INSTITUT FÜR INFORMATIK

Composing Personalised Services on top of Abstract State Services

Hui Ma[1] Klaus-Dieter Schewe[2] Bernhard Thalheim Qing Wang[4]

Bericht Nr. 0804
Mai 2008

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Composing Personalised Services on top of Abstract State Services

Hui Ma[1] Klaus-Dieter Schewe[2] Bernhard Thalheim
Qing Wang[4]

Bericht Nr. 0804
Mai 2008

e-mail: huimanz@gmail.com, k.d.schewe@massey.ac.nz,
thalheim@is.informatik.uni-kiel.de,
q.q.wang@massey.ac.nz

Dieser Bericht ist als persönliche Mitteilung aufzufassen.[a]

---

[a] [1] Victoria University Wellington, School of Mathematics, Statistics and Computer Science, Wellington, New Zealand
[2] Information Science Research Centre,Palmerston North, New Zealand
[3] Christian-Albrechts-University at Kiel, Department of Computer Science, Kiel, Germany
[4] Massey University, Palmerston North, New Zealand

Composing Personalised Services on top of Abstract State Services

Hui Ma[1] Klaus-Dieter Schewe[2] Bernhard Thalheim[3] Qing Wang[4]

[1] Victoria University Wellington, School of Mathematics, Statistics and Computer Science, Wellington, New Zealand, Email: huimanz@gmail.com
[2] Information Science Research Centre,Palmerston North, New Zealand Email: kdschewe@acm.org
[3] Christian-Albrechts-University at Kiel, Department of Computer Science, Kiel, Germany
Email: thalheim@is.informatik.uni-kiel.de
[4] Massey University, Palmerston North, New Zealand
Email: q.q.wang@massey.ac.nz

Abstract

We introduce Abstract State Services (ASSs) as an abstraction of data-intensive services that can be made available for use by other systems, e.g. via the web. An ASS combines a hidden database layer with an operation-equipped view layer, and can be anything from a simple function to a full-fledged Web Information System or a Data Warehouse. We adopt the fundamental approach of Abstract State Machines to model ASSs. Then we show how tailored services can be extracted from available ASSs, integrated with other ASSs and personalised to user preferences.

Table of Contents

Chapter 1

Introduction

Since its introduction the role of the world-wide web has shifted from enabling access to a pool of documents to the provision of services. Such web services can in fact be anything: a simple function, a data warehouse, or a fully functional Web Information System. The unifying characteristic is that content and functionality are made available for use by human users or other services. Therefore, web service integration has become a highly relevant research topic, and a lot of work has been investigated into it [2] including service personalisation [6]. It also appears quite natural that Abstract State Machines (ASMs) [5] have also been used for modelling web services [1].

In this paper we take a more abstract, conceptual approach to service integration, composition and personalisation. We adopt the fundamental idea from the area of dialogue systems [12] that a service can be described by two layers: a hidden database layer consisting of a database schema and transactions, and a visible view layer on top of it providing views and functions based on them. This idea has already been mirrored in development methods for Web Information Systems [14, 13], and also appears as a natural choice for component-based systems development [15]. ASMs have also been suggested as a means for modelling such services [3].

However, we want to go one step further and develop a theory of Abstract State Services (ASSs) following the line of thought of the ASM thesis. Gurevich and Blass [8, 4] formalised sequential and parallel algorithms by requiring a small set of intuitive, abstract postulates to be satisfied, then proved that these postulates are always satisfiable by (sequential) ASMs, i.e. ASMs capture algorithms in the most general sense in a natural way. This has been picked up in [18] and refined in [17] for database transformations exploiting states as meta-finite logical structures [7]. This research can be used as a basis for the model of transactions on the database level, and thus forms the basis of the formal definition of ASSs. In doing so the web as the medium through which a service may become available is of no importance; the notion of ASS can also be applied to enterprise services that are only available to selected clients.

We then address the problem of service integration and composition. Integration means to replace two or more ASSs by a single new one that offers all the functionality of the individual services. We show that this problem can actually be reduced

to database schema and view integration. The second one requires the extraction of service components from existing ASSs that feed a new service without replacing the original ones. For this we can adopt ideas from component composition [15].

In a second step we address the problem of service personalisation according to preference rules. For this we pick up the idea from WIS personalisation [16] to compose personalised tasks, where the preference rules indicate, which choices will be preferred.

The remainder of this papers is organised as follows. In Section 2 we formally introduce the model of Abstract State Services by means of postulates. We then use the language of ASMs in a loose way in our examples for ASSs, leaving the proof that the language actually captures ASSs for publication elsewhere. In Section 3 we show how to integrate ASSs, i.e. replace existing ASSs by a new one that preserves the functionality. In Section 4 we show how to extract components from ASSs in a formal way and how to recompose them into new services. All these operations remain with the framework defined by ASSs. Finally, Section 5 is devoted to the problem of service personalisation. For this we show how the extraction process discussed before can be tailored by preference rules. We conclude with a brief summary and discussion of open research problems.

Chapter 2

Abstract State Services

Traditional database architecture distinguishes at least three layers: a conceptual layer describing the database schema in an abstract way, a physical layer implementing the schema, and an external layer made out of views. The external layer exports the data that can then be used by users or programs. For our purposes here we can neglect the physical layer, but in order to capture data-intensive services, we complete this architecture by adding operations on both the conceptual and the external layer, the former one being handled as database transactions, whereas the latter ones provide the means with which users can interact with a database.

2.1 The Database Layer

In order to abstract from this architecture to obtain a model of abstract services we first formulate postulates for the database layer. Following the general approach of Abstract State Machines [8] we may consider each database computation as a sequence of abstract states, each of which represents the database (instance) at a certain point in time plus maybe additional data that is necessary for the computation, e.g. transaction tables, log files, etc. In order to capture the semantics of transactions we distinguish between a wide-step transition relation and small step transition relations. A transition in the former one marks the execution of a transaction, so the wide-step transition relation defines infinite sequences of transactions. Without loss of generality we can assume a serial execution, while of course interleaving is used for the implementation, but serialisability has to be guaranteed. Then each transaction itself corresponds to a finite sequence of states resulting from a small step transition relation, which should then be subject to the postulates for database transformations [18].

