Orchestrating Composite Web Services Under Data Flow Constraints

Girish Chafle Sunil Chandra Vijay Mann Mangala Gowri Nanda

IBM India Research Laboratory
New Delhi, India
{*cgirish,csunil,vijamann,mgowri*}*@in.ibm.com*

Abstract

A composite service is typically specified using a language such as BPEL4WS and ***orchestrated*** *by a single coordinator node in a* ***centralized*** *manner. The coordinator receives the client request, makes the required data transformations and invokes the component web services as per the specification. However, in certain scenarios businesses might want to impose restrictions on access to the data they provide or the source from which they can accept data.* ***Centralized orchestration*** *can lead to violation of these data flow constraints as the central coordinator has access to the input and output data of all the component web services. In many cases existing methods of data encryption and authentication are not sufficient to handle such constraints. These data flow constraints, thus, present obstacles for composite web service orchestration.*

In this paper we propose a solution for orchestrating composite web services under data flow constraints. The solution is based on ***decentralized orchestration****, in which a composite web service is broken into a set of partitions, one partition per component web service. To overcome data flow constraints, each partition is executed within the same domain as the corresponding component web service and hence, has the same access rights. However, there are, in general, many ways to decentralize a composite web service. We apply a rule based filtering mechanism to choose a set of partitions that does not violate the specified data flow constraints.*

1 Introduction

As web services become ubiquitous, new and complex applications can be created by aggregating the functionality of existing web services (which act as *component web services*). This is referred to as service composition and the aggregated web service is known as a composite web service. Web service composition enables businesses to interact with each other and accomplish complex business processes.

Composite web services may be developed using a specification language such as BPEL4WS [11], WSCI [2] *etc.*, and executed by an engine such as WebSphere Process Choreographer [1]. Typically, a composite web service specification is executed by a single coordinator node. It receives the client requests, invokes the component web services, makes the required data transformations and executes the composite web service according to its specification. We refer to this mode of execution as *centralized orchestration*. In centralized orchestration, all data is routed through the central coordinator, and it has access to the input and output data of all the component web services. However, in certain scenarios web services may apply restrictions on the source and/or destination of the data received or sent, as part of a policy. We term these restrictions as "business defined data flow constraints"[1]. These data flow constraints, thus, present obstacles in the orchestration of composite web services by a central coordinator. Several mechanisms for handling security issues exist in the distributed computing world. These include various methods of authentication and encryption. However, in many cases, as we show later in the paper, these existing methods fail to solve the problem of composite web service orchestration.

In this paper we propose a solution for orchestrating composite web services in presence of business defined data flow constraints. Our solution is based on decentralized orchestration [6]. In decentralized orchestration, a composite web service is broken into a set of partitions, one partition per component web service. The partitions are colocated with the web service. Each partition acts like a proxy that processes, transforms and manages all incoming and outgoing data at the component web service as per the requirements of the composite service. The partitions execute independently and interact with each other directly using asynchronous messaging without any centralized control. Since a partition is colocated with a component web service, it has the same access rights as the web service. Thus it is possible to overcome data flow constraints by ensuring that all

[1] We use the terms "business defined data flow constraints", "data flow constraints" and "constraints" interchangeably in the rest of this paper

constrained data reads and writes are performed within the permitted domains. This is done by automatically decentralizing a composite web service [13]. However, the decentralization algorithm generates all possible partitionings (referred to as a *topology*) which may or may not adhere to data flow constraints. In this paper, we attempt to solve the problem of finding decentralized topologies that do not violate data flow constraints. We propose a rule based filtering mechanism that is capable of selecting topologies that do not violate the specified data flow constraints. Our solution consists of a *Decentralizer* tool to generate all possible topologies, a language to specify data flow constraints, a filtering mechanism to select a topology, and a deployment mechanism to ensure that a partition hosted by a component web service does not violate any constraints specified by that service. We explain our solution in the context of BPEL4WS language. However, our solution is applicable to any composite service specification language.

The organization of the rest of the paper is as follows. We begin by providing the motivation for the problem in Section 2. In Section 3 we give an outline of the solution along with the details of the main components. An overview of the system architecture and a prototype run of the system are given in Section 4. We summarize related work in Section 5 and finally conclude in Section 6 with an outline of future work.

