Automatic Fragment Identification in Workflows Based on Sharing Analysis[⋆]

Dragan Ivanović,[1] Manuel Carro,[1] and Manuel Hermenegildo[1,2]

[1] School of Computer Science, T. University of Madrid (UPM)
(idragan@clip.dia.fi.upm.es, {mcarro, herme}@fi.upm.es)
[2] IMDEA Software Institute, Spain

Abstract. In Service-Oriented Computing (SOC), fragmentation and merging of workflows are motivated by a number of concerns, among which we can cite design issues, performance, and privacy. Fragmentation emphasizes the application of design and runtime methods for clustering workflow activities into fragments and for checking the correctness of such fragment identification w.r.t. to some predefined policy. We present a fragment identification approach based on sharing analysis and we show how it can be applied to abstract workflow representations that may include descriptions of data operations, logical link dependencies based on logical formulas, and complex control flow constructs, such as loops and branches. Activities are assigned to fragments (to infer how these fragments are made up or to check their well-formedness) by interpreting the sharing information obtained from the analysis according to a set of predefined policy constraints.

1 Introduction

Service-Oriented Computing (SOC) enables interoperability of components with low coupling which expose themselves using standardized interface definitions. In that context, service compositions are mechanisms for expressing in an executable form business processes (i.e., wokflows) that include other services, and are exposed as services themselves. Compositions can be described using one of the several available notations and languages [Obj09,Jea07,Wor08,ZBDtH06,vdAP06,vdAtH05] which allow process modelers and designers to view a composition from the standpoint of business logic and processing requirements.

These service compositions are coarse-grained components that normally implement higher-level business logic, and allow streamlining and control over mission-critical business processes inside an organization and across organization boundaries. However, the centralized manner in which these processes are designed and engineered does not necessarily build in some properties which may be required in their run-time environment. In many cases defining subsets of activities (i.e., fragments inside the

[⋆] The research leading to these results has received funding from the European Community's Seventh Framework Programme under the Network of Excellence S-Cube - Grant Agreement n° 215483. Manuel Carro and Manuel Hermenegildo were also partially supported by Spanish MEC project 2008-05624/TIN *DOVES* and CM project P2009/TIC/1465 (*PROMETIDOS*). Manuel Hermegildo was also partially supported by FET IST-231620 *HATS*.

workflow) according to some policy can be beneficial in order to increase reusability (by locating meaningful sets of activities), make it possible to farm, delegate, or subcontract part of the activities (if, e.g., resources and privacy of necessary data make it possible or even advisable), optimize network traffic (by finding out bottlenecks and adequately allocating activities to hosts), ensure privacy (by making sure that computing agents are not granted access to data whose privacy level is higher than their security clearance level), and others. To this end, various fragmentation approaches have been proposed [WRRM08,BMM06,TF07]. In the same line, mechanisms have been defined for refactoring existing monolithic processes into process fragments according to a given fragment definition while respecting the behavior and correctness aspects of the original process [Kha07,KL06].

This paper addresses the automatic identification of fragments given an input workflow, expressed in a rich notation. The kind of fragment identification policies we tackle is based on data accessibility / sharing, rather than, for example, mere structural properties. The latter simply try to deduce fragments or properties by matching parts of workflows with predefined patterns, as [AP08] does for deadlocks. In contrast, the design-time analysis we propose takes into account implicitly and automatically different workflow structures.

At the technical level, our proposal is based on the notion of *sharing* between activities. This is done by considering how these activities handle resources (such as data) that represent the state of an executing composition (i.e., process variables), external participants (such as partner services and external participants), resource identifiers, and mutual dependencies. In order to do so, we need to ensure that the workflow is deadlock free in order to infer a partial order between activities. This is used to construct a Horn clause program [Llo87] which captures the relevant information and which is then subject to sharing and groundness analysis [MH92,JL92,MS93,MH91] to detect the sharing patterns among its variables. The way in which this program is engineered makes it possible to infer, after analysis, which activities must be in the same fragment and which activities need / should not be in the same fragment. Even more interestingly, it can automatically uncover a *lattice* relating fragments of increasing size and complexity to each other and to simpler fragments, while respecting the fragment policy initially set.

2 Structuring Fragments with Lattices

We assume that any fragment definition is ultimately driven by a set of policies which determine whether two activities should / could belong to the same fragment. Fragment identification determines how to group activities so that the fragment policies are respected, while fragment checking ensures that predefined fragments abide by the policies. For example, data with some security level cannot be fed to activities with a lower security clearance level. This can be used to classify activities at different clearance levels according to the data they receive, and also to check that a previous classification is in accordance to the security level of the data received. This may have changed due, for example, to updates in an upstream part of the workflow.

Some approaches to process partitioning [FYG09,YG07] assume that the activities inside an abstractly described process can be *a priori* assigned to different organizational domains depending on the external service that is invoked from the workflow. These are concerned with ensuring that each fragment, corresponding to a projection of the workflow onto organizational domains, is correctly wired with other fragments to preserve both the correct behavior of the original workflow, and to satisfy externally specified constraints that describe legal conversations and data flows between services in the different domains. Rules for correctly separating fragments have also been devised for some concrete executable workflow languages, such as BPEL [Kha07]. In our approach we want to derive the fragmentation constraints from the workflow and the characterization of its inputs, without relying on other external policy sources. Also, we take a more flexible view of workflow activities, including different "local" data processing and structured constructs such as loops and nested sub-workflows, which, in principle, are not *a priori* organization domain specific.

