A sharing-based approach to supporting adaptation in service compositions

Dragan Ivanović · Manuel Carro · Manuel V. Hermenegildo

Abstract Data-related properties of the activities involved in a service composition can be used to facilitate several design-time and run-time adaptation tasks, such as service evolution, distributed enactment, and instance-level adaptation. A number of these properties can be expressed using a notion of sharing. We present an approach for automated inference of data properties based on sharing analysis, which is able to handle service compositions with complex control structures, involving loops and sub-workflows. The properties inferred can include data dependencies, information content, domain-defined attributes, privacy or confidentiality levels, among others. The analysis produces characterizations of the data and the activities in the composition in terms of minimal and maximal sharing, which can then be used to verify compliance of potential adaptation actions, or as supporting information in their generation. This sharing analysis approach can be used both at design time and at run time. In the latter case, the results of analysis can be refined using the composition traces (execution logs) at the point of execution, in order to support run-time adaptation.

D. Ivanović (✉) · M. Carro · M. V. Hermenegildo
Facultad de Informatica, Universidad Politécnica de Madrid,
Campus de Montegancedo s/n,
Boadilla del Monte 28660, Spain
e-mail: idragan@clip.dia.fi.upm.es

M. Carro
e-mail: mcarro@fi.upm.es

M. V. Hermenegildo
e-mail: herme@fi.upm.es

M. Carro · M. V. Hermenegildo
IMDEA Software Institute, Madrid, Spain
e-mail: manuel.carro@imdea.org

M. V. Hermenegildo
e-mail: manuel.hermenegildo@imdea.org

1 Introduction

Service-Oriented Computing (SOC) has become a well-established paradigm for developing, evolving and integrating complex, enterprise-level software systems. The core concept in SOC is that of a service: a software component which is independent of any platform and programming language, with a well defined, standards-based interface exposed on the Internet (or a corporate intranet). In that way, SOC stimulates low coupling between software components. Individual services are usually highly specialized for a particular task, and their interfaces typically include of one or more functionally cohesive sets of operations (called ports). However, the true power of SOC shows in complex, cross-domain and cross-organizational settings, where *service compositions* put together several service components (often provided and maintained by third parties) [11] to perform higher-level or more complex tasks. In turn, exposing service compositions themselves as services and enabling their use by other compositions makes it possible to develop highly complex, large scale, distributed, and flexible software systems.

In this paper we address the problem of *adaptation* at the level of service compositions [11,34] from the data perspective. At design time, adaptation[1] is usually performed in order to meet new functional requirements, adjust non-functional characteristics of the composition (e.g., by removing inefficiencies and bottlenecks), or to enhance interoperability with other systems. At run time, adaptation is typically performed in response to the detection (or prediction) of component failures, extraordinary situations (e.g., exceptions, communication line breakups), or new information about the particular user's context.

In both design-time and run-time adaptation settings, the notion of *correctness*, in the sense that the adapted service composition has to comply with its specification, is crucial. This involves ensuring, among other things, that compatibility with the specified protocols is preserved during conversations with partner (or component) services [35], that the appropriate partner operations are invoked, and that the messages that are exchanged have the correct format and meaning [4]. An adaptation can alter the state of the service composition, replace its components, rearrange activities, reroute messages, fragment the composition into several parts that are executed in a distributed manner or merge several fragments into one, etc., but any adaptation action, simple or complex, must respect the conversation protocols, service interfaces, and message

[1] By convention, and in order to differentiate actions at design time and at run time, any adaptation which impacts the initial assumptions of a SOC, such as those stemming from changes in the requirements and which require a (deep) change in the design of the system, is termed *evolution*. We will in general not use this term unless it is unclear from the context whether we are referring to design time or run time adaptation.

formats and meaning. These, of course, are necessary, but by themselves not sufficient, conditions for correctness of the adaptation.

We argue that the degree of correctness of some adaptations can be improved by taking into account the *data dependencies* present in the specification, i.e., by analyzing how the *content* of messages and the composition state depend on the received messages or behavior of other activities and other factors. In particular, we are interested in *functional dependencies*, which describe what pieces of information and activities determine the content of messages and the internal state, as well as in *data attributes* that describe the contents of data in terms of some domain-specific (i.e., user-defined) properties. Both imply additional correctness criteria that can be used for validating a set of potential adaptation actions, or as a constraint for their generation.

For instance, an adaptation action that breaks a functional dependency by (partially) blocking the flow of information between services in the composition or by replacing one service with another that uses only a subset of inputs risks incurring in information loss and should be (in the absence of more precise information) ruled out as unsafe. On the other hand, if the inputs to two activities do not mutually depend on their state or results, they can in principle be executed independently or in parallel, i.e., isolated into two separate control-flow fragments. Additionally, if the data attributes of a message carry confidential information, then we have to ensure that this information will not be revealed to distrusted participants. Also, the analysis of data attributes allows reasoning about the conceptual information content of the message, which can constrain the search for service candidates by discarding those who expect (or produce) messages whose contents are conceptually incompatible.

Our focus is on supporting an automated analysis of data dependencies that can be used both at run time and as a designer's aid for design-time adaptation. While analysis of protocols relies mostly on automata theory and process algebras [10,35], and analysis of interfaces and message formats is performed mostly by reasoning about service ontologies [13], we base our approach to the analysis of data dependencies on the concept of data sharing and a number of related static analysis techniques based on abstract interpretation [20,29,30]. The application of program analysis techniques is necessary in this case because of two main reasons. Firstly, realistic service compositions involve complex control structures, featuring non-determinism and cyclic execution paths, which may be difficult to analyze without resorting to well-established program abstraction frameworks. Secondly, in a run-time adaptation scenario, the specification is usually the original (pre-adaptation) version of a service composition (typically describing the canonical or the most general case), which is usually expressed using programming language constructs, and a correct adaptation needs to preserve some key properties of the original.

The rest of the paper is organized as follows. Section 2 presents an example that motivates our approach. In Sect. 3 we discuss functional dependencies and data attributes in the context of service compositions, and show how these two aspects of data dependency can be modeled using the same logical framework and the notion of sharing. In Sect. 4 we present the details of sharing analysis, which is the underlying technique used for inferring (safe approximations of) functional dependencies and data attributes in service compositions. We discuss the inputs to the process, and the

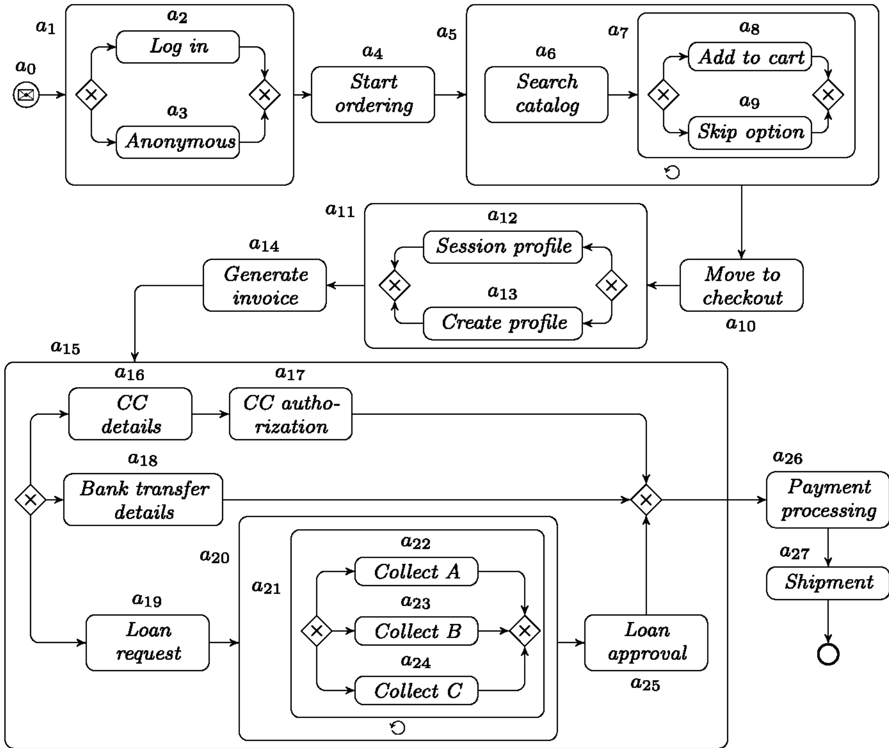


Fig. 1 A sample purchasing end-to-end service composition in BPMN notation

interpretation of its results. Next, we present some applications in Sect. 5. Section 6 offers a review of the related work, and Sect. 7 some conclusions.

2 Motivation

Figure 1 shows an example service composition written in BPMN [33] that realizes a process of purchasing goods form a seller's online site and paying for the goods by credit card, bank transfer, or a loan from a bank. Service invocations and complex activities (XOR-splits and loops) are labeled with the letter a and a subscript, for easier future reference. At the start, the buyer accesses the seller's online site either by logging in to his or her existing account (activity a_2), or by browsing anonymously (a_3). Activity a_4 creates a new shopping cart, and activity a_5 is a loop in which the buyer browses the catalog (a_6) for the base product and its configuration options (e.g., disk, display, memory options for a computer, or the additional equipment for a car) and either adds the item found to the shopping cart (a_8) or skips it (a_9). At the end, the buyer freezes the order (a_{10}) and moves to the check-out stage. At this point, if the buyer had not already logged in, he or she will need to create a profile (a_{13}), by giving the name, address, telephone, and other basic information, which can be saved

for future sessions. Once the buyer is registered, the invoice for the ordered goods is generated (a_{14}).

In the second stage of the process, the buyer chooses among several forms of payment (the exclusive choice a_{15}). One of these choices is to pay using a bank card (a_{16}), which needs to be authorized by the bank (a_{17}). Alternatively, the buyer can pay by debiting his or her bank account (a_{18}). Finally, the buyer may request a loan from a bank that the buyer uses for financing his sales (a_{19}). Depending on whether or not the buyer is a client of the bank and other circumstances, the bank may request one or more documents to be presented (a_{20}) before approving the loan (a_{25}). After fixing the payment option in the exclusive choice activity a_{15}, the payment is made by the bank on behalf of the buyer to the seller (a_{26}). Finally, once the payment is verified, the seller ships the ordered products (a_{27}).

In this fairly standard service composition let us assume that each atomic activity is an invocation of a service (or rather of some operation of some port of a service) which accepts some input and may produce an output message, which is remembered in the state of the composition. The components of the state are shown in Table 1. Where input from the buyer is required, we assume that the corresponding service either takes care of presenting the buyer with a form to fill, or has an interface that the buyer's application can use to supply the required information (and obtain results).

The purchasing composition in Fig. 1 is rather generic and can be adapted in several ways. In particular, there are several classes of *run-time adaptations*, such as parallelization, fragmentation, and compliance checks that can be triggered and applied automatically, ideally without any human intervention. The information on which of these adaptation actions can be used comes from the analysis of both the control and the data dependencies in a service composition. In this paper we argue that both the functional data dependencies and the data attributes (which describe the information content) can be analyzed by means of an (abstract) data sharing analysis. On that basis, we are motivated by the following questions relevant for adaptation:

- *Which activities in the composition do not functionally depend on each other's output, and can therefore be started in parallel?* For instance, in Fig. 1, a loan request activity (a_{19}) can be started without waiting for a_{14} to finish (although a_{25} needs to wait for a_{20}).
- *How can we automatically fragment the composition for distributed enactment, while enforcing information flow constraints?* E.g., the fragments of the composition from Fig. 1 are executed either centrally or on the side of the buyer, the bank, and the seller. The assignment of activities to fragments can be based on data attributes that conceptually describe the information content of the data handled or produced by each activity.
- *How can we automatically choose or replace service components based on information requirements?* Functional dependencies and data attributes can be used as one of the criteria to constrain matching or generation of the replacement, by ensuring that all replacements use all of the relevant data, and by ensuring that the information content of that data (described by data attributes) is adequate.

Note that in all these cases we assume that the usual adaptation constraints related to conversation protocols, message formats, and meaning are correctly preserved, and

Table 1 Data dependencies in the example service composition

	Symbol	Read by	Set/updated by
Data item in composition			
User request	u	a_2, a_3, a_4	a_0
Product query	q	a_5, a_6	a_4, a_6
Buyer info	e	a_{14}, a_{16}	a_{12}, a_{13}
Invoice	i	a_{17}, a_{18}, a_{25}	a_{14}
Credit card info	c	a_{17}	a_{16}
Identification document	d	a_{20}, a_{25}	a_{19}
Additional document A	x		a_{22}
Additional document B	y		a_{23}
Additional document C	z		a_{24}
Transfer order	p	a_{26}	a_{17}, a_{18}, a_{25}
Shipment notice	n		a_{27}
Component service state			
Seller's portal session	w_1	a_4, a_{11}, a_{12}	a_2, a_3, a_{13}
Shopping cart	w_2	a_6, a_8, a_{10}, a_{14}	a_4, a_8, a_{10}
Loan application	w_3	$a_{22}, a_{23}, a_{24}, a_{25}$	a_{19}, a_{20}
Seller's account	w_4	a_{26}, a_{27}	a_{26}

we focus on analyzing the data content, which is not normally taken into account by protocol- and ontology-based approaches.

3 Functional dependencies and data attributes through sharing

In this section we show how the notion of data sharing, in a very general, first-order logical framework, can be used to express two classes of data dependencies: functional dependencies and sharing of the data attributes between input messages sent to the composition, intermediate data items and activities, and outgoing messages from the composition. We start by looking at functional dependencies and data attributes in service compositions, and proceed by presenting the general framework for their common logical representation and generation by means of Horn-clause programs and their operational semantics.

3.1 Functional dependencies in service compositions

Functional dependencies have been widely studied in the field of database systems [2], where they represent (together with multi-valued dependencies) the cornerstone of the most widely applied relational database design and normalization techniques (such as Codd's normal forms) [38]. They have also been extensively studied in logic [17,39], and used for mining of association rules in databases [1].

In the context of service compositions, we are interested in the functional dependencies between some named items, which we call variables, that represent incoming

and outgoing messages in a service composition, the internal data produced and read by the composition activities, and the activities themselves. Unlike the conventional "program variables," these variables do not designate mutable storage locations, but rather act as logical placeholders for values. We shall use the italic letters x, y, z, etc., to denote these variables, and capitals X, Y, Z, etc., to denote sets of variables. In keeping with the usual short-hand notation for functional dependencies, we shall write XY to denote $X \cup Y$ (unless stated otherwise), and wherever a set of variables is expected, we shall allow a variable x to stand for the singleton set $\{x\}$. e.g. $XyZ = X \cup \{y\} \cup Z$, and $xyz = \{x\} \cup \{y\} \cup \{z\} = \{x, y, z\}$.

Definition 1 (*Functional dependency*) A set X of variables is said to determine a set Y of variables (i.e., Y functionally depends on X), which we write $X \multimap Y$, if for each $y \in Y$ there exists a rule that uniquely determines the value of y from values of the variables in X.