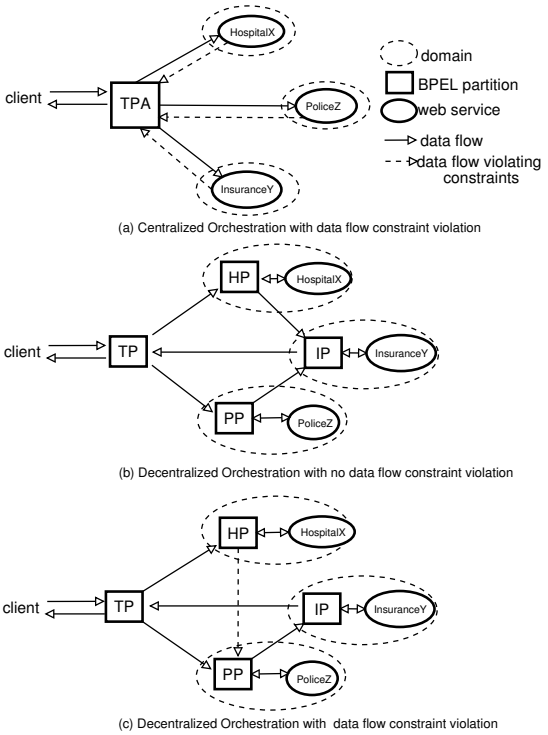


Figure 1. Centralized and Decentralized Orchestration Under Data Flow Constraints

2 Motivation

In this section we introduce a motivating example that is used as a running example throughout the paper. Further, we show that existing security mechanisms do not provide sufficient flexibility to handle orchestration of composite web services in a centralized manner in presence of data flow constraints.

2.1 A Motivating Example

Consider a third party administrator (*TPA*) that provides an accident insurance claim service as shown in Figure 1a. In this case the client submits a request to the TPA with the patient, insurance, and accident details. The TPA settles the claim by interacting with the web services hosted by the hospital - HospitalX, the city police department - PoliceZ and the insurance company - InsuranceY. In the absence of any data flow constraints, the *TPA* can create a composite web service specification and execute it using centralized orchestration. Thus, on receiving a client request, it gets the medical records from the hospital, the police report from the police department, and passes them on to the insurance company to settle the claim. However, in a real world scenario, the hospital might want to maintain the confidentiality of the medical records and reveal them only to the patient himself or to the insurance company. Similarly, the police department might not be willing to share the police report with the TPA and reveal it only to the insurance company. Furthermore, the insurance company might not want to accept the medical records or the police report from any intermediary and only directly from the hospital or the police department respectively. These data flow constraints present obstacles in centralized orchestration of this composite web service.

2.2 Security Mechanisms in Centralized Orchestration

Data flow constraints are typically handled through encryption and related security mechanisms in distributed systems. The web services security (WS-Security) [17] specification deals mainly with aspects of security, privacy and trust between the client and the web services or between two web services. Composition of autonomous web services by a third party may require a rich set of data transformations to be applied in between the sequential invocations of component web services. However, the existing security mechanisms have limited support for this type of data access and manipulation. Consider the example in Figure 1 (a). Given a patient id, the HospitalX web service returns

the corresponding medical record which includes pathological tests, x-rays, dental records, physician's report, billing and a summary. The HospitalX web service encrypts its output for security and this can be decrypted only by the InsuranceY web service. Further, we note that the insurance company is interested only in the billing details and the summary portion of the medical record. Therefore, the *TPA*, needs to extract the billing details and the summary from the medical record before forwarding it to the InsuranceY web service. This implies that the *TPA* needs to be able to decrypt the medical record, which for security reasons is not permissible. Alternatively, if different portions of the medical record are encrypted separately, then it may still be possible for the *TPA* to extract the relevant portions (billing details and summary in this case) from the medical record. Note that, in this case, even though the *TPA* is unable to decrypt any portion of the medical records, it is still able to forward the relevant portions (in their encrypted form) to the InsuranceY web service. Thus, in the presence of encryption, a composite web service can be orchestrated in a centralized fashion *only* if

- no data transformation is required in between different web service invocations, or
- each part in the output message of a component web service is encrypted separately and these parts map directly onto the input parts of the next component web service involved in composition.

Even in the latter case, the coordinator node (*i.e.,* the *TPA*) might want to extract finer details from a part in order to provide a more useful and efficient service. This could include employing some business logic based on the data in the part to decide on the control flow.

