Algebraic Structure of Web Services

Peter Höfner[1] Florian Lautenbacher[2]

Institute of Computer Science, University of Augsburg, Germany

Abstract

Web Services and Service-Oriented Architecture in general are promising concepts to overcome difficulties such as heterogeneity, scalability, etc. In this paper we present an algebraic structure of Web Services which assist users in Web Service composition and formal description of their services. Using relation algebra, tests and iteration offer the possibility of an automatic composition of Web Services based on a specified goal.

Keywords: Web Service; Composition; Relation; Semiring

1 Introduction

The defiances that companies nowadays have to meet have affected their infrastructure and resulted in a need for more loosely-coupled components in distributed heterogeneous environments. The Service-Oriented Architecture (SOA) represents an approach that facilitates this loose coupling while at the same time providing sufficient quality of service necessary for acceptable solutions. Web Services are one possibility to fulfil the requirements of a service-oriented computing world. The W3C defines a Web Service as a software system designed to support interoperable machine-to-machine interaction over a network [7]. This definition encompasses many different systems, but in common usage the term refers to those services that use SOAP-formatted [6] XML envelopes and have their interfaces described by the Web Services Description Language (WSDL) [22]. With standards like WSDL and the Web Service Business Process Execution Language (WSBPEL) [14], both version 2.0, one can describe the data types, messages and flow of processes to model not only a simple Web Service but also the composition (or orchestration) of several Web Services. But this composition is still mostly done manually. There are already first (non-algebraic) approaches to use AI-based planners or different algorithms to achieve an orchestration. On the other hand, formal algebras for Web Services exist, but most of them are not used for Web Service composition so far.

[1] Email: hoefner@informatik.uni-augsburg.de

[2] Email: lautenbacher@informatik.uni-augsburg.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

In this paper we present an algebraic structure for Web Services. It is used for characterising Web Service composition and to determine inputs and outputs of Web Services. The theoretical aspects and definitions are illustrated by a running example in order to assist the readers' understanding. We try to keep the theory simple and to focus on its application.

We start with binary relations and relation algebras which have many applications in mathematics and computer science (e.g. [24,10]); they are well known and provide a rich theory. This paper shows that Web Services are another application.

It is structured as follows: In the next section we describe related work on formal approaches on Web Services as well as on Web Service composition. In Section 3 we define Web Methods and Web Services based on relational algebras and show the benefits in a running example. We go into further detail about the algebraic structure in Section 4. Section 5 defines the composition of Web Services on the algebraic context. We establish the concept of Web Service restriction in Section 6. This paves the way to characterise preconditions and goals as elaborated in more detail in Section 7. Before presenting a small case study in Section 9 we recapitulate the concept of iteration. We conclude with an outlook on on-going and future work. In particular, we sketch how to determine the execution order of Web Services.

2 Related work

There are plenty of approaches to Web Service composition. A composition can be achieved agent-based (as in [17,5]), based on interaction protocols [25], symbolic transition systems [23] or based on some kind of logic [8]. Very often process algebras or Petri nets are used, too. The semantic web community has used planning techniques to address the problem of automated composition of semantic Web Services, e.g. based on OWL-S [18] descriptions of input/output/precondition and effect. In [19] SHOP2, a hierarchical task network (HTN) planner, is employed for Web Service composition. The HTN planner creates workflows by task decomposition.

In [9] a composition algebra is defined which covers inputs and outputs of a Web Service and is based on CCS [20] and CSP [11]. It regards choices, parallel processes and synchronisation. This process algebra solves the composition problem which is generally addressed as finding a composite process showing a requested behaviour. The authors first perform a top-down behaviour decomposition and afterwards a bottom-up process composition and provide an algorithm for the composition of Web Services. Using our algebraic structure one does not need algorithms for the composition anymore, but the composition is automatically inferred via the algebra.

In [8] Web Services are defined based on service nets as a subclass of Petri nets. The created Web Service algebra includes empty services, sequences, choices, iterators, parallel constructs and more advanced workflow patterns like discriminators. Desired properties of that service algebra are described, but the aspect of Web Service composition is not considered. It describes the formal semantics and algebraic properties of single services and the (existing) orchestration of services. Also, it includes advanced workflow patterns, but the composition of services needs to be predefined and can not be inferred through the algebra automatically.

3 Towards a Formalisation of Web Services

In this section we develop an algebraic characterisation of Web Services and present also an algebraic definition of Web Service composition. Obviously, a Web Service consists of an interface and the implementation. In the interface (described in WSDL) several Web Methods are defined. These receive input messages and reply with output messages which both can be of a simple type such as string, integer, etc. or of a complex type. For our first formalisation of Web Services we assume that both, the types of the input data and the output data are known before and therefore there is a *knowledge set* $\mathcal{K}$: a set which includes the input and the output as subsets. Types which might be nested or semantically described are topics for further research and also the concrete binding and port information of Web Services are currently neglected for the sake of simplicity.

Definition 3.1 A *Web Method* is a tuple (I, O), where $\mathcal{K}$ is a knowledge set and $I \subseteq O \subseteq \mathcal{K}$.

The condition $I \subseteq O$ guarantees that we do not lose any information, i.e., any information which is known before the execution of a Web Method is also known afterwards. Mathematically, a Web Method is an *ordered pair*. In the definition, I denotes the set of all data needed by the Web Method. If all input is provided, an execution of the Web Method will produce all data which are given in the set O. Otherwise this specification means that if one element of I is missing, the Web Method cannot be executed and therefore no output is produced. $(\emptyset, O)$ [3] represents a Web Method where no input is needed, i.e., it can be executed at any time.