Obviously, when $X = \varnothing$ then each $y \in Y$ is a constant, i.e., they have a unique value that does not depend on any other variable. For $X \neq \varnothing$ we normally speak of a function or mapping F which, when applied to the values of some or all variables from X, produces a single value for each $y \in Y$, and we write $Y = F(X)$.

Functional dependencies are normally required to obey a standard set of axioms, known as the *Armstrong dependency axioms*:

Definition 2 (*Armstrong dependency axioms*) A binary functional dependency relation "$\multimap$" between subsets of variables needs to satisfy the following set of axioms:[2]

(AD1) $X \multimap X$
(AD2) $(X \multimap Y) \land (U \multimap V) \rightarrow (XU \multimap YV)$
(AD3) $(X \multimap Y) \land (U \multimap V) \land U \subseteq Y \rightarrow (X \multimap V)$

for arbitrary sets of variables X, Y, U and V.

From the point of view of adaptation, reasoning about functional dependencies in service compositions is important in several adaptation settings. If, for instance, the inputs of activity a'' do not depend on the outputs of a', then a'' can proceed independently of a' (i.e., we can parallelize the two), and even the failure or a dynamic replacement of a' with a compatible activity does not affect a''. Conversely, if some activity depends on some set of variables X, then, in general, it cannot be executed before all "ingredients" of X become available (as received messages or as outputs of other activities), and it is generally not safe to replace that activity with another (atomic or complex) one that takes as input some proper subset $Y \subset X$, since that could lead to a loss of data.

Note that in the adaptation examples above we preserve the conversation protocols and the message formats, but, nevertheless, the actual outcomes may differ significantly.

Example 1 In the composition from Fig. 1, and according to Table 1, activity a_{19} (*Request loan*) does not depend on outputs from a_4, a_5, a_{10}, or a_{11}. Therefore, if at the

[2] The set of axioms presented here follows [40] and has been chosen for its minimality. These axioms are equivalent to those originally proposed by Armstrong [2].

start the buyer is determined to take a loan, and knows the approximate price range, he or she can start the loan request process in advance of, or in parallel with, the online ordering. However, a_{25} (*Loan approval*) needs the invoice that is produced by a_{14} (*Generate invoice*).

Example 2 If payment processing (a_{26}) is executed by the bank, and shipment by the seller (a_{27}), without any direct message exchange between the two, then there is an implicit data dependency on the state of the seller's account (which may be in another bank). This is a "hidden" variable that has to be included in the functional dependencies.

Example 3 If in a_{25} we replace the commodity loan approval with a cash loan approval, which does not look at the invoice, the payment would be made to the buyer's account, rather than to the seller's, and the shipment would not commence. That would require an additional bank transfer step by the user to be inserted after a_{25}.

Example 4 If the user logs in to the seller's portal (a_2), we can replace the general catalog search service a_6 with a version that takes into account the buyer's identity, which functionally determines the buyer's purchase history and interests.

Figure 2a shows a conceptual model of functional dependencies arising from a single execution of some activity a. If a reads some data X and produces some outputs Y, we can generally assume that there exists some functional dependency $YU' = F_a(XU)$, where either X or Y (or both) can be empty. U is a representation of the internal state used by a before its execution, and U' is its updated state. If a' is the next activity after a that uses the same internal state, its functional dependencies have the form $Y'U'' = F_{a'}(X'U')$, where X' and Y' are the sets of data items read and written, respectively, by a'.

Our ability to draw conclusions from such a generic functional dependency $XU \twoheadrightarrow YU'$ may be limited for several reasons. Firstly, we may not (and indeed in general do not) know F_a, which expresses the (denotational) semantics of the computation performed by A. Secondly, we may not know the structure of U and U', unlike the structure of X and Y which correspond to the data items in the composition. However, a more precise reasoning can be obtained under specific circumstances or when we are given some additional information:

- If the internal state of A is initialized to a default value at the start of a composition, then in the first use of the state U is a constant, and the dependency reduces to $X \twoheadrightarrow YU'$. This, for instance, happens with the order o after a_4 (Fig. 1; Table 1).
- If we know that A is stateless, the functional dependency reduces to $X \twoheadrightarrow Y$. That is the case with a_9 (*skip item*) and a_{18} (*Get bank transfer details*) that produce their results directly from the inputs (if any).
- If A does not update its state, then we can decompose $XU \twoheadrightarrow YU'$ into $XU \twoheadrightarrow Y$ and $U' = U$. That is the case with a_{14} (*Generate invoice*), which reads the state variable o (the order), but does not change it.
- If we know that for some $Z \subset X$, $ZU \twoheadrightarrow YU'$, we can eliminate from consideration all irrelevant variables from $X \backslash Z$. e.g., loan approval a_{25} may require proof of residence address r, but may not use its content to generate payment authorization z needed by a_{26}.

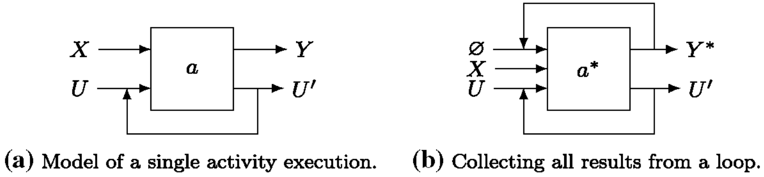


Fig. 2 A conceptual model of functional dependencies in composition activities

Direct functional dependencies, such as those from Fig. 2a, can be given in the form of assertions or meta-data attached to service activities, while indirect (transitive) dependencies are obtained by applying the Armstrong axioms from Definition 2. The general idea is that this kind of reasoning about functional dependencies becomes more precise if in the activity assertions we can narrow the left side of "$\rightarrowtriangle$" (replacing X with $Z \subset X$, or eliminating U), and/or if we know more about the right hand side (e.g., $U' = U$).

If no cycles are involved, the set of all functional dependencies (over a finite set of variables) can be obtained by applying the Armstrong axioms directly a finite number of times. However, if an activity that reads X and writes Y is in fact a loop a^* whose body a can be executed zero or more times, then we are generally interested in some common properties of the set Y^* of all possible values of Y, rather than a Y from a single iteration. This corresponds to the notion of collecting semantics in static program analysis [32]. Figure 2b shows the conceptual model for this case. Starting from an empty set $\varnothing$, each iteration in a^* updates both the internal state and the set of outputs Y^*. In some cases, Y^* may stabilize after a finite set of iterations, but that is generally not guaranteed. Since we are interested in treating this general case, which may involve loops with a generally undecidable number of iterations, we resort to techniques based on abstract interpretation that will be discussed in Sect. 4.

3.2 Data attributes in service compositions

The messages that are sent and received by participants in a service composition are typically dynamic XML documents that may have a complex structure, with nested, optional, and alternative elements. While different technologies (such as DTD and XML Schemas) can be used to constrain the shape and the content of XML messages, these checks can be expensive in terms of computation time, and there is generally no assurance that a particular service infrastructure enforces all the tests for each message that is sent or received. In general, it is the responsibility of the developer to ensure that the messages conform to their XSD specifications, besides being well-formed XML documents. The same applies to the results of XPath or XQuery queries or XSLT transformations used in compositions for computing values of composition state variables.

We are interested in conceptual descriptions of data, where we use the notion of a data attribute to describe some property that holds for a data item represented with a variable (in the sense of variables from Definition 3.1 in the previous section). Some properties (i.e., attributes) can be verified by performing a test on the variable (i.e., by

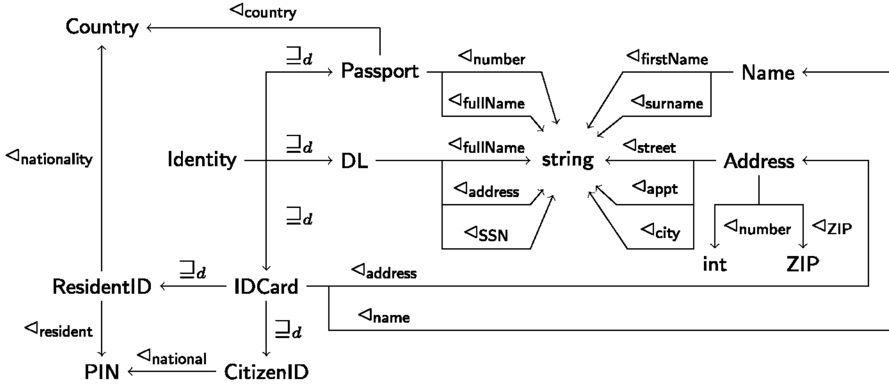


Fig. 3 Structural data description example

inspecting its content), while other properties can be inferred based on domain-specific inference rules from the tests.

Example 5 For instance, variable d (*Identification document*) from Table 1 can be tested for properties such as "contains element address of type string" (property p_1), "contains attribute @allowsResidenceCheck with value true" (property p_2), or "contains element PIN" (property p_3), while an abstract property can be "has a known residence address" (p_4). The inference rules could be $p_1 \rightarrow p_4$ and $p_2 \wedge p_3 \rightarrow p_4$.

We start by defining the notion of structural data description, following an approach similar to that of Data Semantic Structures [4], that originate from semantic data descriptions and ontology matching [13,36].

Definition 3 (*Structural data description*) A conceptual data description is a structure $\textsc{sdd}(\mathbb{D}, \sqsubseteq_d, \lhd, \rightsquigarrow)$ where:

- $\mathbb{D}$ is a set of concepts (conceptual data types), where each $d \in \mathbb{D}$ denotes a set $[\![d]\!]$ of objects (data items) conforming to d;
- $\sqsubseteq_d$ is a partial order on $\mathbb{D}$, called the simulation relation, where $d \sqsubseteq_d d'$ means that an object from $[\![d]\!]$ can be used whenever an object from $[\![d']\!]$ is expected;
- $\lhd \subseteq \mathbb{D} \times F \times \mathbb{D}$ (where F is a set of field names) is a component relation, such that $\langle d, n, d' \rangle \in \lhd$, written as $d \lhd_n d'$, meaning that each object from $[\![d]\!]$ has a field (element or attribute) called n that holds an object from $[\![d']\!]$;
- $\rightsquigarrow$ is a partial transformation functional that maps a pair $\langle d, d' \rangle \in \mathbb{D}^2$ (if a transformation from d to d' is defined) into a function $(d \rightsquigarrow d') : [\![d]\!] \rightarrow [\![d']\!]$ that transforms object $x \in [\![d]\!]$ into its image $x' \in [\![d']\!]$.

Structural data descriptions can be thought of as a sort of typing system for data that is exchanged by services in the composition. The simulation relation $d_1 \sqsubseteq_d d_2$ requires that whenever $d_2 \lhd_n d$, then also $d_1 \lhd_n d$, i.e., objects in $[\![d_2]\!]$ must have all components that the objects in $[\![d_1]\!]$ have, and possibly some more. Also, in that case $(d_1 \rightsquigarrow d_2)$ exists and is an identity function.

Test t	Meaning of $t(x)$
t_1	x : Passport
t_2	x : IDCard
t_3	x : DL
t_4	x/country = "xy"
t_5	x/national exists

Attribute	Meaning
m_1	resident
m_2	national
m_3	known address
m_4	known PIN
m_5	known SSN

Rules	
$t_1 \wedge t_4 \rightarrow m_1$	$t_2 \rightarrow m_3$
$t_1 \wedge t_4 \rightarrow m_2$	$t_2 \rightarrow m_4$
$t_2 \rightarrow m_1$	$t_3 \rightarrow m_5$
$t_2 \wedge t_5 \rightarrow m_2$	
$t_1 \rightarrow m_3$	

Case	t_1	t_2	t_3	t_4	t_5	m_1	m_2	m_3	m_4	m_5
Passport	✓							✓		
XY Passports	✓			✓		✓	✓	✓		
DL		✓								✓
IDCard		✓				✓		✓	✓	
CitizenCard		✓			✓	✓	✓	✓	✓	
ResidentCard		✓				✓		✓	✓	

Fig. 4 Tests, rules and attributes for data structures from Fig. 3

Example 6 Figure 3 shows a simple example of a structural data description for identification documents that may be used for identifying a buyer in some country, i.e., as the data item d from Table 1 which is obtained from activity a_{19} (*Loan request*) in Fig. 1. These include passports (Passport), driving license (DL), and identity cards (IDCards), which are further subdivided into those for citizens of the country (CitizenID) and other residents (ResidentID). While passports and driving licenses represent the person's name and address simply as strings, identity cards use a more structured representation via conceptual data types Name and Address. While Address $\rightsquigarrow$ string and Name $\rightsquigarrow$ string are straightforward, string $\rightsquigarrow$ Address may be difficult, and string $\rightsquigarrow$ Name may be ambiguous. CitizenID and ResidentID are discriminated by fields national and resident, respectively. Some field names have different meanings, e.g., number in Passport and in Address.

Definition 4 (*Data attributes*) For an SSD($\mathbb{D}$, $\sqsubseteq_d$, $\lhd$, $\rightsquigarrow$), a data attribute system is a structure DA($\mathbb{A}$, $\mathbb{T}$, R, $\models_d$), where:

- $\mathbb{A}$ is a set of data attributes, which are propositional symbols that may evaluate to true or false for each $d \in \mathbb{D}$.
- $\mathbb{T}$ is a set of deterministic and terminating Boolean tests that describe the contents of a data object. A set of objects X satisfies $t \in \mathbb{T}$ iff $t(x) = \top$ for each $x \in X$. A $d \in \mathbb{D}$ satisfies t iff $[\![d]\!]$ satisfies t.
- R is a finite set of implication rules of the form $\wedge B \rightarrow a$ (B finite), where $a \in \mathbb{A}$ and $B \subseteq \mathbb{A} \uplus \mathbb{T}$. Circular implications (direct or indirect) are not permitted.
- $\models_d$ is the attribute inference relation that associates a set of data objects S with attributes from $\mathbb{A}$, in such a way that for all $a \in \mathbb{A}$, $S \models_d a$ holds if and only if a can be inferred from the tests that are satisfied by S and the set of rules R.

