SOCA (2009) 3:263–283
DOI 10.1007/s11761-009-0051-x

ORIGINAL RESEARCH PAPER

A theory of data-intensive software services

Hui Ma · Klaus-Dieter Schewe · Bernhard Thalheim · Qing Wang

Received: 23 February 2009 / Accepted: 26 October 2009 / Published online: 17 November 2009

Abstract We introduce Abstract State Services (AS^2s) as an abstraction of data-intensive services that can be made available for use by other systems, e.g. via the web. An AS^2 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 AS^2s and show that AS^2s capture the fundamentals of approaches such as media types, meme media, SOA and web services excluding presentation issues. Then we show how tailored services can be extracted from available AS^2s, combined with other AS^2 components and personalised to user preferences.

Keywords Abstract state machine · Abstract state service · Software service · Extended view · Meme media · Media type · Service composition · Service personalisation

H. Ma
School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand
e-mail: hui.ma@ecs.vuw.ac.nz

K.-D. Schewe (✉)
Information Science Research Centre, Palmerston North, New Zealand
e-mail: kdschewe@acm.org

B. Thalheim
Department of Computer Science, Christian Albrechts University Kiel, Kiel, Germany
e-mail: thalheim@is.informatik.uni-kiel.de

Q. Wang
PBRF Office, University of Otago, Dunedin, New Zealand
e-mail: qing.wang@otago.ac.nz

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, functionality and sometimes even presentation are made available for use by human users or other services. A prominent example for the view of the web as a pool of resources is the meme media architecture [33], which is based on research that started in the second half of the 1980s, and thus, being older than the world-wide-web is not restricted to the web as the only pool of media resources. The general idea is that media resources are extracted from any accessible source, wrapped and thereby brought into the generic form of a meme media object, and stored in a meme pool, from which they can be retrieved, reedited, recombined and redistributed.

Recently, a lot of research has been investigated into service-oriented architectures (SOA) (see e.g. [12,22]) and service-oriented computing (SOC), which adopt the same principle. A quick search on the DBLP bibliography server reveals that in less than a decade, nearly 500 articles were published with keywords "service-oriented" and "architecture" in the title, not counting the far larger number of papers dealing with the subject, but not having it already in the title. Despite this big interest in the area and the many ideas and systems that have been created, many questions have still not been answered. Among them is the fundamental problem of formally defining the notion of service, as most of the research in the area is mainly based on an informal idea of what constitutes a service.

In an effort to consolidate and integrate current research activities, the Service-Oriented Computing Research

Roadmap [24] has been proposed. Service foundations, service composition, service management and monitoring, and service-oriented engineering have been identified as core SOC research themes. In particular, a specific Web Services Description Language (WSDL) has been proposed by W3C [13], a web services standard has been proposed [4], web service integration has become a highly relevant research topic [7] including service personalisation [17] and transaction processing (e.g. WS-Transaction [14] and WS-Coordination [15]) has been proposed to target loosely coupled, dynamic environments. It also appears quite natural that Abstract State Machines (ASMs) [11] have also been used for modelling web services [3,10].

Our research aims at laying the foundations of a theory of service-oriented systems. In particular, we try to answer the following fundamental questions:

– How must a general model for services look like capturing the basic idea and all facets of possible instantiations, and how can we specify such services?
– How can we search for services that are available on the web?
– How do we extract from such services the components that are useful for the intended application, and how do we recombine them?
– How can we optimise service selection using functional and non-functional (aka "quality of service") criteria?

In this article, we address the first of these problems, and partly the third one. The other questions will be addressed by future research. We take an abstract, conceptual approach to service specification, composition and personalisation with a particular focus on data-intensive services, in which not only functional, but also data resources are made available. We adopt the fundamental idea from the area of dialogue systems [25] 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.

In doing so, we assume that services are data-intensive, which will enable us to combine data services with functional services. The assumption of an underlying database is no real restriction, as it is hidden anyway, and data services will be formalised by views, which in the extreme case could be empty to capture pure functional services.

The data model is of secondary importance for this idea, e.g. Entity-Relationship modelling as in [34] or XML [1,16] could be used for this purpose. This idea has already been mirrored in development methods for Web Information System [28,27], and also appears as a natural choice for component-based systems development [29]. ASMs have also been suggested as a means for modelling such services [8]. On the practical side, the IntelligentPad system [33] provides a realisation of a meme media architecture with a similar underlying idea and in addition tools that also capture presentation issues, similar to presentation options in media types [28].

However, we want to go one step further and formally ensure that indeed all (data-intensive) services are captured. For this, we develop a theory of Abstract State Services (AS^{2}s) following the line of thought of the ASM thesis. Gurevich and Blass [9,19] 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 [32] and customised to database transformations exploiting states as meta-finite logical structures [18]. In analogy to the ASM thesis, it could be proven that a variant of ASMs called Abstract Database Transformation Machines (ADTMs) captures exactly all database transformations, while detail of the used data model become part of the background of the computation. 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 AS^{2}s. In doing so, the web as the medium through which a service may become available is of no importance; the notion of AS^2 can also be applied to enterprise services that are only available to selected clients.

The work in [32] actually consists of two parts. The first one describes intuitive postulates for database transformations and discusses the fundamental differences to Gurevich's seminal work on the ASM thesis. These postulates are repeated in this article, though their motivation has been shortened. The reason is that a foundational theory for service-oriented computing has to explain why the language captures all services (even if the formal mathematical proof is done elsewhere). The second part of the MSCS article contains the mathematical proof that database transformations are captured exactly by a variant of ASMs. As for the variant of ASMs, its core, the ADTM-rules, also appears in Sect. 3 of this article. However, the lengthy proof of the main result in [32] is not repeated here.

We then address the problem of service composition and personalisation. Composition requires the extraction of service components from existing AS^{2}s that feed a new service without replacing the original ones. For this, we can adopt ideas from component composition [29]. For service personalisation according to preference rules, we pick up the idea from WIS personalisation [30] to compose personalised tasks, where the preference rules indicate, which choices will be preferred.

Service composition has to be distinguished from service integration, which means to replace two or more AS^{2}s by a single new one that offers all the functionality of the individual services. This problem can actually be reduced to database schema and view integration. For composition, however, we have no access to the hidden database layer of a service. Nevertheless, service integration may be a valuable tool for service design, which is beyond the scope of this article.

The remainder of this paper is organised as follows. In Sect. 2, we formally introduce the model of Abstract State Services by means of postulates. This involves postulates for database transformations in the spirit of the sequential ASM thesis that were developed in [32]. In Sect. 3, we then present a language for AS^{2}s that is based on the model of Abstract Database Transformation Machines (ADTMs), a variant of ASMs that was shown to capture database transformations in [32]. In view of our intention to capture transactional database systems and extended views on top of them, we extend the ADTM language by expressions that cover complex queries as the fixed-point queries on media types in [28]. A preliminary version of this language was presented briefly in [23].

In Sect. 4, we illustrate the power of Abstract State Services by presenting examples taken from areas as diverse as Web Information Systems, Data Warehouses, Web Services and Intelligent Pads. This is followed by a discussion of component extraction and composition to new AS^{2}s in Sect. 5. All composition operations remain within the framework defined by AS^{2}s. In addition, we address the problem of service personalisation. For this, we show how the extraction process discussed before can be tailored by preference rules. Finally, in Sect. 6, we place our work into the context of the literature on Web Information Systems, Web Services and Meme Media. We show that AS^{2}s provide the formal backbone for many other approaches including media types and Meme media. We conclude with a brief summary and discussion of open research problems.

2 Abstract state services

As we consider data-intensive services, we first look at databases. 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 [19], 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 and log files. 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 atomic 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, as long as this is equivalent to the serial execution, i.e. serialisability is guaranteed. Then each transaction itself is a database transformation and as such 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 [32,35]. We will explain these postulates later in this section.

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 $\mathcal{S}$.

With this definition, we do not yet specify what states are. We do, however, already require that states of a database system are states of database transformations. Later, when we discuss the postulates for database transformations, we will further elaborate the notion of state and database transformation.

For now note that differently 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 mainly meant to capture the creation of values such as identifiers as a highly expressive means in query and update languages. This form of non-determinism is common in database transformations.

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

$S_i = S_i^0, \dots, S_i^k = S_{i+1}$ such that $(S_i^j, S_i^{j+1}) \in \tau_{t_i}$ holds for all $j = 0, \dots, k-1$.

Example 1 Let us consider a flight booking system. At its core it may use a database storing data about flights and bookings. For simplicity, assume that we use a relational database, so we may have a relation FLIGHT with attributes flight_no, departure_date, departure_time, origin and destination for the available flights; a relation SEAT with attributes flight_no, departure_date, class, and number for the available seats per class in a flight; and BOOKING with attributes booking_ref, flight_no, departure_date, class, customer_id for the already made bookings. Let us ignore everything else such as customer data and status of bookings.

Then a state of the DBS would contain an instance of the relational database schema, and a booking transaction would change the state by adding further tuples to the booking relation, provided the number of seats booked for each class of each flight does not exceed the number of available seats. The booking transaction itself proceeds stepwise, and each step also changes the database, i.e. the state.