Thus, existing methods of data encryption and authentication [17] suffer from severe limitations in centralized orchestration in the presence of data confidentiality and privacy requirements. Further, use of centralized orchestration can lead to performance degradation (in terms of throughput, response time, scalability, availability) [6, 7].

3 Decentralization and Topology Filtering

Figure 2 gives an outline of the proposed solution. It consists of three main components - the *Decentralizer*, the *Topology Filtering Mechanism*, and the *Deployment mechanism*. The *Decentralizer* takes a composite service specification as input and generates different topologies, some of which may violate constraints. The *Topology Filtering Mechanism* filters out topologies based on data flow constraints. It includes an XML based language to specify data flow constraints as rules and a *Rule based Topology Filter*. Since there may be more than one topology which adheres

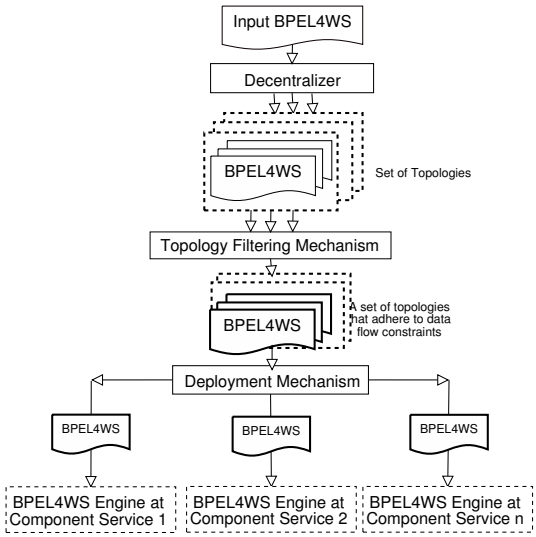


Figure 2. Solution outline

to all the constraints, the *Topology Filter* can output multiple valid topologies, out of which the first one is chosen and and its partitions are deployed. The *deployment mechanism* inside the component web service domain reinsures that a partition hosted by a given component web service domain does not violate any constraints specified by the web service. The details of each of these components are explained in the sections below.

3.1 Decentralizer

In decentralized orchestration, a composite web service is broken into a set of partitions, one partition per component web service. The partitions execute independently and interact with each other directly using asynchronous messaging without any centralized control. Together, the decentralized partitions perform the same task that the centralized coordinator would have done. However, decentralized orchestration offers performance advantages over centralized orchestration [6]. In an earlier paper we have given an algorithm and heuristics to automatically partition a composite web service specified in BPEL4WS [13]. The issues of concurrency and synchronization arising out of decentralization were discussed in [14].

The *Decentralizer* is a tool that implements the partitioning algorithm. It takes the BPEL4WS specification as input and generates a subset of all the possible topologies based on a chosen heuristic. Two of the topologies generated by the tool for the TPA example are explained below.

Example: In Figure 1(a), TPA is a composite web service written in BPEL4WS. It is "decentralized" into partitions TP, HP, IP and PP (refer Figure 1(b) and Figure 1(c)), each

a complete BPEL4WS program in itself, that together perform the same set of operations that TPA does, albeit in a distributed manner. The HP, PP and IP composite service partitions are colocated with the HospitalX, PoliceZ and InsuranceY web services in their respective domains. Therefore, they have not only the same access privileges as the corresponding component web service, but are also subject to the same data flow constraints. In Figure 1(b), TP receives the client request, sends the patient and insurance details to HP and the accident details to PP. HP contacts the HospitalX web service, gets the medical record, extracts the relevant portions from it (billing details and summary in this case) and forwards it along with the insurance details to IP. Similarly PP gets the police report from the PoliceZ web service, and forwards it to IP. IP, in turn, invokes the InsuranceY web service to settle the claim and returns the response to TP. This topology does not violate any of the data flow constraints and can be used for orchestrating the composite web service. However, this is not always the case and some of the topologies can violate data flow constraints (*e.g.,* the topology in Figure 1(c)).

In Figure 1(c), TP receives the client request, sends the accident and insurance details to PP, and patient details to HP. HP extracts the relevant data from the HospitalX web service, and forwards it to PP. PP gets the police report from the PoliceZ web service, and forwards it along with the medical records and insurance details to IP. IP contacts InsuranceY web service to settle the claim and sends the response back to TP. In this topology, medical records are passed from HP to PP, thereby violating HospitalX's data flow constraint. Similarly, the medical record is passed to IP by PP which violates InsuranceY's data flow constraint. This topology would be rejected by the *Topology Filter*.