Many fragmentation approaches assume *flat*, non-structured fragments: activities are just split into non-overlapping sets. However, for the sake of generality (which will be useful later), we assume that fragments can have a richer structure: a lattice, which means that in principle it is possible[3] to join activities belonging to two different fragments in a *super-fragment* which still respects the basic policies. For example, two activities with separate security clearance levels (because one deals with business profit data and the other one with medical problems in a company) can be put together in a new fragment whose clearance level has to be, at least, equal or higher than any of these two. It turns out that it may be possible that in order to have a consistent workflow, an "artificial" security level needs to be created if some activity is discovered to need both types of data. Of course, a lattice can also represent simpler fragmentation schemes, such as "flat" schemes (no order is induced by the properties of the fragments) or "linear" schemes (the properties induce a complete order).

Therefore we will assume that the fragmentation policies can be described using a complete lattice $\langle L, \sqsubseteq, \top, \bot, \sqcup, \sqcap \rangle$, where $\sqsubseteq$ is a partial order relation over the non-empty set L of elements which *tag* the fragments, $\top$ and $\bot$ are the top and bottom elements in the lattice, and $\sqcup$ and $\sqcap$ are the *least upper bound* (LUB) and the *greatest lower bound* (GLB) operations, respectively. In the examples we will deal with in this paper we will only use the LUB operation.

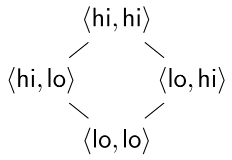


Fig. 1. Data confidentiality: two levels × two domains.

As an example in the domain of data confidentiality, the simplest non-trivial case can be modeled using two levels $L_1 = \{\mathsf{lo}, \mathsf{hi}\}$ such that $\mathsf{lo} \sqsubset \mathsf{hi}$, and activities belong to these two classes depending on the level of data they operate on. In a more complex setting, we can have more degrees of confidentiality $L_2 = \{\mathsf{lo}, \mathsf{med}, \mathsf{hi}\}$, which are still completely ordered ($\mathsf{lo} \sqsubset \mathsf{med} \sqsubset \mathsf{hi}$) or, more interestingly, data belonging to different domains / departments in a company which each have different security levels, e.g., $L_3 = \{\mathsf{lo}, \mathsf{hi}\}^n$, where $n > 1$ is the number of domains. In this case $\sqsubseteq$ is to be defined to represent the company policy: maybe some department "dominates" the

[3] But not *necessarily* permissible: it depends on the particular definition of the lattice.

security in the company and therefore fragments marked with $\langle \mathsf{hi}, \_ \rangle$ (where $\_$ stands for "any value") can access any data, or maybe there is no such a dominance and an activity marked with clearance $\langle \mathsf{hi}, \mathsf{lo} \rangle$ cannot read data marked with security level $\langle \mathsf{lo}, \mathsf{hi} \rangle$, and only activities with clearance level $\langle \mathsf{hi}, \mathsf{hi} \rangle$ can access all data in the organization. The corresponding lattice appears in Fig. 1.

The lattice formalism provides us with the necessary tools for identification of fragments. When a fragment is marked by some element $c \in L$, we can decide whether some activity a can be included in the fragment depending on whether its policy level $\hat{a}$ respects $\hat{a} \sqsubseteq c$ or not. Note that the direction of the partial order in the generic lattice is arbitrary and may need to be adjusted to match the needs of a real situation.

As anticipated earlier, in our approach policies which apply to data are reflected on the results of operations on data and on the activities that perform those operations. Thus, we assign policy levels to activities based on the policy levels of their input data flow. We on purpose abstract from the notion of program variables / storage locations and focus instead on pieces of the information flow, which are more important in distributed workflow enactment scenarios.

3 Derivation of Control and Data Dependencies

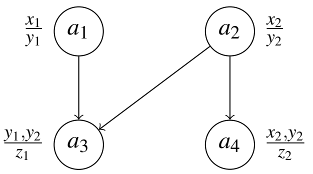
As a first step for the automatic fragmentation analysis proper, we need to find out a feasible order of activities which is coherent with their dependencies and which allows the workflow to finish successfully. How to do this obviously depends on the palette of allowed relationships between activities, with respect to which we opted for a notable freedom (Section 3.1). 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 loops that may result in deadlocks. While there is ample work in deadlock detection [BlZ04,AP08], 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.

3.1 Workflow Representation

We first state which components we highlight in a workflow.

Definition 1 (Workflow). *A workflow W is a tuple $\langle A, C, D \rangle$, where A is a finite set of activities $\{a_1, a_2, \ldots, a_n\}$, $n \geq 0$, C is a set of control dependencies given as pre- and post-conditions for individual activities (see later), and D is a finite set of data dependencies expressed as pairs $\langle a_i, A^{(d)} \rangle \in D$ where $a_i \in A$ is an activity that produces (writes) data item d, and $A^{(d)} \subseteq A$ ($a_i \notin A^{(d)}$) is a set of activities that consume (read) data item d.*

This abstract workflow definition corresponds in general with the most frequently used models for distributed workflow enactment. However, the flexibility of the encoding we will use for the fragmentation analysis allows for two significant extensions compared to other workflow models.



$$A = \{a_1, a_2, a_3, a_4\}$$

$$C = \{\textsf{pre–}a_3 \equiv \textsf{done–}a_1 \land \textsf{done–}a_2\,, \quad \textsf{pre–}a_4 \equiv \textsf{done–}a_2\}$$

$$D = \{y_1, y_2\} \quad y_1 = \langle a_1, \{a_3\}\rangle,\ y_2 = \langle a_2, \{a_3, a_4\}\rangle$$

Fig. 2. An example workflow. Arrows indicate control dependencies.

- 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.