Furthermore, a booking may be issued by a customer after receiving an answer to a query, e.g. asking for flight itineraries from a specified origin airport to a destination airport within a specified timeframe. The answer to such a query would be a set of itineraries, and each itinerary would be specified by a set of flight tuples stored in the database. Thus, the state, in which the booking transaction is started, should also contain the set of itineraries, which is a view on top of the relational database.

The preceding example is of course very simplified, but it illustrates the definition of a database system. Note that if views are considered as part of states of a DBS, then transactions also affect them. We will handle this in the next subsection.

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 $S_0^v, \dots, S_\ell^v$ of some database transformation v with $S_0^v = S_i$ and $S_i \subseteq S_\ell^v$. Traditionally, we would consider $S_\ell^v - S_i$ as the view. Here, we assume that we can write a state of a database system as a set. For instance, if we deal with a relational database system, then each relation is a set of tuples, which can be written as first-order atoms, and the whole database is the union of these sets of atoms. We will later explain that it is also possible in general to view a state as a set.

We can use this to extend a database system by views. For this, 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 (AS^2).

Definition 3 (*extended view postulate*) An *Abstract State Service* (AS^2) 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 $S_0^j, \dots, S_{n_j}^j$ of v_j $(j = 1, \dots, k)$, starting with $S_0^j = S_d$ and terminating with $S_{n_j}^j = S_d \cup V_j$. 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 AS^2 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. We therefore talk of views that are *open* and those that are *closed*. These service operations and the view generating queries 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 [25] or the web interfaces in [28].

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 AS^{2}s, e.g. using the theory developed by Hegner [21].

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 AS^{2}s.

Example 2 The booking operation in Example 1 is a service operation that is associated with a view that produces a list of itineraries for given search criteria such as origin and destination, preferred class and departure time frame. The induced transaction on the DBS updates the BOOKING relation. Initial states for this database transformation can be any consistent database plus any set of open views. The successor state (for the wide-step transition relation τ) would contain the updated database and the same set of views except the one containing the list of itineraries, which would be replaced by a view that simply contains a confirmation message for the selected and booked itinerary.

2.3 Preliminaries: abstract state machines

The notion of Abstract State Service is based on database transformations. Therefore, we obtain a model that is complete for services, i.e. it captures all services, if it captures all database transformations, in particular all database transformations that preserve the input database, i.e. queries. In the next subsection, we will address a model capturing all database transformations, which is based on Abstract State Machines (ASMs) [11]. Before presenting it, let us first look at ASMs without customising them to databases.

ASMs (aka Evolving Algebras) have been introduced as a means to capture algorithms in an abstract, high-level way. This has led to the sequential and parallel ASM theses [9, 19], respectively, showing that sequential and general ASMs, respectively, capture all sequential and parallel algorithms.[1] Such algorithms have been characterised by a small set of intuitive postulates.

Let us look at the three postulates for sequential algorithms without going too much into technical details:

Sequential time postulate: An algorithm proceeds stepwise by means of a transition function on a set of abstract states, starting from an initial state.

Abstract state postulate: States are first-order structures over a fixed signature and a constant base set, i.e. sets of functions, and are closed under isomorphisms.

Bounded exploration postulate: There is a finite set of closed terms, called *bounded exploration witness* such that whenever two states coincide on it, the update sets of the algorithm in these states are equal.

For parallel algorithms, the first two postulates remain the same, but the third one has to be replaced by several more complicated ones.

The decisive novelty of ASMs was the use of first-order structures for the states, while the other two postulates are intuitively clear. In particular, the bounded exploration postulate assures that in any state an algorithm can only manipulate finitely many locations, where a location is understood as one of the functions in the signature plus arguments for it.

On these grounds, an ASM specification is given by a signature Σ, i.e. a set of function symbols, and a set of rules, one of which is marked as the main rule. Rules can be defined as follows (see [11]):

[1] In Gurevich's theory sequential algorithms still permit bounded parallelism, whereas parallel algorithms are understood to capture even unbounded parallelism.

– If $t_0, \dots, t_n$ are terms over Σ, and f is an n-ary function symbol in Σ, then $f(t_1, \dots, t_n) := t_0$ is a rule in $\mathcal{R}$ called *assignment rule*.
– If φ is a Boolean term and $r' \in \mathcal{R}$ is a rule, then **if** φ **then** r' **endif** is a rule in $\mathcal{R}$ called *conditional rule*.
– If φ is a Boolean term and $r' \in \mathcal{R}$ is a rule, then **forall** $x_1, \dots, x_k$ **with** φ **do** r' **enddo** is a rule in $\mathcal{R}$ called *forall rule*.
– If $r_1, \dots, r_n$ are rules in $\mathcal{R}$, then also $r_1 \| \cdots \| r_n$ is a rule in $\mathcal{R}$ called *parallel rule*.
– If r_1, r_2 are rules in $\mathcal{R}$, then also $r_1 \ ; \ r_2$ is a rule in $\mathcal{R}$ called *sequence rule*.
– If $r'(x_1, \dots, x_k) \in \mathcal{R}$ is a rule using variables $x_1, \dots, x_k$, and $t_1, \dots, t_k$ are terms, then $r'(t_1, \dots, t_k) \in \mathcal{R}$ is a rule called *call rule*.

We obtain sequential ASMs by discarding forall rules or by restricting them to formulae φ that can only be satisfied by finitely many $x_1, \dots, x_k$.

Example 3 For the booking transaction from Example 1, we would have function symbols for the relations FLIGHT, SEAT and BOOKING in the signature Σ—interpreted as Boolean-valued functions, i.e. relations, in every state. Then, we specify the following ASM rule:

```
book(set_of_flights,customer) =
    if $\forall f. f \in$ set_of_flights $\Rightarrow$ not_yet_booked($f$,
      customer) $\wedge$ seat_available($f$)
    then
        forall $f$ with $f \in$ set_of_flights
        do store_booking($f$,customer)
        enddo
    endif
```

Here, not_yet_booked(f,customer) and seat_available(f) are used as shortcuts for more complicated conditions expressed in terms of the relations FLIGHT, SEAT and BOOKING. We omit the details. Likewise, store_booking(f,customer) is a rule that contains an assignment BOOKING(...) := true with arguments derived from the input f and customer.

In our previous work in [32,35], we customised the ASM theses to database transformations in general. We followed the same idea as in Gurevich's seminal work to start with intuitive postulates for database transformations and then to define a variant of ASMs, for which we then proved that they capture exactly all database transformations. The major differences to the postulates[2] are the following:

[2] There are also some minor differences, which merely reflect a different taste, but are technically not relevant.

- The notion of state refers to meta-finite structures to capture the finiteness of databases without being limited to finite structures.
- The background of a computation is made explicit (as in the parallel ASM thesis) in order to capture the necessities of any data model the transformation is based on.
- The background may contain so-called location operation to enable structurally determined bounded parallelism by means of multiset operations.
- The genericity in database transformations is captured by a separate postulate.
- A bounded form of non-determinism is permitted, which together with the genericity amounts to semi-determism.

We will explain the modified postulates for database transformations in detail in the following subsection. In Sect. 3, we then show how the customised variant of ASMs for database transformation, the *Abstract Database Transformation Machines* (ADTMs), which capture all database transformations according to the main result in [32], can be exploited to specify AS^{2}s.

2.4 Database transformations

The definition of database systems and by that also the definition of AS^2 refer to postulates for database transformations that have been elaborated in [32]. 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 AS^{2}s 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 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 $S_0, \dots, S_f$ of states with $S_0 \in \mathcal{I}_t$, $S_f \in \mathcal{F}_t$, $S_i \notin \mathcal{F}_t$ for $0 < i < f$ and $(S_i, S_{i+1}) \in \tau_t$ for all $i = 0, \dots, f-1$.

The abstract state postulate is an adaptation of the corresponding postulate for Abstract State Machines [19], 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* S 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 $f_S : B^k \to B$. An isomorphism σ from structure S to structure S' is defined by a bijection $\sigma : B_S \to B_{S'}$ between the base sets that extends to functions by $\sigma(f_S(b_1, \dots, b_k)) = f_{S'}(\sigma(b_1), \dots, \sigma(b_k))$.

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. We will later in the background postulate formulate minimum requirements for the base set such as containing truth values, a value representing undefinedness and more.

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 σ 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. $\Sigma_t = \Sigma_{db} \cup \Sigma_a \cup \{f_1, \dots, f_\ell\}$. 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.

Example 4 In the booking Example 1, we have to deal with finite relations FLIGHT, SEAT and BOOKING, so for the database part a finite structure would be sufficient. However, in the service operations including the view-definining queries, we may need to permit arithmetic operations such as counting, adding prices, determining the time difference between arrival and departure for which we would require the whole set of natural or real numbers. Thus, these infinite sets of numbers have to become part of the set B_a, and in each view that exploits values from these sets we use surrogate identifiers instead, which can be drawn from the finite set B_{db}, and a bridge function assigning the actual values to the surrogate identifiers.

Another example arises, if we use finite XML trees. In this case, each node in the tree would be represented by an identifier, and the tree structure would be expressed by order relations for SUCCESSOR and SIBLING. Thus, B_{db} would have to contain the set of hereditarily finite trees over a finite set $\mathcal{O}$ of node identifiers. For the actual values associated with the tree nodes, we would provide a bridge function.

The postulates in Definitions 4 and 6 are in line with the sequential ASM thesis [19], 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 computation 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 [9], 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.