Due to readability we want to avoid the repetition of the input data in the output as well as the brackets. Therefore we use a grammar-style notation. In particular

$$i_1\ i_2\ \ldots\ i_n \to o_1\ o_2 \ldots\ o_m \Leftrightarrow_{df} (\{i_1, i_2 \ldots, i_n\}, \{i_1, i_2 \ldots, i_n\} \cup \{o_1, o_2 \ldots, o_m\})\,.$$

Like in grammars a choice of rules $u \to v$ and $u \to w$ is abbreviated by $u \to v \,|\, w$. Furthermore a choice of rules $u \to w$ and $v \to w$ is denoted by $u \,|\, v \to w$. If the left hand side or the right hand side of the production rule is empty, i.e., $I = \emptyset$ or $O \backslash I = \emptyset$ $(I = O)$, we write $\emptyset \to o_1 \ldots\ o_m$ and $i_1\ \ldots\ i_n \to \emptyset$ respectively.

Running Example Booking a flight is a very simple example of a Web Method. A customer needs (at least) the date of arrival , the airport of departure, the destination and his credit card number. By using the abbreviations `a`, `dep`, `des`, `cc` for the above information and `etix` for an electronic ticket which is issued during the execution of the Web Method, we get

$$\texttt{flight}_{\texttt{cc}} =_{df} \texttt{a dep des cc} \to \texttt{etix}\,.$$

By definition, this is the same as $(\{\texttt{a}, \texttt{dep}, \texttt{des}, \texttt{cc}\}, \{\texttt{a}, \texttt{dep}, \texttt{des}, \texttt{cc}, \texttt{etix}\})$. □

A Web Method is the simplest form of a Web Service; but it contains neither choice nor does it offer a straightforward composition operation. To eliminate the former deficiency, we define a *simple Web Service*.

[3] The symbol $\emptyset$ denotes the empty set w.r.t. the knowledge set $\mathcal{K}$, whereas we will use $\varnothing$ to denote empty sets w.r.t. other sets.

Definition 3.2 A *simple Web Service* is a collection of Web Methods.

Running Example For our example we now assume that the customer who wants to book a flight has the choice of using his credit card number or his frequent flier card instead (`ff` for short). To characterise the choice we model two different Web Methods $\texttt{flight}_{\texttt{cc}}$ and $\texttt{flight}_{\texttt{ff}}$, where $\texttt{flight}_{\texttt{cc}}$ is defined as above and $\texttt{flight}_{\texttt{ff}} =_{df} \texttt{a dep des ff} \rightarrow \texttt{etix}$. The simple Web Service for booking a flight is then defined as

$$\texttt{flight} =_{df} \{\texttt{flight}_{\texttt{ff}}, \texttt{flight}_{\texttt{cc}}\}\,.$$

By this, we have the choice between different Web Methods. □

Since we assumed that the input I as well as the output O are subsets of $\mathcal{K}$, a simple Web Method becomes a binary relation on $\mathcal{P}(\mathcal{K})$. Hence the algebraic structure of binary relations under union and sequential composition is also interesting. In Section 4 we will give its exact definition.

With the embedding of Web Services into the framework of relations we can now take advantage of all the mathematical background. For example there are two operations on relations, choice and sequential composition. The former one is just the set-theoretic union and describes the choice between Web Methods or simple Web Services, respectively. The sequential composition of two ordered pairs (r, s) and (t, u) is defined by

$$(r, s)\,;(t, u) =_{df} \begin{cases} (r, u) & \text{if } s = t \\ \text{undefined} & \text{otherwise}\,. \end{cases}$$

This composition can be lifted pointwise to a composition of relations. The composition of two relations R and S is defined as

$$R\,;S =_{df} \{r\,;s \mid r \in R,\, s \in S,\, r\,;s \text{ defined}\}\,.$$

Note, that by definition $r\,;s$ is defined only if the output set of r has the same size as the input set of s. Unfortunately, this definition yields a strange behaviour in the setting of Web Services, which is illustrated by the following example. Afterwards we will present a possible solution for this deficit.

Running Example Assuming that the customer does not only want to book a flight but also a hotel room. Therefore, we introduce a Web Service

$$\texttt{hotel} =_{df} \{\texttt{a d cat} \rightarrow \texttt{res}\}\,,$$

where `a` and `d` denotes the date of arrival and departure (we assume that `a` is exactly the same as the day of the flight), `cat` describes the room's category and `res` stands for a reservation number which is given by the hotel after a successful booking.

Intuitively, the result of booking both, a flight and a hotel room *should* be

$$\{\texttt{S cc} \mid \texttt{S ff} \mid \texttt{S cc ff} \rightarrow \texttt{etix res}\}\,,$$

where `S` = `a d dep des cat` is the common set of all input data. Informally this means that if the customer has enough input data he is able to book both a flight

and a hotel. In particular, he needs his credit card, his frequent flier card, or both. But the ordering of booking a flight and booking a hotel should not matter.

Relational composition of `flight` and `hotel` yields an empty set, since the output of `flight` does not match the input of `hotel` and vice versa. □

The problem is, that executing the second Web Method needs more information than the first one provides. Vice versa the second Web Method cannot be executed if the first one provides to many information, like `etix` in the above example. To bridge this gap, we define a Web Service as a collection of Web Methods which is based on a simple Web Service with additional information, which remains untouched during execution of the Web Service and is just added to the output data.

Definition 3.3 Consider a knowledge set $\mathcal{K}$. The *(extended) Web Service* of a simple Web Service W is the relation $\{(I \cup E, O \cup E) : (I, O) \in W, E \subseteq \mathcal{K} \backslash O\}$ and denoted by $\ll W \gg$ [4].