Example 7 Figure 4 shows a data attribute system for the sample structural data description form Fig. 3. Tests $t_1, \ldots, t_5$ are checked at the level of a data instance (x), and here they have the simple format $x : d$ (meaning that x complies with the structural schema for d, in tests t_1–t_3), or an XPath query (t_4 and t_5). The attributes $m_1, \ldots, m_5$ represent domain-specific properties related to the content, rather than to the structure of a data object. They are obtained from the tests using the given set

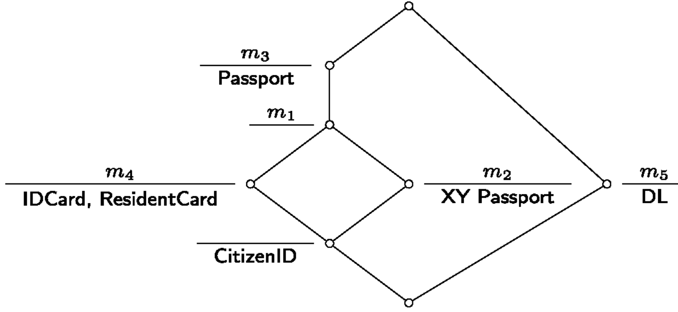


Fig. 5 A sample concept lattice for the context from Fig. 4

of rules (Horn clauses). The bottom part of Fig. 4 shows a matrix of the tests and attributes for several cases of data objects. The cases mainly correspond to the conceptual data types from Fig. 3, but the case called XY Passports is a set of objects which is structurally indistinguishable from Passport, yet its content (i.e., the fact that the passport is issued to the citizen of the hypothetical country XY in question) makes it different from the point of view of the domain-specific attributes.

When considering data at the level of data attributes, we are mostly concerned with the conceptual grouping of the possible cases on the basis of the information they carry with respect to the attributes. For that, we use concepts from Formal Concept Analysis (FCA, a branch of lattice theory [9,15]). In FCA, a context is a triplet (G, M, I) where G is a set of objects (or cases), M is a set of attributes, and $I \subseteq G \times M$ is a relation that associates the objects with the attributes. For arbitrary sets of objects $X \subseteq G$ and attributes $Y \subseteq M$, we define the operators:

$$X' = \{y \in M \mid (\forall x \in X)(\langle x, y \rangle \in I)\} \tag{1}$$

$$Y' = \{x \in G \mid (\forall y \in Y)(\langle x, y \rangle \in I)\} \tag{2}$$

which can be described as follows: X' is the set of all attributes associated (via I) to all objects of X, and Y' is the set of all objects associated to all attributes from Y. The pair $\langle X, Y \rangle$ is called a concept if $X' = Y$ and $Y = X'$, i.e., if X and Y completely determine each other by means of the operator $(\cdot)'$. X is called the extent of the concept, and Y its intent. The fundamental theorem of FCA asserts that concepts form a complete lattice, with the ordering $\langle X_1, Y_1 \rangle \leq \langle X_2, Y_2 \rangle$ iff $X_1 \subseteq X_2$, or, equivalently $Y_2 \subseteq Y_1$. Concept lattices are usually represented with a variation of Hasse diagrams with the greatest concept at the top, and the smallest concept at the bottom. Each node is a concept, and is decorated with attributes that do not appear in greater concepts, and the objects that do not appear in smaller ones.

Example 8 Figure 5 shows a concept lattice based on the attribute matrix from Fig. 4. The concept ordering shows how informative the different cases are from the point of view of attributes $m_1, \ldots, m_5$. Passport is more general than other identity documents (except DL), while CitizenID is more informative than other variations of ID Cards

and passports. Generic IDCard and ResidentCard are conceptually the same. The data item d from Table 1 can be chosen from the set of objects assigned to nodes.

A complex message may consists of several parts that carry their own data attributes. In such a setting, we are interested in all the attributes present in the message. If the components of a message are represented with sets of attributes, then the attributes of the entire message can be combined on the basis of the component relation "$\lhd$". The input X to an activity a can be represented as a set of variables carrying data attributes of the component data objects, which come from the inputs or from previous activities. The output Y from a can be represented as $Y = ZU$, where $Z \subseteq X$, whose attributes Y inherits, and U represents some new components added to Y by a.

3.3 Representing dependencies with substitutions

In our approach we use notions from first-order logic as the underlying mechanism for both functional dependencies and data attributes. A first-order language represents objects in the universe of discourse by means of terms, which are built from variable symbols $(x, y, z, \ldots)$, constants (e.g., 0, $\mathbf{a}$), and function symbols $(f, g, \ldots)$. A complex term of the form $f(t_1, t_2, \ldots, t_n)$ (where $t_1, \ldots, t_n$ are terms) is normally seen as the result of applying some function f of n arguments to terms $t_1, \ldots, t_n$. Therefore, an equation $z = f(x, y)$ is a common mathematical way of expressing a functional dependency of z on x and y by means of f. To actually apply f to its arguments, we need to be equipped with an interpretation that assigns some actual computation procedure to the function symbol f, but even when f is left uninterpreted, $z = f(x, y)$ is a statement of the existence of a functional dependency $xy \twoheadrightarrow z$. On the other hand, from a purely syntactic point of view, $f(x, y)$ can be seen as a grouping of x and y under f, in much the same way as fields are packed together in a record. If x and y carry their sets of data attributes, then $z = f(x, y)$ is also a convenient way to state that z inherits the attributes of both.

We can represent the above equation with a substitution $\sigma = \{z \mapsto f(x, y)\}$. When applied to a term or a statement, a substitution simultaneously replaces all occurrences of the variables on the left-hand side of "$\mapsto$" with the corresponding terms on the right-hand side. With $\mathrm{dom}(\sigma)$ we denote the set $\{x \mid (x \mapsto t) \in \sigma\}$, and with $\mathrm{range}(\sigma)$ the set $\bigcup_{(x \mapsto t) \in \sigma} \mathrm{vars}(t)$, where $\mathrm{vars}(t)$ is the set of variable symbols occurring in term t. To rule out circular references, we require $\mathrm{dom}(\sigma) \cap \mathrm{range}(\sigma) = \varnothing$. Also, to enforce determinism, if a substitution contains mappings $x \mapsto t$ and $x \mapsto t'$, then we require $t \equiv t'$, where "$\equiv$" stands for the syntactical identity of terms as strings of symbols. In our example, $z\sigma \equiv f(x, y)$ and $g(x, z)\sigma \equiv g(x, f(x, y))$.

In Definition 2 (the Armstrong dependency axioms) we have already stated the properties expected of a relation that models functional dependencies. With the following definition we provide the substitution-based functional dependency relation.

Definition 5 Let X and Y be two subsets of variables, and σ a substitution. We say that Y functionally depends on X under σ, and write $X \twoheadrightarrow_\sigma Y$, if $\mathrm{vars}(Z\sigma) \subseteq Y$, where $Z = X \backslash Y$.

We establish the expressiveness of "$\twoheadrightarrow_\sigma$" with the next two lemmas and Theorem 1.

Lemma 1 *For an arbitrary substitution σ, relation $\rightarrowtriangle_\sigma$ satisfies the Armstrong dependency axioms.*

Proof (1) From Definition 5, $\text{vars}((X \setminus X)\sigma) = \text{vars}(\emptyset) = \emptyset \subseteq X$, thus $X \rightarrowtriangle_\sigma X$.

(2) From Definition 5, $X \rightarrowtriangle_\sigma Y$ and $U \rightarrowtriangle_\sigma V$, we have $\text{vars}((Y \setminus X)\sigma) \subseteq X$ and $\text{vars}((V \setminus U)\sigma) \subseteq U$, and, by union, $\text{vars}((Y \setminus X)\sigma) \cup \text{vars}((V \setminus U)\sigma) \subseteq XU$. Now, $\text{vars}((YV \setminus XU)\sigma) = \text{vars}((Y \setminus XU)\sigma) \cup \text{vars}((V \setminus XU)\sigma)$, and because $Y \setminus XU \subseteq Y \setminus X$ and $V \setminus XU \subseteq V \setminus U$, we have $\text{vars}((Y \setminus XU)\sigma) \subseteq \text{vars}((Y \setminus X)\sigma)$ and $\text{vars}((V \setminus XU)\sigma) \subseteq \text{vars}((V \setminus U)\sigma)$. Again, by union, we obtain $\text{vars}((YV \setminus XU)\sigma) = \text{vars}((Y \setminus XU)\sigma) \cup \text{vars}((V \setminus XU)\sigma) \subseteq \text{vars}((Y \setminus X)\sigma) \cup \text{vars}((V \setminus U)\sigma) \subseteq XU$, which means $XU \rightarrowtriangle_\sigma YV$.

(3) Here again from Definition 5, $X \rightarrowtriangle_\sigma Y$ and $U \rightarrowtriangle_\sigma V$, we have $\text{vars}((Y \setminus X)\sigma) \subseteq X$ and $\text{vars}((V \setminus U)\sigma) \subseteq U$. Because $U \subseteq X$, we also have $V \setminus X \subseteq V \setminus U$, and therefore $\text{vars}((V \setminus X)\sigma) \subseteq \text{vars}((V \setminus U)\sigma) \subseteq U \subseteq X$, i.e., $X \rightarrowtriangle_\sigma V$.

□

Indeed, as the Armstrong axioms suggest, some functional dependencies may be evident directly in the substitution, while others can be deduced from them. In our example, $\sigma = \{z \mapsto f(x, y)\}$, the dependency $xy \rightarrowtriangle_\sigma z$ is directly represented, while, e.g., $xyz \rightarrowtriangle_\sigma z$ and $xy \rightarrowtriangle_\sigma y$ are implicit. We formalize the notion of directly represented relationships in the following definition.

Definition 6 For a given substitution σ the functional dependency basis is defined as $[\sigma] = \{\langle X, y\rangle \mid (y \mapsto t) \in \sigma \wedge X = \text{vars}(t)\}$.

Lemma 2 *For an arbitrary substitution σ, no relation that is smaller than $\rightarrowtriangle_\sigma$ and includes $[\sigma]$ satisfies the Armstrong dependency axioms.*

Proof Suppose $\rhd$ is a relation between subsets of variables from language $\mathcal{L}$ that satisfies the Armstrong dependency axioms, such that $\rhd \subseteq \rightarrowtriangle_\sigma$ and $[\sigma] \subseteq \rhd$.

(1) For arbitrary X we have $\text{vars}((X \setminus X)\sigma) = \emptyset \subseteq X$, i.e., $X \rightarrowtriangle_\sigma X$ and also in $X \rhd X$ from (AD1) in Definition 2.

(2) For arbitrary X and Y such that $Y \subseteq X$, from Definition 2 we have $\text{vars}((Y \setminus X)\sigma) = \emptyset \subseteq X$, i.e., $X \rightarrowtriangle_\sigma Y$. Also, from Definition 2 (AD1) we have $X \rhd X$ and $Y \rhd Y$, and with $Y \subseteq X$ from (AD3) we obtain $X \rhd Y$.

(3) For arbitrary X and Y such that $Y \nsubseteq Y$ and $X \rightarrowtriangle_\sigma Y$, from Definition 2 we have $\text{vars}((Y \setminus X)\sigma) \subseteq X$. Therefore, for each $y \in Y \setminus X$, (and there has to be at least one such y), σ must contain a mapping of the form $y \mapsto t$, where $\text{vars}(t) = \text{vars}(y\sigma) \subseteq X$, or, in other words, $X \rightarrowtriangle_\sigma y$. Now, from (2) above, we have $X \rightarrowtriangle_\sigma \text{vars}(y\sigma)$ and also $X \rhd \text{vars}(y\sigma)$. Since $[\sigma] \subseteq \rhd$, we have $\text{vars}(y\sigma) \rhd y$, and, by (AD3), $X \rhd y$. By applying axiom (AD2) over all such y, we conclude $X \rhd Y$.

From (1)–(3) we conclude that all elements from $\rightarrowtriangle_\sigma$ are also present in $\rhd$, and therefore, no proper subset of $\rightarrowtriangle_\sigma$ which contains $[\sigma]$ satisfies the Armstrong dependency axioms. □

Theorem 1 *For an arbitrary substitution σ, $\rightarrowtriangle_\sigma$ is the smallest relation that expresses exactly those functional dependencies that are either present in the base $[\sigma]$, or can be deduced from it using the Armstrong dependency axioms.*

Proof Follows directly from Lemmas 1 and 2.

To prove the adequacy of using substitutions for expressing inheritance of data attributes it suffices to demonstrate that if $x_1, \ldots, x_n$ are variables from $\mathrm{dom}(\sigma)$ that represent data objects, so that $\mathrm{vars}(x_i\sigma)$ is the set of data attributes for x_i under σ, then by extending σ with a mapping $z \mapsto t$, where $\mathrm{vars}(t) = \{x_1, \ldots, x_n\}$, from the definition of $\mathrm{vars}(\cdot)$ it follows that the set of attributes of z under σ, $\mathrm{vars}(z\sigma) = \cup_{i=1}^{n} \mathrm{vars}(x_i\sigma)$.

3.4 Variable sharing

It can be easily seen that expressing functional dependencies and data attribute inheritance by means of substitutions does not depend on the choice of function symbols and the shape of the terms on the right-hand side of "$\mapsto$". As long as the invariant $\mathrm{vars}(t) = \mathrm{vars}(t')$ holds, we can replace any mapping $x \mapsto t$ with $x \mapsto t'$ in σ without loosing any result from the previous subsection. This indicates that a substitution can be presented in a more abstract manner [20,29,30]. The following definitions formalize that notion.

Definition 7 (*Sharing*) A non-empty set S of terms is said to share if $\cap_{t \in S} \mathrm{vars}(t) = X \neq \varnothing$. X is the set of the variables shared in S.

Definition 8 (*Abstract substitution*) Let Y be a set of variables of interest, and σ a substitution. We define the abstract substitution $\alpha_Y(\sigma)$ in the following way:

$$\alpha_Y(\sigma) = \{\{y \in Y \mid x \in \mathrm{vars}(y\sigma)\} \mid x \in Z\},$$

where $Z = (Y \backslash \mathrm{dom}(\sigma)) \cup \mathrm{range}(\sigma)$.

Example 9 Let $Y = xyzu$ and $\sigma = \{x \mapsto f(u, v, w), y \mapsto g(u, v), z \mapsto h(w)\}$. Then, $Z = uvw$ and $\alpha_Y(\sigma) = \{xyu, xy, xz\}$. After applying σ to Y we get $\{f(u, v, w), g(u, v), h(w), u\}$, respecting the order in which the set $xyzu$ is written. The (singleton) set of variables u appears in terms $f(u, v, w), g(u, v), u$ which correspond to the initial variable set xyu after applying σ. The set of variables $\{v, w\}$ appears in terms $f(u, v, w)$ and $g(u, v)$, coming from the initial set of variables xy. The set of variables $\{w\}$ appears in terms $f(u, v, w)$ and $h(w)$, which come from the set of variables xz.

If $Y = xyz$, we get $\alpha_Y(\sigma) = \{xy, xz\}$. If $Y = xyu$, we get $\alpha_Y(\sigma) = \{xyu, xy, x\}$.

Each member of $\alpha_Y(\sigma)$ is called a sharing group, and each sharing group represents a set of variables shared between the members of the group.

Lemma 3 *For each sharing group $S \in \alpha_Y(\sigma)$, there exists a set $X \neq \varnothing$ of variables shared between all members of S, and not shared by any other sharing group.*

Proof First, let us note that from Definition 8, $S \neq \varnothing$. Let X be the set of variables from $Z = (Y \setminus \mathrm{dom}(\sigma)) \cup \mathrm{range}(\sigma)$, for which $x \in X$ implies $\{y \in Y | x \in \mathrm{vars}(y\sigma)\} = S$.