3.2 Topology Filtering Mechanism

The *Topology Filtering Mechanism* is used to reject topologies violating data flow constraints. It consists of a language to specify data flow constraints as rules, the *Rule based Topology Filter* and the *Constraint Reinforcer*.

3.2.1 Data Flow Rules Specification

Each web service specifies a set of rules based on the security policies of the organization. These may be encoded using the WSDL extensibility mechanism. Rules are expressed as a 3-tuple of ⟨*source, destination, Message*⟩. Both the source and the destination are domain names. *Message* is the input message type that a particular `WSDL portType` expects or output message type that it sends back. Rules fall under the *"Allowed"* and *"Not Allowed"* categories. *"Allowed"* rules are those where either a source can send data to a given destination, or where a destination can accept data from a given source. *"Not allowed"* rules are those where either a source cannot send data to a given destination or where a destination cannot receive data from a given source. The source and destinations can also be expressed in terms of domain name sets *e.g., *.co.in* for all the companies in India, *?Police.org* for all police departments. For the TPA example in Figure 1(b) and 1(c), data flow constraints provided by HospitalX web service will consist of *"Allowed"* rules such as ⟨*xHospital.com, yInsurance.com, MedicalRecords*⟩ and *"NotAllowed"* rules such as ⟨*xHospital.com, *, MedicalRecords*⟩. In this case the more specific (*i.e.,* catering to a narrower set of sources or destinations) rule (*i.e.,* the *"Allowed"* rule) will appear first followed by the less specific rule (*i.e.,* the *"NotAllowed"* rule). These rules can be specified using policy languages such as XACML [9].

```
procedure filter (ConstraintList ζ, TopologySet θ)
foreach topology T in θ do
    T ="valid"
    foreach partition P in T do
        P ="valid"
        // search for incoming and outgoing message end-points
        foreach incoming and outgoing end-point E do
            E.src = getSourceDomainName(WSDL List)
            E.dest = getDestinationDomainName(WSDL List)
            E.msg = getMessageType(WSDL List)
            create 3-tuple τ = ⟨E.src, E.dest, E.msg⟩
            if E.src ≠ E.dest then
                if match(τ, ζ) == "not allowed" then
                    P ="discarded"
                    break
                endif
            endif
        done
        if P =="discarded" then
            T="discarded"
            break
        endif
    done
done

function match(Tuple τ, ConstraintList ζ)
returns: "allowed"|"not allowed"
foreach Constraint C in ζ do
    if τ == C and C ∈ {Allowed Constraints} then
        return "allowed"
    elseif τ == C and C ∈ {Not Allowed Constraints} then
        return "not allowed"
    endif
done
return "allowed"
```

Figure 3. Topology Filtering Algorithm

3.2.2 Rule Based Topology Filter

As mentioned earlier, the *Decentralizer* may generate several topologies. It is the task of the *Rule Based Topology Filter* to reject those topologies that do not adhere to

data flow constraints. The *topology filtering* algorithm is given in Figure 3. It takes as input the set of constraints and all the topologies generated by the *Decentralizer*. For each topology, it parses all the partitions and searches for incoming and outgoing message end-points within the partition. In BPEL4WS, these end-points are represented by `receive` or `pick` for an incoming message, `invoke` for both an outgoing message (either a request in synchronous request/response message, or an asynchronous request message) and an incoming message (a response in synchronous request/response message) or `reply` for an outgoing message.

For each `invoke`, the filter extracts the fully qualified name of the corresponding WSDL `message` and `portTypes`. For each `portType`, the filter parses the corresponding WSDL descriptors and extracts the domain name (to which that `portType` is bound). This domain becomes the destination domain for the request portion of `invoke`. The current partition's domain name (inferred from the deployment information) becomes the source domain. This combination of source domain, destination domain and message is used to create a 3-tuple for both the the current partition and the invoked partition. Similarly, for the response portion of `invoke`, the current partition's domain becomes the destination and the domain to which the `portType` is bound becomes the source domain and a 3-tuple for both the partitions is generated. The functions *getSourceDomainName (WSDL List), getDestinationDomainName (WSDL List), and getMessageType (WSDL List)* listed in the algorithm refer to this process. In this way, the *Topology Filter* generates a list of 3-tuples for each partition consisting of the domain of the source of the message, the domain of the destination and the message type. All the 3-tuples where the source and destination domain names are the same are removed from this list (as a communication between different entities within a domain does not violate any data flow constraint by design).