In this definition E is the context and the extension of the simple Web Service W, which just takes any information that is not needed as input for execution and adds this information unchanged to the output. Obviously, each element of a Web Service is again a Web Method. Moreover, the definition implies the following result if $\ll . \gg$ is seen as a function:

Lemma 3.4 *$\ll . \gg$ is additive and idempotent, i.e., $\ll V \cup W \gg = \ll V \gg \cup \ll W \gg$ and $\ll \ll V \gg \gg = \ll V \gg$ for Web Services V and W.*

All proofs can be found in [12]. As a consequence of this lemma, $\ll . \gg$ is also strict, i.e., $\ll \varnothing \gg = \varnothing$.

Running Example Let $\mathcal{K} = \{\mathtt{a}, \mathtt{d}, \mathtt{cat}, \mathtt{res}, \mathtt{dep}, \mathtt{des}\}$. The Web Service based on `hotel` is

$$\ll \mathtt{hotel} \gg = \{\mathtt{S} \mid \mathtt{S\ dep} \mid \mathtt{S\ des} \mid \mathtt{S\ dep\ des} \rightarrow \mathtt{res}\},$$

where $\mathtt{S} = \mathtt{a\ d\ cat}$. The second Web Method hands over information about departure (`dep`), the third information about destination (`des`) and the last one `dep` and `des`. □

Now, we can use the standard sequential composition of relations to formalise the desired behaviour and to define *Web Service composition* in a formal way.

Definition 3.5 Consider a knowledge set $\mathcal{K}$ and two (simple) Web Services V and W over $\mathcal{K}$. The *Web Service composition* of V and W, is defined as

$$V \circ W =_{df} \ll V \gg ; \ll W \gg .$$

Running Example Determining the Web Service composition of the simple Web Services `flight` and `hotel` over $\{\mathtt{a}, \mathtt{d}, \mathtt{dep}, \mathtt{des}, \mathtt{cat}, \mathtt{cc}, \mathtt{ff}, \mathtt{etix}, \mathtt{res}\}$ yields

$$\mathtt{flight} \circ \mathtt{hotel} = \ll \mathtt{flight} \gg ; \ll \mathtt{hotel} \gg = \{\mathtt{S\ cc} \mid \mathtt{S\ ff} \mid \mathtt{S\ cc\ ff} \rightarrow \mathtt{etix\ res}\},$$

[4] When possible, we will skip the set-brackets of W for readability

where $\mathtt{S} = \mathtt{a\ d\ dep\ des\ cat}$ is again the set of common knowledge of all involved Web Methods. Furthermore we get $\mathtt{flight} \circ \mathtt{hotel} = \mathtt{hotel} \circ \mathtt{flight}$. This is exactly the desired behaviour (see above). □

Before discussing some basic properties of Web Service composition in Section 5, we will now set up the theoretical background.

As we will see in the next section, the use of algebra offers abstraction from relations and set theory. One advantage is that it masks all the set-theoretic notation (like brackets) and concentrates on the interesting aspects.

4 Algebraic Structure

As already shown, Web Services can be interpreted as relations. The corresponding abstract algebraic structures of relations are idempotent semirings, which we will discuss in this section.

Definition 4.1 A *semiring* is a quintuple $(S, +, 0, \cdot, 1)$ such that $(S, +, 0)$ is a commutative monoid and $(S, \cdot, 1)$ is a monoid such that $\cdot$ distributes over $+$ and 0 is an annihilator. Concretely, we have the following axioms for semirings.

- additive monoid: $a + (b + c) = (a + b) + c \quad \text{and} \quad a + 0 = a = 0 + a\,,$
- commutativity: $a + b = b + a\,,$
- multiplicative monoid: $a \cdot (b \cdot c) = (a \cdot b) \cdot c \quad \text{and} \quad a \cdot 1 = a = 1 \cdot a\,,$
- distributivity: $a \cdot (b + c) = a \cdot b + a \cdot c \quad \text{and} \quad (a + b) \cdot c = a \cdot c + b \cdot c\,,$
- annihilation: $a \cdot 0 = 0 = 0 \cdot a\,.$

On an *idempotent* semiring (or *i-semiring*) addition is idempotent, i.e., $a + a = a$. In the setting of i-semirings the relation $a \leq b \Leftrightarrow_{df} a + b = b$ is a partial order, i.e., a reflexive, antisymmetric and transitive relation, called the *natural order* on S. It has 0 as its least element. Moreover, $+$ and $\cdot$ are isotone with respect to $\leq$.

It is straightforward that the algebra of binary relations over a set $\mathcal{K}$

$$\mathsf{REL}(\mathcal{K}) =_{df} (\mathcal{P}(\mathcal{K} \times \mathcal{K}), \cup, \varnothing, ;, \Delta)\,,$$

where $\Delta = \{(x, x) : x \in \mathcal{K}\}$ is the identity relation, forms an i-semiring. More details about (idempotent) semirings and examples of their relevance to computer science can e.g. be found in [2].

This structure allows us to express Web Service composition and the choice between Web Services in an abstract way. There are some special elements which have to be discussed: Δ is the Web Service which does nothing than to hand over all input data. From a semantic point of view it can be seen as `skip`. $\varnothing$ is an "improper" Web Service, due to the annihilation laws it stops every calculation and can therefore be seen as `abort`. The last special element is $\mathtt{magic} = \mathcal{K} \times \mathcal{K}$, the Web Service that can do anything.

It is also straightforward to show that $\mathsf{REL}(\mathcal{K})$ forms a *relation algebra* (e.g. [24]) and therefore can be equipped by additional operations. e.g., for calculating the converse. But in the setting of Web Services converse means to undo an already

executed Web Services. Since we do not want such a behaviour, we calculate in the more abstract setting of i-semirings.

5 Web Service Composition

We have already defined Web Service composition in the context of binary relations (cf. Definition 3.5). The composition for two Web Services V and W was to extend each element and then use relational composition, i.e.,

$$V \circ W = \ll V \gg ; \ll W \gg \ .$$

Using the algebraic structure of the previous section we can now derive basic properties of Web Service composition.

