Abstract State Services
A Theory of Web Services

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

[1] Victoria University of Wellington, School of Mathematics, Statistics and Computer Science, Wellington, New Zealand
hui.ma@mcs.vuw.ac.nz
[2] Information Science Research Centre, Palmerston North, New Zealand
kdschewe@acm.org, isrc@xtra.co.nz
[3] Christian-Albrechts-University Kiel, Institute of Computer Science, Kiel, Germany
thalheim@is.informatik.uni-kiel.de
[4] Massey University, Palmerston North, New Zealand
q.q.wang@massey.ac.nz

Abstract. Abstract State Services (ASSs) have been introduced recently as an abstraction of web services that exploit the fundamental approach of Abstract State Machines. An ASS combines a hidden database layer with an operation-equipped view layer, and can be a simple function, a data warehouse or a full-fledged Web Information System (WIS). In this paper we provide a language for ASSs, and show how ASSs capture all these instantiations of "services".

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 from a simple function to a fully functional Web Information System (WIS). 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 [1]. However, despite the existence of conference series on web services and a lot of buzz around "service-oriented architecture" (SOA) the notion of "service" has surprisingly been defined only vaguely.

In this paper we are particularly interested in an abstract, conceptual approach to service integration and composition that has recently been introduced by means of postulates for Abstract State Services (ASSs) [2]. ASSs adopt the fundamental idea from WISs 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 [3]. The postulates for ASSs follow the line of thought of the ASM thesis [4], for which Blass and Gurevich formalised sequential and parallel algorithms by requiring a small set of intuitive, abstract postulates, and proved that these postulates are captured by (sequential) Abstract State Machines (ASMs) [5].

We show that ASSs capture services as diverse as simple functions, data warehouses and WISs. In particular, our focus is on describing WISs in general as particular ASSs. For this we pick up the concept of Abstract Database Transformation Machines (ADTMs), a modification of ASMs that captures the postulates for database transformations [6], and enhance ADTMs by integrating them with a declarative query language that is based on fixed-point construction and identifier creation.

Regarding WISs as ASSs allows us to benefit from the approach to ASS integration and composition that was developed in [2]. In this way we enable the construction of WISs by means of plugging in components from various existing WISs, i.e. we achieve WIS interoperability. Furthermore, as the concept of ASS is not bound to WISs, we may even be able to integrate WISs with any kind of web service. In particular, the close relationship between ADTMs and ASMs permits the easy integration of web services that have been specified by means of ASMs [7].

The remainder of this paper is organised as follows. In Section 2 we briefly introduce the model of Abstract State Services without detailed reference to the postulates for database transformations. We discuss how functional web services, data warehouses and WISs can be seen as a special case of ASSs. In Section 3 we develop an abstract language for ASSs based on the results in [6] and the media type concept for WISs from [3]. We demonstrate the language by a concise WIS example. We conclude with a brief summary and discussion of future research.

2 Abstract State Services

Following the general approach of Abstract State Machines [4] 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. 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 [6].

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 $\mathcal{S}$.

A *run* of a database system DBS is an infinite sequence $S_0, S_1, \dots$ 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 $S_i = S_i^0, \dots, S_i^k = S_{i+1}$ such that $(S_i^j, S_i^{j+1}) \in \tau_{t_i}$ holds for all $j = 0, \dots, k-1$.

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 $S_i = S_0^v, \dots, S_\ell^v$ 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 $S_d \cup V_1 \cup \dots \cup V_k$ 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).

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

A formalisation of database transformations by means of postulates is beyond the scope of this paper and excluded due to space limitations. In a nutshell, the postulates require a one-step transition relation between states (sequential time postulate), states as (meta-finite) first-order structures (abstract state postulate), necessary background for database computations such as complex value constructors (background postulate), limitations to the number of accessed terms in each step (exploration boundary postulate), and the preservation of equivalent substructures in one successor state (genericity postulate) [2]. In [6] the language of Abstract Database Transformation Machines (ADTMs) has been developed, which captures all database transformations.