The major purpose for the explicit constructors in database transformations is the need to capture the constructs of data models. For instance, in complex value and object-oriented databases, we may require the presence of constructors for records, finite sets, lists, multisets, disjoint unions, arrays, maps, etc. Starting from a set of base domains such as *Integer*, *Date and Bool*, we can apply these constructors and nest them arbitrarily to define complex value domains. In tree-based databases such as XML databases, we may even require a colimit constructor leading to hereditarily finite trees, i.e. the domain of all finite trees with nodes in a given base domain such that all subtrees are also trees in the same domain.

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 $\llcorner a_1, \dots, a_m \lrcorner \in \llcorner D \lrcorner$ for every $m \in \mathbb{N}$ and $a_1, \dots, a_m \in D$.
  - 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 $\llcorner D_1, \dots, D_m \lrcorner \in \mathfrak{D}$ for all $m \leq n$ and $D_i \in \mathfrak{D}$ $(1 \leq i \leq m)$, and $\llcorner a_1, \dots, a_m \lrcorner \in \llcorner D_1, \dots, D_m \lrcorner$ for every $m \in \mathbb{N}$ and $a_1, \dots, a_m \in D$.
  - If $\llcorner \lrcorner \in V_K$ has fixed arity n, then $\llcorner D_1, \dots, D_n \lrcorner \in \mathfrak{D}$ for all $D_i \in \mathfrak{D}$ $(1 \leq i \leq n)$, and $\llcorner a_1, \dots, a_n \lrcorner \in \llcorner D_1, \dots, D_n \lrcorner$ for all $a_1, \dots, a_n \in D$.
- and an interpretation of function symbols in V_K over $\mathcal{U}$.

Example 5 The view in Example 1 is to present a set of itineraries, in which each element is a list of flights. In order to model the necessary domain elements, we used constructors $[\cdot]$ and $\{\cdot\}$ for finite lists and finite sets, respectively, both with unfixed arity. Furthermore, we may use a constructor (flight_no, date, departure, origin, destination, class) of fixed arity six.

If *Flight_number*, *Date*, *Time*, *Airport* and *Character* denote base domains, then (flight_no:*Flight_number*, date: *Date*, departure:*Time*, origin:*Airport*, destination:*Airport*, class: *Character*) defines a complex domain for flights. Let this be called *Flight*. Then {[*Flight*]} defines the domain for the set of itineraries.

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 bounded exploration postulate for sequential algorithms requests that only finitely many terms can be updated in an elementary step [19]. For parallel algorithms, this postulate becomes significantly more complicated, as basic constituents not involving any parallelism (so-called "proclets") have to be considered [9].

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 [20], 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 $\rho(m) = \beta(\alpha(b_1) \odot \ldots \odot \alpha(b_n))$ for $m = \langle b_1, \ldots, b_n \rangle \in \mathcal{M}(D)$.

Example 6 A typical location operator is *count* counting the number of elements in a multiset. In this case, α assigns 1 to each element of D, $\odot$ is addition, and β is the identity on D.

If α assigns to b the set $\{b\}$, if b satisfies a formula φ, and $\emptyset$ otherwise, $\odot$ is set union, and β is again the identity, then the location operator defined by α, $\odot$ and β assigns to a multiset $m \in \mathcal{M}(D)$ the set of elements in m satisfying φ.

The definitions of updates, update sets and update multisets are the same as for ASMs [11].

Definition 10 Let t be a database transformation and S be a state of t. A pair $(f, (a_1, \ldots, a_n))$ consisting of an n-ary function symbol f, and arguments $a_1, \ldots, a_n$ in the base set of S for its interpretation f_S in a state is called a *location*, usually written as $f(a_1, \ldots, a_n)$. An *update* of t is a pair (ℓ, v), where ℓ is a location $f(a_1, \ldots, a_n)$ and v is another element in the base set of S. 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 [32].

In order to derive an exploration boundary for a database transformation, we have to be aware of the fact that databases permit associative access. In principle, the claim of unique identifiability applies to databases, as emphasised by Beeri and Thalheim in [6]. More precisely, unique identifiability has to be claimed for the basic updatable units in a database, e.g. objects in [26]. Unique identifiability, however, does not necessarily apply to all elements in a database. Sets of logically indistinguishable locations may be updated simultaneously. Nevertheless, for databases, only logical properties are relevant—this is the so-called "genericity principle" in database theory [5]—and therefore, it must still be possible to use terms to access elements and locations in the database part of a state. These terms, however, may be non-ground.

The exploration boundary postulate in the sequential ASM thesis in [19] uses a finite set of ground terms as bounded exploration witness in the sense that whenever states S_1 and S_2 coincide over this set of ground terms, the update set produced by the sequential algorithm is the same in these states. The intuition behind the postulate is that only the part of a state that is given by means of the witness will actually be explored by the algorithm.

The fact that only finitely many locations can be explored remains the same for database transformations. However, permitting parallel accessibility within the database part of a state forces us to slightly change our view on the bounded exploration witness. For this, we need access terms, which in a sense cover associative access to databases.

In the following definition, we exploit the interpretation of terms α, β in a state (i.e. a structure) S. If α is a ground term, then $val_S(\alpha)$ is the value of the base set resulting from the interpretation of the function symbols in α in the structure S. If β is non-ground, then in addition we require a *variable assignment* ζ, which assigns values of the base set to the variables in β. Then, a variable x is interpreted by $\zeta(x)$, and $val_{S,\zeta}(\beta)$ is the value of the base set resulting from the interpretation of the function symbols and variables in α in the structure S.

Definition 11 An *access term* is either a ground term α or a pair (β, α) of terms, the variables $x_1, \ldots, x_n$ in which refer to the arguments of some $f \in \Sigma_{db}$. The interpretation of (β, α) in a state S is the set of locations

$$\{f(x_1, \ldots, x_n)[a_1/x_1, \ldots, a_n/x_n] \mid val_{S,\zeta}(\beta) = val_{S,\zeta}(\alpha)\}$$

with the variable assignment $\zeta = \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$. Structures S_1 and S_2 *coincide* over a set T of access terms iff the interpretation of each $(\beta, \alpha) \in T$ over S_1 and S_2 are equal.

Due to our request that the database part of a state is always finite, there will be a maximum number m of elements that are accessible in parallel. Furthermore, there is always a number n such that n variables are sufficient to describe the updates of a database transformation, and n can be taken to be minimal. Then for each state S, the upper boundary of exploration is $\mathcal{O}(m^n)$, where m depends on S. Taking these together, we obtain our fourth postulate.

Definition 12 (*bounded exploration postulate*) For a database transformation t, there exists a fixed, finite set T of access 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.

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 (the interpretation of f in state S) to B' results in $f_{S'}$. Substructures $S_1, S_2 \preceq S$ are *equivalent* (notation: $S_1 \equiv S_2$) iff there exists an automorphism $\sigma \in Aut(S)$ with $\sigma(S_1) = S_2$.

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 $(S, \sigma'(S')) \in \tau_t$. Furthermore, for each state S'' with $(S, S'') \in \tau_t$, there exists some substructure $Y \preceq S$ with $X \equiv Y$ and $Y \preceq S''$.

3 A language for abstract state services

In this section, we develop an abstract language for the specification of AS^{2}s. As AS^{2}s are based on database transformations, we first adapt Abstract Database Transformation Machines (ADTMs), which have been proven to capture database transformations in general [32]. In doing so, we can specify the database layer by

- a background class specifying additional base types, each associated with a base domain, constructor symbols and function symbols associated with these constructors,
- a signature comprising function symbols for the database and algorithmic parts of states and for the bridge functions,
- a set of initial states for the database system,
- a set of transactions, each of which will be defined by an ADTM-rule and
- a set of auxiliary ADTM-rules.

In the following, we simply use the term *rule* to mean ADTM-rule. On top of such specification of a database system, we define the view layer by a set of extended views. Each view is defined by

- a signature defined similarly to the signature for the underlying database system,
- a defining query that is defined by another ADTM-rule possibly using auxiliary ADTM-rules, and
- a set of operations that are specified similar to transactions, but in addition include details on how to handle views.

While such a definition would capture all AS^{2}s, it does not exploit declarative query languages as emphasised in [32]. Therefore, in a second step, we extend the language by adding declarative query expressions taken from a complete fixed-point query language [28]. This is merely "syntactic sugar", as by the background postulate, we have at least multiset and tuple constructors available, and thus could emulate any higher-order structure, in particular, those resulting from inflationary fixed-point constructions.

3.1 Database systems specifications

We first define ADTM-rules on top of a signature Σ and some background class satisfying the requirements in Definitions 6 and 8. Furthermore, ADTM-rules may involve variables, so in the following definition, we also refer to *database variables* as variables that must be interpreted by values in B_{db}.