Since Web Service composition is defined in terms of sequential composition of relations and therefore in terms of multiplication of i-semirings in general, we get

Corollary 5.1 *Web Service composition is associative and distributes over choice.*

Associativity follows by associativity of relations and $\ll V \circ W \gg = V \circ W$. The distributivity law follows from Lemma 3.4. These properties are of course necessary for Web Service composition; but in contrast to other approaches there is no need to add these as axioms, since they can be derived in our setting.

Lemma 5.2 *For arbitrary Web Services V and W, the composed Web Service $V \circ W$ is again an extended Web Service, i.e. there is a simple Web Service X with $\ll X \gg = V \circ W$.*

In particular, $V \circ W$ is again a Web Service. Note that if $V \circ W = W \circ V$, then the two Web Services can be executed in parallel (when neglecting possible dependencies on some shared resources).

Running Example Let us expand the above example by a "planning the trip" Web Service. Therefore we assume that there is a simple Web Service which collects all necessary information, but needs no input data at all.

$$\mathtt{plan} =_{df} \{\emptyset \to \mathtt{S\ cc} \mid \mathtt{S\ ff} \mid \mathtt{S\ cc\ ff}\} \ ,$$

where $\mathtt{S} = \mathtt{a\ d\ dep\ des\ cat}$. In fact there are three different outputs depending on the information on the credit and the frequent flier card. (The customer has to specify at least one.) Furthermore let $\mathcal{K} =_{df} \{\mathtt{a}, \mathtt{d}, \mathtt{dep}, \mathtt{des}, \mathtt{cc}, \mathtt{ff}, \mathtt{cat}, \mathtt{etix}, \mathtt{res}\}$ be the knowledge set. In the remainder we denote the Web Services $\ll\mathtt{flight}\gg$, $\ll\mathtt{hotel}\gg$ and $\ll\mathtt{plan}\gg$ by $\mathtt{f}$, $\mathtt{h}$ and $\mathtt{p}$, resp. Composing these Web Services yields

$$\mathtt{p\,;h\,;f} = \mathtt{p\,;f\,;h} = \{\emptyset \to \mathtt{S\ P\ cc\ ff} \mid \mathtt{S\ P\ cc} \mid \mathtt{S\ P\ ff},\quad \mathtt{cc} \mid \mathtt{ff} \to \mathtt{S\ P\ cc\ ff}\} \ ,$$

$$\mathtt{f\,;p\,;h} = \mathtt{f\,;h\,;p} = \mathtt{h\,;p\,;f} = \mathtt{h\,;f\,;p} = \varnothing \ ,$$

where $\mathtt{S} = \mathtt{cat\ a\ d\ dep\ des}$ is the set of data which has to be collected by the Web Service under all circumstances and $\mathtt{P} = \mathtt{etix\ res}$ is the set of data produced by the Web Services after successful execution. The composed Web Services in the

last line coincide with the empty service, since `p` "produces" knowledge which is already needed by `f` or `h` before and therefore yields a conflict. For example, after the execution of `f` the date of departure (`dep`) is known, since this is in the output set. But then it is not possible that the Web Service for planning (`p`) specifies this date. This conflict yields an abortion and an empty result set. Such a behaviour seems quite natural for us, since otherwise the customer would for example be able to change the date for travelling after booking the hotel and before booking the flight. Note if we add more elements to the knowledge set $\mathcal{K}$ then $\mathtt{p}\,;\mathtt{h}\,;\mathtt{f}$ contains even more elements. □

6 Web Service Restriction

Our aim is not only to characterise Web Service composition but also algebraic notions to express the needed input data to perform a certain action or to use assertions to guarantee a certain knowledge.

Therefore, we introduce the concept of tests and will show lateron how to use them in modal operators in order to search for Web Services that achieve a specified goal and detect the data that is needed to invoke these Web Services.

Running Example We assume the Web Service $\mathtt{flight} \cup \mathtt{hotel}$ which either books a hotel room or a flight. To test a successful booking (if a customer has an e-ticket at the end of execution) we use the term $\ll\mathtt{flight} \cup \mathtt{hotel}\gg\,;\ll\mathtt{etix} \to \emptyset\gg$. □

In REL a test can be modelled as a subrelation of the identity relation; meet and join of such partial identities coincide with their composition and union. Generalising this, one defines a *test* in an i-semiring [16] to be an element $p \leq 1$ that has a complement q relative to 1, i.e., $p + q = 1$ and $p \cdot q = 0 = q \cdot p$. The set of all tests of an i-semiring S is denoted by $\mathtt{test}(S)$. It is not hard to show that $\mathtt{test}(S)$ is closed under $+$ and $\cdot$ and has 0 and 1 as its least and greatest elements. Moreover, the complement $\neg p$ of a test p is uniquely determined by the definition. Hence $\mathtt{test}(S)$ forms a Boolean algebra. In the remainder we will consistently write $a, b \ldots$ for arbitrary semiring elements and $p, q, \ldots$ for tests. Furthermore, we freely use the Boolean laws for tests; e.g. an important property is

$$p \cdot a \cdot q \leq 0 \;\Leftrightarrow\; a \cdot q \leq \neg p \cdot a\,. \tag{1}$$

With the above definition of tests we deviate slightly from [16], in that we do not allow an arbitrary Boolean algebra of subidentities as $\mathtt{test}(S)$ but only the maximal complemented one. The reason is that the axiomatisation of modal operators, presented below, forces this maximality anyway (see [4]).

