rightarrow \text{dom}(b)$ for all base types b . This can be extended to bijections ι_t for any type t as follows:

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

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 closed under isomorphisms.

Definition 2. A refinement \mathfrak{M}^* of a TASM \mathfrak{M} with abstraction predicate \mathcal{A} is called a *strong 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 $x \in \text{dom}(t_i)$ 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 $x \in \text{dom}(t_i^*)$ 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 $g(f(s(\mathcal{S}))) = s(\mathcal{S})$.

Definition 2 provides a stronger proof obligation for refinements in the application area we are interested in. Furthermore, this notion of strong refinement heavily depends on the presence of types.

Let us finally discuss refinement rules for data warehouses and OLAP systems. According to [21] the integration of schemata and views on all three tiers is an important part of such refinement rules. Furthermore, the intention is to set up rules of the form

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

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

Example 1. As a sample refinement rule adopted from [11] take

$$\frac{\mathfrak{M} \triangleright R : (t_{11} \oplus \dots \oplus t_{1k}) \times t_2 \times \dots \times t_n \rightarrow \{\mathbb{1}\}}{\begin{array}{l} \mathfrak{M}^* \triangleright R_1^* : t_{11} \times t_2 \times \dots \times t_n \rightarrow \{\mathbb{1}\} \\ R_2^* : t_{12} \times t_2 \times \dots \times t_n \rightarrow \{\mathbb{1}\} \\ \vdots \\ R_k^* : t_{1k} \times t_2 \times \dots \times t_n \rightarrow \{\mathbb{1}\} \end{array}}$$

Then the corresponding abstraction predicate \mathcal{A} is as

$$\forall x_1, \dots, x_n (R_i^*(x_1, \dots, x_n) = 1 \leftrightarrow R((i, x_1), \dots, x_n) = 1)$$

The corresponding computable queries f and g can be obtained as

$$\begin{aligned} R_1 &:= \{(x_1, \dots, x_n) \mid ((1, x_1), x_2, \dots, x_n) \in R\} \parallel \dots \parallel \\ R_k &:= \{(x_1, \dots, x_n) \mid ((k, x_1), x_2, \dots, x_n) \in R\} \end{aligned}$$

and

$$R := \bigcup_{i=1}^k \{((i, x_1), x_2, \dots, x_n) \mid (x_1, \dots, x_n) \in R_i\},$$

respectively.

6 Conclusion

In this paper we introduced a typed version of Abstract State Machines. The motivation for this is that our research aims at applying the ground model / refinement method supported by ASMs in the area of data warehouses and OLAP systems. For this it is advantageous to have bulk data types (sets, relations) and declarative query expressions available, but these are not available in ASMs. So, the major reason for adding them is to increase the applicability of ASMs. We did, however, show that these extensions do not increase the expressiveness of ASMs, as each typed ASM is equivalent to an “ordinary” one.

In a second step we approached refinement in typed ASMs. We clarified what we want to achieve by refinements in data-intensive application areas. In particular, we distinguish between weak and strong refinement, the latter one being more restrictive with respect to changes to the signature, in order to preserve the semantics of data in accordance with schema dominance as discussed in [11].

This brings us closer to the goal of our research project to set up sound and complete refinement rules for the development of distributed data warehouses and OLAP systems. The general approach to defining such rules has already been discussed in [21], while the current work clarifies, what kind of rules we have to expect. In [15] we already discussed rules for distribution, but the more challenging task consists in rules for data integration. So our next steps will address such rules following among others the work on view integration in [11].

References

1. Abrial, J.-R.: *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
2. Blass, A., Gurevich, J.: Abstract State Machines Capture Parallel Algorithms, *ACM Transactions on Computational Logic*, **4**(4), 2003, 578–651.

3. Börger, E.: The ASM Ground Model Method as a Foundation for Requirements Engineering, *Verification: Theory and Practice*, 2003.
4. Börger, E.: The ASM Refinement Method, *Formal Aspects of Computing*, **15**, 2003, 237–257.
5. Börger, E., Stärk, R.: *Abstract State Machines*, Springer-Verlag, Berlin Heidelberg New York, 2003.
6. Del Castillo, G., Gurevich, Y., Stroetmann, K.: Typed Abstract State Machines, 1998, Unpublished, available from <http://research.microsoft.com/gurevich/Opera/137.pdf>.
7. Gurevich, J.: Sequential Abstract State Machines Capture Sequential Algorithms, *ACM Transactions on Computational Logic*, **1**(1), 2000, 77–111.
8. Gyssens, M., Lakshmanan, L.: A foundation for multidimensional databases, *Proc. 22nd VLDB Conference, Mumbai (Bombay), India*, 1996.
9. Kimball, R.: *The Data Warehouse Toolkit*, John Wiley & Sons, 1996.
10. Lewerenz, J., Schewe, K.-D., Thalheim, B.: Modelling Data Warehouses and OLAP Applications Using Dialogue Objects, in: *Conceptual Modeling – ER’99* (J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, E. Métais, Eds.), vol. 1728 of *LNCS*, Springer-Verlag, 1999, 354–368.
11. Ma, H., Schewe, K.-D., Thalheim, B., Zhao, J.: View Integration and Cooperation in Databases, Data Warehouses and Web Information Systems, *Journal on Data Semantics*, **IV**, 2005, 213–249.
12. Schewe, K.-D.: *Specification and Development of Correct Relational Database Programs*, Technical report, Clausthal Technical University, Germany, 1997.
13. Schewe, K.-D.: On the Unification of Query Algebras and their Extension to Rational Tree Structures, in: *Proc. Australasian Database Conference 2001* (M. Orłowska, J. Roddick, Eds.), IEEE Computer Society, 2001, 52–59.
14. Schewe, K.-D., Zhao, J.: ASM Ground Model and Refinement for Data Warehouses, *Proc. 12th International Workshop on Abstract State Machines – ASM 2005* (D. Beauquier, E. Börger, A. Slissenko, Eds.), Paris, France, 2005.
15. Schewe, K.-D., 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* (S. Hartmann, M. Stumptner, Eds.), vol. 43 of *CRPIT*, Australian Computer Society, 2005, 97–105.
16. Schmidt, J. W.: Some High Level Language Constructs for Data of Type Relation, *ACM Transactions on Database Systems*, **2**(3), 1977, 247–261.
17. Tannen, V., Buneman, P., Wong, L.: Naturally Embedded Query Languages, in: *Database Theory (ICDT 1992)* (J. Biskup, R. Hull, Eds.), vol. 646 of *LNCS*, Springer-Verlag, 1992, 140–154.
18. Thomson, E.: *OLAP Solutions: Building Multidimensional Information Systems*, John Wiley & Sons, New York, 2002.
19. Widom, J.: Research Problems in Data Warehousing, *Proceedings of the 4th International Conference on Information and Knowledge Management*, ACM, 1995.
20. Zamulin, A. V.: Typed Gurevich Machines Revisited, *Joint Bulletin of NCC and IIS on Computer Science*, **5**, 1997, 1–26.
21. Zhao, J., Ma, H.: ASM-Based Design of Data Warehouses and On-Line Analytical Processing Systems, *Journal of Systems and Software*, **79**, 2006, 613–629.
22. Zhao, J., Schewe, K.-D.: Using Abstract State Machines for Distributed Data Warehouse Design, in: *Conceptual Modelling 2004 – First Asia-Pacific Conference on Conceptual Modelling* (S. Hartmann, J. Roddick, Eds.), vol. 31 of *CRPIT*, Australian Computer Society, Dunedin, New Zealand, 2004, 49–58.