Definition 15 The set $\mathcal{R}$ of *rules* over a signature $\Sigma = \Sigma_{db} \cup \Sigma_a \cup \{f_1, \ldots, f_\ell\}$ are defined as follows:

- If $t_0, \ldots, t_n$ are terms over Σ, and f is an n-ary function symbol in Σ, then $f(t_1, \ldots, t_n) := t_0$ is a rule in $\mathcal{R}$ called *assignment rule*.
- If φ is a Boolean term and $r' \in \mathcal{R}$ is a rule, then **if** φ **then** r' **endif** is a rule in $\mathcal{R}$ called *conditional rule*.
- If φ is a Boolean term with only free database variables $x_1, \ldots, x_k$ and $r' \in \mathcal{R}$ is a rule, then **forall** $x_1, \ldots, x_k$ **with** φ **do** r' **enddo** is a rule in $\mathcal{R}$ called *forall rule*.
- If $r_1, \ldots, r_n$ are rules in $\mathcal{R}$, then also $r_1 \| \cdots \| r_n$ is a rule in $\mathcal{R}$ called *parallel rule*.
- If φ is a Boolean term with only free database variables $x_1, \ldots, x_k$ and $r' \in \mathcal{R}$ is a rule, then **choose** $x_1, \ldots, x_k$ **with** φ **do** r' **enddo** is a rule in $\mathcal{R}$ called *choice rule*.
- If r_1, r_2 are rules in $\mathcal{R}$, then also $r_1 \ ; \ r_2$ is a rule in $\mathcal{R}$ called *sequence rule*.

- If $r' \in \mathcal{R}$ is a rule and ϑ is a location function that assigns location operators ϱ to terms t, then **let** $\vartheta(t) = \varrho$ **in** r' **endlet** is a rule in $\mathcal{R}$ called *let rule*.
- If $r'(x_1, \dots, x_k) \in \mathcal{R}$ is a rule using variables $x_1, \dots, x_k$, and $t_1, \dots, t_k$ are terms, then $r'(t_1, \dots, t_k) \in \mathcal{R}$ is a rule called *call rule*.

On the grounds of this definition, we can now define a database system specification as indicated earlier. First, we assume a fixed *background specification* $\mathcal{BS}$ by means of a type systems, e.g.

$$t = b \mid (a_1 : t_1, \dots, a_n : t_n) \mid \{t\} \mid \langle t \rangle \mid \dots ,$$

i.e. we take base types b and constructors $(\cdot)$, $\{\cdot\}$, $\langle\cdot\rangle$, etc. for records, finite sets, multisets and maybe more. According to Definition 8, the type *BOOL* for truth values must be one of the base types. The same applies to at least one type for database values. Furthermore, the constructors $(\cdot)$ for records–pairs would be sufficient – and $\langle\cdot\rangle$ for multisets must be present.

With each base type b, we associate a *domain* $dom(b)$, so the base types collectively define the set $\mathcal{D}$ of base domains requested in Definition 7. The domain association *dom* is then extended for the type constructors in the way defined in Definition 7.

In addition to the types, $\mathcal{BS}$ must contain functions symbols, each of which is associated with an arity that is defined by input- and output-types. These functions may be parametric polymorphic, i.e. type variables will be permitted. For instance, $\wedge : BOOL\ BOOL \rightarrow BOOL$ defines a function symbol for conjunction, and $\uplus : \langle x \rangle\ \langle x \rangle \rightarrow \langle x \rangle$ defines a function symbol for polymorphic multiset union (with multiplicities added up). These function symbol are then interpreted over the domains as requested in Definition 7. Note that some function symbols for truth values, records and multisets are requested in Definition 8.

On top of a background specification, we can define a *signature* Σ as in Definition 5, but we permit the modification to use types. For instance, $even : NAT \rightarrow BOOL$ would define a function symbol in the algorithmic part of Σ, which defines a function on natural numbers for testing whether a given number is even or not.

Definition 16 A *database system specification* DBSS over a background specification $\mathcal{BS}$ consists of

- a signature Σ over $\mathcal{BS}$ fulfilling the requests from Definition 6,
- a set $\mathcal{I}$ of states over Σ called initial states of DBSS that is closed under isomorphisms,
- a finite set $\mathcal{T}$ of parameterised transactions, each of which is defined by a rule as in Definition 15 with free variables equal to the parameters and
- a finite set $\mathcal{A}$ of auxiliary rules defined in the same way as $\mathcal{T}$.

Note that our definition of ADTM-rules permits calling rules. We do not exclude in Definition 16 that a transaction $r \in \mathcal{T}$ is called by another rule in $\mathcal{T}$ or even $\mathcal{A}$, but in this case, it is only treated as an auxiliary rule.

3.2 Extended view specifications

We now approach the requirements of Definition 3 to define the extended views on top of a database system specification. In doing so, we have to address the signature, the defining query and operations on views.

If $\Sigma = \Sigma_{db} \cup \Sigma_a \cup \{f_1, \dots, f_n\}$ is the signature of a database system specification DBSS, we extend the signature by adding database function symbols and bridge functions to obtain the extended signature $\Sigma_{\text{ext}} = \Sigma'_{db} \cup \Sigma_a \cup \{f_1, \dots, f_k\}$ (with $k \geq n$). Then the added function symbols, i.e. $\Sigma'_{db} - \Sigma_{db} \cup \{f_{n+1}, \dots, f_k\}$ define a *view signature*, denoted as Σ_v.

An ADTM-rule r_v over the extended schema Σ_{ext} will be called a *query* over Σ, iff the input database is preserved, and the result only depends on it. Formally, the following two conditions must be satisfied:

- For all state pairs (S, S') produced by r_v, i.e. there is a finite run $S_0, \dots, S_\ell$ of r_v with initial state $S_0 = S$ and final state $S_\ell = S'$ such that the restrictions of S and S' to Σ coincide.
- For all state pairs (S_1, S'_1) and (S_2, S'_2) produced by r_v such that the restrictions of S_1 and S_2 to Σ coincide we have $S'_1 = S'_2$.

Though this definition of query is semantical, as it is based on the states and not on the text of the rule r_v, the conditions are easily satisfied, if r_v only contains assignment rules with function symbols in the view signature Σ_v.

Definition 17 If DBSS is a database system specification with signature Σ, then a *view* v over DBSS is defined by a view signature Σ_v over Σ and a defining query r_v over $\Sigma \cup \Sigma_v$.

In order to address operations associated with views, we need *selection conditions*, which are Boolean terms that can be evaluated on structures over Σ_v and thus define substructures. If φ is a selection condition, we permit the use of *restriction terms* $t[\varphi]$. Then, a *v-rule* over view v with selection condition φ is given by a parametrised ADTM-rule without assignments, but with the possibility to use restriction terms, to open views by means of rules $open(v')$ for $v' \neq v$ and to close the view v using the rule $close(v)$. Opening a view v means to initiate the functions in the corresponding view signature Σ_v, while closing it can be expressed simply by letting all functons in Σ_v be totally undefined. Technically, the use of restriction terms can be replaced by using conditional rules with the term φ.

Definition 18 An *extended view* over a database system specification DBSS consists of a view v over DBSS and a set $\mathcal{O}_v$ of v-rules over v. An *Abstract State Service Specification* (A3S) $\mathcal{AS}$ consists of a database system specification DBSS and a set $\mathcal{V}$ of extended views over DBSS.

3.3 Fixed-point queries

As ADTMs capture database transformations, A3Ss capture AS^{2}s, so in a theoretical sense we have achieved completeness, which in most cases is far more than we need. On the other hand, the database transformations that are used as defining queries do not fully exploit the possibilities that are given by declarative query language, though using forall rules allows us to adopt a calculus-style of defining queries, i.e. the Boolean term φ in such rules would in fact be used to express the query in a declarative way. As multisets and maybe also sets can be used in these terms, any higher-order construction would be enabled. However, writing queries as higher-order logical expressions is uncommon; therefore, we provide the possibility to exploit fixed-point queries in Definition 17 to define views. Such queries have been defined in the Identity Query Language (IQL) from [2] and used in [28].

In order to formalise this, we assume that one of the base types in the type system of $\mathcal{BS}$ is a type *ID*, the domain of which is a countable set of abstract identifiers. Then, take countable sets of variables V_t for each type t. These sets are to be pairwise disjoint. Variables and constants of type t are *terms* of that type. In addition, for each variable ι of type *ID*, there is a term $\hat{\iota}$ of some type $t(\iota)$. If $\tau_1, \ldots, \tau_k$ are terms of type t, then $\{\tau_1, \ldots, \tau_k\}$ is a term of type $\{t\}$, and $\langle\tau_1, \ldots, \tau_k\rangle$ is a term of type $\langle t\rangle$. If $\tau_1, \ldots, \tau_k$ are terms of type $t_1, \ldots, t_k$, respectively, then $(a_1 : \tau_1, \ldots, a_k : \tau_k)$ is a term of type $(a_1 : t_1, \ldots, a_k : t_k)$. Each function symbol f in the signature Σ of arity n defines a relation symbol Rf of arity $n + 1$.

If τ_1, τ_2 are terms of type $\{t\}$ (or $\langle t\rangle$) and t, respectively, then $\tau_1(\tau_2)$ is a positive literal (also called a *fact*) and $\neg\tau_1(\tau_2)$ is a negative literal. If τ_1, τ_2 are terms 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 fact without variables.

A *clause* is an expression of the form $L_0 \leftarrow L_1, \ldots, L_k$ with a fact L_0 (called the *head* of the clause) and literals $L_1, \ldots, L_k$ (called the *body* of the clause), such that each variable in L_0 not appearing in the rule's body is of type *ID*.

A *logic program* is a sequence $P_1; \ldots; P_\ell$, in which each P_i is a set of clauses.