Running Example Assume a user who has not executed a Web Service and who wants to plan a trip (see above). Therefore he has not specified any data before the execution of the Web Services. But, $\mathtt{p}\,;\mathtt{h}\,;\mathtt{f}$ can also contain Web Methods which start with some data. To distinguish Web Services with no input, we insert a test $t =_{df} \{(\emptyset, \emptyset)\}$ at the beginning of the calculation. Since the result is not empty for

$$t\,;\mathtt{p}\,;\mathtt{f}\,;\mathtt{h} \quad \text{and} \quad t\,;\mathtt{p}\,;\mathtt{h}\,;\mathtt{f}$$

the execution of the Web Services in this order yields a positive result (a hotel room and a flight is booked). Note that we use t and not the extended Web Service $\ll t \gg$, since we want to guarantee that an execution starts without any input. Usually, one has to use the extended one if a test occurs inside an execution (see above). □

Of course similar to a restriction at the beginning of a calculation, we can also use tests to enforce particular results (e.g., $a \cdot p$) or to enforce knowledge inside a computation or intermediate results (e.g., $a \cdot p \cdot b$). Since tests do not determine any new data, they contain only Web Methods of the form $i_1 \ \ldots \ i_n \to \emptyset$.

By the above examples, we have seen that tests form sets of possible current information or sets of possible configurations.

In Section 7 we show another criterion to guarantee a positive result. For this we further need an additional property about extended Web Services.

Lemma 6.1 *Any extended Web Service which is also a test in the i-semiring can be split into parts. That is, for any subsets X, Y, Z of a given knowledge set $\mathcal{K}$ with $X = Y \cup Z$ we have*

$$\ll(X,X)\gg = \{(Y,Y)\} \circ \{(Z,Z)\} = \ll(Y,Y)\gg \ ; \ll(Z,Z)\gg \ .$$

Since on tests ; coincides with meet, informally the lemma describes that the test "X holds" can be replaced by two tests, namely "Y holds" and "Z holds".

Running Example Given the test $\ll \texttt{etix res} \to \emptyset \gg$. By the above lemma this Web Service can be split into $\ll \texttt{etix} \to \emptyset \gg \ ; \ll \texttt{res} \to \emptyset \gg$. □

7 Preconditions and Modal Operators

As we have seen, tests can be used to model assertions for Web Services. But they are also the basis for defining modal operators [3] which are used for modelling termination and an abstract version of the wlp-operator [21]. The resulting formalism is similar to propositional dynamic logic but also strongly related to temporal logics. In this section we discuss these operators with respect to Web Services. In particular we show how to determine necessary information which has to be specified by a customer when a certain goal is given, e.g. to receive an e-ticket in the end.

Definition 7.1 An i-semiring S is called *modal* [3] if it can be endowed with a total (forward) box operation $|a] : \texttt{test}(S) \to \texttt{test}(S)$, for each $a, b \in S$, that satisfies

$$p \leq |a]q \Leftrightarrow p \cdot a \cdot \neg q \leq 0 \quad \text{and} \quad |a \cdot b]p = |a](|b]p) \, .$$

A (forward) diamond is defined as the de Morgan dual of the box; $|a\rangle p =_{df} \neg|a]\neg p$.

Informally, in the context of Web Services, $|a\rangle p$ characterises the set of possible information with at least one successor in p when executing the Web Service a, i.e., the preimage of the set p under a. $|a]p$ characterises the situation where there is no execution of a, that starts in p and terminates in $\neg q$. Using Equation (1) shows that whenever an execution of a terminates in $\neg q$, the execution has to start in $\neg p$ and therefore $|a]p$ models the possible infomation from which execution of a is guaranteed to terminate in an element of p or the execution is not possible. Formally,

in REL and also in Web Services, one has $(x,x) \in |R]q \;\Leftrightarrow\; (\forall y : xRy \Rightarrow (y,y) \in q)$. Furthermore, in [21] it is shown that the box operator coincides with the wlp-operator, i.e., $\mathsf{wlp}(a,p)q = |a]q$.

For a better understanding let us have a look at our running example.

Running Example A customer needs an electronic ticket and a reservation number for a successful booking. Therefore the aim after execution is to reach the set $q =_{df} \ll \texttt{etix res} \to \emptyset \gg$. This example determines all elements which either yield no execution or, if an execution exists, it leads to a successful booking of a hotel and a flight. Determining $|\texttt{f} \,;\, \texttt{h}]q$ yields in total 512 elements.

Let us have a closer look at the results. There are a lot of elements like $\texttt{res} \to \emptyset$ for which an execution of $\texttt{f} \,;\texttt{h}$ yields an abortion and the result is the empty set. On the other hand there are elements like $\texttt{a d cc des dep cat} \to \emptyset$ where the execution of the Web Services `flight` and `hotel` yields the desired result. □

In contrast to the box operator $|a\rangle q$ is characterised by $(x,x) \in |R\rangle q \;\Leftrightarrow\; \exists (y,y) \in q : xRy$ in REL and Web Services.

Running Example $|\texttt{f} \,;\, \texttt{h}\rangle q$ determines all possible starting configurations that have at least one successful execution path, i.e, there is at least one possibility of execution where the involved Web Services yield an e-ticket and a reservation number. □

The combination of both operators guarantees that at least one result of the Web Service a exists and all resulting information is in p. This is expressed by

$$|a\rangle p \cdot |a]p \,.$$

Running Example Determining $|\texttt{f} \,;\, \texttt{h}\rangle q \,;\, |\texttt{f} \,;\, \texttt{h}]q$ with $q =_{df} \ll \texttt{etix res} \to \emptyset \gg$ yields indeed the desired result; namely exactly the information which is needed to use both Web Services:

$$|\texttt{f} \,;\, \texttt{h}\rangle q \,;\, |\texttt{f} \,;\, \texttt{h}]q = \{\texttt{S cc} \,|\, \texttt{S ff} \,|\, \texttt{S cc ff} \to \emptyset\} \;,$$

where $\texttt{S} = \texttt{cat a d dep des}$ is again the set of data which has to specified by the user under all circumstances. Interpreting that result we now know that the customer has to give next to `S` either his credit card number (`cc`), the number of his frequent flier card (`ff`) or both. □

Of course this result is not a surprise since we constructed the Web Services in exactly that way, but since the modal operators can be applied to any Web Services they can be used to determine the necessary data. For this purpose it is useful to provide some basic laws for boxes and diamonds. All the presented laws as well as many more can be found in [4].