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

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

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 [8] 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

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

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 [8] 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 [3], 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.

3 A Language for ASSs

In this section we develop a formal language for ASSs. Different from our work in [6] our emphasis is not so much completeness, but simply providing a language that is handy for capturing a wide class of WISs as ASSs.

ADTMs. As the definition of ASSs is mainly built on top of database transformations, we adopt the language of ADTMs from [6], which captures all database transformations. In the following we use the notation *atom(p)* referring to all atomic elements appeared in terms of a program p.

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.

An update set $\mathcal{U} = \{(l_1, v_1), ..., (l_n, v_n)\}$ is *consistent* iff the locations of updates in $\mathcal{U}$ are distinct. With respect to a given state S, the execution of a non-trivial update set $\mathcal{U}$ over S is denoted as $S \oplus \mathcal{U}$ such that $S \oplus \mathcal{U} = S'$ where $l_i^{S'} = v_i$ $(i = 1, ..., n)$ iff $\mathcal{U}$ is consistent, otherwise it is undefined.

An *Abstract Database Transformation Machine* (ADTM) Λ has a fixed vocabulary H over a finite set of base domains, consisting of a set S_Λ of states of vocabulary H, a set $I_\Lambda \subseteq S_\Lambda$ of initial states and a set $F_\Lambda \subseteq S_\Lambda$ of final states, states in I_Λ, F_Λ and S_Λ are closed under Z-isomorphisms for $Z = atom(P_\Lambda)$ and $Z \subset H$, a program P_Λ of vocabulary H, and a relation π_Λ over S_Λ is determined by P_Λ such that $\{S_{i+1} | (S_i, S_{i+1}) \in \pi_\Lambda\} = \{S_i \oplus u | u \in \mathcal{U}(P_\Lambda, S_i)\}$.

The vocabulary of an Abstract Database Transformation Machine is constituted by a state vocabulary and a background vocabulary. If Λ is an Abstract Database Transformation Machine, Z is a finite set of elements from the base domains of Λ. Then taking sets $\mathcal{X}$ and $\mathcal{F}$ of dynamic nullary function names (i.e. variables) and a set of non-nullary function names in the vocabulary of Λ, respectively, then a set $\mathcal{T}$ of terms of Λ is defined as the smallest set satisfying $t \in \mathcal{T}$ for all $t \in U$ with $atom(t) \subseteq Z$, $\mathcal{X} \subseteq \mathcal{T}$, and $f(t_1, ...t_n) \in \mathcal{T}$, where $f \in \mathcal{F}$ and $t_i \in \mathcal{T}$ $(i \in [1, n])$.

We still have to define the programs P_Λ used in the definition above. For this we will need location operators. For this 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 $m = \langle b_1, ..., b_n \rangle \in \mathcal{M}(D)$.

Using a location function that assigns a location operator or $\perp$ to each location, an update multiset can be reduced to an update set. For convenience, we use $\varDelta^{\varUpsilon}(r)$ and $\ddot{\varDelta}^{\varUpsilon}(r)$ to denote an update set and an update multiset produced by executing a rule $r \in \mathcal{R}$ on structure $\varUpsilon$, respectively. A term t is a Boolean term iff $valueSet(t, \varUpsilon) = \{true\}$ or $valueSet(t, \varUpsilon) = \{false\}$.

For an Abstract Database Transformation Machine $\varLambda$, let $\varUpsilon$ be a state of $\varLambda$, $\mathcal{T}$ and $\mathcal{X}$ be the sets of terms and variables of $\varLambda$, respectively, then $P_{\varLambda}$ is inductively defined by a set $\mathcal{R}$ of rules.

