

# Typed Abstract State Machines in Data-Intensive Applications

Klaus-Dieter Schewe      Jane Zhao

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

# Outline

- 1 Introduction
- 2 Typed Abstract State Machines
- 3 Translation TASM  $\rightarrow$  ASM
- 4 Weak and Strong Refinements
- 5 Concluding Remarks

# 1 Introduction

- ASMs (and likewise any other “formal method”) should be applicable for the development of large software systems
- Applying ASMs (or likewise any other “formal method”) for data-intensive applications is somehow a bit cumbersome
- The reason is the existence of advanced (declarative) high-level languages for these applications
- The obvious solution is to introduce an application-specific intermediate layer
- Our aim is to provide this layer

# Background

- early 90s: work on B-based development methodology for data-intensive applications:
  - back- and front-end translations
  - standard (provably correct) refinement rules plus “method” for their use
- early 90s: integrated typed specification language with topos semantics
- recently: work on ASM-based development methodology for data warehouse applications
- focus on “refinement calculus”, i.e. standard (application-dependent) refinement rules embedded in a pragmatic method for their application

## 2 Typed Abstract State Machines

- basically, we require relations on the database and the data warehouse tiers, while the OLAP tier requires sets of complex values
- start with a *type system*, e.g.

$$t = b \mid \{t\} \mid a_1 : t_1 \times \cdots \times a_n : t_n \mid a_1 : t_1 \oplus \cdots \oplus a_n : t_n \mid \mathbb{1}$$

- for this type systems we obtain the usual notation of *subtyping* denoted by  $\leq$
- subtyping  $t \leq t'$  induces a canonical projection mapping  $\pi_{t'}^t : \text{dom}(t) \rightarrow \text{dom}(t')$

# Signatures

- The *signature* of a TASM is defined by a finite list of function names  $f_1, \dots, f_m$
- Each function  $f_i$  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  $f_i : \text{dom}(t_i) \rightarrow \text{dom}(t'_i)$
- Functions can be *dynamic*, i.e. updatable
  - 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
  - by none of both, in which case we get a *derived* function

## States and “Relations”

- Each pair  $\ell = (f_i, x)$  with  $x \in \text{dom}(t_i)$  defines a *location* with  $v = f_i(x)$  as its *value*
- A *state* of a TASM is a set of location/value pairs
- (by abuse of notation) call a function  $R$  of kind  $t \rightarrow \{\mathbb{1}\}$  a *relation*
- As  $\{\mathbb{1}\}$  can be considered as a truth value type, we may identify  $R$  with a subset of  $\text{dom}(t)$
- This makes sense, in particular, if  $t$  is a record type

# Update Rules

- the skip rule **skip**
- the assignment rule  $f(\tau) := \tau'$  with a function  $f : t \rightarrow t'$  and terms  $\tau, \tau'$  of type  $t$  and  $t'$ , respectively
- the sequence rule  $r_1; \dots; r_n$
- the block rule  $r_1 \parallel \dots \parallel r_n$
- the conditional rule  $\varphi_1\{r_1\} \boxtimes \varphi_2\{r_2\} \boxtimes \dots \boxtimes \varphi_n\{r_n\}$
- the let rule  $\Lambda x = \tau\{r\}$
- the forall rule  $\mathbf{A}x \bullet \varphi\{r\}$
- the choice rule  $\textcircled{x} \bullet \varphi\{r\}$
- the call rule  $r(\tau)$ , i.e. the execution of rule  $r$  with parameters  $\tau$

## Terms in Update Rules

- 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  $e \in \text{dom}(t')$  and static functions  $f : t \rightarrow t'$  and  $g : t' \times t' \rightarrow t'$  we get  $\text{src}[e, f, g](\tau) \in \mathbb{T}_{t'}$  (**structural recursion**)
- 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  $\varphi$  we get  $\mathbf{I}x.\varphi \in \mathbb{T}_t$