Definition 2 (Activity preconditions). *A precondition of an activity $a_i \in A$ is a propositional formula which can use the full set of logical connectives ($\land$, $\lor$, $\neg$, $\rightarrow$, and $\leftrightarrow$), the values 1 (for true) and 0 (for false), and the propositional symbols $\textsf{done–}a_j$ and $\textsf{succ–}a_j$, $a_j \in A$, where $\textsf{done–}a_j$ holds if a_j has completed and $\textsf{succ–}a_j$ stands for a successful outcome of a_j when $\textsf{done–}a_j$ holds (i.e., $\textsf{succ–}a_j$ is only meaningful when $\textsf{done–}a_j$ is true).*

Definition 3 (Dependencies). *A set of control dependencies C that associates each $a_i \in A$ with its precondition, which we will term $\textsf{pre–}a_i$. We write C as a set of identities of the form $\textsf{pre–}a_i \equiv \langle\text{formula}\rangle$. Trivial cases of the form $\textsf{pre–}a_i \equiv 1$ are omitted.*

Commonly, the preconditions use "done" symbols, whereas "succ" symbols can be added to reflect the business logic in the structure of the workflow, and 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).

Figure 2 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 the 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 $\textsf{pre–}a_j \equiv \textsf{done–}a_i$. An AND-join after a_i and a_j into a_k becomes $\textsf{pre–}a_k \equiv \textsf{done–}a_i \land \textsf{done–}a_j$. An (X)OR-join of a_i and a_j into a_k is encoded as $\textsf{pre–}a_k \equiv \textsf{done–}a_i \lor \textsf{done–}a_j$. And an XOR split of a_i into

a_j and a_k (based on the business outcome of a_i) becomes $\mathsf{pre}\text{–}a_j \equiv \mathsf{done}\text{–}a_i \wedge \mathsf{succ}\text{–}a_i$, $\mathsf{pre}\text{–}a_k \equiv \mathsf{done}\text{–}a_i \wedge \neg\mathsf{succ}\text{–}a_i$. 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.

3.2 Validity of Control Dependencies

The relative freedom given for specifying logic formulae for 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.

Whether a deadlock can happen or not depends on both topology and the logic of control dependencies. Figure 3 shows a simple example where the dependency arrows are drawn from a_i and a_j whenever $\mathsf{pre}\text{–}a_j$ depends on a_i finishing. That topological information is not sufficient for inferring deadlock freeness, unless there are no loops in the graph. If the connective marked with $\bullet$ in $\mathsf{pre}\text{–}a_3$ is $\vee$, there is no deadlock: indeed, there is a possible execution sequence, $a_1 - a_2 - a_3 - a_4$. If, however, $\bullet$ denotes $\wedge$, there is a deadlock between a_3 and a_4.

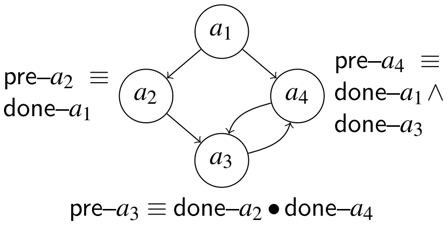

a_1
a_2
a_4
a_3
$\mathsf{pre}\text{–}a_2 \equiv \mathsf{done}\text{–}a_1$
$\mathsf{pre}\text{–}a_4 \equiv \mathsf{done}\text{–}a_1 \wedge \mathsf{done}\text{–}a_3$
$\mathsf{pre}\text{–}a_3 \equiv \mathsf{done}\text{–}a_2 \bullet \mathsf{done}\text{–}a_4$

Fig. 3. An example of deadlock dependency on logic formula: $\bullet$ can be either $\wedge$ or $\vee$.

Therefore, in general, checking for deadlock-freeness needs looking at the formulas. We present one such approach that relies on simple proofs of propositional formulas. We start by forming a logical theory Γ from the workflow by including all preconditions from C and adding axioms of the form $\mathsf{done}\text{–}a_i \rightarrow \mathsf{pre}\text{–}a_i$ 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 4. *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:*

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

For every data dependency $\langle a_j, A^{(d)} \rangle \in D$, and for each $a_i \in A^{(d)}$, we wish to ensure that a_i cannot start unless a_j has completed, since otherwise the data item d would not be ready. Expressed with a logic formula, that condition is $\mathsf{pre}\text{–}a_i \rightarrow \mathsf{done}\text{–}a_j$, as in the definition of Δ, above. Therefore, we require $\delta_{ij} = 1$.

The computation of Δ involves proving propositional formulas, which is best achieved using some form of SAT solvers, which 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 is deadlock-free if and only if* $\forall a_i \in A,\ \delta_{ii} = 0$.

Proposition 2 (Partial ordering). *In a deadlock-free workflow, 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 Translation and Analysis

To apply sharing analysis, we first transform the workflow into an appropriate logic program. 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.

4.1 Workflows as Horn Clauses

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 particular order does not impact 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:

$$w(V) \leftarrow T(a_1), T(a_2), \ldots, T(a_N)$$

where V is the set of all logic variables used in the clause, and $T(a_i)$ stands for the translation of the activity a_i.

As mentioned before, the logic program aims at representing data flow and dependencies in a sharing analysis-amenable way, which we will detail later. Logic variables are used to represent input data, data dependencies, output data, and the data sets read by individual activities. For each activity $a_i \in A$, we designate a set R_i of logic variables that represent data items read by a_i, and a set W_i of logic variables that stand for data items produced by a_i. We also designate a special variable $\hat{a}_i$ that represents the total inflow of data into a_i. The task of the translation is to connect R_i and $W_i \cup \{\hat{a}_i\}$ correctly.