X cannot be empty, because otherwise S would have to be empty. Let $S' \neq S$ be another sharing group from $\alpha_Y(\sigma)$, and X' such that $x' \in X'$ implies $\{y \in Y | x \in \text{vars}(y\sigma)\} = S'$. If $x' \in X$, then $S = S'$, which is a contradiction. Therefore, X and X' must be disjoint. □

It is easy to see that there exists an infinite number of substitutions θ such that $\alpha_Y(\theta) = \alpha_Y(\sigma)$—for example, one for each unique renaming of variables in range(σ). We now consider how expressive the sharing information contained in the abstract substitution $\alpha_Y(\sigma)$ is compared to the (concrete) substitution σ.

Definition 9 (*Sharing ordering*) For two variables $y, y' \in Y$ and an abstract substitution $\alpha_Y(\sigma)$, we write $y \sqsubseteq_s y'$ if for all $S \in \alpha_Y(\sigma)$, $y \in S$ implies $y' \in S$.

Theorem 2 *For a set of variables of interest Y and a substitution σ:*

(1) *For arbitrary $y, y' \in Y$, $y, y' \notin X$, if $y \sqsubseteq_s y'$ then $X \rightarrow_s y'$ implies $X \rightarrow_s y$.*
(2) *For arbitrary $y \in Y$, $X \rightarrow_\sigma y$, $y \notin X$, implies* $\{x \in Y | x \sqsubseteq_s y \wedge x \notin \text{dom}(\sigma)\} \subseteq X$.
(3) *If* range$(\sigma) \subseteq Y$, *then* $[\sigma] = \{\langle X(y), y\rangle \,|\, y \in Y \cap \text{dom}(\sigma)\}$, *where* $X(y) = \{x \in Y | x \sqsubseteq_s y \wedge x \notin \text{dom}(\sigma)\}$.

Proof (1) Assume $y \sqsubseteq_s y'$ and $X \rightarrow_\sigma y'$, and let $S_1, S_2, \ldots, S_n$ $(n \geq 0)$ be all sharing settings containing y. From Lemma 3, there are n non-empty and pairwise disjoint sets $V_1, V_2, \ldots, V_n$ such that $\cup_{i=1}^n V_i = \text{vars}(y\sigma)$. Since $y' \in S_i$, for each $i = 1 \ldots n$, we have $\text{vars}(y\sigma) = \cup_{i=1}^n V_i \subseteq \text{vars}(y'\sigma) \subseteq X$, i.e., $X \rightarrow_\sigma y$.

(2) Assume $X \rightarrow_\sigma y$, i.e., $\text{vars}(y\sigma) \subseteq X$. For an arbitrary $x \in Y$, $x \notin \text{dom}(\sigma)$, $vars(x\sigma) = \{x\}$. If $x \sqsubseteq_s y$, then x shares with y in at least one sharing group. Therefore, $x \in \text{vars}(y\sigma)$, i.e., $\{x\} \subseteq \text{vars}(y\sigma)$. By disjointedness over all such x, we have $\{x \in Y | x \sqsubseteq_s y \wedge x \notin \text{dom}(\sigma)\} \subseteq \text{vars}(y\sigma) \subseteq X$.

(3) From Definition 6, we know that $[\sigma] = \{\langle X, y\rangle \mid (y \mapsto t) \in \sigma \wedge X = \text{vars}(t)\}$. Now we need to prove that for each mapping $(y \rightarrow t) \in \sigma$, $\text{vars}(t) = X(y)$.

(3.a) Assume $x \in \text{vars}(t)$. Therefore, $x \notin \text{dom}(\sigma)$, because circular substitutions in σ are forbidden. Since range$(\sigma) \subseteq Y$, we have $x \in Y$. Finally, let $S \in \alpha_Y(\sigma)$ be a sharing group containing x. By definition, $S = \{w \in Y | z \in \text{vars}(w\sigma)\}$ for some z. Because $x \in S$ this implies $z \in \text{vars}(x\sigma) = \{x\}$. Therefore, z must be the same as x. And, therefore, y must belong to S. In other words $x \sqsubseteq_s y$. This completes all conditions needed for $x \in X(y)$.

(3.b) Assume $x \in X(y)$. Therefore, $x \in Y$, $x \sqsubseteq_s y$ and $x \notin \text{dom}(\sigma)$. Since $x \in Y$ and $x \notin \text{dom}(\sigma)$, x has to appear in at least one sharing setting $S \in \alpha_Y(\sigma)$. We also know $y \in S$. Following the same argument from (3.a), $S = \{w \in Y | x \in \text{vars}(w\sigma)\}$, and from $y \in S$, we conclude $x \in \text{vars}(y\sigma) = \text{vars}(t)$.

□

Theorem 2 tells us what we can infer about functional dependencies from $\alpha(\sigma)$, without knowing σ directly, but with the knowledge of $\alpha(\sigma)$. In the most basic case, from $y \sqsubseteq_s y'$ we conclude that whatever functionally determines y' also determines y. We can draw more informative conclusions if we are equipped with what is usually

called *freeness* information [30]: whether some $x \in Y$ belongs to $\text{dom}(\sigma)$ or not. With freeness information we can (at least partially) reconstruct the left side of "$\rightarrowtriangle_\sigma$". And, by both having the freeness information and extending Y to include all variables from $\text{range}(\sigma)$, we can in fact reconstruct the sharing basis $[\sigma]$. This tells us that, under these conditions, the abstract substitution $\alpha_Y(\sigma)$ is as expressive as σ when it comes to functional dependencies.

4 Sharing analysis

From the previous sections, we conclude that the abstract substitution $\alpha_Y(\sigma)$ is enough to derive the functional dependencies we need. The question now is how to infer this abstract substitution. The way we do it is by using a *sharing analysis*, applied to a Horn clause (logic program) version of the service composition.

In this section, we present the steps necessary for preparing inputs for the sharing analysis, describe briefly the analysis itself, and discuss the interpretation of the analysis outputs. Sharing analysis is an instance of program analysis, and therefore we start by describing the process of transforming a definition of a service composition into a Horn-clause program as appropriate for the analysis. We use the approach from our previous work on fragmentation analysis [18].

4.1 Derivation of control and data dependencies

As a first step towards creating a Horn clause representation of a service composition, we find out a feasible order of activities which is coherent with their dependencies and which allows the composition to finish successfully. How to do this obviously depends on the palette of allowed relationships between activities, with respect to which we opted for a notable freedom by adopting a relatively general abstract composition model. To find such an order we first establish a partial order between workflow activities which respects their dependencies; in doing this we also detect whether there are dependency conflicts that may result in deadlocks. While there is ample work in deadlock detection [3,5], we think that the technique we propose is clean, can be used for arbitrarily complex dependencies between activities, and uses well-proven, existing technology, which simplifies its implementation.

Definition 10 (*Abstract composition model*) An abstract composition model is a tuple $\text{AC}(A, C, D)$, where:

- A is a finite set of activities;
- C is a set of formulas expressing control flow preconditions for each $a \in A$, of the form $\textsf{pre-}a \equiv \phi$, where ϕ is a propositional formula built from the usual logical connectives ($\vee$, $\wedge$, $\neg$, $\rightarrow$, and $\leftrightarrow$) and the propositional symbols $\textsf{done-}a'$ and $\textsf{succ-}a'$ for $a' \in A$, with the following meaning:
  - $\textsf{done-}a'$ is true if a_j has completed;
  - $\textsf{succ-}a'$ (when $\textsf{done-}a'$ holds) indicates that the outgoing condition from a' has evaluated to true;
  - The combination $\textsf{done-}a' = 0$ and $\textsf{succ-}a' = 1$ is illegal.

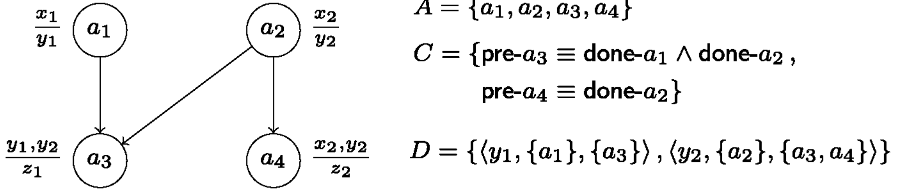


Fig. 6 An example workflow. *Arrows* indicate control dependencies

– D is a finite set of data items in the compositions (giving the core data dependencies), consisting of tuples of the form $\langle x, W, R\rangle$, where x is a data item, $W \subseteq A$ is the set of activities that write x, and $R \subseteq A$ is the set of activities that read x.

This abstract composition model is able to express some of the most frequently used composition workflow patterns, such as AND/OR/XOR splits and joins. However, thanks to the flexibility of the encoding we will use for the sharing analysis, it introduces two significant extensions compared to other workflow models:

– In our approach, the activities inside a workflow can be simple or structured. The latter include branching (*if-then-else*) and looping (*while* and *repeat-until*) constructs, arbitrarily nested. The body of a branch or a loop is a sub-workflow, and activities in the main workflow cannot directly depend on activities inside that sub-workflow. Of course, any activity in such a sub-workflow is subject to the same treatment as activities in the parent workflow.
– Second, we allow an expressive repertoire of control dependencies between activities besides structured sequencing: AND split-join, OR split-join and XOR split-join. We express dependencies similarly to the link dependencies in BPEL but with fewer restrictions, thereby supporting OR- and XOR-join.

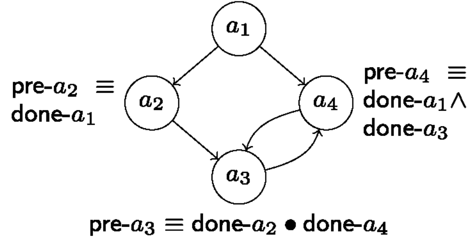
Commonly, the preconditions in C use "done" symbols, whereas "succ" symbols are added to distinguish mutually exclusive execution paths. We do not specify here how the "succ" indicators are exactly computed.

Note that each activity in the workflow is executed at most once; repetitions are represented with the structured looping constructs (yet, within each iteration, an activity in the loop body sub-workflow can also be executed at most once).

Example 10 Figure 6 shows an example. The activities are drawn as nodes and control dependencies indicated by arrows. Data dependencies are textually shown in a "fraction" or "production rule" format next to the activities: items above the bar are used (read) by the activity, and items below are produced. Note that only items y_1 and y_2 are data dependencies; others either come from the input message (x_1, x_2), or are the result of the workflow (z_1, z_2). Item y_1 is produced by a_1 and used by a_3, and y_2 is produced by a_2 and used by a_3 and a_4.

Many workflow patterns can be expressed in terms of such logical link dependencies. For instance, a sequence "a_j after a_i" boils down to $\text{pre-}a_j \equiv \text{done-}a_i$. An AND-join after a_i and a_j into a_k becomes $\text{pre-}a_k \equiv \text{done-}a_i \wedge \text{done-}a_j$. An (X)OR-join of a_i and a_j into a_k is encoded as $\text{pre-}a_k \equiv \text{done-}a_i \vee \text{done-}a_j$.

Fig. 7 An example of deadlock dependency on logic formula: • can be either ∧ or ∨



And an XOR split of a_i into a_j and a_k (based on the business outcome of a_i) becomes pre-$a_j \equiv$ done-$a_i \wedge$ succ-a_i, pre-$a_k \equiv$ done-$a_i \wedge \neg$succ-a_i. In terms of execution scheduling, we take the assumption that a workflow activity a_i may start executing as soon as its precondition is met.

4.2 Validity of control dependencies

The relative freedom allowed in the use of logic formulas to specify control dependencies comes at the cost of possible anomalies that may lead to deadlocks and other undesirable effects. These need to be detected beforehand, i.e., at design/compile time using some sort of static analysis. Here, we are primarily concerned with *deadlock-freeness*, i.e., elimination of the cases when activities can never start because they wait on events that cannot happen.

Example 11 Whether a deadlock can happen or not depends on both the topology and the logic of control dependencies. Topological information is in general not enough to determine deadlock freeness, unless there are no loops in the graph. Figure 7 shows a simple example where the dependency arrows are drawn from a_i and a_j whenever pre-a_j depends on a_i finishing. If the connective marked with • in pre-a_3 is $\vee$, there is no deadlock: indeed, there is a possible execution sequence, $a_1 - a_2 - a_3 - a_4$. If, however, • denotes $\wedge$, there is a deadlock between a_3 and a_4.

Therefore, in general, checking for deadlock-freeness requires looking at the formulas. We present one approach that relies on simple proofs of propositional formulas. We start by forming a propositional logical theory Γ from the workflow by including all preconditions from C and adding axioms of the form done-$a_i \rightarrow$ pre-a_i for each $a_i \in A$. These additional axioms simply state that an activity a_i cannot finish if its preconditions were not met. On that basis, we introduce the following definition to help us detect deadlocks and infer a task order which respects the data and control dependencies:

Definition 11 (*Dependency matrix*) For a given composition model AC(A, C, D), the dependency matrix Δ is a square Boolean matrix such that its element δ_{ij}, corresponding to $a_i, a_j \in A$, is defined as:

$$\delta_{ij} = \begin{cases} 1, & \text{if } \Gamma, \text{pre-}a_i \vdash \text{done-}a_j \\ 0, & \text{otherwise} \end{cases}$$

For every data dependency $\langle x, R, W\rangle \in D$, and for each $a \in R$, we wish to ensure that a cannot start unless at least one of $b \in W$ has completed, since otherwise the data item x would not be ready. Expressed with a logic formula, that condition is $\text{pre-}a \rightarrow \vee_{b \in W\setminus\{a\}} \text{done-}b$.

The computation of Δ involves proving propositional formulas, which is best achieved using some form of SAT solver. Such solvers are nowadays very mature and widely available either as libraries or standalone programs. It follows from the definition that $\delta_{ij} = 1$ if and only if the end of a_j is a necessary condition for the start of a_i. It can be easily shown that Δ is a transitive closure of C, and that is important for the ordering of activities in a logic program representation. However, the most important property can be summarized as follows.

Proposition 1 (Freedom from deadlocks) *The given workflow* $\text{AC}(A, C, D)$ *with dependency matrix* Δ *is deadlock-free if and only if* $\forall a_i \in A, \delta_{ii} = 0$.

Proposition 2 (Partial ordering) *In a deadlock-free workflow* $\text{AC}(A, C, D)$*, the dependency matrix* Δ *induces a strict partial ordering* $\prec$ *such that for any two distinct* $a_i, a_j \in A, a_j \prec a_i$ *iff* $\delta_{ij} = 1$.