Lemma 7.2 *For elements $a \in S$ and $p, q \in \texttt{test}(S)$*

$$\begin{aligned} |a](p \cdot q) &= |a]p \cdot |a]q, & |a\rangle(p+q) &= |a\rangle p + |a\rangle q, \\ |a+b]p &= |a]p \cdot |b]p, & |a+b\rangle p &= |a\rangle p + |b\rangle p, \\ |p]q &= \neg p + q, & |p\rangle q &= p \cdot q, \\ & & |a \cdot b\rangle p &= |a\rangle |b\rangle p \end{aligned}$$

The first line explains how to decompose a Web Service if the goal ($p \cdot q$ or $p + q$) can be split. The second line splits the Web Service itself, the third row calculates the test if the execution step is a test itself. The last row shows that the diamond satisfies the same law for composition as the box operator.

Running Example We want to determine the necessary information to receive an e-ticket and a reservation number after the execution of a Web Service $W = \texttt{flight} \circ \texttt{hotel}$ (provided execution is possible). Therefore we have to determine

$$|W] \ll \texttt{etix res} \to \emptyset \gg \ .$$

By Lemma 6.1 and the first equation of Lemma 7.2 this expression is equivalent to $|W] \ll \texttt{etix} \to \emptyset \gg \ ; \ |W] \ll \texttt{res} \to \emptyset \gg$. By simple calculations this is the same as $|\ll \texttt{flight} \gg] \ll \texttt{etix} \to \emptyset \gg \ ; \ |\ll \texttt{hotel} \gg] \ll \texttt{res} \to \emptyset \gg$. This shows that the calculation can be splitted into a part for booking the flight and one for booking the hotel. □

Obviously, the splitting rules of Lemma 6.1 and 7.2 cannot be applied in each situation. In particular if a single Web Service produces two dependent results, the splitting is not useful.

Note, that backwards boxes $[a|p$ and diamonds $\langle a|p$, which describes all possible ending states of an element, is easily defined as a domain operator in the opposite semiring (i.e., the one that swaps the order of composition).

In particular, the backward box is defined by $p \leq [a|q \Leftrightarrow \neg q \cdot a \cdot p \leq 0$ and $[a \cdot b|p = [b|([a|p)$. The backward diamond is again defined as the de Morgan dual $\langle a|p =_{df} \neg[a|\neg p$.

Using backwards modal operators one can now characterise goals for Web Services instead of preliminaries.

Running Example Usually, a system starts with no information. Therefore consider the test $t =_{df} \{(\emptyset, \emptyset)\}$. Then $[\texttt{p} \ ; \texttt{h} \ ; \texttt{f}|t \ ; \langle \texttt{p} \ ; \texttt{h} \ ; \texttt{f}|t$ yields the set of possible knowledge after successful execution of all three Web Services. □

8 Web Service Iteration

To round off the discussion about the algebraic structure of Web Services, we briefly discuss a possibility how to formalise the iteration of Web Services in the algebra setting without giving too many details. Obviously it seems useful to characterise an arbitrary, but finite number of iterations in the setting of Web Services.

To iterate a Web Services V twice, we can use the expression V^2. But how can we formalise an arbitrary iteration? We have to determine the reflexive and transitive closure of a Web Service which is expressed and denoted by the Kleene star *. Therefore the expression

$$V^* = \bigcup_{i \in \mathbb{N}} V^i$$

determines the desired behaviour.

Running Example Instead of giving the concrete order of the Web Services `p`, `f`

and h, one might say that the user is allowed to execute each Web Service which he wants in any ordering. Hence, $(\mathtt{p}+\mathtt{f}+\mathtt{h})^*$ would give all iterations where either the flight, the hotel or the overall planning would be executed first and then the others. The result of this iteration is skip (the element Δ) or p, f, h, pf, ph, fp, fh, hp, hf, pfh, phf, fph, fhp, hpf, hfp, ... [5] or any other combination of these services. It would also yield to the result with multiple occurrences of one single Web Service like pfp (or any similar), but this cannot be executed, i.e., $\mathtt{pfp} = \varnothing$. Moreover we have

$$(\mathtt{p} \cup \mathtt{f} \cup \mathtt{h})^* ; q = \mathtt{p} ; \mathtt{f} ; \mathtt{h} ; q \cup \mathtt{p} ; \mathtt{h} ; \mathtt{f} ; q = \mathtt{p} ; (\mathtt{f} \cup \mathtt{h}) ; q \,.$$

The last step is by distributivity. Therefore we now have the possible sequences that yield the desired result. Moreover, since

$$\mathtt{p} ; \mathtt{f} ; \mathtt{h} ; q = \mathtt{p} ; \mathtt{h} ; \mathtt{f} ; q = \mathtt{p} ; (\mathtt{f} \cup \mathtt{h}) ; q \tag{2}$$

we also know that f and h can be executed in parallel. □

We do not want to discuss this operation and structure (which is known as Kleene algebra) and its connection to Web Services. Instead we will present some longer examples in the next section. More details concerning the reflexive, transitive closure within relations can be found e.g. in [24], about Kleene algebra in [1,15].

9 Simple Case Study

We have implemented relations, Web Service composition, tests and modal operators in Haskell. With this implementation we have build up a small case study to show that using the presented theory is useful to determine information about Web Services. The Haskell code, the encodings of our examples and the result sets can be found at the web site of [12].