Definition 1 database postulate. A *database system* DBS consists of

- a set $\mathcal{S}$ of states, together with a subset $\mathcal{I} \subseteq \mathcal{S}$ of initial states,
- a wide-step transition relation $\tau \subseteq \mathcal{S} \times \mathcal{S}$, and
- a set $\mathcal{T}$ of transactions, each of which is associated with a small-step transition relation $\tau_t \subseteq \mathcal{S} \times \mathcal{S}$ $(t \in \mathcal{T})$ satisfying the postulates of a database transformation over $\mathcal{S}$.

We will further elaborate the notion of state and database transformation. For now note that different from the sequential time postulate in Gurevich's work we permit non-determinism both in the wide-step transition relation and in the small-step transition relations. For the first one this is due to the fact that transactions may be started anytime, and the database system will schedule them in a serialisable way thereby defining a (serial) run. The non-determinism in the small-step transition relations is far more limited, as it is only meant to capture the creation of values such as identifiers as a highly expressive means in query and update languages.

Definition 2. A *run* of a database system DBS is an infinite sequence $S_0, S_1, \ldots$ of states $S_i \in \mathcal{S}$ starting with an initial state $S_0 \in \mathcal{I}$ such that for all $i \in \mathbb{N}$ $(S_i, S_{i+1}) \in \tau$ holds, and there is a transaction $t_i \in \mathcal{T}$ with a finite run $S_i = S_i^0, \ldots, S_i^k = S_{i+1}$ such that $(S_i^j, S_i^{j+1}) \in \tau_{t_i}$ holds for all $j = 0, \ldots, k-1$.

2.2 The View Layer

Views in general are expressed by queries, i.e. read-only database transformations. Therefore, we can assume that a view on a database state $S_i \in \mathcal{S}$ is given by a finite run $S_i = S_0^v, \ldots, S_\ell^v$ of some database transformation v with $S_i \subseteq S_\ell^v$ – traditionally, we would consider $S_\ell^v - S_i$ as the view. We can use this to extend a database system by views.

In doing so we let each state $S \in \mathcal{S}$ to be composed as a union $S_d \cup V_1 \cup \cdots \cup V_k$ such that each $S_d \cup V_j$ is a view on S_d. As a consequence, each wide-step state transition becomes a parallel composition of a transaction and an operation that switches views on and off. This leads to the definition of an Abstract State Service (ASS).

Definition 3 extended view postulate. An *Abstract State Service* (ASS) consists of a database system DBS, in which each state $S \in \mathcal{S}$ is a finite composition $S_d \cup V_1 \cup \cdots \cup V_k$, and a finite set $\mathcal{V}$ of (extended) views. Each view $v \in \mathcal{V}$ is associated with a database transformation such that for each state $S \in \mathcal{S}$ there are views $v_1, \ldots, v_k \in \mathcal{V}$ with finite runs $S_d = S_0^j, \ldots, S_{n_j}^j = S_d \cup V_j$ of v_j $(j = 1, \ldots, k)$. Each view $v \in \mathcal{V}$ is further associated with a finite set $\mathcal{O}_v$ of (service) operations $o_1, \ldots, o_n$ such that for each $i \in \{1, \ldots, n\}$ and each $S \in \mathcal{S}$ there is a unique state $S' \in \mathcal{S}$ with $(S, S') \in \tau$. Furthermore, if $S = S_d \cup V_1 \cup \cdots \cup V_k$ with V_i defined by v_i and o is an operation associated with v_k, then $S' = S'_d \cup V'_1 \cup \cdots \cup V'_m$ with $m \geq k-1$, and V'_i for $1 \leq i \leq k-1$ is still defined by v_i.

In a nutshell, in an ASS we have view-extended database states, and each service operation associated with a view induces a transaction on the database, and may change or delete the view it is associated with, and even activate other views. These service operations are actually what is exported from the database system to be used by other systems or directly by users, in which case we obtain the dialogue interfaces described in [12] or the web interfaces in [14].

The abstract handling of service operations that induce transactions avoids the view update problem, which has to be taken into account when dealing with concrete specifications for ASSs, e.g. using the theory developed by Hegner [10].

What is exported can be very limited such as simple aggregation functions, in which case most of the data in the database would be hidden. The other extreme would be to export the complete database and define operations that take a query text as input and then process the query. Both extremes (and anything between them) are supported by the definition of ASSs. We will illustrate this later in this section by several examples.

2.3 Database Transformations

The definition of database systems and by that also the definition of ASS refer to postulates for database transformations that have been elaborated in [18, 17]. We will briefly describe these postulates here, though a full motivation will not be possible due to space limitations. In total, there will be five postulates: the *sequential time postulate*, the *abstract state postulate*, the *background postulate*, the *exploration boundary postulate*, and the *genericity postulate*. An object satisfying these postulates will be a data transformation. Together with the *database postulate* in Definition 1 and the *extended view postulate* in Definition 3 we obtain the complete definition of ASSs by means of postulates.

Definition 4 sequential time postulate. A database transformation t is associated with a non-empty set of states $\mathcal{S}_t$ together with a non-empty subsets $\mathcal{I}_t$ and $\mathcal{F}_t$ of initial and final states, respectively, and a one-step transition relation τ_t over $\mathcal{S}_t$, i.e. $\tau_t \subseteq \mathcal{S}_t \times \mathcal{S}_t$.

Analogously to Definition 2 a *run* of a database transformation t is a finite sequence $S_0, \ldots, S_f$ of states with $S_0 \in \mathcal{I}_t$, $S_f \in \mathcal{F}_t$, $S_i \notin \mathcal{F}_t$ for $i < f$, and $(S_i, S_{i+1}) \in \tau_t$ for all $i = 0, \ldots, f-1$.

The abstract state postulate is an adaptation of the corresponding postulate for Abstract State Machines [8], according to which states are first-order structures, i.e. sets of functions. These functions are interpretations of function symbols given by some signature.

Definition 5. A *signature* Σ is a set of function symbols, each associated with a fixed arity. A *structure* over Σ consists of a set B, called the *base set* of the structure together with interpretations of all function symbols in Σ, i.e. if $f \in \Sigma$ has arity k, then it will be interpreted by a function from B^k to B. An isomorphism from structure X to structure Y is defined by a bijection $\sigma : B_X \to B_Y$ between the base sets that extends to functions by $\sigma(f_X(b_1, \ldots, b_k)) = f_Y(\sigma(b_1), \ldots, \sigma(b_k))$. A *$Z$-isomporphism* for $Z \subseteq B_X \cap B_Y$ is an isomorphism σ from X to Y that fixes Z, i.e. $\sigma(b) = b$ for all $b \in Z$.