Update Rule: $f(t_1, ...t_n) := t_0$, where $f(t_1, ...t_n)$ and t_0 are terms in $\mathcal{T}$, with update set $\varDelta^{\varUpsilon}(f(t_1, ...t_n) := t_0) = \{(f(a_1, ...a_n), a_0)\}$, where $a_i \in valueSet$ $(t_i, \varUpsilon)$ for $i \in [0, n]$.

Conditional Rule: if t **then** r **endif**, where t is a Boolean term and $r \in \mathcal{R}$. The update set is $\varDelta^{\varUpsilon}($**if** t **then** r **endif**$) = \varDelta^{\varUpsilon}(r)$ if $valueSet(t, \varUpsilon) = \{true\}$, otherwise $\varDelta^{\varUpsilon}($**if** t **then** r **endif**$) = \emptyset$.

Forall Rule: forall x **with** $\varphi(x)$ **do** r **enddo**, where $x \in \mathcal{X}$, $\varphi(x)$ is a Boolean term containing x and $r \in \mathcal{R}$. The rule first produces the update multiset $\ddot{\varDelta}^{\varUpsilon}($**forall** x **with** $\varphi(x)$ **do** r **enddo**$) = \biguplus_{i \in [1,n]} \ddot{\varDelta}^{\varUpsilon}([a_i/x]r)$ where $\{x \mid valset(\varphi(x), \varUpsilon) = \{true\}\} = \{a_1, ..., a_n\}$ and $[a_i/x]r$ means that variable x is bounded to value a_i within the rule r. Then this update multiset is reduced to the update set $\varDelta^{\varUpsilon}($**forall** x **with** $\varphi(x)$ **do** r **enddo**$)$.

Choose Rule: choose x **with** $\varphi(x)$ **do** r **enddo**, where $x \in \mathcal{X}$, $\varphi(x)$ is a Boolean term containing x and $r \in \mathcal{R}$. $\varDelta^{\varUpsilon}($**choose** x **with** $\varphi(x)$ **do** r **enddo**$)$ $= \varDelta^{\varUpsilon}([a/x]r)$, where $a \in \{x|valueSet(\varphi(x), \varUpsilon) = \{true\}\}$ and $[a/x]r$ means that variable x is bounded to value a within the rule r.

Parallel Rule: $r_1 \| r_2$ with $r_1, r_2 \in \mathcal{R}$ and the update set $\varDelta^{\varUpsilon}(r_1 \| r_2) = \varDelta^{\varUpsilon}(r_1) \cup$ $\varDelta^{\varUpsilon}(r_2)$.

Sequence Rule: $r_1; r_2$ with $r_1, r_2 \in \mathcal{R}$. In this case the update set is $\varDelta^{\varUpsilon}(r_1; r_2)$ $= \varDelta^{\varUpsilon}(r_1) \oslash \varDelta^{\varUpsilon}(r_2)$, which results from first applying $\varDelta^{\varUpsilon}(r_1)$ on structure $\varUpsilon$ and then applying $\varDelta^{\varUpsilon}(r_2)$ on the resulting structure.

Let Rule: let $\theta(t) = \rho$ **in** r, where θ is a location function, ρ a location operator, $t \in \mathcal{T}$ and $r \in \mathcal{R}$. This gives $\varDelta^{\varUpsilon}($**let** $\theta(t) = \rho$ **in** $r) = \varDelta^{\varUpsilon}([\rho/\theta(t)]r)$ with $l \in \{x|x \in valueSet(t, \varUpsilon)\}$, and $[\rho/\theta(t)]r$ means that the location operator associated with l is bounded to ρ within the rule r.

Queries as Database Transformations. Let us now take a look at the fixed-point query language in [3] that has been used for defining views in WISs. In order to adapt this language to the formalism of ASSs we have to make some assumptions regarding the background. One of the domains used in the background should correspond to a set of abstract identifiers, so we can use a base type *ID* to refer to values in this domain. Identifiers in views will be interpreted as abstract surrogates for URIs.