When planning a business trip, it is essential to know which data are necessary in order to book a flight, a hotel, etc. Since the used Web Services are mostly not known in advance, it would be nice if this could be computed. Imagine the following Web Service that needs the departure, the destination, the date of a flight and either a credit card number or a frequent flyer number to book a flight. The return value is an electronic ticket number for the booked flight. We want to book two flights: one to the destination des and additionally a return flight (getting etix and etix2):

$$\begin{aligned} \ll\mathtt{flight}\gg =_{df} \{&\mathtt{a\ dep\ des\ cc} \mid \mathtt{a\ dep\ des\ ff} \rightarrow \mathtt{etix}, \\ &\mathtt{d\ dep\ des\ cc} \mid \mathtt{d\ dep\ des\ ff\ des} \rightarrow \mathtt{etix2}\} \,. \end{aligned}$$

Hence, the expression

$$\neg(\mathtt{etix} \cup \mathtt{etix2}) ; |\mathtt{flight}^*\rangle(\mathtt{etix} ; \mathtt{etix2}) ; |\mathtt{flight}^*](\mathtt{etix} ; \mathtt{etix2})$$

would compute all input parameters that are possible on the knowledge set $\mathcal{K} = \{\mathtt{a}, \mathtt{d}, \mathtt{dep}, \mathtt{des}, \mathtt{cc}, \mathtt{ff}, \mathtt{etix}, \mathtt{etix2}, \mathtt{smt}\}$, where smt describes something additional.

[5] Due to readability we leave ; implicit.

The test $\neg(\texttt{etix} \cup \texttt{etix2})$ guarantees that the customer has not bought any ticket before the execution of the Web Service. The query yields six results: All include `a`, `d`, `dep` and `des` and (not surprisingly) all recombinations of `cc`, `ff` and `smt`. Since we know that two iterations of `flight` yield two tickets, the star in the above expression could be replaced by $\texttt{flight}^2$. Nevertheless, since normal users do not have such knowledge we modelled the desired behaviour with an arbitrary finite iteration.

But much more interesting are the input parameters that are necessary for different Web Services: what is needed in order to book two flights, a hotel and a car, getting the result of a reservation number of the hotel (`res`), of the car (`resnrc`) and of the two e-tickets? In addition to that we also want our car to be insured against accidents. On the extended knowledge set $\mathcal{K} \cup \{\texttt{cat}, \texttt{kind}, \texttt{resnrc}, \texttt{insurenr}, \texttt{res}\}$ with the additional Web Services for renting a car $\ll\texttt{car}\gg =_{df} \{\texttt{a d des cc} \to \texttt{resnrc}\}$ and for insuring the car $\ll\texttt{insure}\gg =_{df} \{\texttt{resnrc cc} \to \texttt{insurenr}\}$ the query

$$\neg(\texttt{etix} \cup \texttt{etix2} \cup \texttt{res} \cup \texttt{resnrc} \cup \texttt{insurenr});$$
$$|\langle(\texttt{flight} \cup \texttt{hotel} \cup \texttt{car} \cup \texttt{insure})^*\rangle(\texttt{etix} ; \texttt{etix2} ; \texttt{res} ; \texttt{resnrc} ; \texttt{insurenr});$$
$$|[(\texttt{flight} \cup \texttt{hotel} \cup \texttt{car} \cup \texttt{insure})^*](\texttt{etix} ; \texttt{etix2} ; \texttt{res} ; \texttt{resnrc} ; \texttt{insurenr})$$

will return this information whereas we already prevent getting results when the goal parameters are existing in advance. The query returns the following four results:

$$\{\texttt{S} \,|\, \texttt{S ff} \,|\, \texttt{S ff smt} \,|\, \texttt{S smt} \ \to \emptyset\},$$

where $\texttt{S} = \texttt{a}, \texttt{d}, \texttt{dep}, \texttt{des}, \texttt{cat}, \texttt{kind}, \texttt{cc}$.

The results show that using the given inputs (`S`) one can achieve the results using the Web Services `flight`, `hotel`, `car`, `insure`. More precisely, it can be shown that the Web Services can be executed in a given order (similar to Equation (2)). Collecting the inputs (`S`) in a Web Service $\texttt{plan} = \{\emptyset \to \texttt{S}\}$ the following process model would be the result of the computation (cf. Figure 1): At the beginning you need to plan the business trip, then you can book the hotel and the flight as well as the car and additionally an insurance for the car. The hotel booking, flight booking and car booking are independent from each other and therefore could be modelled in parallel, whereas the insurance depends on the car booking Web Service.

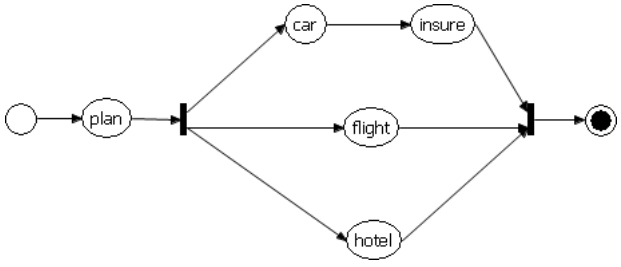

Fig. 1. Process model for the case study.

10 Conclusion and Outlook

In this paper we presented a first step towards an algebra of Web Services and showed how to make advantage of the resulting algebraic structures. In particular, Web Services can be embedded into the well-known structure of relations which by adding operations for composition and choice form an i-semiring. Henceforth we considered Web Services from a syntactical point of view and started to see them as tuples of input and output data.

This embedding led to a definition of Web Service composition on an algebraic level. After that we were able to add restrictions to Web Services, e.g., to select those Web Services satisfying a specific condition. Furthermore we used modal operators to determine necessary information which guarantee to reach certain goals. Throughout the paper we tried to illustrate the developed theory by an on-going example, which was expanded to a simple case study.