For each such 3-tuple for a partition, the *Topology Filter* applies a constraint matching function which takes as input the 3-tuple and a list of rules (in the form of 3-tuples) for a particular partition. The constraint matching function is also given in Figure 3.

3.2.3 Constraint Reinforcer

The task of the *Constraint Reinforcer* is to ensure that the original data flow constraints are strictly adhered to. There are two reasons that a *Constraint Reinforcer* is required.

- The web service generates data flow constraints based on the incoming and outgoing messages at the web service. However, during composite service creation and its subsequent decentralization, parts of these messages may get copied into other messages within the composite service partitions.
- A malicious program may copy the contents of a message, that has data flow rules associated with it, into another message in order to access data without authorization.

In both these cases, the modified messages need to be detected and additional set of data flow rules for these messages need to be generated automatically. The *Constraint Reinforcer* generates these rules which are in addition to the given set of constraints specified by a web service. These are passed on to the *Topology Filter* to add to its rules set. The *Constraint Reinforcer* is first executed during the topology filtering process. It is again invoked by the component service runtime infrastructure at the time of deployment at the component web service's domain to ensure that a deployed partition does not violate the constraints stipulated by that component web service.

Example: In the TPA example, if the data transformations generate a new message type *ModifiedMedicalRecords* that consists of only a portion (*billing details and summary*) of the original message type *MedicalRecords*, then a new set of rules for *ModifiedMedicalRecords* should be generated and passed on to the *Topology Filter*.

The *Constraint Reinforcer* takes all the rules specified by the web service for a given message type (*MedicalRecords* in this case) and creates a new set of rules in which the name of the message type (*i.e., MedicalRecords*) is replaced by the newly generated message type (*i.e., ModifiedMedicalRecords*). For example, for the *"Allowed"* data flow rule - ⟨*xHospital.com, yInsurance.com, MedicalRecords*⟩, the *Constraint Reinforcer* generates a new additional *"Allowed"* rule of the form - ⟨*xHospital.com, yInsurance.com, ModifiedMedicalRecords*⟩and passes it to the *Topology Filter*.

Algorithm The *Constraint Reinforcer* uses data flow analysis to detect messages that need additional rules. For each partition it builds a Data Dependence Graph (DDG). The DDG is a graph that consists of one node for each activity in a partition. There is an edge from a node n_i to another node n_j, if there is a data dependence [3] from n_i to n_j. To generate additional data flow rules, the *Constraint Reinforcer* applies the following algorithm

- For each variable, v_i that accepts data at an incoming end-point, trace the DDG forward along data dependence edges. If we reach a variable, v_c that has a data flow constraint specified by the web service, then generate a corresponding rule for v_i.
- For each variable, v_o that sends data at an outgoing end-point, trace the DDG backward along data depen-

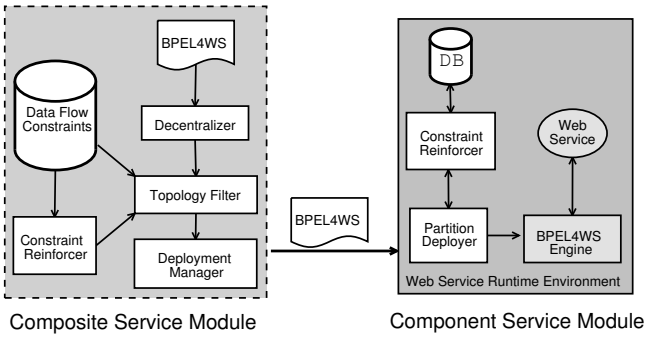


Figure 4. System Architecture

dence edges. If we reach a variable, v_c that has a data flow constraint specified by the web service, then generate a corresponding rule for v_o.

3.3 Deployment Mechanism

The *Deployment Mechanism* consists of a *Deployment Manager* used by an administrator composing the web service, a *Partition Deployer* and the *Constraint Reinforcer* (as shown in Figure 4). The *Partition Deployer* and the *Constraint Reinforcer* are part of the web services' runtime environment.