Finally, a *query* Q is defined by a view signature Σ_v and a logic program $\mathcal{P}_Q$, in which the function symbols in Σ_v correspond to predicate symbols that only occur in heads of clauses.

Example 7 Let us illustrate fixed-point queries by an example adapted from [23] dealing with a paper submission and reviewing system. For such a system, we could model a simple relational database schema. At its core, we would have the following signature (using functions with arity in parentheses to represent relations): paper(7), member(9), assigned(2), review(17).

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 and 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 and recommendation. Paper authors are handled separately. So we may have paper (19,“Abstract State Services”,kdschewe@acm.org, “dr0w33@p”,“…”,28-02-09,accepted) = 1 in some state indicating that on 28 February 2009 a paper with title “Abstract State Services” and abstract “…” was submitted. The email-address kdschewe@acm.org is the contact e-mail address, the paper received the id 19, the chosen password is “dr0w33@p”, and the paper has been accepted.

For the task of PC-members to discuss papers after they have been reviewed, we need a set of tuples each representing a paper with its paper_id, title, abstract and the set of reviews associated with it. The query needed to produce this relation can be expressed by using the following logic program:

$\mathrm{pap}(i, p, t, ab, R) \leftarrow \mathrm{paper}(i, (p, t, e, pw, ab, d, c));$

Here, the arguments of paper in the rule body refer to an abstract identifier i and the attributes paper_id p, title t, contact_email e, password pw, abstract ab, submission_date d and accept_code c. In the first step, we “forget” most of these, keeping only paper_id p, title t and abstract ab and creating a new identifier R for the set of reviews that is still to be constructed.

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

In this second step, we use the predicate $\hat{R}$ associated with the identifier R constructed in the first step and collect the reviews to the corresponding paper. As in the first step, the arguments of review and member refer again to abstract identifiers and the attributes given earlier in the respective order. Most of these arguments are simply "forgotten", i.e. they do not appear in the head of the clause. We only keep for each review the abstract paper identifier i, the reviewer name n, the subreviewer n', contribution c, positive_aspects pos, negative_aspects neg, details dc, confidence co, originality o, significance s, technical_quality q, relevance r, presentation pr and recommendation or.

$$\mathrm{ans}(p, t, ab, \hat{R}) \leftarrow \mathrm{pap}(i, p, t, ab, R).$$

In the last step, we replace the abstract identifier R in the result by the set of reviews $\hat{R}$ constructed in the previous step.

4 Examples

Let us now look at examples for AS^{2}s. We will concentrate on functions, which are quite often taken as Web Services, Data Warehouses, Web Information Systems and systems using Intelligent Pads.

4.1 Functional web services

We consider services that mainly consist of some service operations that are made available, i.e. the view they are defined on is trivial.

Example 8 Suppose we have a database with employee information, in particular salaries. Individual salaries will be kept hidden, but building averages for groups of 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 [11], 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

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

The next example was used in [33] to illustrate the combination of meme media objects by means of Intellingent Pads.

Example 9 Another simple example is given by a currency converter, in which case the database schema would simply need a single relation schema `Rate` with three attributes source_currency, target_currency and exchange_rate.

The conversion operation would be made available in combination with an empty view. We expect two currencies and a value for the amount as input, so the service operation can be expressed by the simple rule convert(in, out, a), which is defined by

choose p **with** `Rate`(in, out, p) = true **do** result := $p \cdot a$ **enddo**

4.2 Data warehouses

More interesting examples of AS^{2}s are given by data warehouses, which could be turned this way into web warehouses. The ASM-based approach in [36] 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. 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. A third ASM models the OLAP interface on the basis of the idea that datamarts can be represented as extended views.

Example 10 For instance, here we could have controlled functions sales, product, and store all of arity 3 and a static ternary function time. Similar to Example 7, 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.

For example, a view may 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; they must be defined as part of the background signature) by the simple rule

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

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 [36] 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.

4.3 Web information systems

Another even more complex example is given by Web Information Systems (WISs), following the modelling approach in [28], 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 AS^{2}s.

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.

Example 11 A stock market database stores stock values by means of a relation `Stock` with attributes for company, date time and stock_value. In a view, we would represent for a selected company the current stock value plus the development of the value since the last opening. Thus, the view with parameter c for the company name can be simply defined by the rule

```
choose $v$ with Stock($c$,today,now,$v$) = true
        do Result := ($v$, $\pi_{\text{time,value}}(\sigma_{\text{company}=c,\text{date}=\text{today}}$
          (Stock)))
        enddo
```

Service operations could be buy(x) or sell(x) with x indicating the number of shares to be bought or sold, or average and predict to determine the average value over the day, and a predicted value at the end of the day. Other service operations could be for changing the company, switching to a larger timeframe for the development of the share, etc.

In the next section, we want to show how to combine the finance data service in Example 11 with the currency conversion service from Example 9. This would constitute an easy example of service composition. A more difficult situation arises in epidemiology, e.g. in the prediction of bushfire spreading, in which case we would have to combine a GIS service and a weather service with a forecasting model for the spreading of a fire. The next example provides a glimpse of one of the input services needed for this.

Example 12 A weather service may provide a view MEASUREMENT containing a set of 5-tuples $(loc, date, time, wind, rain)$, in which loc denotes a location by means of a pair of coordinates, $date$ and $time$ denote the date and time of measurement, $wind$ denotes a wind-vector indicating direction and speed of the wind, and $rain$ denotes the amount of rain in millimetres per hour. That is, loc is of type *Point*, which is defined as $(x : FLOAT, y : FLOAT)$, $date$ is of type *DATE*, $time$ is of type *TIME*, $wind$ is of type *Vector*, which is also defined as $(x : FLOAT, y : FLOAT)$, and $rain$ is of type *FLOAT*.

A service operation on this view may be predict(t) with t denoting a timeframe in hours. The service operation would produce a similar set of triples as predicted to develop within the next t hours. This could be specified as:

```
predict($t$) =
    close_view;
    advance(now,$t$,predict_date,predict_time);
    let $\vartheta(set\_of\_predictions) = \cup$ in
        forall $loc$ with $\exists d, t', w, r$.MEASUREMENT
          $(loc, d, t', w, r) = true$
        do last_five_measurements($loc$,$M$);
           extrapolate($M$,predict_date,predict_time,
             predict_wind,predict_rain);
           set_of_predictions := { ($loc$,
             predict_wind,predict_rain)}
        enddo
    endlet
```

This uses other rules advance(now,t,predict_date, predict_time) to compute the date and time t hours from now, last_five_measurements(loc,M) to determine the last five measurements at the given location loc and return them in a list M, and the core rule extrapolate (M,predict_date, predict_time,predict_wind,predict_rain) to estimate (by extrapolation) the wind and rain at the given date and time and the basis of the measurements in M (and maybe other data in the database).

5 Abstract state services composition and personalisation

In this section, we discuss how to extract components from AS^{2}s and recompose them to form new AS^{2}s. In doing so, we introduce constructions for parallel composition with feedback and sequential composition. The discussion is rounded up by a brief indication of how component extraction can be personalised, though personalisation is not the major focus of this paper.

5.1 Extraction of service components

While AS^2 integration replaces given AS^{2}s by new ones preserving their functionality, the composition of AS^{2}s does not aim at replacing any existing AS^2. Instead, the goal is to define new services that exploit functionality of existing ones. That is, we will have to extract components from existing AS^{2}s and recompose these components. A simple form of recomposition can be component integration as discussed in the previous section—the extracted components will be AS^{2}s as well. However, we may also exploit other mechanisms of component composition, e.g. those discussed in [29].

In order to extract components from an AS^2, 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 AS^2 with the same underlying database system as before.

In a second step, we actually restrict the views $v \in \mathcal{V}'$ themselves by defining views 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 $\{v_1, \dots, v_k\}$, then $p_v \circ v$ would have to be replaced by $\{p_{v_i} \circ v_i \mid i \in \{1, \dots, k\}, v_i \in \mathcal{V}'\}$. In this way, the collection of views p_v defines an AS^2 with the same underlying database system as before. We will call this an AS^2 component.

Definition 19 Let $\mathcal{A} = (DBS, \mathcal{V}) = (\mathcal{S}, \tau, \{\tau_t\}_{t \in \mathcal{T}}, \{(v, \{o_1, \dots, o_{n_v}\})\}_{v \in \mathcal{V}})$ be an AS^2. A *component* of $\mathcal{A}$ is an AS$^2(\mathcal{S}, \tau, \{\tau_t\}_{t \in \mathcal{T}}, \{(p_v \circ v, \{o'_1, \dots, o'_{n'_v}\})\}_{v \in \mathcal{V}'})$ with $\mathcal{V}' \subseteq \mathcal{V}$ and $\{o'_1, \dots, o'_{n'_v}\} \subseteq \{o_1, \dots, o_{n_v}\}$.

Note that in order to define a component of an AS$^2\mathcal{A}$, it is not necessary to know anything about the DBS of $\mathcal{A}$, as only the views and associated service operations are affected. In a practical sense, this reflects the request that the database layer of an AS^2 should be hidden.

5.2 Parallel composition of abstract state services

After extracting components from several AS^{2}s, their integration along the lines discussed in the previous section is one way of recomposing them. Another one is parallel composition.