4.3 Generating Horn clause representations

The Horn clause representation of a service composition introduced in this section is essentially a logic program with semantics corresponding to that of a subset of standard Prolog with operational semantics based on SLD resolution [25]. The goal of that program is to represent the computation of **all** the substitutions that express functional dependencies and data attribute sharing of the composition, as described in Sect. 3. In other words, the purpose of such program is not to operationally mimic the scheduling of workflow activities, but to express and convey relevant data and control dependency information to the sharing analysis stage.

Based on the strict partial ordering $\prec$ induced by the dependency matrix Δ, in the deadlock-free case it is always possible to *totally* order the activities so that $\prec$ is respected. The choice of a particular order has no impact on our analysis, because we assume that the control dependencies, from which the partial ordering derives, include the data dependencies. From this point on we will assume that activities are renumbered to follow the chosen total order. The workflow can then be translated into a Horn clause of the form:

$$w(V) \leftarrow T(a_1), T(a_2), \ldots, T(a_n) \tag{3}$$

where V is the set of all logic variables used in the clause, and $T(a_i)$ stands for the translation of activity a_i into a logic (Prolog) goal. For simple activities (such as a service invocation or an assignment), $T(a_i)$ gives a sequence of equations (explained below) that relate its inputs and outputs. The complex activities, such as branches or loops, along with their constituent parts (e.g., loop body and *then*/*else* parts) are recursively translated into separate clauses following the scheme (3) above, and $T(a_i)$ is a call to such generated clause.

Logic variables in V are used to represent input and output messages, variables comprising the composition state, internal state of component services (as in Table 1), and the data sets read by individual activities. For each activity $a_i \in A$ we designate a set X_i of logic variables that represent data items read by a_i, a set Y_i of logic variables that stand for data items produced by a_i, as well as the sets U_i and U_i' that represent the state of a component service to which a belongs before and after its execution. We designate a variable set $A_i \subseteq X_i U_i$ that represents the total inflow of data into a_i. The task of the translation is to connect X_i, Y_i, A_i, U_i, and U_i' correctly.

The translation scheme is mechanical. We first present the scheme for simple activities. Using Prolog notation, where variable names start in upper case, we use A*i* to denote all inputs to activity a_i, and Y*i* to denote its output. We use X*i*a to denote the part of A*i* used in computing Y*i*. It is always safe to assume that all data an activity accesses is also used in producing its output, i.e., A*i* = X*i*a, but if we can obtain sufficient guarantees to safely exclude some item in A*i* from X*i*a, we can draw more precise conclusions about the functional dependencies, as discussed in Sect. 3.1. e.g., if a_i contains a loop internally, data used in evaluating the loop condition (included in A*i*) may not be used in the computation of the loop output Y*i*. Next, if a_i is an invocation of a stateful component service, we use W*i*_0 to symbolically denote its previous internal state, and W*i* for its state after executing a_i. The part of A*i* used for updating the internal state is denoted by X*i*b. Again, a safe assumption is A*i* = X*i*b, while more precision can be obtained by safely restricting X*i*b when possible. We use X*i* = [X*i*$_1$, X*i*$_2$, ..., X*i*$_m$] ($m \geq 0$) to denote the union of X*i*a and X*i*b. For analyzing data attributes that may be injected by an external activity, we optionally introduce M*i* to represent those attributes injected into the output Y*i*, and N*i* for those injected into the internal state W*i*.

To round up our translation scheme for simple activities, we need to take into account that they can be placed inside a loop, and in that case we are interested in the most general sharing that includes all potential loop iterations. For that reason, we include in A*i* and Y*i* their previously collected values A*i*_0 and Y*i*_0, respectively (which are ground on first iterations and outside the loops). Thus, $T(a_i)$ can be put in the shape of a sequence:

```
Ai = [Xi, Wi_0 | Ai_0],
Yi = [Xia, Wi_0, Mi | Yi_0],
Wi = [Xib, Ni | Wi_0]
```

where the Prolog notation [A, B | C] means a list that starts with A and B, and continues with the elements of list C. Note that for convenience we are using lists here, since the shape of the data structure is not significant for the abstract substitutions presented in Section 3.4, on which the sharing analysis is based. Each of the equations of the form X = t at the point of execution in a Prolog program where the currently computed substitution is σ, provided that X $\notin \mathrm{dom}(\sigma)$ (which is ensured by construction in our scheme), extends σ with a new mapping (X $\mapsto t$). The translation scheme above is rather generic and can be simplified in several ways, depending on the activity:

- If a_i is stateless, we remove the third equation from the scheme and replace W*i*_0 with [] (the empty list).

```
1   a16c(A16,A17,A26,A27,E,C,I,P,N,W4) :-
      % a16
3     A16 = [E],
      C = [E,_],
5     % a17
      A17 = [C,I],
7     P = [C,I],
      a26c(A26,A27,P,N,W4).
9
    a25c(A25,A26,A27,D,P,N,W3,W4) :-
11    % a25
      A25 = [D,W3],
13    P = [D,W3],
      a26c(A26,A27,P,N,W4).
15
    a26c(A26,A27,P,N,W4) :-
17    % a26
      A26 = [P],
19    W4 = [P,_],
      % a27
21    A27 = [W4],
      N = [W4].
```

```
23  a5_(A6_0,A8_0,A9_0,Q_0,W2_0,A6,A8,A9,Q,W) :-
24    % exit loop
      A6 = A6_0,
26    A8 = A8_0,
      A9 = A9_0,
28    Q = Q_0,
      W = W_0.
30  a5_(A6_0,A8_0,A9_0,Q_0,W2_0,A6,A8,A9,Q,W) :-
      % a6
32    A6_1 = [Q_0,W2_0|A6_0],
      Q_1 = [W2_0|Q_0],
34    a7c(A8_0,A9_0,Q_1,W2_0,A8_1,A9_1,Q_2,W2_1),
      % loop
36    a5_(A6_1,A8_1,A9_1,Q_2,W2_1,A6,A8,A9,Q,W).

38  a7c(A8_0,A9_0,Q_0,W2_0,A8,A9,Q,W2) :-
      % case a8
40    A8 = [Q_0,W2_0|A8_0],
      Q = [Q_0],
42    W2 = [Q_0|W2_0],
      % pass a9
44    A9 = A9_0.
    a7c(A8_0,A9_0,Q_0,W2_0,A8,A9,Q,W2) :-
46    % case a9
      A9 = [[]|A9_0],
48    % pass a8, q and w2
      A8 = A8_0,
50    Q = Q_0,
      W2 = W2_0.
```

Fig. 8 Fragments of translation to Horn clause form of the composition from Fig. 1

- If a_i has a state, but does not update it, we replace Xib and Ni in the third equation with [ ]. With respect to the abstract substitutions, Wi_0 and Wi $\equiv$ [[ ] , [ ] | Wi_0] are indistinguishable.
- If a_i updates its state without first reading it (e.g., by issuing an UPDATE SQL command), we replace Wi_0 in the first equation with [ ].
- If a_i is not inside a loop, we replace Ai_0 and Yi_0 with [ ].
- If we are not interested in data attributes, we replace Mi and Ni with [ ].
- When Mi or Ni need to be represented, but their content is not important, we can replace them with the underscore symbol "_" that represents an anonymous variable in Prolog.
- The scheme is easily extended to the activities that have several pieces of state and/or several outputs.

Example 12 Figure 8 shows several fragments of the Horn clause representation of the composition from Fig. 1, in Prolog notation (comments start with "%", and "$\leftarrow$" is written as ":-"). Lines 16–22 show a clause for predicate a26c that models data dependencies from a_{26} to the finish, i.e., for a_{26} and a_{27}. We use the same labels for data items and service state from Table 1, but in uppercase. Lines 18–19 model activity a_{26} (*Payment processing*). Line 18 indicates that the activity reads the transfer order p, and line 19 indicates that the state of the seller's account w_4 now depends on p and its earlier state (unknown to a_{26} and thus represented with the underscore). Note that a_{26} does not have a direct output, so the second equation is missing. However, w_4 is accessed by a_{27} (*Shipment*, modeled by lines 21–22), which checks that the payment has settled, and serves as the input for the shipment notice n. Since a_{27} does not modify

state w_4, the third equation is missing. All named variables from lines 17–22 are also found in the list of arguments to `a26c` in line 16, to propagate substitutions.

Example 13 Lines 1–14 in Fig. 8 show the clauses for predicates `a16c` and `a25c` that model data dependencies from a_{16} and a_{25} to the finish, respectively. `a16c` models the data dependencies of a_{16} and a_{17}, and calls `a26c` at its end. Likewise, `a25c` models the dependencies of a_{25} and calls `a26c`.

For complex constructs, such as loops and XOR-splits, the translation generates additional Prolog clauses depending on the type of construct. These are illustrated in the examples that follow.

Example 14 Lines 23–36 in Fig. 8 show the translation of the loop construct a_5 as predicate `a5_`. The first clause (lines 23–29) models the case of exiting from the loop where the initial values from a previous iteration (with suffix "`_0`") are propagated to the exit. The second clause (lines 30–36) models a loop iteration. Its body consists of the translation of the loop body: activity a_6 (*Search catalog*) in lines 32–33, and a_7 which is an XOR-split implemented in line 34 as a call to a specially generated predicate `a7c` (see the next example). The variables in the iteration use suffix "`_0`" if they come from a previous iteration, and suffix "`_1`" if they result from the current one. The final line (36) recursively calls `a5_` and passes to it the "`_1`" versions as the new initial ones.

Example 15 Lines 38–51 in Fig. 8 show the translation for the XOR-split (*if-then-else*) construct a_7. Each of the two branches is translated as a separate clause of `a7c`. Lines 40–42 in the first clause model activity a_8 (*Add to cart*), which uses the previous values of q and w_2 (line 40, see Table 1) to update q and w_2, respectively (lines 41–42). Line 47 in the second clause shows the translation for a_9 (*Skip option*), which does not alter the state, nor produces any outputs. Since both clauses need to have the same interface to the calling code (in line 34), they enumerate the union of all variables from translations of both a_8 and a_9. The variables that are not used in a clause are propagated from the previous values with the suffix "`_0`". This is the case with `A9` in the first clause (line 44), and with `A8`, `Q` and `W2` in the second clause (lines 49–51).

4.4 Sharing analysis proper

In this subsection we give a brief overview of the actual sharing analysis employed in our approach. It is an instance of abstract interpretation [7], a static analysis technique that interprets a program by mapping concrete, possibly infinite sets of variable values onto (usually finite) abstract domains, together with data operations, in a way that is correct with respect to the original semantics of the programming language. In the abstract domain, computations usually become finite and easier to analyze, at the cost of lack of precision, because abstract values typically cover (sometimes infinite) subsets of concrete values. However, the abstract approximations of the concrete behavior are *safe*, in the sense that properties proven in the abstract domain necessarily hold in the concrete case. Whether abstract interpretation is precise enough for proving a given property depends on the problem and on the choice of the abstract domain.

Yet, abstract interpretation provides a convenient and finite method for calculating approximations of otherwise, and in general, infinite fixpoint program semantics, as is typically the case in the presence of loops and/or recursion.

We use a combined, abstract interpretation-based sharing, freeness, and groundness analysis for logic programs [30], which computes abstract substitutions and freeness information as described in Sect. 3.4. For a deterministic Horn clause program (i.e., with a single possible execution path) without loops, the sharing analysis computes abstract substitution $\Theta = \alpha_Y(\sigma)$, where σ is the substitution computed by that program, and Y are argument variables to a predicate that is called (e.g., arguments of `a26c` in Fig. 8). For programs with non-determinism (i.e., where several control flows are possible, as in the case of the two clauses of `a7c`), the sharing analysis computes $\Theta = \bigcup_{i=1}^{n} \alpha_Y(\sigma_i)$, where $\sigma_i, \ldots, \sigma_n$ are the substitutions computed by the alternatives. And, for the cases of looping (such as `a5_`), the sharing analysis computes a fixed point Θ that is either equal or a superset of any $\alpha_Y(\sigma)$ where σ can be computed by the loop. Therefore, the result Θ is a safe approximation, in the sense that it includes all possible sharing groups. The sharing analysis is combined with a freeness analysis, which infers which variables are unbound, i.e., have not been substituted with a non-variable term. The sharing analysis also infers groundness information, determining with variables are bound to terms that do not contain any variables (note that those variables can be excluded from any sharing group in which they may appear). Some logic program analysis tools, like CiaoPP [16], have been developed which give users the possibility of running different analysis algorithms on input programs. We build on one of these analysis available in CiaoPP: `shfr` [6,30].

When analyzing attributes, the inputs to the Horn clause program that represent incoming messages and internal activity attributes are normally initialized to contain a configuration of variables that represents conceptually the content in terms of data attributes of such messages. For instance, input `D` in predicate `a25c` from Fig. 8 (lines 10–14) corresponds to one of the cases of identification documents from Figure 5, which are characterized by the attributes $m_1, \ldots, m_5$ from Fig. 4. If `D` represents a passport, we can add `D=[M1,M2,M3]` before calling `a25c`. This is not necessary if `D` is the only input that uses these attributes.

4.5 Interpretation of sharing results

As mentioned above, the result of sharing analysis is an abstract substitution Θ such that $\alpha_Y(\theta) \subseteq \Theta$ for all concrete substitutions θ that can be computed by the Horn clause program. In other words, no potential sharing group is left from Θ. Therefore we can construct a relation $\leq_s$ from Θ in the same way as $\sqsubseteq_s$ is constructed from $\alpha_Y(\theta)$ in Definition 9. However, it can be easily verified that such $\leq_s$ must be a subset of $\sqsubseteq_s$ for each θ for which $\alpha_Y(\theta) \subseteq \Theta$.

Proposition 3 *For an abstract substitution Θ and its relation $(\leq_s) \subseteq Y^2$ between variables of interest from Y, and for any concrete substitution θ such that $\alpha(\theta) \subseteq \Theta$ and the relation $(\sqsubseteq_s) \subseteq Y^2$ induced by it, it holds that $(\leq_s) \subseteq (\sqsubseteq_s)$.*

Proof First, note that $x \sqsubseteq_s x$ and $x \leq_s x$ hold trivially for each $x \in Y$. Next, suppose that for arbitrary distinct $x, y \in Y$ we have $x \leq_s y$, but not $x \sqsubseteq_s y$. That is only possible if $\alpha_Y(\theta)$ contains some sharing group S such that $x \in S$, but $y \notin S$. But since $\alpha_Y(\theta) \subseteq \Theta$, then $S \in \Theta$ also, which conflicts with the assumption $x \leq_s Y$. Therefore, no such S can exist, i.e., it follows that $x \sqsubseteq_s y$ must hold.

Example 16 Suppose that $Y = \{x, y\}$ and two possible concrete substitutions are $\theta_1 = \{x \mapsto f(u), y \mapsto g(u, v)\}$ and $\theta_2 = \{x \mapsto g(u, v), y \mapsto f(v)\}$. Then, $\alpha_Y(\theta_1) = \{xy, y\}$, i.e., $(\sqsubseteq_s)_1 = \{\langle x, y\rangle\}$, and $\alpha_Y(\theta_2) = \{x, xy\}$, *i.e.*, $(\sqsubseteq_s)_2 = \{\langle y, x\rangle\}$. However, for $\Theta = \alpha_Y(\theta_1) \cup \alpha_Y(\theta_2) = \{x, xy, y\}$, we have $(\leq_s) = \varnothing$.