A primer on logic programs. Data items in logic programs are called *terms*. A term t can be either a *variable*, an *atom* (i.e., a simple non-variable data item, such as the name of a object or a number), or a compound term of the form $f(t_1, t_2, \ldots, t_n)$, $(n \geq 0)$, where f is the *functor*, and $t_1, t_2, \ldots, t_n$ are the argument terms. $f(a, b)$ is not a function call, — it can be seen instead as a record named f with two fields containing items a and b. Terms that are not variables and do not contain variables are said to be *ground*. Procedure calls in a logic program are called *goals* and have the syntactic form $p(t_1, t_2, \ldots, t_n)$ $(n \geq 0)$, where p is a *predicate name* of *arity* n (usually denoted as p/n), and the terms $t_1, t_2, \ldots, t_n$ are its arguments. In goals, the infix predicate $=/2$ is used to denote unification of terms (described later), as in $t_1 = t_2$.

In the execution of a logic program, variables serve as placeholders for terms that satisfy the logical conditions encoded in the program. Each successful execution step may map a previously free variable to a term. The set of such mappings at a program point is called a *substitution*. Thus, a substitution θ is a finite set of mappings of the form $x \mapsto t$ where x is a variable and t is a term. Each variable can appear only on one side of the mappings in a substitution. At any point during execution, the actual value of the variables in the program text is obtained by applying the current substitution to these (syntactical) variables. These applications may produce terms that are possibly more concrete (have fewer variables) than the ones wich appear in the program text. If $\theta = \{x \mapsto 1, y \mapsto g(z)\}$, and $t = f(x,y)$, then the application $t\theta$ gives $f(1, g(z))$.

Substitutions at a subsequent program points are composed together to produce an aggregated substitution for the next program point (or the final substitution on exit). E.g., for the previous θ and $\theta' = \{z \mapsto a+1\}$ (with a being an atom, not a variable), we have $\theta\theta' = \{x \mapsto 1, y \mapsto g(a+1)\}$.

Substitutions are generated by *unifications*. Unifying t_1 and t_2 gives a substitution θ which ensures that $t_1\theta$ and $t_2\theta$ are identical terms by introducing the least amount of new information. Unifying x and $f(y)$ gives $\theta = \{x \mapsto f(y)\}$; unifying $f(x, a+1)$ and $f(1, z+y)$ gives $\theta = \{x \mapsto 1, z \mapsto a, y \mapsto 1\}$; and the attempt to unify $f(x)$ and $g(y)$ fails, because the functors are different. When a goal calls a predicate, the actual and the formal arguments are unified, which may generate further mappings to be added to the accumulated ones.

Take, for instance, the Horn clause translation of the example workflow on Fig. 4. Variables in the listing are written in uppercase, and comments that start with "%" indicate workflow activity. The comma-separated goals in the body (after ":-") are executed one after another. If the initial substitution is $\theta_0 = \{x_1 \mapsto u_1, x_2 \mapsto u_2\}$, then the first unification produces $\theta_1 = \{\hat{a}_1 \mapsto f_1(x_1)\}$, so the aggregate substitution before the next goal is $\theta_0\theta_1 = \{x_1 \mapsto u_1, x_2 \mapsto u_2, \hat{a}_1 \mapsto f_1(u_1)\}$. The second unification in the body adds $\theta_2 = \{y_1 \mapsto f_2(x_1)\}$, and the result is $\theta_0\theta_1\theta_2 = \{x_1 \mapsto u_1, x_2 \mapsto u_2, \hat{a}_1 \mapsto f_1(u_1), y_1 \mapsto f_{1y_1}(u_1)\}$.

The process continues until the final substitution is reached: $\theta_0\theta_1 \cdots \theta_8 = \{x_1 \mapsto u_1,\ x_2 \mapsto u_2,\ \hat{a}_1 \mapsto f_1(u_1),\ y_1 \mapsto f_2(u_1),\ \hat{a}_2 \mapsto f_3(u_2),\ y_2 \mapsto f_4(u_2),\ \hat{a}_3 \mapsto f_5(f_2(u_1), f_4(u_2)),\ z_1 \mapsto f_6(f_2(u_1), f_4(u_2)),\ \hat{a}_4 \mapsto f_7(u_2, f_4(u_2)),\ z_2 \mapsto f_8(u_2, f_4(u_2))\}$.

Note that the program point substitutions are expressed based on u_1 and u_2, the terms to which x_1 and x_2 were initialy bound to. In this case it is interesting that some variables (for example, y_1 and z_1) are bound to terms that contain a common term (variable u_1, in this case), and so we say that y_1 and z_1 *share*.

Definition 5 (Sharing). *Given a runtime substitution θ, two syntactical variables x and y are said to share if the terms $x\theta$ and $y\theta$ contain some common variable z.*

In the preceding example $\hat{a}_1$ shares with x_1 via u_1; y_2 shares with x_2 via u_2; $\hat{a}_3$ shares both with x_1 via u_1, and with x_2 via u_2; etc. The basis for all sharing are u_1 and u_2, yet they do not appear in the program, but in the initial substitution. The key to the use of sharing analysis for the definition of fragments is, precisely, the introduction of such "hidden" variables, which act as "links" between workflow variables that represent data flows and activities.

Assignments, expression evaluations, and service invocations are translated into unifications that enforce sharing between the input and the output data items of the activity. Complex activities are translated into separate predicates, and an example of such translation (for a *repeat-until* loop construct) is given in the example in Section 5.