Definition 20 Let $\mathcal{A}^i = (\mathcal{S}^i, \tau^i, \{\tau_t\}_{t \in \mathcal{T}^i}, \{(v, \{o_1, \dots, o_{n^i_v}\})\}_{v \in \mathcal{V}^i})$ $(i = 1, \dots, n)$ be AS^{2}s. Their *parallel composition* $\mathcal{A}^1 \oplus \cdots \oplus \mathcal{A}^n$ is an AS^2 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 composition, 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 $i = 1, \dots, n$.
– The set of transactions is the product $\mathcal{T} = \{t_1 \| \cdots \| t_n \mid t_i \in \mathcal{T}^i\}$.
– Small step transition relations are defined by parallel composition, i.e. $(S_1 \cup \cdots \cup S_n, S'_1 \cup \cdots \cup S'_n) \in \tau_{t_1 \| \cdots \| t_n}$ iff $(S_i, S'_i) \in \tau_{t_i}$ for all $i = 1, \dots, n$.
– The set of views is also defined as a product $\mathcal{V} = \{v_1 \| \cdots \| v_n \mid v_i \in \mathcal{V}^i\}$.
– The sets of service operations are defined by parallel composition $\mathcal{O}_{v_1 \| \cdots \| v_n} = \{o_1 \| \cdots \| o_n \mid o_i \in \mathcal{O}_{v_i}\}$.

The obvious drawback of parallel composition is that we still make merely the service operations of the original AS^{2}s available. In order to obtain new service operations by composition of extracted ones, we follow the approach in [29] to distinguish between retrieval and update operations.

Definition 21 A service operation $o \in \mathcal{O}_v$ in a component of an AS$^2\mathcal{A}$ 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 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.

In a one-sided composition, all the service operations associated with a view v_2 come first, provided they open the view v_1, and are composed with the update operations associated with view v_1. It does make no sense to compose them with retrieval operations, as these merely extract data, but leave the underlying database unchanged, so from a service user point-of-view any composition does the same as the retrieval operation alone. In a double-sided composition, we use the same composition of service operations in both directions. That is, compose service operations associated with v_2 that open v_1 with update operations on v_1, compose service operations associated with v_1 that open v_2 with update operations on v_2 and preserve all retrieval operations on v_1 and v_2. In this way, we build all meaningful compositions of the service operations associated with views v_1 and v_2.

Definition 22 Let $\mathcal{A}$ be an AS^2.

- 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 AS^{2}s with double-sided composition of views, i.e. instead of taking $\mathcal{O}_{v_1 \| v_2}$ as in Definition 20, 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 [29].

In this definition we made the restriction that views belong to the same AS^2. However, this is only a technical restriction, because we can always extend a view to become a view on an extended DBS; it will simply ignore the extension. Using this, we can use single-sided and double-sided composition in Definition 22 loosely for views defined on different AS^{2}s. This is exploited in the next definition of parallel composition with feedback.

Here, "feedback" refers to nothing more than the analogy to circuits, as the output of an operation from one service feeds in as input for the operation of the other service and vice versa. This analogy is also exploited in some Meme Media tools [33].

Definition 23 Let $\mathcal{A}^i = (\mathcal{S}^i, \tau^i, \{\tau_t\}_{t \in \mathcal{T}^i}, \{(v, \{o_1, \dots, o_{n^i_v}\})\}_{v \in \mathcal{V}^i})$ $(i = 1, 2)$ be AS^{2}s. Their *parallel composition with feedback* $\mathcal{A}^1 \bowtie \mathcal{A}^2$ is an AS^2 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 composition, 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 22.

Note that same as component extraction parallel composition (with feedback) does not require knowledge of the underlying database systems, as only views and service operations are affected. For parallel composition with feedback, however, it is necessary to know which service operations are retrieval operations and which are update operations.

Example 13 Let us look again at the flight booking service in Examples 1 and 2. Let the view, on which this service operation is defined be v_1. Obviously, the flight-booking operation (rename it to flight_booking) is an update operation.

Take another service dealing with hotel bookings. Here, we would use a view v_2 presenting available hotel for a specified timeframe. The booking operation hotel_booking is also an update operation.

The double-sided composition $v_1 \bowtie v_2$ is a view that simultaneously creates a set of itineraries and a set of hotel options for specified timeframes. In addition, the compositions hotel_booking $\circ$ flight_booking and flight_booking $\circ$ hotel_booking are service operations associated with $v_1 \bowtie v_2$. The first of these feeds the output of flight_booking, i.e. the booked itinerary, as input into hotel_booking. Only the arrival and departure dates at the destination are needed for this composition. Similarly, the second composed service operation feeds the output of hotel_booking, restricted to the booked arrival and departure dates, as input into flight_booking.

Note that in Example 13, the two composed service operations offer almost the same functionality, which reflects that in this case we could have done almost equally well with parallel composition as defined in Definition 20. Example 16 below shows another example of double-sided composition with greater dependencies between the composed services.

5.3 Sequential composition of abstract state services

Parallel composition with feedback replaces update operations by compositions with service operations from another AS^2, and this composition is used in both directions. If we only want to consider compositions with operations from one AS^2 executed first, we obtain a sequential composition. This is defined next exploiting single-sided composition.

Definition 24 Let $\mathcal{A}^i = (\mathcal{S}^i, \tau^i, \{\tau_t\}_{t\in\mathcal{T}^i}, \{(v, \{o_1, \dots, o_{n_v^i}\})\}_{v\in\mathcal{V}^i})$ $(i = 1, 2)$ be AS^{2}s. Their *sequential composition* $\mathcal{A}^1 \circ \mathcal{A}^2$ is an AS^2 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 composition, 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 $\mathcal{V} = \{v_1 \ltimes v_2 \mid v_i \in \mathcal{V}^i\}$.
– The sets of service operations are defined by single-sided composition $\mathcal{O}_{v_1 \ltimes v_2}$ as in Definition 22.

With respect to the hidden database layer, sequential composition behaves in the same way as parallel composition (with feedback).

Example 14 Similar to Example 13, we could use the single-sided composition of the flight-booking view v_1 with a conference registration service v_2. Assume that v_2 is associated with a service operation register. Thus, flight_booking $\circ$ register becomes a service operation associated with $v_1 \ltimes v_2$. The conference dates resulting as output from the register operation feed as input into the flight_booking service operation.

5.4 Final remarks

A few final remarks on service extraction and composition are due now. The composition constructs we defined, i.e. sequential, parallel composition and parallel composition with feedback operate on sets of views and associated sets of service operations, and in all cases, single- and double-sided composition is applied uniformly. This is, however, no restriction, as the constructs are applied to AS^2 components (which are again AS^{2}s), and these components can be as small as desired, i.e. they can even consist of a single view with a single service operation.

If different compositions of service operations are needed, service components can be composed in different ways, and the results could be combined by parallel composition. Note that if parallel composition is used for AS^{2}s with the same underlying DBS, this could actually be identified with an AS^2 on this DBS instead of the some with itself. As the database layer is hidden, this identification would not be visible to service users. Of course, for practical convenience, it may be desirable to define composition constructs that deal directly with these cases. Theoretically, however, these are merely derived from the constructs we defined, and thus can be dispensed with.

Our definitions also neglect "local" operations, i.e. the possibility to compose the extracted service operations with any other operation defined elsewhere. If this composition is in the form $o_1 \circ o_\ell \circ o_2$ for operations o_1, o_2 as in Definition 22 and a local operation o_ℓ, this composition can again be mimiqued by the constructs mentioned earlier, if o_ℓ is defined as a service operation associated with a (possibly empty) view on some local database. In this way, the usage of such local operations does not add anything, but of course practically, we would avoid defining an AS^2 for the only purpose to extract a service operation that is then subject to service composition.

A more general case arises, if an AS^2 is defined, and the defining queries of views as well as service operations are simply used as part of the definition of views and service operations. In this case, the underlying database system of the extracted component would become part of the underlying database system of the AS^2 to be defined, but no general rules apply to forming views and service operations. The only restriction is that access to the database system of the extracted component is limited to the views and service operations of the component, but this is within the hidden database layer.

Therefore, in order to recombine the data extraction and service operations from several ASSs, we may exploit functional composition in general, if input and output are compatible. In addition, we may use aggregation operations and other locally defined auxiliary functions. For instance, if q_{v_1} and q_{v_2} are defining queries of two views v_1 and v_2 resulting in relations, we can aggregate them by building the natural join of the corresponding results. We denote this view by $v_1 \bowtie v_2$, and call it an *aggregated view* of $\{v_1, v_2\}$ with aggregate functions $\{\bowtie\}$. More generally, any functional composition of given views with other functions defines an aggregated view, provided the views have to be executed first.

Similarly, any such functional composition (without the restriction that views have to come first) can define a new service operation. This leads to the following definition of an aggregated ASS.