This means that $\leq_s$ derived from Θ can be used as a lower approximation for $\sqsubseteq_s$ induced by any concrete θ. A natural way to compute the upper bound for $\sqsubseteq_s$ is given using the following definition:

Definition 12 (*Maximal sharing ordering*) Let Θ be an abstract substitution, and Y a set of variables of interest. For arbitrary $x, y \in Y$, we say that $x \bar{<}_s y$ if either no $S \in \Theta$ contains x, or there exists some $S \in \Theta$ such that $x \in S$ and $y \in S$.

Proposition 4 *For an abstract substitution Θ and its relation $(\bar{<}_s) \subseteq Y^2$ between variables of interest from Y, and for any concrete substitution θ such that $\alpha_Y(\theta) \subseteq \Theta$ and all variables from Y appear in $\alpha(\theta)$, with the relation $(\sqsubseteq_s) \subseteq Y^2$ induced by it, $(\sqsubseteq_s) \subseteq (\bar{<}_s)$.*

Proof Suppose that for arbitrary $x, y \in Y, x \sqsubseteq_s y$. Since all variables from Y must appear in at least one sharing group in $\alpha_Y(\theta)$, and $x \sqsubseteq_s y$, then there has to exist $S \in \alpha_Y(\theta)$ such that $x \in S$ and $y \in S$. From $\alpha_Y(\theta) \subseteq \Theta$, we conclude that $S \in \Theta$, and from Definition 12, we obtain $x \bar{<}_s y$.

To summarize, $x \leq_s y$ implies that all components of x are necessarily components of y, while $x \bar{<}_s y$ implies that all components x may possibly be components of y. Both $\leq_s$ and $\bar{<}_s$ are directly obtained from the result of sharing analysis Θ.

4.6 Complexity and precision of the sharing analysis

The shfr sharing and freeness analysis for logic programs is known to produce the most precise sharing results (i.e., the least over-approximation of Θ), but its computational cost may grow, in the worst case, exponentially with the number of variables. Other, more efficient, but less precise sharing analysis techniques have been proposed, such as the clique sharing analysis [31], or the pair-sharing analysis [24]. An abstract substitution Θ' obtained from such a less precise sharing analysis technique is generally a superset of Θ obtained from shfr. It can be easily verified that in that case:

$$(\leq_s') \subseteq (\leq_s) \subseteq (\sqsubseteq_s) \subseteq (\bar{<}_s) \subseteq (\bar{<}_s')$$

where $\leq_s'$ and $\bar{<}_s'$ correspond to Θ'.

On the other hand, one way to increase the precision of the approximation with $\leq_s$ and $\bar{<}_s$ is to remove some of the alternative clauses from the Horn clause program whose effect on Θ is to inflate it due to the union of abstract substitutions for each alternative. That can be done, for instance, when a (partial) trail of the execution of the composition is known.

Example 17 Let us take a look at the clauses of predicate `a7c` from Fig. 8, lines (38–51). The abstract substitution from the first clause Θ_1 is (in Prolog syntax) `[[A8_0,A8], [A9_0,A9], [Q0_0,A8,Q,W2], [W2_0,A8,Q,W2]]`, while Θ_2 from the second clause is `[[A8_0,A8], [A9_0,A9], [Q_0,Q], [W2_0,W2]]`. Their union $\Theta = \Theta_1 \cup \Theta_2$ is more general, but less precise than both Θ_1 and Θ_2. Learning which branch was taken from the traces, or predicting which branch will necessarily be taken eliminates either Θ_1 or Θ_2 and gives a more precise result.

5 Examples of application

In this section we show how the framework for functional dependencies from Sect. 3 and the analysis method from Sect. 4 can be used to address the problems mentioned in Sect. 2 as motivation for this work.

The steps described in Sect. 4 are currently almost completely automated and we have developed prototype tools centered around the CiaoPP program analysis and transformation system [16] that accept a description of a service composition in an abstract composition form (introduced in Sect. 4.1), prepare the Horn clause representation of the composition which is subject to the sharing analysis, and extract the sharing results from the analysis outputs. We are working on a set of pre-processing tools that accept composition definitions written in (fragments of) widely accepted composition languages, such as BPMN, WS-CDL [41], and BPEL [21]. The prototype tools also export the sharing results in a form suitable for FCA-based concept lattice visualization using external tools.

5.1 Parallelization

The general control structure of a service composition can often be adapted to provide more flexibility while not violating the control and data dependencies. One example is parallelization, which allows composition activities to start as soon as their control and data dependencies allow. Automatic parallelization can be performed by interpreting the results of the sharing analysis over variables that represent data items and or activities (Sect. 4.5). Our criterion for parallelization will be based on the following: an activity can start as soon as all the necessary data (including the internal state of the component services) is available. Note that the necessary control and data dependencies are already encoded in the ordering of activities in the composition in the translation to the Horn clause program.

Table 2 A representation of a sharing result Θ as a context

Group/attrib.	Activity inputs										
	a_0	a_2	a_4	a_5	a_{10}	a_{12}	a_{14}	a_{16}	a_{17}	a_{26}	a_{27}
S_1		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S_2			✓	✓	✓	✓	✓	✓	✓	✓	✓
S_3				✓	✓		✓		✓	✓	✓
S_4									✓	✓	✓
S_5										✓	✓

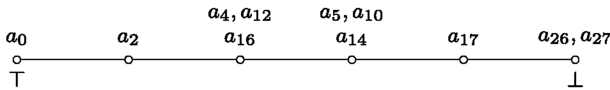


Fig. 9 Conceptual lattice (laid out horizontally for convenience) grouping together activities and data based on functional dependencies

Let us look at a class of use cases of the composition in Fig. 1 where the user always logs in (instead of starting an anonymous session), and where payments are always made by credit card. In that class of use cases, from the composition model in Fig. 1 we prune branches with activities a_3, a_{13}, a_{18}, and a_{19}. The sharing analysis of the pruned composition returns an abstract substitution Θ with five sharing groups S_1, S_2, S_3, S_4, and S_5, over the set Y of variables of interest which represent activity inputs. The membership of the variables from Y in the sharing groups is shown schematically in Table 2.

The necessary condition for parallelization of two activities is that both draw all of their inputs from the same set of previously computed data items and component states. We use the relation $\leq_s$ derived from Θ, because we are interested only in functional dependencies that hold under any concrete substitution. For two variables $x, y \in Y$, $x \leq_s y$ guarantees that all data needed to compute x is included in the data needed to compute y. Or, equivalently, if there is not enough data to compute x, then y cannot be computed either. If we recall the notion of FCA concept lattices from Sect. 3.2, with Y as the set of objects, Θ as the set of attributes, and the membership of variables from Y in the sharing groups from Θ, then we can easily verify that for each $x \in Y$, $\{x\}'' = \{y \in Y \mid x \leq_s y\}$. Therefore, we can represent the conceptual hierarchy of activities, induced by the guaranteed sharing ordering $\leq_s$, in the form of a concept lattice, such as in Fig. 9 for our parallelization example.

The relationship between this conceptual hierarchy and the functional relationships modeled with "$\rightarrowtriangle$" is the following: if the inputs of a conceptually higher set of activities A_1 depend on some set of inputs X and component states U (i.e., $XU \rightarrowtriangle A_1$), then for a conceptually "lesser" set of activities A_2, we have $XUV \rightarrowtriangle A_2$, where V is some non-empty set of additional inputs or updated states of component services. This property holds even in a general case, when the lattice does not have a linear form as in Fig. 9. Within a group of activities at the same level in this conceptual hierarchy, control dependencies can be freely rearranged as long as they do not clash with data

dependencies: if an activity a receives the output of a', then a must come after a', but otherwise a and a' can be parallelized.

In Fig. 9, activity a_0 at node $\top$ waits for the input u, without needing any previous information. Activity a_2 receives u and produces w_4 (the next concept node after $\top$). In the second concept node from $\top$, activities a_4, a_{16}, and a_{12} can all be executed in parallel, while for the fourth node, a_5, a_{10}, and a_{14} must be executed sequentially in that order because of the data dependencies shown in Table 1. Note that in the latter case, the sequence a_5–a_{10}–a_{14} represents a sub-workflow for an iterative *browse-select-iterate-exit* process typical of e-commerce Web portals, but can in principle be replaced by another set of activities that implement a different procedure for creating an invoice, e.g., based on a list of items supplied by the buyer, without the activities that come before or later observing any difference.

When evaluating the usability of the sharing approach for the automatic parallelization of activities, we need to take into account the complexity and accuracy of the sharing analysis on the one hand, and the quality of information about data dependencies in the workflow on the other hand. Using our prototype tools, the translation of a service workflow encoded as an abstract composition model (Sect. 4.1) is linear in the number of activities and data items in the composition, and does not present a major computational overhead. Also, the interpretation of the results from the sharing analyzer is straightforward, since the structure of the resulting abstract substitution is directly transposed into the shape used by Table 2 (unless there is a need for visualization which involves more complex lattice construction). The greatest part of the computational complexity when applying the proposed approach is consumed by the sharing analysis proper using CiaoPP. On a low-end personal computer running Mac OS X v.1.7.5, this stage consumes approximately between 1,100 and 1,800 ms, depending on the run.

When it comes to the quality of information on data dependencies in the composition, it should be noted that we may not normally have full information about how each invocation of a component service affects its internal state, which was in our motivation example explicitly represented in the lower part of Table 1. In the case of several operations on the same service (unless we know it is stateless), to ensure correctness we need to make a safe assumption that each operation may modify its internal state. That may introduce additional data dependencies which tend to reduce the level of parallelization, by losing opportunities to parallelize activities when that is not safe.

The left-hand side of Fig. 10 shows a simple sequence of six service invocations, which refer to operations on three distinct services (1), two services (2), and a single service. If the operation s_1 updates the state of Service A, this in case (1) creates an additional data dependency between s_1 and s_2, in case (2) between s_1 and s_2 and between s_1 and s_3, in case (3) between s_1 and each s_i, $i = 2..6$, etc. The graph on the right-hand side of Fig. 10 shows the proportion of possible parallelization opportunities used, depending on the quality of the assumptions about the impact of the invocations on the state of the respective services. The values in the graph are averages across all possible combinations of state impact for the six activities and the corresponding safe assumptions, crossed with all possible combinations of forward data dependencies between the operations on different services.

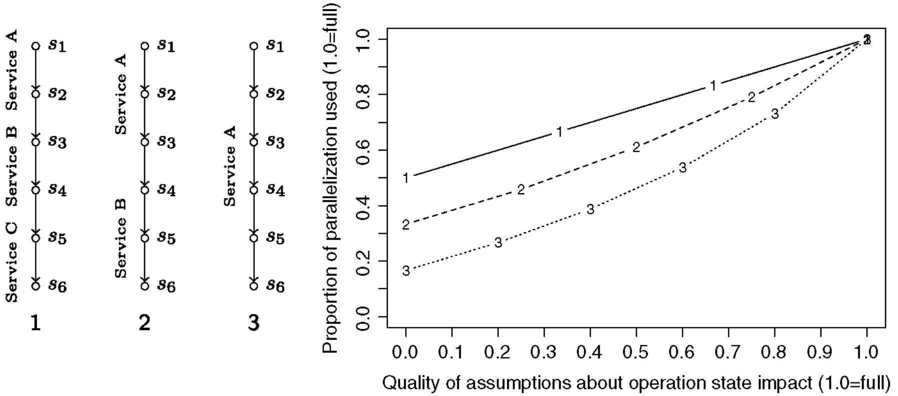


Fig. 10 Effect on the information on service statefulness on the level of parallelization

The rightmost point on the graph unsurprisingly shows that the completely correct assumptions lead to utilization of all opportunities for parallelization in the sequence in all three cases. However, as the quality of the assumptions decreases, by assuming more impact on the state than necessary, more and more parallelization opportunities are lost, depending on how many actual services are involved. In case (3), where all operations belong to the same service, assuming that they all affect its state (when that is not the case) leads to a loss of about 83 % of all parallelization opportunities (counting parallelization of each s_1–s_6 separately), while in case (1) the loss is smaller (about 50 % on average) because the additional dependencies caused by the wrong assumptions play a smaller role compared to all other potential data dependencies.

5.2 Fragmentation

It is often of interest to take a service composition that is designed and represented as an orchestration, i.e., with a centralized control flow, and to break it into parts (called fragments) that can be executed in a distributed manner, possibly on servers that belong to different organizational domains. That process is called fragmentation, and it is a form of adaptation that can be applied at design time or at run time [26]. We can use the sharing approach to support fragmentation by assigning activities to organizational-domain-based fragments based on the content of the data they handle. This time we model data attributes that describe the content of data, in the sense discussed in Sect. 3.2. We extend our earlier work on automatic attribute inference and fragment identification based on sharing [18,19].

To illustrate the approach, we look at the part of the service composition from Fig. 1 which starts with the activity a_{15} (an XOR-split), and look only at the branches that correspond to credit card payment (activities a_{16}, a_{17}) and bank transfer (a_{18}). We modify slightly the generation of Horn clauses to expose the component state w_4 and the credit card information c as input variables of interest for the analysis, along with the invoice i and the user info e that are inputs to that part of the composition from Fig. 1.

Table 3 The sharing result Θ for a case of fragmentation based on data sharing

Data object	Sharing groups from Θ											
	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}
e	✓						✓					
c		✓						✓				
i			✓						✓			
w_4				✓								
a_{16}	✓										✓	
a_{17}		✓	✓									✓
a_{18}						✓			✓			
a_{26}		✓	✓						✓	✓		
a_{27}		✓	✓	✓					✓	✓		
n		✓	✓						✓	✓		
p		✓	✓	✓					✓	✓		

$$\begin{aligned} e\,\bar{<}_s &= \{e, a_{16}\} \\ c\,\bar{<}_s &= \{c, a_{17}, a_{26}, a_{27}, p, n\} \\ i\,\bar{<}_s &= \{i, a_{17}, a_{18}, a_{26}, a_{27}, p, n\} \\ w_4\,\bar{<}_s &= \{w_4, a_{27}, n\} \end{aligned}$$

(a) The summary of $\bar{<}_s$ for attribute carriers.

Activity	Inherits from			
	e	c	i	w_4
a_{16}	✓	×		
a_{17}		✓	✓	
a_{18}			✓	
a_{26}		✓	✓	
a_{27}		✓	✓	✓

(b) The attribute inheritance relation.