Taking structures as states reflects common practice in mathematics, where almost all theories are based on first-order structures. Variables are special cases of function symbols of arity 0, and constants are the same, but unchangeable.

Definition 6 abstract state postulate. All states $S \in \mathcal{S}_t$ of a database transformation t are structures over the same signature Σ_t, and whenever $(S, S') \in \tau_t$

holds, the states S and S' have the same base set. The sets $\mathcal{S}_t$, $\mathcal{I}_t$ and $\mathcal{F}_t$ are closed under isomorphisms, and for $(S_1, S_1') \in \tau_t$ each isomorphism from S_1 to S_2 is also an isomorphism from S_1' to $S_2' = \sigma(S_1')$ with $(S_2, S_2') \in \tau_t$.

Furthermore, the signature Σ_t is composed as a disjoint union out of a database signature Σ_{db}, an algorithmic signature Σ_a, and a finite set of unary bridge function symbols, i.e. $\Sigma_t = \Sigma_{db} \cup \Sigma_a \cup \{f_1, \ldots, f_\ell\}$. The base set of a state is $B = B_{db} \cup B_a$ with interpretation of function symbols in Σ_{db} and Σ_a over B_{db} and B_a, respectively. The interpretation of a bridge function symbols defines a function from B_{db} to B_a. With respect to such states the restriction to Σ_{db} is a finite structure.

The postulates in Definitions 4 and 6 are in line with the sequential ASM thesis [8], and with the exception of allowing non-determinism in the sequential time postulate and the reference to meta-finite structures in the abstract state postulate there is nothing in these postulates that makes a big difference to postulates for sequential algorithms. The next postulate, however, is less obvious, as it refers to the background of a computation, which contains everything that is needed to perform the permutation that is not yet captured by the state. For instance, truth values and their connectives, and a value $\perp$ to denote undefinedness constitute necessary elements in a background.

For database transformations, in particular, we have to capture constructs that are determined by the used data model, e.g. relational, object-oriented or semi-structured, i.e. we will have to deal with type constructors, and with functions defined on such types. Furthermore, when we allow values, e.g. identifiers to be created non-deterministically, we would like to take these values out of an infinite set of reserve values. Once created, these values become active, and we can assume they can never be used again for this purpose.

Following [4] we use background classes to define backgrounds, which will then become part of states. Background classes themselves are determined by background signatures that consist of constructor symbols and function symbols. Function symbols are associated with a fixed arity as in Definition 5, but for constructor symbols we also permit the arity to be unfixed or bounded.

Definition 7. Let $\mathcal{D}$ be a set of base domains and V_K a background signature, then a *background class* $\mathcal{K}$ with V_K over $\mathcal{D}$ is constituted by

- the universe $\mathcal{U} = \bigcup_{D \in \mathfrak{D}} D$ of elements, where $\mathfrak{D}$ is the smallest set with $\mathcal{D} \subseteq \mathfrak{D}$ satisfying the following properties for each constructor symbol $\llcorner\lrcorner \in V_K$:
  - If $\llcorner\lrcorner \in V_K$ has unfixed arity, then $\llcorner D \lrcorner \in \mathfrak{D}$ for all $D \in \mathfrak{D}$, and $\llcorner a_1, \ldots, a_m \lrcorner \in \llcorner D \lrcorner$ for every $m \in \mathbb{N}$ and $a_1, \ldots, a_m \in D$.
  - If $\llcorner\lrcorner \in V_K$ has unfixed arity, then $A_{\llcorner\lrcorner} \in \mathfrak{D}$ with $A_{\llcorner\lrcorner} = \bigcup_{\llcorner D \lrcorner \in \mathfrak{D}} \llcorner D \lrcorner$.
  - If $\llcorner\lrcorner \in V_K$ has bounded arity n, then $\llcorner D_1, \ldots, D_m \lrcorner \in \mathfrak{D}$ for all $m \leq n$ and $D_i \in \mathfrak{D}$ $(1 \leq i \leq m)$, and $\llcorner a_1, \ldots, a_m \lrcorner \in \llcorner D_1, \ldots, D_m \lrcorner$ for every $m \in \mathbb{N}$ and $a_1, \ldots, a_m \in D$.
  - If $\llcorner\lrcorner \in V_K$ has fixed arity n, then $\llcorner D_1, \ldots, D_n \lrcorner \in \mathfrak{D}$ for all $D_i \in \mathfrak{D}$ $(1 \leq i \leq n)$, and $\llcorner a_1, \ldots, a_n \lrcorner \in \llcorner D_1, \ldots, D_n \lrcorner$ for all $a_1, \ldots, a_n \in D$.
- and an interpretation of function symbols in V_K over $\mathcal{U}$.

That is, given the base set of a structure $\mathcal{S}$, we can add the required Booleans and $\perp$, partition it into base domains $\mathcal{D}$, then apply the construction in Definition 7 to obtain a much larger base set, and interpret functions symbols with respect to this enlarged base set.

Definition 8 background postulate. Each state of a database transformation t must contain

- an infinite set of reserve values,
- truth values and their connectives, the equality predicate, the undefinedness value $\perp$, and
- a background class $\mathcal{K}$ defined by a background signature V_K that contains at least a binary tuple constructor $(\cdot)$, a multiset constructor $\langle\cdot\rangle$, and function symbols for operations on pairs such as pairing and projection, and on multisets such as empty multiset $\langle\rangle$, singleton $\langle x\rangle$, and multiset union $\uplus$.

The exploration boundary postulate for sequential algorithms requests that only finitely many terms can be updated in an elementary step [8]. For parallel algorithms this postulate becomes significantly more complicated, as basic constituents not involving any parallelism (so-called "proclets") have to be considered [4].