Then we assume the existence of a tuple constructor $(\cdot)$ and a set constructor $\{\cdot\}$. Note that the minimum requirement for a background structure given in [2] already contains the existence of a pair-constructor that would be sufficient for building the tuple constructor on top of it.

Furthermore, let us assume that the database part in each state consists of relations and that identifiers are used for all tuples. These assumptions allow us to define *extended terms* (eterms), and each term will be typed. For each type t we take a countable set of variables V_t. These sets are to be pairwise disjoint. Blurring the distinction between a value in a domain and a constant term that is always interpreted by that value, variables and constants of type t become eterms of that type. In addition, each relation $R \in \mathcal{S}$ is an eterm of type $\{(\text{ident} : ID, \text{value} : t_R)\}$. For each variable $\imath$ of type ID there is a term $\hat{\imath}$ of some type $t(\imath)$. If $\tau_1, \dots, \tau_k$ are eterms of type t, then $\{\tau_1, \dots, \tau_k\}$ is an eterm of the set type $\{t\}$. If $\tau_1, \dots, \tau_k$ are eterms of type $t_1, \dots, t_k$, respectively, then $(a_1 : \tau_1, \dots, a_k : \tau_k)$ is an eterm of the tuple type $(a_1 : t_1, \dots, a_k : t_k)$.

If τ_1, τ_2 are eterms of type $\{t\}$ and t, respectively, then $\tau_1(\tau_2)$ is a positive literal and $\neg\tau_1(\tau_2)$ is a negative literal. If τ_1, τ_2 are eterms of the same type t, then $\tau_1 = \tau_2$ is a positive literal and $\tau_1 \neq \tau_2$ is a negative literal. A *ground fact* is a positive literal without variables.

A *rule* is an expression of the form $L_0 \leftarrow L_1, \dots, L_k$ with a fact L_0 (called the *head* of the rule) and literals $L_1, \dots, L_k$ (called the *body* of the rule), such that each variable in L_0 not appearing in the rule's body is of type ID. A *logic program* is a sequence $P_1; \dots; P_\ell$, in which each P_i is a set of rules.

Finally, a *query* Q on $\mathcal{S}$ is defined by a type t_Q and a logic program $\mathcal{P}_Q$ such that a variable *ans* of type $\{(\text{url} : ID, \text{value} : t_Q)\}$ is used in $\mathcal{P}_Q$. A *Boolean query* can be described as a query Q with type $t_Q = \mathbb{1}$ assuming a trivial type $\mathbb{1}$ in the background with exactly one element in its domain.

A program $P_1; \dots; P_\ell$ is evaluated sequentially. Each set of rules is evaluated by computing an *inflationary fixed point*. That is, start with the set of ground facts given by a database as part of a state S. Whenever variables in the body of a rule can be bound in a way that all resulting ground literals are satisfied, then the head fact is used to add a new ground fact. Whenever variables in the head cannot be bound in a way that they match an existing ground fact, the variables of type ID will be bound to new identifiers.

According to the result in [6] this language can of course be expressed by ADTMs, but due to its declarative nature we prefer to preserve it as a language for expressing the views in WISs. In order to link the views or any other query to ADTMs, which we will need for the transactional database transformations, we want to incorporate query expressions as some form of syntactic sugar into ADTMs.

This can be done by extending the definition of terms in ADTMs by expressions of the form $\langle\!\langle P \rangle\!\rangle(ans)$ with a program P using the answer variable *ans*. Such a term will be interpreted by the value associated with *ans* after executing the logic program.