Deployment Manager The *Deployment Manager* takes as input a set of valid topologies for deployment from the *Topology Filter*. It selects the first topology for deployment and sends partitions of that topology to the corresponding component web service nodes.

Partition Deployer At each component web service node there is an agent, the Partition Deployer, which is colocated with the component web service. The Partition Deployer has two main functions: (1) constraint checking and verification, and (2) deployment. Constraint checking and verification is essential because the partition is generated by an external entity and after deployment the partition executes within the domain as a trusted piece of code and has full access to unencrypted output data of the component web service even if encryption is being used. The *Partition Deployer* accepts the incoming BPEL4WS partitions and hands them to the *Constraint Reinforcer* to generate the additional set of data flow rules. In cases where encryption is being used through WS-Security, the *Constraint Reinforcer* will be utilized to generate additional security policies so that any confidential data that is flowing in or out of that node in the form of newly created message types is also encrypted. The *Partition Deployer* then uses the same algorithm as the *Topology Filter* to verify that the partition submitted for deployment at this domain satisfies all the data flow constraints specified for this domain. After constraint checking and verification, partition is deployed on to the BPEL4WS engine.

Table 2. Discarded Topologies

Topology	Edge	Data	Constraints Violated
2	HP-PP	Medical Records	x.3, y.4
8	PP-HP	Police Report	y.3, z.2
3	HP-PP	Medical Records	x.3, y.4

4 Implementation

4.1 Architecture Overview

The system for orchestrating composite web services in constrained data flow environments has two modules (refer Figure 4). The *composite service module* is used by an administrator to create a valid topology(that doesn't violate any data flow constraints) from the original input specification. It consists of the *Decentralizer*, the *Topology Filter*, the *Constraint Reinforcer*, and the *Deployment Manager*. The *component service module*, consists of the *Constraint Reinforcer* and the *Partition Deployer*, which reside inside the same domain as the component web service (preferably as part of the web service's runtime environment).