Using a relational approach works fine for calculating inputs and/or outputs. But, so far, we have not mentioned in detail how to determine any execution order for Web Services. Nevertheless, since we have lifted Web Services to an abstract algebraic level, a replacement of the relational model by any other model which is also based on i-semirings is possible without recalculating properties again. This is another advantage of our approach. In [12] we introduce the algebra of traces, another (well-known) i-semiring, which is useful for Web Service composition. Informally, traces simply save information about the execution order of Web Services.

One of the great advantages of our approach is certainly the simplicity and the well-known theory. For example, using relations allows us to apply all the well-known and efficient algorithm for determining certain relations like the reflexive and transitive closure (e.g. [24]). Such applications will be part of our further work.

Overall, this paper is only a very first step towards a full algebraic characterisation. Nevertheless it shows the basics and provides the ground for on-going and further work. There are various open questions which can hopefully be solved.

First of all it might be interesting to see if the algebra leads to simplifications which can be used to optimise and reorganise Web Services. We already touched this question at the end of Section 7.

Secondly, a problem of our approach is that semantic mismatches may lead to an empty result set. We have not considered situations where the input does not match the output exactly. For example a Web Service might have the birthdate as output whereas the "next" Web Service only requires a date in general. To solve such situations one has to include some taxonomy or ontology (which says that every birthdate is also a general date). This ontology of course has to be combined with the Web Service composition. Re-using an idea of [13] where the composition of an i-semiring is enriched by an additional relation seems quite promising to solve this deficit. Another idea would be to introduce a type system to determine dependencies between data like birthdate or date.

Probably the most challenging open issue is a formal characterisation of *Semantic Web Services*. But, we are quite optimistic that our approach can be re-used for those, too. We hope that tests can be used to model e.g. preconditions and effects.

Acknowledgements.

We are most grateful to Bernhard Bauer and Bernhard Möller for fruitful discussions and remarks.

References

[1] J. H. Conway. *Regular Algebra and Finite Machines*. Chapman & Hall, 1971.

[2] J. Desharnais, B. Möller, and G. Struth. Modal Kleene Algebra and Applications — A Survey —. *Journal on Relational Methods in Computer Science*, 1:93–131, 2004.

[3] J. Desharnais, B. Möller, and G. Struth. Termination in modal Kleene algebra. In J.-J. Lévy, E. W. Mayr, and J. C. Mitchell, editors, *IFIP TCS2004*, pages 647–660. Kluwer, 2004. Revised version: *Algebraic Notions of Termination*. Technical Report 2006-23, Institut für Informatik, Universität Augsburg, 2006.

[4] J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *ACM Trans. Computational Logic*, 7(4):798–833, 2006.

[5] V. Ermolayev, N. Keberle, O. Kononenko, and V. Y. Terziyan. Proactively composing web services as tasks by semantic web agents. In L.-J. Zhang, editor, *Advanced Technologies in Web Services Research*. Cybertech Publishing, 2007.

[6] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. Soap version 1.2 part 1: Messaging framework. Recommendation, W3C, 2003.

[7] H. Haas. Architecture and future of web services: From soap to semantic web services. *WWW 2004, New York, USA*, 2004.

[8] R. Hamadi and B. Benatallah. A petri net-based model for web service composition. In K.-D. Schewe and X. Zhou, editors, *Database Technologies 2003 (ADC 2003)*, volume 17 of *CRPIT*, pages 191–200. Australian Computer Society, 2003.

[9] S. V. Hashemian and F. Mavaddat. Composition algebra. In F. de Boer and V. Mencl, editors, *Formal Aspects of Component Software (FACS'06) –Preliminary Proceedings*, number 344 in UNU-IIST Technical Reports, pages 247–264, 2006.

[10] U. Hebisch and H. J. Weinert. *Semirings — Algebraic Theory and Applications in Computer Science*. World Scientific, Singapur, 1998.

[11] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[12] P. Höfner and F. Lautenbacher. Algebraic structure of web services. Technical Report 2007-12, Universität Augsburg, Institut für Informatik, 2008. <`http://www.informatik.uni-augsburg.de/forschung/reports/`>.

[13] P. Höfner and B. Möller. Non-smooth and zeno trajectories for hybrid system algebra. Technical Report 2006-07, University of Augsburg, Institute of Computer Science, 2006.

[14] D. Jordan and J. Evdemon. Web services business process execution language version 2.0. *OASIS Working Group documents*, 2006.

[15] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[16] D. Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.

[17] P. Küngas. *Distributed Agent-based Web Service Selection, Composition and Analysis through Partial Deduction*. PhD thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, 2006.

[18] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, M. Narayanan, Sriniand Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-S: Semantic markup for web services. Member submission, W3C, 2004.

[19] S. A. McIlraith and T. Cao Son. Adapting golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, editors, *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 482–496. Morgan Kaufmann, 2002.

[20] R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[21] B. Möller and G. Struth. wp is wlp. In W. MacCaull, M. Winter, and I. Duentsch, editors, *Relational Methods in Computer Science*, volume 3929 of *Lecture Notes in Computer Science*, pages 200–211, 2006.

[22] J. J. Moreau, R. Chinnici, A. Ryman, and S. Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. Candidate recommendation, W3C, 2006.

[23] J. Pathak, S. Basu, and V. Honavar. Modeling web services by iterative reformulation of functional and non-functional requirements. In A. Dan and W. Lamersdorf, editors, *International Conference on Service-Oriented Computing – ICSOC 2006*, volume 4294 of *Lecture Notes in Computer Science*, pages 314–326. Springer, 2006.

[24] G. Schmidt and T. Ströhlein. *Relations and Graphs: Discrete Mathematics for Computer Scientists*. Springer, 1993.

[25] Z. Wu, J. Harney, K. Verma, J. Miller, and A. Sheth. Composing semantic web services with interaction protocols. Technical report, LSDIS lab, University of Georgia, USA, 2006.