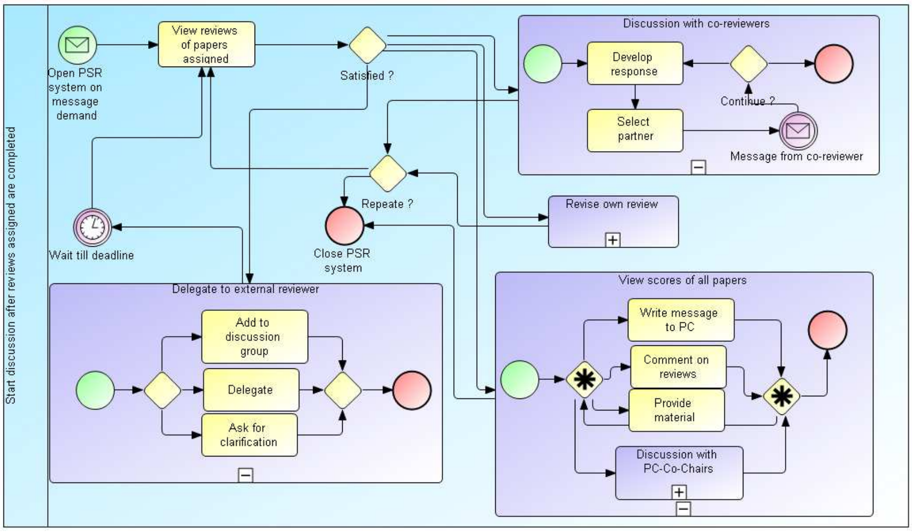

Fig. 1. The start of the discussion process for PC members

Example. Let us take a web-based conference management system (paper submission and reviewing PSR) as a familiar example. Let us further concentrate on the task of PC-members to discuss papers after they have been reviewed, and reviews of other reviewers for the same paper become visible. In this case an appropriate view would present the reviews and comments on a given paper, and the basic operations available to reviewers would be to enter a comment, and to revise the review. In order to do so, it may further be appropriate to obtain an anonymised overview of the ranking of all papers. If access rights permit, a discussant (usually a PC-co-chair, but the role of discussion convenor may also be created) may access all reviews of the participating PC-members to get an idea of their general attitude, i.e. whether they are usually hard or more generous towards authors.

The BPMN (Business Process Modelling Notation) diagram in Figure 1 displays the opportunities of the system for PC members. A PC member starts the subsystem and views first the reviews for the papers he or she reviewed. If the reviews are contradicting and the PC member wants to contribute to the discussion the discussion can either be directly started with all those PC members that have been reviewing at least one of the papers the PC member had or may revise the reviews accordingly or may delegate the discussion to the external reviewers the PC member asked or may first have a look onto all scores of the papers.

We left out the details of revision for own reviews. The diagram does not contain the message flow to other associated processes. The start process is only used for the start of the PC discussion. In a similar form we may model the discussion session of the PC. The diagram may also be enhanced by processes aiming in negotiation among reviewers of a paper and in mediating in the case of conflicts among the PC members.

Furthermore, the system may have a formalised treatment of subreviewers so that in case a review was actually done by a subreviewer, the PC-member has the choice to delegate the discussion to the subreviewer or to involve the subreviewer in the discussion or to only communicate with the subreviewer without letting him participate in the discussion. However, in order to keep things simple we will only add the subreviewer's name to a review. We also ignore the possibility of comments.

For such a system we could model a simple relational database schema. At its core – omitting all parts that are not relevant for the discussion task – we would have the following signature (using functions with arity in parentheses to represent relations): paper(7), member(9), assigned(2), review(17).

All these functions are dynamic and controlled, i.e. they can be updated by the system. The seven components of paper correspond to attributes such as paper_id, title, contact_email, password, abstract, submission_date and accept_code. The components of member correspond to attributes member_id, name, address, email, phone, rights, user_id, password, type. Components of assigned correspond to member_id and paper_id. The 17 attributes for review could be id, member_id, subreviewer, paper_id, submission_date, contribution, positive_aspects, negative_aspects, confidential_remarks, details, confidence, originality, significance, technical_quality, relevance, presentation, recommendation. Paper authors are handled separately.

Using these functions we may have paper(19,"Abstract State Services",kdschewe@acm.org,"dr0w33@p","...",10-04-08,und) = 1 in some state indicating that on 10 April 2008 a paper with title "Abstract State Services" and abstract "..." was submitted. The contact email-address is kdschewe@acm.org, the paper received the id 19, and the chosen password is "dr0w33@p". The acceptance of the paper is undecided.