For each output data item $x \in W_i$ of the translated activity a_i, we introduce a unification between x and a compound term that involves the variables that are used in producing it, and which form a subset of R_i. If we do not know exactly which variables from R_i are necessary to produce x, we can safely use them all, at the cost of over-approximating sharing. The choice of functor name in the compound term is not significant. The same applies to the activity-level variable $\hat{a}_i$, which is unified with a compound term containing all variables from R_i, to model the dependency of a_i on all information that it uses as input.

```
w(X1,X2,A1,Y1,A2,Y2,A3,Z1,A4,Z2):-
  A1=f1(X1), % a1
  Y1=f1Y1(X1),
  A2=f2(X2), % a2
  Y2=f2Y2(X2),
  A3=f3(Y1,Y2), % a3
  Z1=f3Z1(Y1,Y2),
  A4=f4(X2,Y2), % a4
  Z2=f4Z2(X2,Y2).
```

Fig. 4. Logic program encoding of the workflow from Fig. 2

4.2 Sharing Analysis

The sharing analysis we use here is an instance of abstract interpretation [CC77], 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 the 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 abstract interpretation-based sharing, freeness, and groundness analysis for logic programs. Instead of analyzing the infinite universe of possible substitutions during execution of a logic program, sharing analysis is concerned just with the question of which variables may possibly share in a given substitution. This analysis is helped by freeness and groundness analysis, because the former tells which variables are not substituted with a compound term, and the latter helps exclude ground variables from sharing. Some logic program analysis tools, like CiaoPP [HBC^{+}10], have been devel-

```
function RecoverSubstVars($V, \Theta$)
    $n \leftarrow |\Theta|$; $U \leftarrow \{u_1, u_2, ..., u_n\}$          ▷ $n = |\Theta|$ fresh variables in $U$
    $S : V \rightarrow \wp(U)$; $S \leftarrow \mathrm{const}(\emptyset)$          ▷ the initial value for the result
    for $x \in V$, $i \in \{1..n\}$ do          ▷ for each variable and subst. setting
        if $x \in \Theta[i]$ then          ▷ if the variable appears in the setting
            $S \leftarrow S[x \mapsto S(x) \cup \{u_i\}]$          ▷ add $u_i$ to its resulting set
        end if
    end for
return $U$, $S$
end function
```

Fig. 5. The minimal substitution variable set recovery algorithm.

```
init1(f(W1,W2),f(W2)).
init2(f(W1),f(W2)).
init3(f(W2),f).

caseN(X1,X2,A1,Y1,A2,Y2,A3,Z1,A4,Z2):-
  initN(X1,X2),
  w(X1,X2,A1,Y1,A2,Y2,A3,Z1,A4,Z2).
```

Case	Sharing results
1	[[X1,A1,Y1,A3,Z1], [X1,X2,A1,Y1,A2,Y2,A3,Z1,A4,Z2]]
2	[[X1,A1,Y1,A3,Z1], [X2,A2,Y2,A3,Z1,A4,Z2]]
3	[[X1,A1,Y1,A3,Z1]]

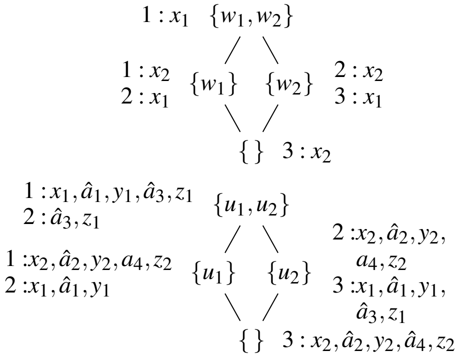

$1 : x_1$ $\{w_1, w_2\}$
$1 : x_2$
$2 : x_1$
$\{w_1\}$
$\{w_2\}$
$2 : x_2$
$3 : x_1$
$\{\}$ $3 : x_2$
$1 : x_1, \hat{a}_1, y_1, \hat{a}_3, z_1$
$2 : \hat{a}_3, z_1$
$\{u_1, u_2\}$
$1 : x_2, \hat{a}_2, y_2, a_4, z_2$
$2 : x_1, \hat{a}_1, y_1$
$\{u_1\}$
$\{u_2\}$
$2 : x_2, \hat{a}_2, y_2, a_4, z_2$
$3 : x_1, \hat{a}_1, y_1, \hat{a}_3, z_1$
$\{\}$ $3 : x_2, \hat{a}_2, y_2, \hat{a}_4, z_2$

Fig. 6. The initial settings and the sharing results.

oped which give users the possibility of running different analysis algorithms on input programs. We build on one of the sharing analyses available in CiaoPP.

In the sharing and freeness domain, an abstract substitution Θ is a set of all possible sharing sets. Each sharing set is a subset of the variables in the program. The meaning of this set is that those variables may be bound at run-time to terms which share some common variable. E.g., if $\theta = \{x \mapsto f(u), y \mapsto g(u,v), z \mapsto w\}$, then the corresponding abstract substitution (projected over x, y, and z) is $\Theta = \{\{x,y\},\{y\},\{z\}\}$. Note that if u is further substituted (with some term t), x and y will be jointly affected; if v is further substituted, only y will be affected, and if w is further substituted only z will be affected.

Although an abstract substitution represents an infinite family of concrete substitutions (e.g., $\theta' = \{x \mapsto v, y \mapsto h(w,v), z \mapsto k(u)\}$ in the previous example), it is always possible to construct a minimal concrete substitution exhibiting that sharing, by taking as many auxiliary variables as there are sharing sets, and constructing terms over them. Furthermore, if one is only interested in the sets of shared auxiliary variables, not the exact shape of the substituted terms, the simple algorithm from Figure 5 suffices. Note that only a ground variable $x \in V$ can have $S(x) = \emptyset$.

4.3 Deriving Fragment Identification Information from Sharing

To derive fragment identification information from the sharing analysis results, the logic program workflow representation has to be analyzed in conjunction with some initial conditions that specify the policy levels of the input data. In our example from Figures 2 and 4, this applies to the input data items x_1 and x_2. We will assume that the policy lattice is isomorphic to or a sublattice of some lattice L induced by the powerset of a non-empty set $W = \{w_1, ..., w_n\}$ with the set inclusion relation.