For database transformations the problem lies somehow in between. Computations are intrinsically parallel, even though implementations may be sequential, but the parallelism is restricted in the sense that all branches execute de facto the same computation. We will capture this by means of location operators, which generalise aggregation functions and cumulative updates. Furthermore, depending on the data model used and thus on the actual background signature we may use complex tree-structured values. As a consequence we have to cope with the problem of partial updates [9], e.g. the synchronisation of updates to different parts of the same tree values.

Definition 9. Let $\mathcal{M}(D)$ be the set of all non-empty multisets over a domain D, then a *location operator* ρ over $\mathcal{M}(D)$ consists of a unary function $\alpha : D \to D$, a commutative and associative binary operation $\odot$ over D, and a unary function $\beta : D \to D$, which define $\rho(m) = \beta(\alpha(b_1) \odot \cdots \odot \alpha(b_n))$ for $m = \langle b_1, ..., b_n\rangle \in \mathcal{M}(D)$.

The definitions of updates, update sets and update multisets are the ame as for ASMs [5].

Definition 10. For a database transformation t let S be a state of t, f a dynamic function symbol of arity n in the state signature of t, and $a_1, ..., a_n, v$ be elements in the base set of S, then an *update* of t is a pair (l, v), where l is a location $f(a_1, ..., a_n)$. An *update set* is a set of updates; an *update multiset* is a multiset of updates.

Using a location function that assigns a location operator or $\perp$ to each location, an update multiset can be reduced to an update set. It is further possible to construct for each $(S, S') \in \tau_t$ a minimal update set $\Delta(t, S, S')$ such that applying this update set to the state S will produce the state S'. Then $\Delta(t, S)$ denotes the set of all such update sets for t in state S, i.e. $\Delta(t, S) = \{\Delta(t, S, S') \mid (S, S') \in \tau_t\}$. The problem

of partial updates is then subsumed by the problem of providing consistent update sets, in which there cannot be pairs (l, v_1) and (l, v_2) with $v_1 \neq v_2$ – details are discussed in [17].

In order to derive at an exploration boundary for database transformation we have to be aware of the fact that for databases only logical properties are relevant – this is the so-called "genericity principle" in database theory – and therefore, it must must be possible to use non-ground terms to access elements of a state. Elements matching a ground term in a state S are then accessible in parallel.

Definition 11. Let S be a state of the database transformation t. Then elements $a_1, \ldots, a_n$ in the base set of S are *accessible in parallel* iff there exists a term α with values $a_1, \ldots, a_n$ in S.

Then there will be a maximum number m of elements that are accessible in parallel. Furthermore, there is always a minimum number n such that n variables are sufficient to describe the updates of a database transformation. Taking these together we obtain our fourth postulate.

Definition 12 exploration boundary postulate. For a database transformation t there exists a fixed, finite set T of terms of t such that $\Delta(t, S_1) = \Delta(t, S_2)$ holds whenever the states S_1 and S_2 coincide over T, and for each state S of t there exist $n, m \in \mathbb{N}$ such that the upper boundary of exploration is $\mathcal{O}(m^n)$, where m depends on S.

The last postulate addresses genericity. For queries genericity means that queries should preserve isomorphisms. In order to capture also queries that use constants this genericity request has to be relaxed to the preservation of Z-isomorphisms, where Z contains all constants appearing in the query. In generalising this request to general database transformations we concentrate on equivalent substructures in the following sense, and leave the generalisation to Z-isomorphisms to elsewhere.

Definition 13. A structure S' is a *substructure* of the structure S (notation: $S' \preceq S$) iff the base set B' of S' is a subset of the base set B of S, and for each function symbol f of arity n in the signature Σ the restriction of f_S to B' results in $f_{S'}$. Substructures $S_1, S_2 \preceq S$ are *equivalent* (notation: $S_1 \equiv S_2$) iff interchanging them gives rise to an automorphism of S.

This allows us to formulate our genericity postulate, which requires that whenever a substructure is preserved by a one-step transition, then all equivalent substructures will appear as substructure in one of the states reachable by the one-step transition. This postulate puts a severe restriction on the non-determinism in the transition relation τ_t.

Definition 14 genericity postulate. Let X be a substructure of state $S \in \mathcal{S}_t$ with $X \preceq S'$ for $(S, S') \in \tau_t$. Then for each $Y \preceq S$ with $X \equiv Y$ the isomorphism $\sigma : X \to Y$ extends to an isomorphism $\sigma' : S' \to \sigma'(S')$ with $(S, \sigma'(S')) \in \tau_t$ and σ' being the identity on every substructure Z that is not equivalent to X.

2.4 Examples

Let us now look at examples for ASSs. We will concentrate on functions, which are quite often taken as web services, Data Warehouses and Web Information Systems.

Functional Web Services. Suppose we have a database with employee information, in particular salaries. Individual salaries will be kept hidden, but building averages for groups for employees will be offered as a service. In this case, we could have a quaternary relation with employee_id, name, department and salary in the database schema. Using ASMs [5] we would model this by a controlled 4-ary function employee. Then employee(43,Lisa,Cheese,4100)=1 means that there is an employee with id 43, name Lisa, and salary 4100 in the Cheese department, while employee(552,Bernd,Milk,8000)=0 means that in the Milk department there is no employee named Bernd with id 552 and salary 8000.

The averaging operation would be made available in combination with an empty view. We would allow either a grouping by department or no grouping at all. The result would leave the database as it is but display a new view with the resulting relation and the same operation associated to it. Using ASM notation we could define the averaging operation by department simply by the rule

$$\begin{aligned}\text{result} :=& \{(d,a) \mid \exists i,n,s.\,\text{employee}(i,n,d,s) = 1 \wedge \\ & a = \text{avg}\langle s \mid \exists i,n,s.\,\text{employee}(i,n,d,s) = 1\rangle\}\end{aligned}$$

Data Warehouses. A more interesting example of ASSs is given by data warehouses, which could be turned this way into web warehouses. The ASM-based approach in [19] used three linked ASMs to model data warehouse and OLAP applications. At its core we have an ASM modelling the data warehouse itself using star or snowflake schemata. For instance, here we could have controlled functions sales, product, and store all of arity 3, and a static ternary function time. As before, sales(003,14,27-2-2008)=1 represents the fact that product 003 was sold in store 14 on 27 February 2008, product(003,hammer,27.5)=1 means that the product with id 003 is a hammer, which is sold at a price of 27.5, store(14,Awapuni,Palmerston)=1 means that the store with id 14 is located in Awapuni in the city of Palmerston, and time(27,2,2008)=1 indicates that 27 February 2008 is a valid time point.