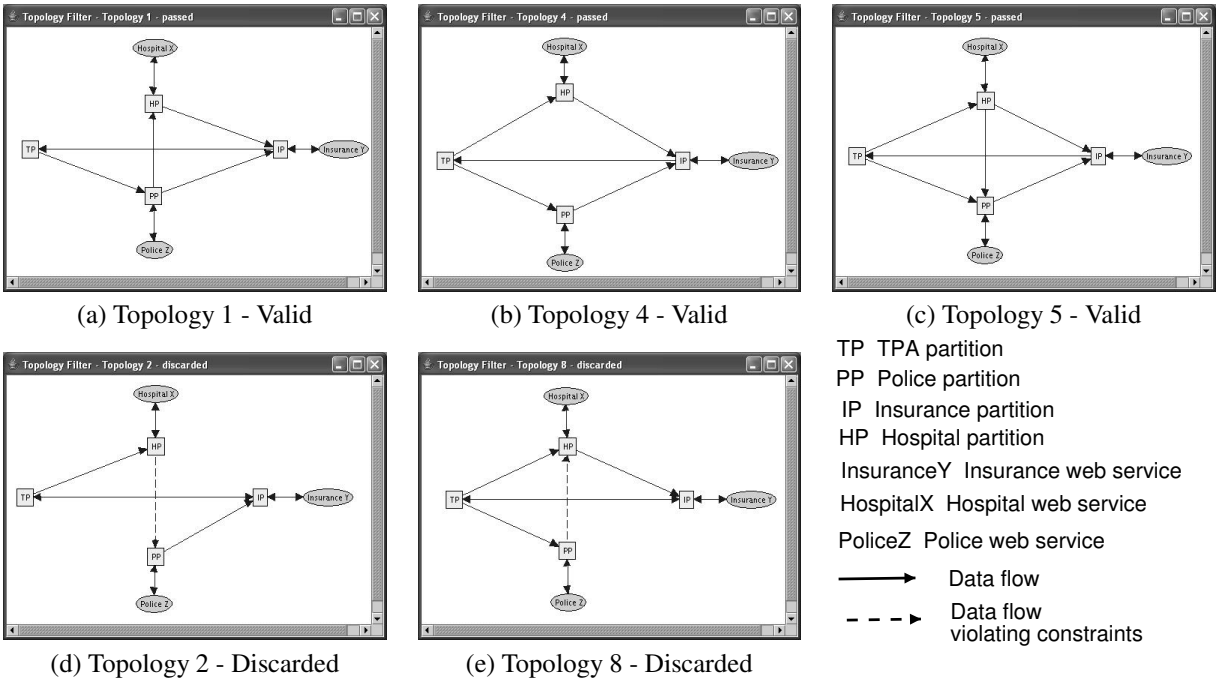
4.2 A Prototype Run

This section describes a sample run of the system for the TPA example. The *Decentralizer* tool generated 8 different topologies. These topologies were given to the *Topology Filter* and the rules given in Table 1 were used for filtering. The XML schema that was used for specifying these rules is given in Appendix A.

The *Topology Filter* discarded 3 topologies and validated 5 topologies. Some of the valid and discarded topologies are shown in Figure 5. The remaining topologies are not shown due to lack of space. The discarded topologies violate one or more data flow constraints indicated in Table 2 and shown as dashed edge in Figure 5. Among the valid topologies, 1, and 5 require some explanation as they have a data flow edge resembling the data flow edges violating some constraints in discarded topologies. In topology 8 the data flow edge from PP to HP represents police report being transferred from PP to IP via HP whereas in topology 1 it represents patient details (to get medical records from HP) that is routed via PP and hence is not a violation. Similarly, in topology 2 the data flow edge from HP to PP represents medical records being transferred from HP to IP via PP whereas in topology 5 it represents accident details from

Table 1. Rules for Prototype Run

Web Service	Tuple ⟨ source, destination, message type ⟩	Category	Id
HospitalX	⟨ xHospital.com,iInsurance.com, MedicalRecords ⟩	Allowed	x.1
	⟨ xHospital.com,yInsurance.com, MedicalRecords ⟩	Allowed	x.2
	⟨ xHospital.com,*, MedicalRecords ⟩	NotAllowed	x.3
InsuranceY	⟨ ?Hospital.com, yInsurance.com, hospitalRecords ⟩	Allowed	y.1
	⟨ ?Police.org, yInsurance.com, policeReport ⟩	Allowed	y.2
	⟨ *, yInsurance.com, policeReport ⟩	NotAllowed	y.3
	⟨ *, yInsurance.com, hospitalRecords ⟩	NotAllowed	y.4
PoliceZ	⟨ zPolice.org, ?Insurance.com, investigationReport ⟩	Allowed	z.1
	⟨ zPolice.org, *, investigationReport ⟩	NotAllowed	z.2



(a) Topology 1 - Valid (b) Topology 4 - Valid (c) Topology 5 - Valid

(d) Topology 2 - Discarded (e) Topology 8 - Discarded

Figure 5. Topologies Generated

TP(to get police report from PP) that is routed via PP and hence is not a violation.

5 Related Work

There are four primary areas of research that are related to this work: reinforcement of information-flow policies, decentralized orchestration of composite web services, rule driven web service composition and web services and workflow security.

Information flow policies [8, 10] are used to specify confidentiality and integrity requirements and control the *end-to-end* use of data in a secure system. Secure program partitioning [18] is a language based technique for protecting confidential data during computation in distributed systems containing mutually untrusted hosts. Confidentiality and integrity policies are expressed by annotating the programs with confidentiality labels. The program can then be partitioned automatically to run securely on heterogeneously trusted hosts. In our solution, data flow constraints expressed in terms of *source, destination and message type* are similar to confidentiality labels used in secure partitioning. Web services provide the added advantage of defining them as extensibility elements in the WSDL rather than annotating the source program with security policies.

Decentralization is a relatively new technique for orchestrating web services [13, 6, 14], although it has been applied in earlier research for enabling distributed workflow execution [12]. SELF-SERV [5] is a framework for dynamic provisioning of web services based on the ideas of decentralized orchestration and peer to peer execution. Most of the earlier approaches have studied decentralization from the angle of performance. To the best of our knowledge no one has used it for overcoming data flow constraints

in orchestrating composite web services.

[15] describes a rule driven approach for flexible and dynamic service composition in an automated fashion. Zeng *et al.* [19] uses a rule inference framework, where end users declaratively define their business objectives to dynamically compose web services. Ponnekanti *et al.* [16] propose a rule-based algorithm that permits semi-automation of workflow composition. All these systems generate a valid centralized composite service specification that adheres to all the given rules. We go one step further and try to solve the problem where no centralized composite service specification can be generated that adheres to the given rules. Atluri *et al.* [4] discuss security issues in decentralized workflows. Their focus is on data security problems created due to exposure of business logic in decentralized workflows. Some of these security concerns can be expressed in terms of data flow constraints provided by the coordinator node orchestrating the composite service and can be then handled by our proposed solution.