Definition 25 Let $\mathcal{A}^i = (\mathcal{S}^i, \tau^i, \{\tau_t^i\}_{t\in\mathcal{T}^i}, \{(v^i, \{o_1^i, \dots, o_{n_{v^i}}^i\})\}_{v^i\in\mathcal{V}^i})$ be ASS components $(i = 1, \dots, n)$. An AS$^2\mathcal{A} = (\mathcal{S}, \tau, \{\tau_t\}_{t\in\mathcal{T}}, \{(v, \{o_1, \dots, o_{n_v}\})\}_{v\in\mathcal{V}})$ is an *aggregation* of $\mathcal{A}^1, \dots, \mathcal{A}^n$ with a set of local functions $\mathcal{F}$ iff each view $v \in \mathcal{V}$ is an aggregated view of $\bigcup_{i=1}^n \mathcal{V}^i$ with the aggregate functions $\mathcal{F} \cup \bigcup_{i=1}^n \bigcup_{v^i\in\mathcal{V}^i} \{o_1^i, \dots, o_{n_{v^i}}^i\}$, and each service operation $o \in \mathcal{V}$ is composed out of $\bigcup_{i=1}^n \bigcup_{v^i\in\mathcal{V}^i} \{o_1^i, \dots, o_{n_{v^i}}^i\} \cup \mathcal{F} \cup \bigcup_{i=1}^n \{q_v \mid v \in \mathcal{V}^i\}$.

5.5 Examples

Let us now illustrate the composition of AS^{2}s by two simple examples that draw on the Examples in Sect. 4.

Example 15 Suppose stock values in Example 11 are produced in US dollars. In order to produce values in a different currency, e.g. Euro, we must apply a function f to all such values, i.e. the view would be defined by the rule

```
choose $v$ with Stock($c$,today,now,$v$) = true
    do Result := $(f, \mathtt{map}(id, f))(v, \pi_{\text{time,value}}(\sigma_{\text{company}=c,\text{date}=\text{today}}(\mathtt{Stock})))$
enddo
```

Here, the function f can be defined by the service operation convert (USD,Euro,a) from Example 9. Thus, the conversion takes the role of the projection function p_v in Definition 19. Alternatively, if we consider view-formation also a service function, the composed service would also appear as the result of parallel composition with feedback.

Example 16 Another more complex example for service composition arises, if we want to combine a weather service (as sketched in Example 12) with a GIS service and a bushfire prediction model. We could first produce a parallel composition with feedback between the GIS service and the weather service, which would give us a prediction of the weather by means of wind-vectors and expected rainfall in the geographical area we are interested in, which would then be combined with "local functions" predicting locations under fire on the basis of known fire locations. If these locations contain inhabited areas, which again can be taken from the GIS service, corresponding action plans can be developed.

A sketch of the service operation predict_fire can be obtained as follows:

```
predict_fire(area_of_interest, hours, affected_area) =
  let $\vartheta$(affected_area) $= \cup$ in
  forall $loc, t$ with $loc \in$ area_of_interest $\wedge\ 0 \leq t \leq$ hours
    do forall $d, t', w, r$ with $(loc, d, t', w, r) \in$ predict($t$)
      do PREDICT$(loc, d, t', w, r) := t$
      enddo;
      affected_area := $\bigcup_{t'=0}^{\text{hours}-t} \{loc' + (t+t')\cdot const \cdot w \mid \exists r, d', t''.$
                 PREDICT$(loc', d', t'', w, r) = t \wedge d' = date(\text{advance}(now, t))$
                 $\wedge\ t'' = time(\text{advance}(now, t))\}$
    enddo
  endlet
```

Here the check, whether a location is in the area of interest and producing a different representation for the resulting set of locations is done by a service operation from the GIS service, while the predict service operation is taken from the weather service as sketched in Example 12. The prediction model is based on simple linear spreading in the direction of the wind with a constant *const*.

5.6 AS^2 personalisation

With the concept of AS^{2}s, 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 AS^{2}s, new services can be defined by extracting AS^2 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 AS^{2}s 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 automatically to user preferences, i.e. how can only views and associated operations be selected out of those on offer 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 AS^2, 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 AS^2. 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. `skip` is needed to express optionality, as $p + 1$ expresses or choice between p

and `skip`, i.e. in this case p is optional. `abort` is needed to permit reasoning with equations. For instance, an equation $\bar{\alpha} p = 0$ expresses that α is a precondition for p. Similarly, $pq = 0$ expresses that q cannot follow p.

Definition 26 Let $\mathcal{O}$ denote the set of service operations associated with an AS^2 $\mathcal{A}$, and let $\mathcal{C}$ be a set of Boolean conditions. Then, 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 identified with an operation that tests it, so if p and q are Booleans, then 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 [30], 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 AS^2. 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 [30], and generalised to parameterised plots in [31]. Then, p' would define a personalised plot, and the operations in it are the natural choice for the selection.

Example 17 Consider a service for ordering products with a plot

$$(\alpha_1(\varphi_1\alpha_2\varphi_2 + \alpha_3\varphi_3 + \alpha_4\varphi_4)\alpha_5(\alpha_6\varphi_5 + 1) + 1)^*,$$

in which $\alpha_1, \ldots, \alpha_6$ represent service operations select_product, payment_by_card, payment_by_bank_transfer, payment_by_cheque, provide_address and confirm_order, and the Boolean conditions $\varphi_1, \ldots, \varphi_5$ express price_in_range, payment_by_credit_card, payment_by_bank_transfer, payment_by_cheque and order_confirmed.

The equations $\alpha_4 = 0$ and $\varphi_1(\alpha_3 + \alpha_4) = 0$ express preferences not to pay by cheque, and if possible to pay by card. This can be used to rewrite the plot to

$$(\alpha_1(\varphi_1\alpha_2\varphi_2 + \bar{\varphi}_1\alpha_3\varphi_3)\alpha_5(\alpha_6\varphi_5 + 1) + 1)^*.$$

In this personalised plot, the service operation α_4 = payment_by_cheque has completely disappeared, and the service operation α_3 = payment_by_bank_transfer has received a precondition $\bar{\varphi}_1$ expressing that payment by bank transfer is only applied, if the amount is out of range for card payment.

6 Related work

Let us finally take a closer look at how AS^{2}s fit into the literature. We are particularly interested in meme media [33], web information systems [28] and web services (see e.g. [3, 7, 12, 13, 22]).

6.1 Media types

Media types are a core concept in the co-design approach to web information systems modelling and development. They provide the means to define a conceptual model of a WIS as a whole [28].

At its core, a media type is defined by an extended view on some database schema. The data model and the query language used for defining views are left as open choices. The only request is that abstract identifiers can be created by the queries. These identifiers are to represent URLs, thus serve both as a mechanism to create abstract page content and links. That is, when a query is evaluated against the underlying database, the result will be a set of *media objects*, each of which represents some abstract page content with an identifying URL and possibly links to other media objects. However, media types are not bound to page abstraction, but can also be used to capture larger portions of a WIS, thereby supporting context-awareness, session objects and collaboration.

The most important extension to such views is provided by operations, through which a dialogue interface to the media objects is enabled. In this way, media types also capture aspects of WIS functionality. As media types are first of all operation-extended views defined on some database schema, and the operations induce database transactions, media types are equivalent to the operation-extended views in Definition 3 addressing the view layer of an AS^2, in other words: our model of AS^{2}s is an abstraction of media types.

The difference is that media types were defined to capture the interaction of users with a WIS, while AS^{2}s also highlight the aspect of using the offered service operations to define new services by means of component extraction and composition. Furthermore, the approach of AS^{2}s aims at a theory of data-intensive services, thus instead of focusing on the usability of conceptual languages, the main goal is to define services in general and to show that this definition can be exploited to enable service-oriented computing in the sense that as much as possible functionality (and data) can be borrowed from existing services.

Another difference is that media types are extended in various other ways addressing granularity of the content by means of hierarchical versions, adaptivity by means of cohesion that enables automatic separation of important content from less important one and presentation option that determine the layout and playout of media objects. These extensions are helpful for the web-based interaction, but they are of minor importance, when it comes to component extraction and recomposition of services. Nevertheless, extending AS^{2}s in these directions is an idea to be addressed further, as finally services will also need to pay attention to dialogue interaction.

6.2 Meme media

Meme media technologies have been developed since the second half of the 1980s, i.e. even prior to the development of the WWW. They are centred around a model for worldwide publication, reediting and redistribution of intellectual resources, depicted in [33, Figure 1.2]. The idea is to extract these resources and to use wrappers to bring them into the uniform shape of a meme object, which is then stored in a meme pool, from which it can be taken for the purpose of defining new media resources. Thus, the basic idea is very similar to the one underlying AS^{2}s.

The major difference is that the work on meme media technologies is primarily focused on technical solutions without any intent to provide a general theoretical framework for the work. Key tools are wrappers that enable hiding details of extracted resources and provide a common frame-based interface. For instance, if a resource is made available via the web maybe in form of an HTML-document, then the frame representation may provide input-slots linked to input-field on the HTML-page and coupled with scripts handling the input. In this way, meme objects support the model-view-controller (MVC) architecture. Furthermore, meme objects are coupled with a presentation, whereas AS^{2}s abstract from all aspects of representation.

However, despite these apparent differences, meme media can be considered as the practical counterpart of AS^{2}s. Wrapping technology will be needed to extract AS^2 components and bring them into a form that allows the composition constructs to be applied. The redistribution corresponds to turning any resulting AS^2 into a running systems maybe by exploiting media types.