# Bulk Updates

- For relations  $R$  of kind  $t \rightarrow \{\mathbb{1}\}$  we further permit *bulk assignments*:
  - $R := \tau$  (for replacing)
  - $R :+_k \tau$  (for inserting)
  - $R :-_k \tau$  (for deleting)
  - $R :&_k \tau$  (for updating)
- using each time a term  $\tau$  of type  $\{t\}$  and a supertype  $k$  of  $t$

### 3 Translation TASM $\rightarrow$ ASM

- First establish a correspondence between states
- For a function  $f : t \rightarrow t'$  in the signature of  $\mathfrak{M}$  with  $t = a_1 : t_1 \times \cdots \times a_n : t_n$  define an  $n$ -ary function  $\Phi(f) = \hat{f}$  in the signature of  $\Phi(\mathfrak{M})$  with

$$\hat{f}(x_1, \dots, x_n) = \begin{cases} f(x_1, \dots, x_n) & \text{if } \bigwedge_{1 \leq i \leq n} \hat{f}_{t_i}(x_i) = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

- In all other cases define a unary function  $\Phi(f) = \hat{f}$  with

$$\hat{f}(x) = f(x) \text{ if } \hat{f}_t(x) = 1 \quad \text{and} \quad \hat{f}(x) \text{ undefined otherwise}$$

## Translation TASM $\rightarrow$ ASM (cont.)

- In both cases  $\hat{f}_t$  is a unary function that checks whether a value  $x$  is of type  $t$ , i.e. we have  $\hat{f}_t(x) = 1$  for  $x$  of type  $t$ , while otherwise  $\hat{f}_t(x)$  is undefined. Then

$$\hat{f}_{t_1 \times \dots \times t_n}(x) = 1 \text{ iff } x = (x_1, \dots, x_n) \wedge \bigwedge_{1 \leq i \leq n} \hat{f}_{t_i}(x_i) = 1$$

$$\hat{f}_{\{t\}}(x) = 1 \text{ iff } x = \{x_1, \dots, x_k\} \wedge \bigwedge_{1 \leq i \leq k} \hat{f}_t(x_i) = 1$$

$$\hat{f}_{t_1 \oplus \dots \oplus t_n}(x) = 1 \text{ iff } \bigvee_{1 \leq i \leq n} (x = (i, x_i) \wedge \hat{f}_{t_i}(x_i) = 1)$$

$$\hat{f}_{\mathbf{1}}(x) = 1 \text{ iff } x = \mathbf{1}$$

- In doing so, each state of the TASM  $\mathfrak{M}$  corresponds in a canonical way to a state of the ASM  $\Phi(\mathfrak{M})$

# Translation of Update Rules

- The TASM  $\mathfrak{M}$  to be *equivalent* to an ASM  $\Phi(\mathfrak{M})$  iff both machines have the same runs
- For the rules in  $\mathfrak{M}$  we mainly have to consider assignment  $f(\tau) := \tau'$ , which have to be replaced in  $\Phi(\mathfrak{M})$  by assignments  $\hat{f}(\tau) := \hat{\tau}'$ , where  $\llbracket \tau' \rrbracket_s = \llbracket \hat{\tau}' \rrbracket_s$  holds for all states  $s$
- Ensure that the right term is computed by the assignment
- For this we have to translate the complex term language of TASM into extensions to the rules in  $\Phi(\mathfrak{M})$

# Translation of Terms

- For subtype functions  $\pi_{t'}$  we may assume these are defined in the signature of  $\Phi(\mathfrak{M})$  by static unary functions
- Then a rule  $r$  in  $\mathfrak{M}$  containing  $\pi_{t'}^t(\tau)$  can be replaced by  $\Lambda y = \pi_{t'}^t(\tau) \{r[\pi_{t'}^t(\tau)/y]\}$  with a fresh variable  $y$  that is substituted for the term
- Similarly, for projections  $\pi_i$  on tuple types we can replace a rule  $r$  containing  $\pi_i(\tau)$  by the conditional rule  $\tau = (\tau_1, \dots, \tau_n) \{ \Lambda y = \pi_i \{r[\pi_i(\tau)/y]\} \}$ , again using a fresh variable  $y$
- If a rule  $r$  in  $\mathfrak{M}$  involves a term  $\mathbf{I}x.\varphi$ , this can be replaced in  $\Phi(\mathfrak{M})$  by the choice rule  $@y \bullet \{x \mid \varphi\} = \{y\} \{r[\mathbf{I}x.\varphi/y]\}$  with a new variable  $y$