A second ASM would be used for modelling operational databases with rules extracting data from them and refreshing the data warehouse. This ASM is of no further relevance for us and thus will be ignored.

The third ASM models the OLAP interface on the basis of the idea that datamarts can be represented as extended views. Such a view may e.g. extract all sales in store 14 in 2008 together with product description and price. That is, the view defining database transformation could be described (using relational algebra operations liberally) by the simple rule

$$\begin{aligned}\text{result} :=& \pi_{\text{p-id,description,price,day(date),month(date)}} \\ & (\sigma_{\text{s-id}=14 \wedge \text{year(date)}=2008}(\text{sales}) \bowtie \text{product})\end{aligned}$$

Service operations associated with such a view could be roll-up and drill-down operations, e.g. aggregating sales per day or month, or slicing operations, e.g. concentrating on sales of a particular product. Furthermore, we could permit operations for changing the selected year or store. We omit further details.

Furthermore, the main rule in the OLAP ASM in [19] mainly serves the purpose of opening and closing datamarts and selecting operations associated with them. This has been already captured by the notion of a run.

Web Information Systems. Another even more complex example is given by Web Information Systems (WISs), following the modelling approach in [14], which among others provides the notion of media type. At its core, a media type is a view on a database schema that is extended by operations (and more), which is exactly what we capture with ASSs.

However, in this case the view-defining queries must be able to create the link structure between instances of media types, the so-called media objects, which implies that the creation of identifiers is a desirable property in such queries. As already stated the non-determinism in database transformation is motivated by such identifier creation. In this sense WISs provide an example for the necessity of non-determinisn in the small-step transition relations in Definition 1.

Chapter 3

ASS Integration

The integration of ASSs aims at replacing two given ASSs by a new one that supports the functionality of both original ASSs. As databases, data warehouses and WISs appear as examples of ASSs, the approach to view integration in [11] should be promising for a generalisation to ASSs.

Therefore, let us start with an integration of database systems DBS^1 and DBS^2. We use superscripts for the wide-step and small-step transition relations as well as for the transactions when referring to these two systems. For integration we need another set $\mathcal{S}\downarrow$ of states together with projection functions $p : \mathcal{S}^1 \to \mathcal{S}\downarrow$ and $q : \mathcal{S}^2 \to \mathcal{S}\downarrow$. In view of the abstract state postulate, p and q should be invariant under isomorphisms, i.e. isomorphic states are to be mapped to isomorphic states.

Speaking less formally, the projection functions p and q model the components of states $S^1 \in \mathcal{S}^1$ and $S^2 \in \mathcal{S}^2$ that should be identified. Then state integration requires the existence of a set $\mathcal{S}\uparrow$ of integrated states together with projection functions $\bar{p} : \mathcal{S}\uparrow \to \mathcal{S}^1$ and $\bar{q} : \mathcal{S}\uparrow \to \mathcal{S}^2$ that are "universal" in the following sense (in fact, this is a "pullback" definition):

- The diagram defined by p, q, $\bar{p}$ and $\bar{q}$ permutes, i.e. $p \circ \bar{p} = q \circ \bar{q}$ holds, and
- for any other set of states $\mathcal{S}$ together with projections $p' : \mathcal{S} \to \mathcal{S}^1$ and $q' : \mathcal{S} \to \mathcal{S}^2$ that satisfy $p \circ p' = q \circ q'$ there exists a unique function $r : \mathcal{S} \to \mathcal{S}\uparrow$ with $\bar{p} \circ r = p'$ and $\bar{q} \circ r = q'$.

Example 1. Let states in $\mathcal{S}^1$ be defined by relations over a relation schema R^1 with attributes A, B, C, D, and let states in $\mathcal{S}^2$ be defined by relations over a relation schema R^2 with attributes C, D, E, F. Then the projections p and q can be simply defined by $\pi_{C,D}$ in both cases, i.e. we project onto the common attributes. Then $\mathcal{S}\uparrow$ obviously consists of joint relations $r_1 \bowtie r_2$ with $r_1 \in \mathcal{S}^1$ and $r_2 \in \mathcal{S}^2$, i.e. relations over a relation schema with attributes A, B, C, D, E, F. The functions $\bar{p}$ and $\bar{q}$ are obviously the projections π_{R^1} and π_{R^2}, respectively.

The interesting effect of this pullback definition is that it carries over to the transactions and the transition relations that are used in Definition 1. Say, if S_1 is the start state of a transaction $t \in \mathcal{T}^1$, then we have a transition $(S_1, S_1') \in \tau^1$ that is defined by a run of t. Let $S \in \mathcal{S}\uparrow$ be a state that results from integrating S_1 with

$S_2 \in \mathcal{S}^2$, then the corresponding function $\bar{p}$ maps S onto S_1. Similarly, there is a state $S' \in \mathcal{S}\uparrow$ that results from integrating S'_1 with S_2. The pair (S, S') is then the natural extension of (S_1, S'_1) to a transition on states $\mathcal{S}\uparrow$.

The definition also allows us to preserve views. For this let v be a view on $\mathcal{S}^1$, which transforms a state S^1_d into a state $S^1_d \cup V^1$. If S_d is a state after integrating S^1_d with some state S^2_d originating from DBS^2, then $S_d \cup V^1$ results from integrating $S^1_d \cup V^1$ with $S^2_d \cup V^1$, and thus v extends to a view that transforms S_d into $S_d \cup V^1$.

We are, however, not only interested in preserving views, but in integrating them as well, which can be approached in the same way. For views v^1 on DBS^1 transforming S^1 into $S^1 \cup V^1$ and v^2 on DBS^2 transforming S^2 into $S^2 \cup V^2$ we can first integrate S^1 and S^2 into the integrated state S. As explained this turns v^1 and v^2 both into views over S. We can then separately integrate V^1 and V^2 into V, which means that we can replace v^1 and v^2 by an integrated view that will transform S into $S \cup V$.