Fig. 11 Attribute inheritance arising from by Θ in Table 3

The result of the sharing analysis in this case is an abstract substitution Θ with twelve sharing groups, shown schematically in Table 3. A horizontal line visually separates the variables that are the source of data attributes (e, c, i and w_4) from others whose attributes we wish to infer. In this case, Θ is a union of abstract substitutions arising from two alternative paths of execution (a_{16}–a_{17}–a_{26}–a_{27} and a_{18}–a_{26}–a_{27}), and includes sharing groups from both. We wish to assign data attributes in a safe way, i.e., without leaving any potential attribute out. Therefore, this time we use the relation $\bar{<}_s$, which is an upper approximation of the actual sharing ordering $\sqsubseteq_s$.

For $x \in Y$, we write $(x\bar{<}_s)$ to denote the set of all $y \in Y$ such that $x\bar{<}_s y$. Figure 11a shows the summary of $\bar{<}_s$ for variables that carry data attribute. Part (b) of the figure shows that summary turned into an attribute inheritance relation where the "✓ " marks come from part (a), and the "×" mark come from the fact that c is an output of a_{16} in Table 1 (note that a variable a_i in Table 3 and Fig. 11a represents only the inputs to the activity with the same name). The attribute inheritance relation tells us that the activities can be grouped into four groups:

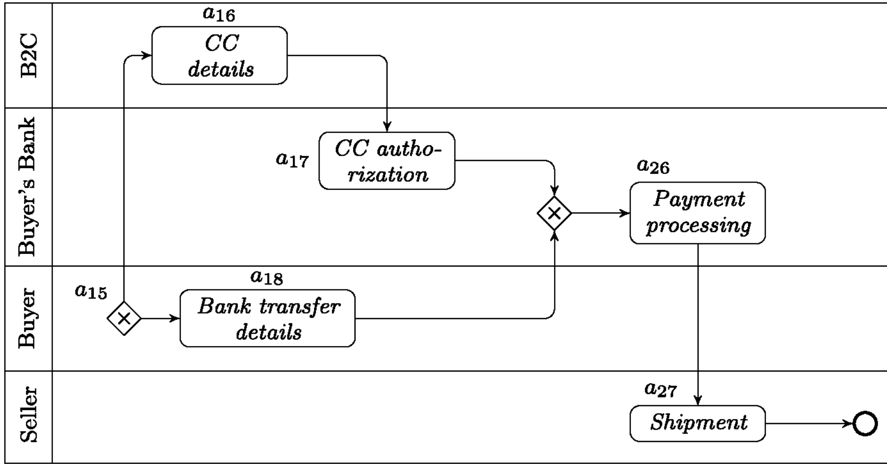


Fig. 12 A sample fragmentation scheme between four domains

- a_{16} handles only the user profile (e) and his or her credit card details (c), without needing to know about the goods purchased (i) or the state of the seller's bank account (w_4). This job can be outsourced to any freely or commercially available B2C online payment portal.
- a_{18} handles the information from the invoice i to collect the buyer's bank account information and produce the bank transfer order p. This can, in principle, be done by the buyer.
- a_{17} and a_{26} handle the information on the buyer's credit card (c) and the invoice (i) to issue and process the payment order (p), and to realize the order (a_{26}). This job is best suited for the buyer's bank. Note that a_{26} does not always see c, but only when the a_{16}–a_{17} branch is taken. However, our goal is to assign attributes for the most general case.
- a_{27} handles the buyer's credit card details (c), the invoice (i), and the state of the seller's account (w_4) to produce the shipment notice (n). This job is best handled by the seller. Note again that the credit card information is only potentially present.

Figure 12 shows a sample fragmentation scheme where part of the original composition is remodeled as a choreography that involves four communicating fragments placed in domain/role swim-lanes: Seller, Buyer, Buyer's Bank, and B2C, following the above classification of activities based on inheritance of data attributes.

Note that in principle the fragmentation based on data attributes can be combined with the parallelization approach from the preceding subsection, to obtain finer-grained fragments that reflect both data attribute inheritance and functional dependencies. e.g., using the lattice from Fig. 9, we can further subdivide the fragment in the Buyer's bank swim-lane into two, a_{17} and a_{26}. A basis of fine-grained composition fragments can be useful for automatic, on-demand merging of fragments based on the desired functionality or Quality of Service (QoS) constraints [44].

Using our prototype implementation, in our previous work [19] we have evaluated the sharing-based approach to fragmentation using an E-Health case study collected

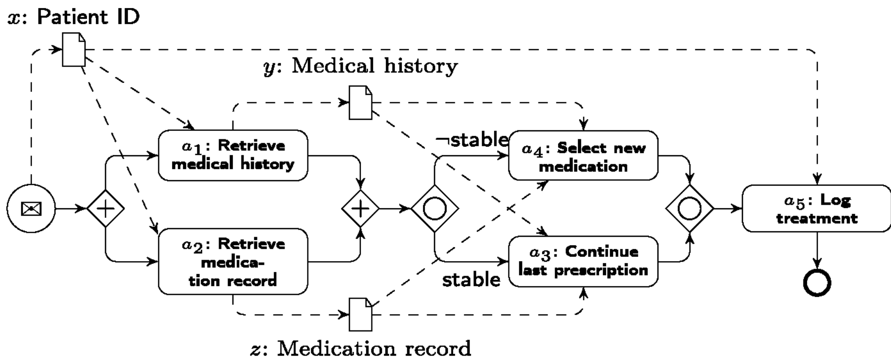


Fig. 13 A drug prescription workflow in BPMN from E-Health scenario

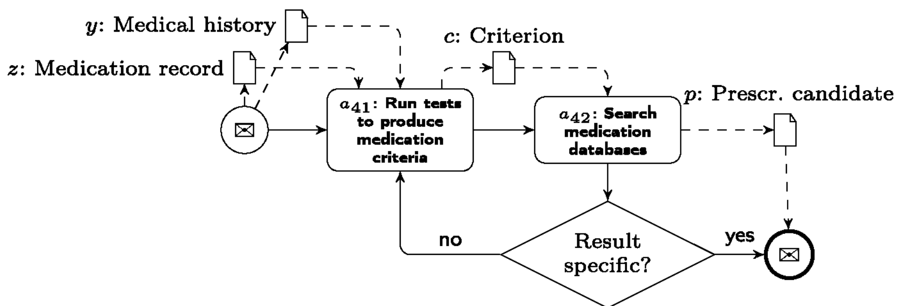


Fig. 14 BPMN diagram of the composite activity a_4 from Fig. 13

within the S-Cube project [12]. Figure 13 depicts a drug prescription workflow in BPMN notation, annotated with data dependencies. The process is initiated by the arrival of a patient with an appropriate identification (labeled as x in the figure). Next, two parallel activities (a_1 and a_2) are run to retrieve the patient's medical history and medication record. The data items resulting from these two activities are respectively marked y and z. Additionally, while retrieving the medical history, activity a_1 informs about the stability of the health of the patient. Depending on it, either the last prescription is continued (activity a_3) or new medication is selected (activity a_4). Finally, the treatment of the patient is logged (activity a_5). Activity a_4 is in itself a service composition, shown in Fig. 14. It contains a loop that iteratively refines the prescription based on medical tests.

The organization responsible for medicine prescription may want to split the workflow among several partners, based on what kind of information they are allowed to handle. *Registry and Archive* cannot look into the patient's symptoms, tests, or insurance coverage data. *Medical examiners* can at most see the symptoms and tests, without reference to the coverage information. *Medication providers* can only take care of symptoms and coverage, without reference to the medical tests. All tasks that cannot be assigned to the partners according to these rules are kept by the central *Health organization.*

Item	Name	Address	SSN	Symp.	Tests	Cover.
x	✓	✓	✓			
d				✓	✓	
e				✓		✓
a_2, z	✓	✓	✓	✓		✓
a_1, y, p, a_{42}, c	✓	✓	✓	✓	✓	
a_3, a_4, a_{41}	✓	✓	✓	✓	✓	✓
a_5	✓	✓	✓			

Fig. 15 The resulting context for the drug prescription workflow analysis

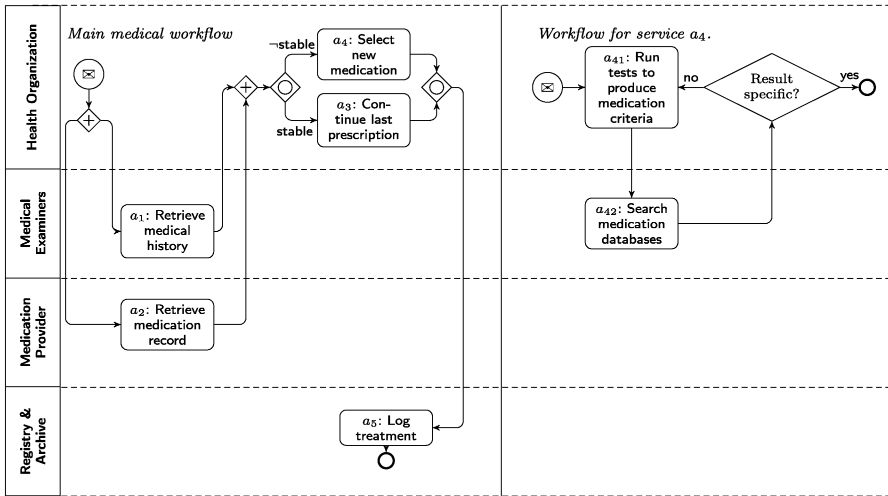


Fig. 16 An example fragmentation for the drug prescription workflow

Figure 15 shows the sharing analysis results for the drug prescription workflow. The upper part of the table (above the line) shows inputs, where the columns represent data attributes, x stands for PatientID in Fig. 13, while d and e represent the contents of the databases used by activities a_1 and a_2 to produce (on the basis of x) data items y (Medical history) and z (Medication record), respectively. The lower part of the table shows the inferred attributes of all intermediate data items and activities from the main workflow and the sub-workflow for activity a_4. Figure 16 shows the assignment of activities to fragments that correspond to the health organization and its partners.

The table in the upper part of Fig. 15 shows an alternative assignment of the attributes to the inputs to the drug description workflow, where insurance coverage information also appears in the medical history database, and forms a part of the medical history record for the patient. In this case, the sharing analysis results give rise to a reassignment of activities to the swim-lanes as shown in the lower part of Fig. 17. In this case, the health organization keeps to itself the activities that were delegated to *Medical Examiners* in Fig. 16, since it is not safe to entrust such external entity with the insurance coverage details.

Item	Name	Address	SSN	Symp.	Tests	Coverage
$\mathbf{x}$, a_5	✓	✓	✓			
$\mathbf{d}$				✓	✓	✓
$\mathbf{e}$				✓		✓
a_2, z	✓	✓	✓	✓		✓
y, p, other a.	✓	✓	✓	✓	✓	✓

Swimlane	*Activities*
Health Organization	a_1, a_3, a_4, a_{41}, a_{42}
Medical Examiners	(*empty*)
Medication Provider	a_2
Registry & Archive	a_5

Fig. 17 Alternative context and fragmentation scheme for the drug prescription workflow

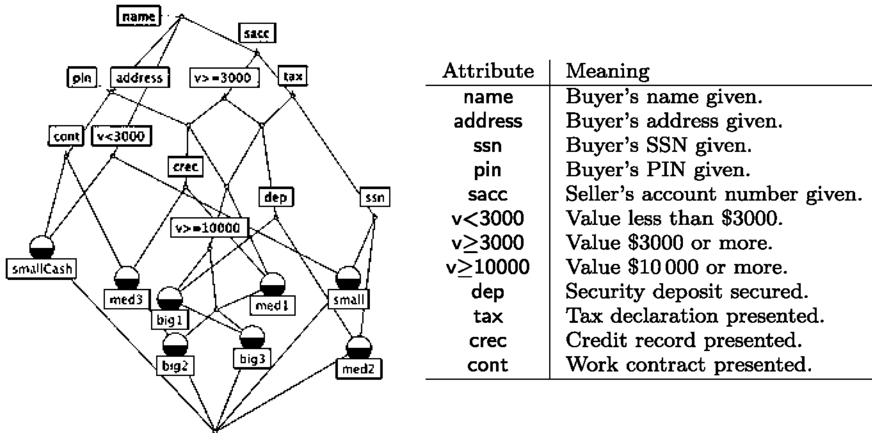
5.3 Constraining component search and validation

As mentioned in Sect. 3.2, the compliance of a message with a structural data description (a semantic data type accepted by the service) in itself does not guarantee that an invoked service will be able to perform its task successfully. The reason for that is that XML messages may have many optional or alternative parts that may be present or absent in a structurally compliant message, yet whose presence or absence may cause the service to fail. Or, the message may contain references (foreign keys) to non-existent or wrong entities. In our approach, we use data attributes to represent the content, rather than the structure of a message, which, on top of the structural matching, may help us in reasoning about whether some service implementation is suitable for the given task.

Let us look again at the example composition from Fig. 1, where activity a_{25} performs loan approval based on the documents collected in a_{19} and a_{20}. Let us suppose that, as it usually happens in reality, these documents are not passed repeatedly to a_{25}, but are rather stored in a "loan request file" for the buyer (a "logical" file, not a file in the O.S. sense), represented with w_3 in Table 1, which is created by a_{19}, updated by a_{20}, and consumed by a_{25}. For the sake of argument here, we shall also assume that the seller's invoice i is inserted into w_3, so that a_{25} accesses w_3 as an integrated super-document with all pieces of information placed inside.

Figure 18 presents a hypothetical concept lattice of eight credit approval candidate services, for "small," "medium," and "big" consumer loans. The small loans are up to 3000 monetary units, the medium ones are between 3000 and 10000, and the big loans are 10000 monetary units or more. The candidate services are characterized by means of the required data attributes of the loan request file w_3 at their inputs. Small loans can be approved in cash by smallCash. It requires name, pin, address, and cont (besides $\mathsf{v} < 3000$), while all other loan approval services pay directly to the seller, and therefore require sacc. All attributes required by a candidate can be collected from the lattice diagram by following all lines that go from it to the top. e.g., form med2, that is the set {ssn, tax, sacc, name, dep, $\mathsf{v} \geq 3000$}.

Let us now suppose that, as a result of a previous data attribute analysis, we conclude that the identity document d supplied by a_{19} at the start has some set of attributes $D \subseteq$ {name, address, ssn, pin}, and that the set I of attributes for the invoice i is



Attribute	Meaning
name	Buyer's name given.
address	Buyer's address given.
ssn	Buyer's SSN given.
pin	Buyer's PIN given.
sacc	Seller's account number given.
v<3000	Value less than $3000.
v≥3000	Value $3000 or more.
v≥10000	Value $10 000 or more.
dep	Security deposit secured.
tax	Tax declaration presented.
crec	Credit record presented.
cont	Work contract presented.