6.3 Web services

While media types and meme media refer to research done by only one group, web services are ubiquitous, and it is impossible to relate AS^{2}s to all work that has been done in this area. Service foundations, composition, management and monitoring and service-oriented engineering have been identified as core research themes in [24], but surprisingly, little work has been devoted to foundations, whereas the research on AS^{2}s mainly addresses exactly the theoretical foundations. In doing so, the core features of data-intensive services have been highlighted by means of postulates, and it is hard to deny that these intuitive postulates capture the intent behind services.

The contribution of AS^{2}s for now is to give a precise formal clarification of what a service is. As it is based on a variant of ASMs that capture all database transformations, it captures all services. It further clarifies how services can be extracted and recomposed remaining completely within the same formal framework.

Of course, AS^{2}s do not solve all problems around web services. They provide the theoretical backbone for the many research problems addressed in the web services community. Among these research problems is the management of offered services by means of uniform interfaces that make clear what can be expected from the service. As this also becomes a commercial issue, pricing, conditions of usage, liability, etc. are non-negligible questions. Another group of research problems in this area is linked to the design of services, which is out of the scope of our current work on AS^{2}s.

Finally, despite AS^{2}s being considered as theoretical backbone for service-oriented system development, we would like to emphasise that AS^{2}s treat not only functions, but also data as part of services, which many approaches to web services and web services composition do not.

7 Conclusion

In this paper, we introduced Abstract State Services (AS^{2}s) as an abstraction of data-intensive services that can be made available for use by other systems, e.g. via the web. An AS^2 combines a hidden database layer with an operation-equipped view layer, thereby integration data and functionality offerings into our notion of service. An AS^2 can be almost anything, e.g. a simple function, a data warehouse or a Web Information System. We adopted the approach taken for the proof of the sequential ASM thesis, i.e. we defined AS^{2}s by means of intuitive, abstract postulates. The rationale behind this approach is that we did not simply want to define "just another service language", whatever expressiveness such a language might have, but to actually define what a data-intensive service is. We believe the postulates we came up with are intuitive, and capture the intent behind services.

We defined a language for AS^{2}s based on Abstract Database Transformation Machines (ADTMs), which have been proven to capture database transformations in general [32]. In order to do so, we obtain a language that captures all data-intensive services as stipulated by the postulates. However, for the sake of easier handling of database queries, we integrated expressions from a complete, fixed-point-based query language. Such a language captures indeed all AS^{2}s. We exemplified this by a diverse range of AS^2 examples, and by comparing our work with web services, media types and meme media.

We then discussed the problems of service 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 AS^{2}s the approach may contribute to web interoperability, but should be linked more tightly to web application development methods. With respect to data warehouses, AS^2 integration and composition will contribute to web warehousing, but this also has to be investigated in more detail.

With respect to media types and meme media, we emphasised one particular difference. While our theory captures data and functionality in a very satisfactory way, the chosen abstract approach does not integrate aspects of presentation, which, however, are available with media types by means of presentation options, and are an essential feature of meme media. We believe that AS^{2}s can be extended to capture also presentation aspects, which was beyond the scope of this article and has to be addressed by follow-on research. The same applies to granularity and adaptivity, for which media types provide the concepts of hierarchical versions and cohesion, whereas AS^{2}s do yet take care of these nice properties.

With respect to service-oriented architectures in general and web-services in particular, research also addressed problems of which services to select, in particular with respect to optimisation of expressiveness, performance or costs. These kinds of problems have not yet been addressed in this article, but will be in follow-on research. For this, the formal framework of AS^{2}s will help defining load and cost models, i.e. AS^{2}s can form a basis for research in this direction.

References

1. Abiteboul S, Buneman P, Suciu D (2000) Data on the Web: from relations to semistructured data and XML. Morgan Kaufmann Publishers, San Francisco
2. Abiteboul S, Kanellakis PC (1998) Object identity as a query language primitive. J ACM 45(5):798–842
3. Altenhofen M, Börger E, Lemcke J (2005) An abstract model for process mediation. In: Lau K-K, Banach R (eds) Formal methods and software engineering, 7th international conference on formal engineering methods (ICFEM 2005) Lecture Notes in Computer Science, vol 3785. Springer, pp 81–95
4. Alves A, et al (2007) Web services business process execution language, version 2.0. OASIS Standard Committee, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
5. Beeri C, Milo T, Ta-Shma P (1996) On genericity and parametricity (extended abstract). In: PODS '96: proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems (New York, NY, USA), ACM, pp 104–116
6. Beeri C, Thalheim B (1999) Identification as a primitive of data models. In: Polle T, Ripke T, Schewe K-D (eds) Fundamentals of information systems. Kluwer, Boston pp 19–36
7. Benatallah B, Casati F, Toumani F (2006) Representing, analysing and managing web service protocols. Data Knowl Eng 58(3): 327–357
8. Binemann-Zdanowicz A, Thalheim B (2003) Modeling information services on the basis of ASM semantics. In: Börger E, Gargantini A, Riccobene E (eds) Abstract state machines. Lecture notes in computer science, vol 2589. Springer, Berlin, pp 408–410
9. Blass A, Gurevich J (2003) Abstract state machines capture parallel algorithms. ACM Trans Comput Logic 4(4):578–651
10. Börger E (2007) Modeling workflow patterns from first principles. In: Parent C, Schewe K-D, Storey V, Thalheim B (eds) Conceptual modeling—ER 2007. Lecture notes in computer science, vol 4801. Springer, Berlin pp 1–20
11. Börger E, Stärk R (2003) Abstract state machines. Springer, Berlin
12. Brenner MR, Unmehopa MR (2007) Service-oriented architecture and web services penetration in next-generation networks. Bell Labs Tech J 12(2):147–159
13. Christensen E et al (2001) Web services description language (WSDL) 1.1 http://www.w3c.org/TR/wsdl
14. Cox W et al (2004) Web services transaction (WS-Transaction). BEA Systems, IBM, Microsoft, http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html
15. Feingold W, Jeyaraman R (2007) Web services coordination (WS-Coordination), version 1.1, 2007. OASIS Web Services Transaction WS-TX TC, http://docs.oasis-open.org/ws-tx/wstx-wscoor1.1-spec.pdf
16. Goldfarb CF, Prescod P (1998) The XML handbook. Prentice Hall, New Jersey
17. Gómez J, Cachero C, Pastor O (2003) Modelling dynamic personalization in web applications. In: Third international conference on web engineering—ICWE 2003. LNCS, vol 2722. Springer, pp 472–475
18. Grädel E, Gurevich Y (1995) Metafinite model theory. In: LCC '94: selected papers from the international workshop on logical and computational complexity. Springer, London, pp 313–366
19. Gurevich J (2000) Sequential abstract state machines capture sequential algorithms. ACM Trans Comput Logic 1(1):77–111

20. Gurevich Y, Tillmann N (2005) Partial updates. Theor Comput Sci 336(2–3):311–342
21. Hegner SJ (2008) Information-optimal reflections of view updates on relational database schemata. In: Hartmann S, Kern-Isberner G (eds) Foundations of information and knowledge systems—5th international symposium (FoIKS 2008). Lecture Notes in Computer Science, vol 4932. Springer, pp 112–131
22. Kumaran S et al (2007) Using a model-driven transformational approach and service-oriented architecture for service delivery management. IBM Syst J 46(3):513–530
23. Ma H, Schewe K-D, Thalheim B, Wang Q (2008) Abstract state services. In: Song I-Y et al (eds) Advances in conceptual modeling—Challenges and opportunities, ER 2008 workshops. LNCS, vol 5232. Springer, pp 406–415
24. Papazoglou MP, van den Heuvel W-J (2007) Service oriented architectures: approaches, technologies and research issues. VLDB J 16(3):389–415
25. Schewe K-D, Schewe B (2000) Integrating database and dialogue design. Knowl Inf Syst 2(1):1–32
26. Schewe K-D, Thalheim B (1993) Fundamental concepts of object oriented databases. Acta Cybernetica 11(4):49–84
27. Schewe K-D, Thalheim B (2005) The co-design approach to web information systems development. Int J Web Inf Syst 1(1):5–14
28. Schewe K-D, Thalheim B (2005) Conceptual modelling of web information systems. Data Knowl Eng 54(2):147–188
29. Schewe K-D, Thalheim B (2006) Component-driven engineering of database applications. In: Stumptner M, Hartmann S, Kiyoki Y (eds) Conceptual modelling 2006—third Asia-Pacific conference on conceptual modelling (APCCM 2006). CRPIT, vol 53. Australian Computer Society, pp 105–114
30. Schewe K-D, Thalheim B (2007) Personalisation of web information systems—a term rewriting approach. Data Knowl Eng 62(1):101–117
31. Schewe K-D, Thalheim B, Wang Q (2009) Customising web information systems according to user preferences. World Wide Web 12(1):27–50
32. Schewe K-D, Wang Q (2008) A customised ASM thesis for database transformations (submitted for publication)
33. Tanaka Y (2003) Meme media and meme market architectures. IEEE Press, USA
34. Thalheim B (2000) Entity-relationship modeling: foundations of database technology. Springer, Berlin
35. Wang Q, Schewe K-D (2007) Axiomatization of database transformations. In: Proceedings of the 14th international ASM workshop (ASM 2007) University of Agder, Norway
36. Zhao J, Ma H (2006) ASM-based design of data warehouses and on-line analytical processing systems. J Syst Softw 79(5):613–629