Example 2. The result of integrating relations over $R^1 = \{\text{id}, \text{name}, \text{dept}, \text{salary}\}$ and $R^2 = \{\text{id}, \text{first}, \text{last}, \text{height}\}$ gives relations over schema $R = \{\text{id}, \text{first}, \text{last}, \text{height}, \text{dept}, \text{salary}\}$. Suppose a view v^1 produces relations with $\{\text{dept}, \text{avg\_salary}\}$, while a view v^2 produces relations over $\{\text{max\_height}\}$. The integration of these views results in a view v that produces relations over $\{\text{dept}, \text{avg\_salary}, \text{max\_height}\}$ with constant value for the max_height attribute.

Finally, operations associated with a view carry over to the views after integration, as they merely induce a transaction and a change to the active views.

Chapter 4

ASS Composition

While ASS integration replaces given ASSs by new ones preserving their functionality, the composition of ASSs does not aim at replacing any existing ASS. Instead the goal is to define new services that exploit functionality of existing ones. That is, we will have to extract components from existing ASSs and recompose these components. A simple form of recomposition can be component integration as discussed in the previous section – the extracted components will be ASSs as well. However, we may also exploit other mechanisms of component composition, e.g. those discussed in [15].

In order to extract components from an ASS we first build a subset $\mathcal{V}' \subseteq \mathcal{V}$ of the set of views, and for each view $v \in \mathcal{V}'$ we restrict the service operations to a subset $\mathcal{O}'_v \subseteq \mathcal{O}_v$. These subset restrictions obviously produce an ASS with the same underlying database system as before.

In a second step we actually restrict the views $v \in \mathcal{V}'$ themselves by defining a view p_v on top of it, i.e. p_v is a database transformation that will transform a state V into a state $V \cup V'$. Practically speaking, service extraction can only be performed by service users, and they only have access to the view layer, not to the underlying database system. Nevertheless, by forgetting the original view V, the composed database transformation $p_v \circ v$ defines a view on top of the original database system transforming states $S \in \mathcal{S}$ into states $S \cup V'$. Furthermore, $o \in \mathcal{O}'_v$ still induces the same transaction, and if v would be replaced by $\{v_1, \ldots, v_k\}$, then $p_v \circ v$ would have to be replaced by $\{p_{v_i} \circ v_i \mid i \in \{1, \ldots, k\}, v_i \in \mathcal{V}'\}$. In this way, the collection of views p_v defines an ASS with the same underlying database system as before. We will call this an ASS component.

Definition 1. Let $\mathcal{A} = (DBS, \mathcal{V}) = (\mathcal{S}, \tau, \{\tau_t\}_{t \in \mathcal{T}}, \{(v, \{o_1, \ldots, o_{n_v}\})\}_{v \in \mathcal{V}})$ be an ASS. A *component* of $\mathcal{A}$ is an ASS $(\mathcal{S}, \tau, \{\tau_t\}_{t \in \mathcal{T}}, \{(p_v \circ v, \{o'_1, \ldots, o'_{n'_v}\})\}_{v \in \mathcal{V}'})$ with $\mathcal{V}' \subseteq \mathcal{V}$ and $\{o'_1, \ldots, o'_{n'_v}\} \subseteq \{o_1, \ldots, o_{n_v}\}$.

After extracting components from several ASSs, their integration along the lines discussed in the previous section is one way of recomposing them. Another one is parallel composition.

Definition 2. Let $\mathcal{A}^i = (\mathcal{S}^i, \tau^i, \{\tau_t\}_{t \in \mathcal{T}^i}, \{(v, \{o_1, \ldots, o_{n_v^i}\})\}_{v \in \mathcal{V}^i})$ $(i = 1, \ldots, n)$ be ASSs. Their *parallel composition* $\mathcal{A}^1 \oplus \cdots \oplus \mathcal{A}^n$ is an ASS that is defined as follows:

- The set of states is the sum $\mathcal{S} = \{S_1 \cup \cdots \cup S_n \mid S_i \in \mathcal{S}^i\}$.
- The wide-step transition relation τ is defined by parallel cmposition, i.e. $(S_1 \cup \cdots \cup S_n, S'_1 \cup \cdots \cup S'_n) \in \tau$ iff $(S_i, S'_i) \in \tau^i$ for all $i = 1, \ldots, n$.
- The set of transactions is the product $\mathcal{T} = \{t_1 \| \ldots \| t_n \mid t_i \in \mathcal{T}^i\}$.
- Small step transition relations are defined by parallel composition, i.e. $(S_1 \cup \cdots \cup S_n, S'_1 \cup \cdots \cup S'_n) \in \tau_{t_1 \| \ldots \| t_n}$ iff $(S_i, S'_i) \in \tau_{t_i}$ for all $i = 1, \ldots, n$.
- The set of views is also defined as a product $\mathcal{V} = \{v_1 \| \ldots \| v_n \mid v_i \in \mathcal{V}^i\}$.
- The sets of service operations are defined by parallel composition $\mathcal{O}_{v_1 \| \ldots \| v_n} = \{o_1 \| \ldots \| o_n \mid o_i \in \mathcal{O}_{v_i}\}$.

The obvious drawback of both component integration and parallel composition is that in both cases we still make merely the service operations of the original ASSs available. In order to obtain new service operations by composition of extracted ones we follow the approach in [15] to distinguish between retrieval and update operations.

Definition 3. A service operation $o \in \mathcal{O}_v$ in a component of an ASS $\mathcal{A}$ is a *retrieval operation* iff the induced transaction on the database system underlying $\mathcal{A}$ is the identity, otherwise it is an *update operation*.

A retrieval operation does only affect the views that are open or closed, but it may nevertheless affect the presentation, from which in our definition abstracts. An update operation on the other hand may change the underlying database state. The view v_1 the update operation $o_1 \in \mathcal{O}_{v_1}$ is associated with provides data that affect the service operation. Therefore, if o_1 opens another view v_2, we may compose any update operation $o_2 \in \mathcal{O}_{v_2}$ to define a new update operation $o_2 \circ o_1$. We can use this to define the one-sided and double-sided composition of views as follows.

Definition 4. Let $\mathcal{A}$ be an ASS.