Fig. 18 A hypothetical concept lattice of credit approval procedures

a_{22}	a_{23}	a_{24}	Validated candidates
−	−	−	(none)
+	−	−	small, med1
−	+	−	(none)
+	+	−	small, med1, med2, big1, big3
−	−	+	(none)
+	−	+	smallCash, small, med1, med3, big2
−	+	+	(none)
+	+	+	smallCash, small, med1, med2, med3, big1, big2, big3

Activity	Contributes attributes
a_{22}	pin, tax, crec
a_{23}	dep
a_{24}	cont, ssn

Fig. 19 Adaptive search for loan approval service candidates

one of {sacc, v $<$ 3000}, {sacc, v $\geq$ 3000}, or {sacc, v $\geq$ 3000, v $\geq$ 10000}. The question we ask is what loan service candidate we need to chose to ensure that enough information is provided to the service to perform its function.

Figure 19 shows an adaptive search for service candidates for $D = \{$name, address, ssn$\}$, depending on a combination of a_{22}, a_{23}, and a_{24} executed within a_{20}. Each of these activities contributes some additional information, represented in the lower table in the figure. In this case, a_{22} must always be executed, and for a "big" loan, also at least one of a_{23} or a_{24}. If D was computed based on the minimal sharing (relation $\leq_s$), we obtain candidates that certainly comply, and if it was computed based on $\bar{<}_s$, we obtain the set of all potential candidates.

6 Related work

This paper builds on the previous publications by the authors that focused on fragmentation and attribute inference for service compositions [18,19]. In this paper we expand the approach by providing a common, logic-based foundation for both representing

and reasoning about functional dependencies as well as data attributes in service compositions, with the goal of supporting adaptation. We also introduce and formalize the upper and lower approximation of the sharing ordering which are, e.g., used for parallelization, fragmentation, and component search and validation.

Service adaptation has been widely studied, and [11] gives a good overview of the methodological framework and some of the techniques proposed. The problem of automatic adaptation of service interfaces at the level of protocols in terms of protocol realizability and compatibility has been extensively studied, for instance by Ponge et al. [35], and other authors. In our approach, we start from the assumption that the adapted composition, or its fragments, need to preserve the original protocol present before the adaptation. The difference is that we do not use transition systems and automata to reason about the protocol invariants, but instead introduce data dependencies on the shared component state between the invocations. While well suited for analyzing functional dependencies and data attributes, more advanced interactions involving, e.g., timed conversations and transactional behavior would require combining our approach with the protocol-based techniques.

Another interesting area in service adaptation research relates to automatic conversion of operations and message formats based on semantic descriptions. Describing the semantics of Web services, message types, and operations, as well as Web service search and matching, has been well studied, and a good overview and guide through the current state of the art can be found in Euzenat et al. [13]. In this paper we deal with the issue of semantic matching indirectly, by using Boolean tests to detect properties of data (Definition 3 in Sect. 3.2), while not making a direct use of more powerful data abstraction techniques based on, e.g., XQuery and XSLT. However, our discussion of component search and compliance testing (Sect. 5.3) relies on the assumption that adequate semantic descriptions and registries of prospective component candidates are readily available.

Automatic service composition, based on adaptive planning, has been studied by Beauche and Poizat [4]. It represents an alternative to our parallelization approach (Sect. 5.1) in that it starts from a set of service components and their pre- and post-condition and data, and tries to combine them into a composition (with parallelization when possible). Our sharing-based analysis, in comparison, starts from an already existing composition, which is then analyzed and decomposed into subsets of activities that can be parallelized. Therefore, in our approach, the pre- and post-conditions of the basic blocks and the properties of their data are inferred rather than given in advance, which is well suited for finer-grain parallelization based on *ad hoc* artifacts.

Service fragmentation, as a form of adaptation, has been surveyed by Mancioppi et al. [27]. In this paper, we are dealing with the problem of deciding which activities in the original composition should be assigned to which fragments, and we choose the information content described by data attributes to be the criterion. Other authors have proposed different criteria for fragmentation. Tan and Fan [37] proposed a technique for workflow fragmentation in a way that maximizes the distribution of process activities among nodes of a grid or cluster of process execution (enactment) engines. The proposed approach is a form of run-time fragmentation that is transparent to the user/designer and is meant to be automatically applied by the nodes of the distributed process execution engine. The work by Yildiz et al. [14,42] concentrates

on fragmenting workflows between services in different business domains. The data items that are passed between the external services have different security or confidentiality levels, and therefore need to be protected from unauthorized lookup in different domains. Similarly to ours, their approach is also motivated by information flow control, but is restricted to acyclic workflows. There are also other approaches to fragmentation, such as that by Zaplata et al. [43], which instead of partitioning the composition, assign different execution paths to nodes in the distributed enactment environment. Of course, after assigning activities to fragments using some criteria, the actual work of creating and deploying executable fragments involves many technical details and is very dependent on the composition language. For BPEL, a detailed discussion can be found in the work by Khalaf and Leymann [22,23].

The technical foundations of variable sharing analyses for logic programs have been proposed by Jacobs and Langen [20] and by Muthukumar et al. [28–30] and have been used effectively in program parallelization by Bueno et al. [6,28]. These analyses are instances for logic programs of the framework of abstract interpretation, a general approach to program analysis that was originally proposed by Cousot and Cousot [8] and has been since applied to a great variety of languages and properties. An overview of its general application to several analysis problems can be found in the book on program analysis by Nielson and Hankin [32]. A clique sharing analysis that offers interesting cost trade-offs has been described by Navas et al. [31].

Using Formal Concept Analysis (FCA) for representing and reasoning about conceptual properties of objects is described in the standard texts by Ganther and Stumme [15] and Davey and Priestley [9].

7 Conclusions

Sharing analysis can be used as an underlying technique for ensuring correctness of adaptation actions in service compositions, by taking into account and analyzing both the control and data dependencies. Two important classes of data dependencies—functional dependencies and data attributes—can be captured using a single representation framework that centers on the notions of logic variables, substitutions, and Horn clauses, for which well-developed sharing analysis techniques and tools exist. The technique is well suited for compositions involving complex control structures, including loops, branches and parallel flows. Parallelization, fragmentation, and component selection and compliance checking are some forms of adaptation whose correctness depends on respecting—and can be informed by—the data sharing invariants inferred by means of sharing.

In this paper we have presented a sharing-based framework for supporting adaptation of service compositions by means of analysis of functional dependencies and data attributes pertaining to data objects, component states, and activities inside a composition. The results of the analysis can be used for several adaptation related tasks: e.g., for rearranging or parallelizing activities, fragmenting a composition based on the attributes of data handled by its activities, or constraining the search for the replacement components.

The logical basis of representation allows us to derive pure Horn clause programs, a subset of standard Prolog programs, to capture both data and control dependencies, in the presence of complex control structures, such as branches and loops. On such programs we apply an analysis of sharing of logic variables which produces results that aggregate all possible groups of variables (representing data objects, component states, and activities) for all possible control paths in the composition. We introduce the notions of minimal and maximal sharing ordering to approximate sharing in any particular run, and use these approximations to reason about functional dependencies and data attribute inheritance. The precision of the approximation can be improved by eliminating unused control paths based on process traces or behavior prediction.

Acknowledgments The authors were partially supported by Spanish MEC project 2008-05624/TIN *DOVES* and CM project P2009/TIC/1465 (*PROMETIDOS*).

References

1. Agrawal R, Imieliński T, Swami A (1993) Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIGMOD international conference on management of data, SIGMOD '93, pp 207–216. ACM, New York. doi:10.1145/170035.170072. http://doi.acm.org/10.1145/170035.170072
2. Armstrong WW (1974) Dependency structures of data base relationships. In: IFIP congress, pp 580–583 (1974)
3. Awad A, Puhlmann F (2008) Structural detection of deadlocks in business process models. In: Abramowicz W, Fensel D (eds) International conference on business information systems, LNBIP, vol 7. Springer, Berlin, pp 239–250
4. Beauche S, Poizat P (2008) Automated service composition with adaptive planning. In: Proceedings of the 6th international conference on service-oriented computing, ICSOC '08. Springer, Berlin. doi: 10.1007/978-3-540-89652-4-42. http://dx.doi.org/10.1007/978-3-540-89652-4-42
5. Bi HH, Zhao JL (2004) Applying propositional logic to workflow verification. Inf Technol Manage 5:293–318
6. Bueno F, García de la Banda M, Hermenegildo M (1999) Effectiveness of abstract interpretation in automatic parallelization: a case study in logic programming. ACM Toplas 21(2):189–238
7. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM symposium on principles of programming languages (POPL'77). ACM Press, New York
8. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of POPL'77. ACM Press, New York, pp 238–252
9. Davey BA, Priestley HA (2002) Introduction to lattices and order, 2nd edn. Cambridge University Press, Cambridge
10. Dezani-Ciancaglini M, De'Liguoro U (2010) Sessions and session types: an overview. In: Proceedings of the 6th international conference on Web services and formal methods, WS-FM'09. Springer, Berlin, pp 1–28. http://dl.acm.org/citation.cfm?id=1880906.1880907
11. Di Nitto E, Ghezzi C, Metzger A, Papazoglou M, Pohl K (2008) A journey to highly dynamic, self-adaptive service-based applications. Autom Softw Eng 15:313–341. doi:10.1007/s10515-008-0032-x. http://dx.doi.org/10.1007/s10515-008-0032-x
12. DiNitto E (2009) S-Cube deliverable CD-IA-2.2.2: collection of industrial best practices, scenarios and business cases. Tech. rep., S-Cube Consortium
13. Euzenat J, Shvaiko P (2007) Ontology matching. Springer, Heidelberg
14. Fdhila W, Yildiz U, Godart C (2009) A flexible approach for automatic process decentralization using dependency tables. In: ICWS, pp 847–855
15. Ganter B, Stumme G, Wille R (eds) (2005) Formal concept analysis, foundations and applications. Lecture notes in computer science, vol 3626. Springer, Berlin

16. Hermenegildo MV, Bueno F, Carro M, López P, Mera E, Morales J, Puebla G (2010) An overview of Ciao and its design philosophy. Tech. Rep. CLIP2/2010.0, Technical University of Madrid (UPM), School of Computer Science. Under consideration for publication in Theory and Practice of Logic Programming (TPLP)
17. Hintikka J (2004) Independence-friendly logic and axiomatic set theory. Ann Pure Appl Logic 126(1–3):313–333
18. Ivanović D, Carro M, Hermenegildo M (2010) Automatic fragment identification in workflows based on sharing analysis. In: Weske M, Yang J, Maglio P, Fantinato M (eds) Service-oriented computing–ICSOC 2010, no. 6470. LNCS. Springer, Berlin, pp 350–364
19. Ivanović D, Carro M, Hermenegildo M (2011) Automated attribute inference in complex service workflows based on sharing analysis. In: Proceedings of the 8th IEEE conference on services computing SCC 2011. IEEE Press, New York, pp 120–127
20. Jacobs D, Langen A (1989) North American conference on logic programming. MIT Press, Cambridge
21. Jordan D et al (2007) Web services business process execution language version 2.0. Tech. rep., IBM, Microsoft et al
22. Khalaf R (2007) Note on syntactic details of split BPEL-D business processes. Tech. Rep. 2007/2, IAAS, U. Stuttgart
23. Khalaf R, Leymann F (2012) Coordination for fragmented loops and scopes in a distributed business process. Inf Syst 37(6):593–610
24. Lagoon V, Stuckey P (2002) Precise pair-sharing analysis of logic programs. In: Principles and practice of declarative programming. ACM Press, New York, pp 99–108
25. Lloyd J (1987) Foundations of logic programming, second, extended edn. Springer, Berlin
26. Ma Z, Leymann F (2009) Bpel fragments for modularized reuse in modeling bpel processes. In: Mauri JL, Giner VC, Tomas R, Serra T, Dini O (eds) ICNS. IEEE Computer Society, New York, pp 63–68
27. Mancioppi M, Danylevych O, Karastoyanova D, Leymann F (2011) Towards classification criteria for process fragmentation techniques. In: BDP2011, colocated with BPM'11
28. Muthukumar K, Bueno F, de la Banda MG, Hermenegildo M (1999) Automatic compile-time parallelization of logic programs for restricted, goal-level, independent and-parallelism. J Logic Program 38(2):165–218
29. Muthukumar K, Hermenegildo M (1989) Determination of variable dependence information at compile-time through abstract interpretation. In: North American conference on logic programming. MIT Press, Cambridge, pp 166–189
30. Muthukumar K, Hermenegildo M (1991) Combined determination of sharing and freeness of program variables through abstract interpretation. In: ICLP'91. MIT Press, Cambridge, pp 49–63
31. Navas J, Bueno F, Hermenegildo M (2006) Efficient top-down set-sharing analysis using cliques. In: Eight international symposium on practical aspects of declarative languages, no. 2819. LNCS. Springer, Berlin, pp 183–198
32. Nielson F, Nielson HR, Hankin C (2005) Principles of program analysis, 2nd edn. Springer, Berlin
33. Object Management Group (2011) Business process model and notation (BPMN), Version 2.0. http://www.omg.org/spec/BPMN/2.0/PDF
34. Papazoglou MP, Pohl K, Parkin M, Metzger A (eds) (2010) Service research challenges and solutions for the future internet–S-Cube–towards engineering, managing and adapting service-based systems. Lecture notes in computer science, vol 6500. Springer, Berlin
35. Ponge J, Benatallah B, Casati F, Toumani F (2007) Fine-grained compatibility and replaceability analysis of timed web service protocols. In: Parent C, Schewe KD, Storey VC, Thalheim B (eds) ER, Lecture notes in computer science, vol 4801. Springer, Berlin, pp 599–614
36. Shvaiko P (2005) A classification of schema-based matching approaches. J Data Seman 4:146–171
37. Tan W, Fan Y (2007) Dynamic workflow model fragmentation for distributed execution. Comput Ind 58(5):381–391. http://dx.doi.org/10.1016/j.compind.2006.07.004
38. Ullman JD (1988) Database and knowledge-base systems, vol 1. Computer Science Press, Maryland
39. Väänänen J (2007) Dependence logic: a new approach to independence friendly logic. Cambridge University Press, Cambridge (London mathematical society student texts)
40. Valtchev P, Missaoui R, Godin R (2004) Formal concept analysis for knowledge discovery and data mining: The new challenges. In: ICFCA, pp 352–371
41. World Wide Web Consortium (2005) Web services choreography description language version 1.0. http://www.w3.org/TR/ws-cdl-10/

42. Yildiz U, Godart C (2007) Information flow control with decentralized service compositions. In: ICWS, pp 9–17
43. Zaplata S, Kottke K, Meiners M, Lamersdorf W (2009) Towards runtime migration of ws-bpel processes. In: ICSOC/ServiceWave Workshops, pp 477–487
44. Zemni MA, Benbernou S, Carro M (2010) A soft constraint-based approach to qos-aware service selection. In: Weske M, Yang J, Maglio P, Fantinato M (eds) Service-oriented computing–ICSOC 2010, no. 6470. LNCS. Springer, Berlin, pp 596–602