The initial conditions are then represented by means of sharing between the input data variables and the corresponding subsets of W. Several initial conditions (labeled with 1, 2, and 3) are shown in Figure 6 (left). Each `case`N (where N is in this case 1, 2, or 3) predicate starts with all variables independent (i.e., no sharing), and calls `init`N to set up the initial sharing pattern for x_1 and x_2 in terms of the hidden variables w_1 and w_2. How these different patterns place variables x_1 and x_2 in the policy lattice is shown in the figure at the right, top, by displaying the case number just before each variable. Below this lattice are the sharing results (as Prolog lists) obtained from the shfr analysis.[4] Right on the same figure are the projections of the initial sharing settings for x_1 and x_2 on the original policy lattice $L = \wp(W)$, $W = \{w_1, w_2\}$, and projections of the sharing results on the lattice $L' = \wp(U)$, $U = \{u_1, u_2\}$ derived using the algorithm in Figure 5.

Let us, for instance, interpret case 2. If we use variable names to mean their policy levels (with primes in L'), we see that initially $x_1 = \{w_1\}$ and $x_2 = \{w_2\}$ are disjoint, i.e., incomparable w.r.t. $\sqsubseteq$. Activity a_1 uses x_1 as the input, and produces y_1 from x_1; hence, $\hat{a}'_1 = y'_1 = x'_1 = \{u_1\}$. Analogously, $\hat{a}'_2 = y'_2 = x'_2 = \{u_2\}$. For a_3, both y_1 and y_2 are used to produce z_1. Hence, $\hat{a}'_3 = z'_1 = y'_1 \sqcup y'_2 = \{u_1, u_2\}$. Finally, a_4 uses x_2 and y_2 to produce z_2, and thus $\hat{a}'_4 = z'_2 = x'_2 \sqcup y'_2 = \{u_2\}$. Therefore, a_2 and a_4 are at the same clearance level, and a_3 is at a different (but non-comparable) level.

The most important feature of the derived lattice L' is that if for $x, y \in L$ we have $x \sqsubseteq y$, then for their respective images x', y' derived in L', we also have $x' \sqsubseteq y'$. This feature follows from the structure of the translation for simple activities (linear unifications), the fact that variables in W are hidden inside the initialization predicate (and thus remain mutually independent and free), and the semantics of unification in the shfr domain. Therefore, the two typical fragmentation inference tasks are:

- The policy level $\sqcup B$ of a subset of activities $B \subseteq A$ in L' is $\sqcup\{\hat{a}'_i \mid a_i \in B\}$.
- To check constraint compliance for $B \subseteq A$ in L, one needs to represent c as an input data item in the workflow, and then check $\sqcup B \sqsubseteq c'$ in L'.

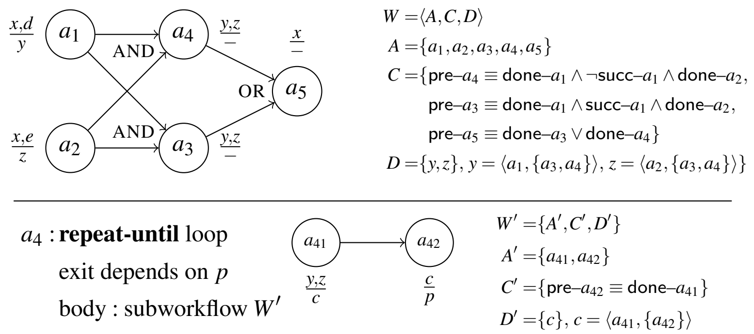
Note that we have not defined at any moment the exact shape of the finally inferred lattice: we merely stated the relationship between two input flow streams (for three different possibilities, in this case) and the analysis algorithm built, for each of these cases, the abstract substitution which places every relevant program variable in its point.

The next section will present in more detail an example involving privacy and two types of data.

5 An Example of Application to Data Privacy

Figure 7 shows a simplified workflow for drug prescription in a health care organization. The input data are the identity of the patient (x), authorization to access the patient's medical history (d) and the authorization to access the patient's medication record (e).

[4] These results were obtained in 3.712ms (total) on a Intel Core Duo 2GHz machine with 2GB of RAM running CiaoPP 1.12 and Mac OS X 10.6.3.



$W = \langle A, C, D\rangle$

$A = \{a_1, a_2, a_3, a_4, a_5\}$

$C = \{\mathsf{pre}\text{–}a_4 \equiv \mathsf{done}\text{–}a_1 \wedge \neg\mathsf{succ}\text{–}a_1 \wedge \mathsf{done}\text{–}a_2,$
$\mathsf{pre}\text{–}a_3 \equiv \mathsf{done}\text{–}a_1 \wedge \mathsf{succ}\text{–}a_1 \wedge \mathsf{done}\text{–}a_2,$
$\mathsf{pre}\text{–}a_5 \equiv \mathsf{done}\text{–}a_3 \vee \mathsf{done}\text{–}a_4\}$

$D = \{y, z\},\ y = \langle a_1, \{a_3, a_4\}\rangle,\ z = \langle a_2, \{a_3, a_4\}\rangle\}$

a_4 : **repeat-until** loop
exit depends on p
body : subworkflow W'

$W' = \{A', C', D'\}$

$A' = \{a_{41}, a_{42}\}$

$C' = \{\mathsf{pre}\text{–}a_{42} \equiv \mathsf{done}\text{–}a_{41}\}$

$D' = \{c\},\ c = \langle a_{41}, \{a_{42}\}\rangle$

Fig. 7. A simplified drug prescription workflow.