## Translation of Structural Recursion Terms

- If a rule  $r$  in  $\mathfrak{M}$  involves the term  $src[e, f, g](\tau)$  with  $\tau$  of type  $\{t\}$ , then replace this by a rule

$$\tau = \emptyset \{r[src[e, f, g](\tau)/\emptyset]\}$$

$$\boxtimes \exists z(\tau = \{z\}) \{ @z \bullet \tau = \{z\} \{ \Lambda y = f(z) \{ r[src[e, f, g](\tau)/y] \} \} \}$$

$$\boxtimes \exists z_1, z_2(\tau = z_1 \cup z_2 \wedge z_1 \cap z_2 = \emptyset \wedge z_1 \neq \emptyset \wedge z_2 \neq \emptyset)$$

$$\{ \mathbf{A}z_1 \bullet \hat{f}_{\{t\}}(z_1) = 1 \wedge z_1 \neq \emptyset$$

$$\{ \mathbf{A}z_2 \bullet \hat{f}_{\{t\}}(z_2) = 1 \wedge z_2 \neq \emptyset \wedge z_1 \cap z_2 = \emptyset \wedge \tau = z_1 \cup z_2$$

$$\{ \Lambda y = g(src[e, f, g](z_1), src[e, f, g](z_2)) \{ r[src[e, f, g](\tau)/y] \} \} \}$$

## 4 Weak and Strong Refinements

- The general notion of *refinement* in ASMs relates two ASMs  $\mathfrak{M}$  and  $\mathfrak{M}^*$  in the following way:
  - a correspondence between the 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.
- For data-intensive applications we like to have a more specialised definition of refinement

# Correspondence Between States

- 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
- Require 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

## (Weak) Refinement

- 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
  - a correspondence  $\triangleright$  between main rules  $r$  of  $\mathfrak{M}$  and main rules  $r^*$  of  $\mathfrak{M}^*$  such that

for all states  $s, s_n$  of  $\mathfrak{M}$ , where  $s_n$  is the successor state of  $s$  with respect to the update set  $\Delta_r$  defined by the main rule  $r$ , there are states  $s^*, s_n^*$  of  $\mathfrak{M}^*$  with  $(s, s^*) \models \mathcal{A}$ ,  $(s_n, s_n^*) \models \mathcal{A}$ , such that  $s_n^*$  is the successor state of  $s^*$  with respect to the update set  $\Delta_{r^*}$  defined by the main rule  $r^*$

- weak refinements still permit too much latitude for data-intensive applications

## Additional Requirements

- the refined schema  $\mathcal{S}^*$  of the refining TASM  $\mathfrak{M}^*$  should dominate the schema  $\mathcal{S}$  of the refined TASM  $\mathfrak{M}$
- For this we need a notion of computable query
- We first define isomorphisms starting from bijections  $\iota_b : \text{dom}(b) \rightarrow \text{dom}(b)$  for all base types  $b$ , which can be extended to bijections  $\iota_t$  for any type  $t$
- Then  $\iota$  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 such that for all isomorphisms  $\iota$  of  $\mathcal{S} \cup \mathcal{S}^*$  we have  $f \circ \iota = \iota \circ f$

# Strong Refinement

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

- $\mathfrak{M}$  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  $i = 1, \dots, n$ .
- $\mathfrak{M}^*$  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  $i = 1, \dots, m$ .
- 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  $g(f(s(\mathcal{S}))) = s(\mathcal{S})$ .

## 5 Concluding Remarks

- We presented a typed variant of ASMs that is tailored to declarativity in data-intensive applications
- The typing does (of course) not increase expressiveness, but translation is effective
- We tailored refinement emphasising the preservation of “database” content
- For this we referred to schema dominance with computable queries
- Defining a system of standard rules has been done elsewhere, yet not in the context of ASMs