Now take the query expressed by the following logic program:

$$\begin{aligned}
\mathrm{pap}(p,t,ab,R) &\leftarrow \mathrm{paper}(i_p,(p,t,e,pw,ab,d,c));\\
\hat{R}(i,n,n',c,pos,neg,dc,co,o,s,q,r,pr,or) &\leftarrow \mathrm{pap}(p,t,ab,R),\\
&\quad \mathrm{review}(i_r,(m,n,n',p,d,c,pos,neg,cf,dc,co,o,s,q,r,pr,or)),\\
&\quad \mathrm{member}(i_m,(m,n,a,e,ph,rg,u,pw,ty));\\
\mathrm{ans}(i,(p,t,ab,\hat{R})) &\leftarrow \mathrm{pap}(i,p,t,ab,R).
\end{aligned}$$

This will produce a set of tuples, each of which represents a paper with its paper_id, title, abstract and the set of reviews associated with it. So each tuples represents the data presented to a PC-member when working on the paper discussion task.

An obvious service operation is revise_review, which would replace the review in the database – a simple assignment rule will be sufficient for this. In the view only the modified review would be replaced.

Another service operation is overview, which would create an additional view with an overview of the ranking of papers. We omit the details of the logic program needed for this view.

4 Conclusion

In this paper we used Abstract State Services (ASSs) as a formal abstraction of web services in a very general sense, capturing functional services, data warehouses and WISs. The fact that ASSs are grounded in the theory of Abstract State Machines (ASMs) shows the potential of ASMs to push Conceptual Modelling to new levels [9]. On the other hand, ASSs offer the opportunity to develop a theory for Meme media [10] thereby linking the theory to a very powerful practical instrumentarium for editing, extracting and composing services.

We exploited the language of ADTMs as a very powerful instrument for modelling ASSs, as it captures all database transformations encompassing queries and uopdates, and enriched it with integrated declarative query expressions. In doing so, we opened the pathway for WIS integration and composition along the lines discussed for ASSs in general, which will lead us to web interoperability. Nevertheless, the means by which ASSs can be composed still have not been explored in full detail, thus providing opportunities for further research. Furthermore, applying ASSs for web interoperability in practical case studies is another field that could be fruitfully explored.

References

1. Benatallah, B., Casati, F., Toumani, F.: Representing, Analysing and Managing Web Service Protocols. Data and Knowledge Engineering 58(3), 327–357 (2006)
2. Ma, H., Schewe, K.D., Thalheim, B., Wang, Q.: Composing Personalised Services on top of Abstract State Services (submitted for publication)
3. Schewe, K.-D., Thalheim, B.: Conceptual Modelling of Web Information Systems. Data and Knowledge Engineering 54(2), 147–188 (2005)
4. Gurevich, J.: Sequential Abstract State Machines Capture Sequential Algorithms. ACM Transactions on Computational Logic 1(1), 77–111 (2000)
5. Börger, E., Stärk, R.: Abstract State Machines. Springer, Heidelberg (2003)
6. Wang, Q., Schewe, K.D.: Axiomatization of Database Transformations. In: Prinz, A., Börger, E. (eds.) ASM 2007, University of Agder, Norway (2007)
7. Altenhofen, M., Börger, E., Lemcke, J.: An Abstract Model for Process Mediation. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 81–95. Springer, Heidelberg (2005)
8. Zhao, J., Ma, H.: ASM-based Design of Data Warehouses and On-line Analytical Processing Systems. Journal of Systems and Software 79(5), 613–629 (2006)
9. Börger, E.: Modeling Workflow Patterns from First Principles. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 1–20. Springer, Heidelberg (2007)
10. Tanaka, Y.: Meme Media and Meme Market Architectures. IEEE Press, Wiley-Interscience, USA (2003)