Based on the patient id and the corresponding authorization, activity a_1 retrieves the patient's medical history (y), and signals success ($\mathsf{succ}\text{–}a_1$) iff the patient's health has been stable. Simultaneously, activity a_2 uses the patient's id and the corresponding authorization to retrieve the patient's medication record (z). Depending on the outcome of a_1, either a_3 or a_4 is executed. Activity a_3 continues the last medical prescription, based on the medical history and the medication record. Activity a_4, on the other hand, tries to select a new medication. Activity a_{41} runs some additional tests based on the medical history and the medication record to produce criteria c for the new medicine. Medication record z is used just for cross-checking, and does not affect the result c. Activity a_{42} searches the medication databases for a medicine matching c, which is prescribed if found; the search may fail if the criterion c is too vague. The search result p is used as the exit condition from the loop. Finally, activity a_5 records that the patient has been processed.

The fragmentation policies used in this example are based on the assumption that we want to distribute execution of this centralized workflow so that the fragments can be executed inside domains that need not have access to all patient's information. For instance, the health care organization may delegate some of the activities to an outside (or partner) service that keeps and integrates medical histories and runs medical checks, but should not (unless expressly authorized) be able to look into the patient's medication record, to minimize influence that the types and costs of earlier prescribed may have on the choice of medical checks. Other activities can be delegated to partners that handle only medication records. Finally, the organization owning the workflow wants to reserve to itself the right to access both the medical history and the medication record at the same time.

To formalize the policies, we introduce a set of two data privacy tags $W = \{w_1, w_2\}$, and the lattice of policies $L = \wp(W)$. The presence of tag w_1 indicates authorization to access to medical history related data and the presence of tag w_2 indicates authorization to access to medication record related data. Using variable names to indicate policy levels, we start with $d = \{w_1\}$, $e = \{w_2\}$, and $x = \{\,\}$; the latter implies that consulting the identity of a patient does not require specific clearances.

It can be easily demonstrated that this workflow does not have deadlocks, and that one compatible ordering of activities is $\langle a_1, a_2, a_3, a_4, a_5\rangle$. The translation of the workflow into a logic program is shown in Figure 8. Note the initial sharing setting in the

```
analysis(X,D,E,T,A1,Y,A2,Z,A3,A4,A41,C,A42,P,A5):-
        init(X,D,E,T),
        w(X,D,E,T,A1,Y,A2,Z,A3,A4,A41,C,A42,P,A5).

init(f,f(W1),f(W2),f(W1,W2)).

w(X,D,E,T,A1,Y,A2,Z,A3,A4,A41,C,A42,P,A5):-
        A1=f1(X,D), % a_1
        Y=f1_Y(X,D),
        A2=f2(X,E), % a_2
        Z=f2_Z(X,E),
        A3=f3(Y,Z), % a_3
        a_4(Y,Z,A4,A41,C,A42,P), % a_4
        A5=f5(X). % a_5
```

```
a_4(Y,Z,A4,A41,C,A42,P):-
        w2(Y,Z,A41,C2,A42,P2),
        A4=f(P2),
        a_4x(Y,Z,C2,P2,C,P,A4,A41,A42).

a_4x(_,_,C,P,C,P,_,_,_).
a_4x(X,Z,_,_,C,P,A4,A41,A42):-
        a_4(X,Z,A4,A41,C,A42,P).

w2(Y,Z,A41,C,A42,P):-
        A41=f41(Y,Z), % a_41
        C=f41_C(Y),
        A42=f42(C), % a_42
        P=f42_P(C).
```

Fig. 8. Logic program encoding for the medication prescription workflow.

predicate init before calling the workflow translation in predicate w. It corresponds to the lattice L from Figure 9. We have introduced the element $\top$ (variable T in the listing) that corresponds to the top element of the original lattice.

Also, note how the *repeat-until* has been translated into two predicates, a_4 and a_4x. Predicate a_4 invokes the sub-workflow w2, to produce data items c and p from the single iteration (logic variables C2 and P2). It then enforces sharing between $\hat{a}_4$ and the latest p on which the exit condition depends, and invokes a_4x. The latter predicate treats two cases: the first clause models exit from the loop, by passing the values of c and p from the last iteration as the final ones. The second clause of a_4x models repetition of the loop. The sub-workflow comprising the body of the loop is translated to w2 using the same rules as the main workflow.

The abstract substitution that is the result of the sharing analysis, as returned by the CiaoPP analyzer, is shown on Figure 11.[5] By interpreting these results using the algorithm for recovery of the minimal sets of variable substitutions from Figure 5, we obtain the resulting lattice L', shown on Figure 10. Variables X and A5 are ground and do not appear in the result. Note that the relative ordering of the input data items (x, d, e, and $\top$) has been preserved in L'. Also, the assignment of policy levels to activities shows that the activities a_3 and a_{41} are critical because they need to have both the d and e clearance levels—i.e., the level $d \sqcup e$. Activities a_1, a_4 and a_{42} can be safely delegated to a partner that is authorized to look at the medical history of a patient, but not at

[5] These results were obtained in 2.130ms on 2GHz Intel Core Duo machine with 2GB of RAM, running Mac OS X 10.6.3 and CiaoPP 1.12.

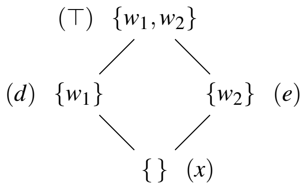
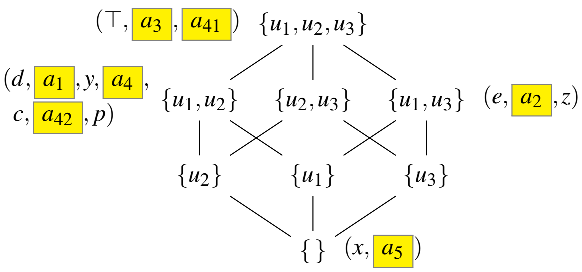


Fig. 9. The base sharing setting for the code from Fig. 11.