- Let $v_1 \in \mathcal{V}$ be a view on $\mathcal{A}$ with service operations $\mathcal{O}_{v_1}$ that are decomposed into the sets $\mathcal{O}^r_{v_1}$ and $\mathcal{O}^u_{v_1}$ of retrieval and update operations, respectively. Let $v_2 \in \mathcal{V}$ be another view on $\mathcal{A}$ with service operations $\mathcal{O}_{v_2}$, and let $\mathcal{O}^1_{v_2} \subseteq \mathcal{O}_{v_2}$ denote the set of service operations that open v_1. Then the *one-sided composition* $v_1 \ltimes v_2$ is the view with the database transformation v_1 and the associated set of service operations $\mathcal{O}^r_{v_1} \cup \{o_1 \circ o_2 \mid o_1 \in \mathcal{O}^u_{v_1}, o_2 \in \mathcal{O}^1_{v_2}\}$.
- Let $v_1, v_2 \in \mathcal{V}$ be views on $\mathcal{A}$ with service operations $\mathcal{O}_{v_i}$ that are decomposed into the sets $\mathcal{O}^r_{v_i}$ and $\mathcal{O}^u_{v_i}$ of retrieval and update operations, respectively ($i = 1, 2$). Furthermore, let $\mathcal{O}^1_{v_2} \subseteq \mathcal{O}_{v_2}$ denote the set of service operations that open v_1, and $\mathcal{O}^2_{v_1} \subseteq \mathcal{O}_{v_1}$ denote the set of service operations that open v_2. Then the *double-sided composition* $v_1 \bowtie v_2$ is the view with the database transformation $v_1 \| v_2$ and the associated set of service operations

$$\begin{aligned}\mathcal{O}_{v_1 \bowtie v_2} = {} & \mathcal{O}^r_{v_1} \cup \mathcal{O}^r_{v_2} \cup \\ & \{o_1 \circ o_2 \mid o_1 \in \mathcal{O}^u_{v_1}, o_2 \in \mathcal{O}^1_{v_2}\} \cup \{o_2 \circ o_1 \mid o_2 \in \mathcal{O}^u_{v_2}, o_1 \in \mathcal{O}^2_{v_1}\}\end{aligned}$$

If we combine parallel composition of ASSs with double-sided composition of views, i.e. instead of taking $\mathcal{O}_{v_1 \| v_2}$ as in Definition 2 we take the double-sided composition $\mathcal{O}_{v_1 \bowtie v_2}$, we obtain parallel composition with feedback in analogy to the definition in [15].

Definition 5. Let $\mathcal{A}^i = (\mathcal{S}^i, \tau^i, \{\tau_t\}_{t\in\mathcal{T}^i}, \{(v, \{o_1, \ldots, o_{n_v^i}\})\}_{v\in\mathcal{V}^i})$ $(i = 1, 2)$ be ASSs. Their *parallel composition with feedback* $\mathcal{A}^1 \bowtie \mathcal{A}^2$ is an ASS that is defined as follows:

- The set of states is the sum $\mathcal{S} = \{S_1 \cup S_2 \mid S_i \in \mathcal{S}^i\}$.
- The wide-step transition relation τ is defined by parallel cmposition, i.e. $(S_1 \cup S_2, S_1' \cup S_2') \in \tau$ iff $(S_i, S_i') \in \tau^i$ for $i = 1, 2$.
- The set of transactions is the product $\mathcal{T} = \{t_1 \| t_2 \mid t_i \in \mathcal{T}^i\}$.
- Small step transition relations are defined by parallel composition, i.e. $(S_1 \cup S_2, S_1' \cup S_2') \in \tau_{t_1\|t_2}$ iff $(S_i, S_i') \in \tau_{t_i}$ for $i = 1, 2$.
- The set of views is also defined as a product $\mathcal{V} = \{v_1 \bowtie v_2 \mid v_i \in \mathcal{V}^i\}$.
- The sets of service operations are defined by doubled-sided composition $\mathcal{O}_{v_1\bowtie v_2}$ as in Definition 4.

Chapter 5

ASS Personalisation

With the concept of ASSs we provide a mechanism to export data and services that can be used by others within a more or less open community. The ultimate open community would be given by the web. Using the offered ASSs new services can be defined by extracting ASS components and recomposing them in various ways as described in the previous section. At first sight the extraction and composition process is a manual activity: discover available services, decide which components might be relevant, extract them and recombine them as needed. In a sense the resulting new ASSs will be personalised, as the selected components and the used composition method reflect the preferences of the service user. Nevertheless, the question arises how the selection process can be tailored in a way that out of the views and associated operations on offer only those are selected that are relevant for the intended use.

In order to address this problem of personalisation support we concentrate on the selection process as outlined at the beginning of the previous section, i.e. building a subset $\mathcal{V}' \subseteq \mathcal{V}$ of the set of views and restricting the service operations $\mathcal{O}_v$ associated with $v \in \mathcal{V}'$ to a subset $\mathcal{O}'_v$. This reflects the part of the process that is determined by the preferences of the service user, while follow-on steps are more of a technical nature and aim at rearranging the selected views and operations in the best suitable way. These steps of restricting the selected views and operations to define components and composing these components will be left for manual treatment following the selection.

We further concentrate on the service operations treating the views they are associated with as necessary basis. That is, if a service operation $o \in \mathcal{O}_v$ is to be selected, then of course v has to be selected as a view, and if no operation in $\mathcal{O}_v$ is considered to be relevant, there is no need to select v.

To support the automatic or semi-automatic selection of service operations from a given ASS we have to know more about it, in particular, how it is supposed to be used. For this purpose we associate an action scheme or plot with an ASS. Such a plot will be an algebraic expression composed out of the service operations together with Boolean pre- and postconditions that prescribes meaningful sequences of operations – in the case of a WIS this would constitute the possible navigation paths. For technical reasons we will also need operations `skip` and `abort` with the

usual meaning, which we will denote as 1 and 0, respectively.

Definition 1. Let $\mathcal{O}$ denote the set of service operations associated with an ASS $\mathcal{A}$, and let $\mathcal{C}$ be a set of Boolean conditions. The the set $\mathcal{P}$ of *plots* over $\mathcal{O}$ and $\mathcal{C}$ is the smallest set with $\mathcal{O} \cup \mathcal{C} \cup \{0,1\} \subseteq \mathcal{P}$ satisfying the following conditions:

- For $p, q \in \mathcal{P}$ we also have $pq \in \mathcal{P}$, $p + q \in \mathcal{P}$, $p \| q \in \mathcal{P}$ and $p^* \in \mathcal{P}$.
- For $p \in \mathcal{P}$ not involving any operation in $\mathcal{O}$ we also have $\bar{p} \in \mathcal{P}$.