$(\top, a_3, a_{41})$ $\{u_1,u_2,u_3\}$
$(d, a_1, y, a_4, c, a_{42}, p)$
$\{u_1,u_2\}$
$\{u_2,u_3\}$
$\{u_1,u_3\}$ (e, a_2, z)
$\{u_2\}$
$\{u_1\}$
$\{u_3\}$
$\{\}$ (x, a_5)

Fig. 10. Interpretation of the sharing results from Fig. 11.

```
[[D,E,T,A1,Y,A2,Z,A3,A4,A41,C,A42,P],     (Corresponding to $u_1$)
 [D,T,A1,Y,A3,A4,A41,C,A42,P],            (Corresponding to $u_2$)
 [E,T,A2,Z,A3,A41]]                       (Corresponding to $u_3$)
```

Fig. 11. Sharing analysis result for the code from Fig. 8.

the medication record. Activity a_2, by contrast, can be delegated to a partner that is authorized to look at the medication record, but not the medical history of a patient; and finally, a_5 can be entrusted to any partner, since it does not handle any private information.

Going in the other direction, we can look at the resulting lattice L' to deduce the policy level that corresponds to any subset of workflow activities. For $B_1 = \{a_1, a_2\}$, $\sqcup B_1 = \top$; for $B_2 = \{a_2, a_5\}$, $\sqcup B_2 = e$, etc.

6 Conclusions

We have shown how sharing analysis, a powerful program analysis technique, can be effectively used to identify fragments in service workflows. These are in our case represented using a rich notation featuring control and data dependencies between workflow activities, as well as nested structured constructs (such as branches and loops) that include sub-workflows. The policies that are the basis for the fragmentation are represented as points in a complete lattice, and the fragments to which input data / activities belong are stated with initial sharing patterns. The key to this use of sharing analysis is how workflows are represented: in our case we have used Horn clauses designed to adequately enforce sharing between inputs and outputs of the workflow activities. The results of the sharing analysis lead to the construction of a lattice that preserves the ordering of items from the original policy lattice and which contains inferred information which can be used for both deciding the compliance of individual activities with given fragmentation constraints, and to infer characteristics of potential fragments.

As future work, we want to attain a closer correspondence between the abstract workflow descriptions and well-known workflow patterns, as well as to provide better support to languages used for workflow specification. Another line of the future work concerns aspects of data sharing in stateful service conversations (such as accesses to databases, updates of persistent objects, etc.), as well as on composability of the results of sharing analysis across services involved in cross-domain business processes.

References

[AP08] Ahmed Awad and Frank Puhlmann. Structural Detection of Deadlocks in Business Process Models. In Witold Abramowicz and Dieter Fensel, editors, *International Conference on Business Information Systems*, volume 7 of *LNBIP*, pages 239–250. Springer, May 2008.

[BlZ04] Henry H. Bi and J. leon Zhao. Applying Propositional Logic to Workflow Verification. *Information Technology and Management*, 5:293–318, 2004.

[BMM06] Luciano Baresi, Andrea Maurino, and Stefano Modafferi. Towards Distributed BPEL Orchestrations. *ECEASST*, 3, 2006.

[CC77] P. Cousot and R. Cousot. 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)*, pages 238–252. ACM Press, 1977.

[FYG09] Walid Fdhila, Ustun Yildiz, and Claude Godart. A Flexible Approach for Automatic Process Decentralization Using Dependency Tables. In *ICWS*, pages 847–855, 2009.

[HBC$^+$10] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. Technical Report CLIP2/2010.0, Technical University of Madrid (UPM), School of Computer Science, March 2010. Under consideration for publication in *Theory and Practice of Logic Programming (TPLP)*.

[Jea07] D. Jordan and et. al. Web Services Business Process Execution Language Version 2.0. Technical report, IBM, Microsoft, et. al, 2007.

[JL92] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314, July 1992.

[Kha07] Rania Khalaf. Note on Syntactic Details of Split BPEL-D Business Processes. Technical Report 2007/2, Institut für Architektur von Anwendungssystemen, Universität Stuttgart, Universitätsstrasse 38, 70569 Stuttgart,Germany, July 2007.

[KL06] R. Khalaf and F. Leymann. E Role-based Decomposition of Business Processes using BPEL. In *IEEE International Conference on Web Services (ICWS'06)*, 2006.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd Ext. Ed., 1987.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *International Conference on Logic Programming (ICLP 1991)*, pages 49–63. MIT Press, June 1991.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[MS93] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. Technical report 93/7, Univ. of Melbourne, 1993.

[Obj09] Object Management Group. *Business Process Modeling Notation (BPMN), Version 1.2*, January 2009.

[TF07] Wei Tan and Yushun Fan. Dynamic Workflow Model Fragmentation for Distributed Execution. *Comput. Ind.*, 58(5):381–391, 2007.

[vdAP06] Wil van der Aalst and Maja Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In *The Role of Business Processes in Service Oriented Architectures*, number 06291 in Dagstuhl Seminar Proceedings, 2006.

[vdAtH05] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, June 2005.

[Wor08] The Workflow Management Coalition. *XML Process Definition Language (XPDL) Version 2.1*, 2008.

[WRRM08] Barbara Weber, Manfred Reichert, and Stefanie Rinderle-Ma. Change Patterns and Change Support Features - Enhancing Flexibility in Process-Aware Information Systems. *Data Knowl. Eng.*, 66(3):438–466, 2008.

[YG07] Ustun Yildiz and Claude Godart. Information Flow Control with Decentralized Service Compositions. In *ICWS*, pages 9–17, 2007.

[ZBDtH06] Johannes Maria Zaha, Alistair P. Barros, Marlon Dumas, and Arthur H. M. ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. In *OTM Conferences (1)*, pages 145–162, 2006.