The informal meaning of the operators is the following: pq denotes a sequence (q follows p), $p + q$ denotes a choice between p and q, $p \| q$ denotes parallel execution of p and q, and p^* denotes iteration of p. A Boolean condition is identied with an operation that tests it, so if p and q are Booleans, the pq denotes conjunction and $p + q$ disjunction. Furthermore, $\bar{p}$ denotes negation, and 0 and 1 correspond to false and true, respectively. Finally, as there is no interaction between the operations, $p \| q$ can be considered as a shortcut for $pq + qp$. Then, according to [16] the set $\mathcal{P}$ must satisfy the axioms of Kleene algebras with tests.

Suppose now we are given the plot $p \in \mathcal{P}$ associated with an ASS. Then we can define preference rules by means of equations on $\mathcal{P}$ as follows:

- $\alpha(p + q) = \alpha p$ means that under the condition α, if there is a choice between p and q, then p will be preferred.
- $p(q + r) = pq$ means that after p, if there is a choice between q and r, then q will be preferred.
- $\alpha p^* = \alpha p$ means that under the condition α the preference is to execute p exactly once instead of iterating it arbitrarily often.
- $\bar{\alpha} p = 0$ means that α is a precondition for p.
- $p\bar{\alpha} = 0$ means that α is a postcondition for p.

This list of equations expressing preference rules is not exhaustive. Together with the conditional equations that define the axioms for Kleene algebras with tests we can use the given plot p and a postcondition β that we want to reach (it could simply be 1), and apply the equations as term rewriting rules to turn $p\beta$ into a simpler form, say P'. This approach to rewriting on the basis of Kleene algebras with tests has been handled in detail in [16]. Then p' would define a personalised plot, and the operations in it are the natural choice for the selection.

Chapter 6

Conclusion

In this paper we introduced Abstract State Services (ASSs) as an abstraction of services that can be made available for use by other systems, e.g. via the web. An ASS combines a hidden database layer with an operation-equipped view layer, and can be anything, e.g. a simple function, a data warehouse or a Web Information System. We adopted the approach taken for the proof of the ASM thesis, i.e. we defined ASSs by means of intuitive, abstract postulates, leaving the definition of languages for ASSs to future work. We strongly believe that it will be possible to prove that a version of ASMs would turn out again to represent the kind of languages capturing ASSs.

We then discussed the problems of service integration, composition and personalisation leading to new services defined on top of existing ones. This in principle shows the power of the concept, but will require further elaboration in future research. For instance, concentrating on Web Information Systems as ASSs the approach may contribute to web interoperability, but should be linked more tightly to web application development methods. With respect to data warehouses ASS integration and composition will contribute to web warehousing, but this also has to be investigated in more detail.

Bibliography

1. M. Altenhofen, E. Börger, and J. Lemcke. An abstract model for process mediation. In K.-K. Lau and R. Banach, editors, *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 81–95. Springer-Verlag, 2005.
2. B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing web service protocols. *Data and Knowledge Engineering*, 58(3):327–357, 2006.
3. A. Binemann-Zdanowicz and B. Thalheim. Modeling information services on the basis of ASM semantics. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines*, volume 2589 of *Lecture Notes in Computer Science*, pages 408–410. Springer-Verlag, 2003.
4. A. Blass and J. Gurevich. Abstract state machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 4(4):578–651, 2003.
5. E. Börger and R. Stärk. *Abstract State Machines*. Springer-Verlag, Berlin Heidelberg New York, 2003.
6. J. Gómez, C. Cachero, and O. Pastor. Modelling dynamic personalization in web applications. In *Third International Conference on Web Engineering – ICWE 2003*, volume 2722 of *LNCS*, pages 472–475. Springer-Verlag, 2003.
7. E. Grädel and Y. Gurevich. Metafinite model theory. In *LCC '94: Selected Papers from the International Workshop on Logical and Computational Complexity*, pages 313–366, London, UK, 1995. Springer-Verlag.
8. J. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, 2000.
9. Y. Gurevich and N. Tillmann. Partial updates. *Theoretical Computer Science*, 336(2-3):311–342, 2005.
10. S. J. Hegner. Information-optimal reflections of view updates on relational database schemata. In S. Hartmann and G. Kern-Isberner, editors, *Foundations of Information and Knowledge Systems – 5th International Symposium (FoIKS 2008)*, volume 4932 of *Lecture Notes in Computer Science*, pages 112–131. Springer-Verlag, 2008.
11. H. Ma, K.-D. Schewe, B. Thalheim, and J. Zhao. View integration and cooperation in databases, data warehouses and web information systems. *Journal of Data Semantics*, IV:213–249, 2005.
12. K.-D. Schewe and B. Schewe. Integrating database and dialogue design. *Knowledge and Information Systems*, 2(1):1–32, 2000.

13. K.-D. Schewe and B. Thalheim. The co-design approach to web information systems development. *International Journal on Web Information Systems*, 1(1):5–14, 2005.
14. K.-D. Schewe and B. Thalheim. Conceptual modelling of web information systems. *Data and Knowledge Engineering*, 54(2):147–188, 2005.
15. K.-D. Schewe and B. Thalheim. Component-driven engineering of database applications. In M. Stumptner, S. Hartmann, and Y. Kiyoki, editors, *Conceptual Modelling 2006 – Third Asia-Pacific Conference on Conceptual Modelling (APCCM 2006)*, volume 53 of *CRPIT*, pages 105–114. Australian Computer Society, 2006.
16. K.-D. Schewe and B. Thalheim. Personalisation of web information systems - a term rewriting approach. *Data and Knowledge Engineering*, 62(1):101–117, 2007.
17. Q. Wang and K.-D. Schewe. A tailored ASM thesis for database transformations. submitted for publication.
18. Q. Wang and K.-D. Schewe. Axiomatization of database transformations. In *Proceedings of the 14th International ASM Workshop (ASM 2007)*, University of Agder, Norway, 2007.
19. J. Zhao and H. Ma. ASM-based design of data warehouses and on-line analytical processing systems. *Journal of Systems and Software*, 79(5):613–629, 2006.