6 Conclusions and Future Work

In this paper we have extended our previous work on decentralized orchestration to use it for orchestrating composite web services under data flow constraints. To the best of our knowledge, no one has used decentralized orchestration to solve this problem. Our solution includes an XML based language for specifying the data flow constraints, a rule based filtering mechanism to filter topologies generated by decentralization, and a deployment mechanism to ensure enforcement of constraints at run time.

The solution described in this paper, raises several interesting issues which need to be addressed. Remote deployment of BPEL4WS partitions at the location of component web services requires further investigation.

Similarly, future versions of BPEL or other workflow languages might allow embedded program code necessitating a more rigorous verification of the partitions. Decentralization of a composite service can also result in new security concerns as the business logic is exposed to multiple parties [4]. We handle some of these concerns (which can be expressed in terms of data flow constraints) in our current solution. We are working on enhancing our XML based language to include other types of constraints that can capture these security issues.

References

[1] WebSphere Application Server Enterprise Proess Choreographer. http://www7b.software.ibm.com/wsdd/zones/was/wpc.html.

[2] Web Service Choreography Interface (WSCI) 1.0. http://www.w3.org/TR/wsci, 2002.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] V. Atluri, S. A. Chun, and P. Mazzoleni. A Chinese Wall Security Model for Decentralized Workflow Systems. In *Proceedings of the Conference on Computer and Communications Security,Philadelphia*, November 2001.

[5] B. Benatallah, M. Dumas, Q. Sheng, and A. Ngu. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, February 2002.

[6] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized Orchestration of Composite Web Services. In *Proceedings of the 13th International World Wide Web Conference (WWW)*, New York, USA, May 2004.

[7] Q. Chen and M. Hsu. Inter-Enterprise Collaborative Business Process Management. In *Proceedings of 17th International Conference on Data Engineering (ICDE)*, 2001.

[8] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

[9] eXtensible Access Control Markup Language (XACML). http://www-106.ibm.com/developerworks/xml/library/x-xacml/.

[10] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–86, 1984.

[11] R. Khalaf, N. Mukhi, and S. Weerawarana. Service-Oriented Composition in BPEL4WS. In *Proceedings of the Twelfth International World Wide Web Conference (WWW)*, 2003.

[12] P. Muth, D. Wodtke, J. Weissenfels, A. K. Dittrich, and G. Weikum. From Centralized Workflow Specification to Distributed Workflow Execution. *Journal of Intelligent Information Systems*, 10(2), April 1998.

[13] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing Execution of Composite Web Services. In *Proceedings of Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2004.

[14] M. G. Nanda and N. Karnik. Synchronization Analysis for Decentralizing Composite Web Services. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2003.

[15] B. Orriens, J. Yang, and M. Papazoglou. A Framework For Business Rule Driven Web Service Composition. 22nd International Conference on Conceptual Modeling (ER 2003).

[16] S. R. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Building Composite Web Services. In *Proceedings of the 11th International World Wide Web Conference*, 2002.

[17] Web Services Security (WS-Security). http://www.ibm.com/developerworks/webservices/library/ws-secure/.

[18] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted Hosts and Confidentiality: Secure Program Partitioning. In *Symposium on Operating Systems Principles (SOSP)*, 2001.

[19] L. Zeng, B. Benatallah, H. Lei, A. Ngu, D. Flaxer, and H. Chang. Flexible Composition of Enterprise Web Services. In *Electronic Markets*, volume 13, pages 141–152, 2003.

A Schema for Specifying Rules

```
<schema targetNamespace="http://tempuri.org/dfc"
  xmlns:tns="http://tempuri.org/dfc"
  xmlns="http://www.w3.org/2000/10/XMLSchema">

 <complexType name="dataType">
  <attribute name="type" type="QName"
         use="required"/>
 </complexType>

 <complexType name="endpoint">
  <attribute name="domainName" type="uriReference"
         use="required"/>
 </complexType>

 <complexType name="ruleType">
  <sequence>
   <element name="source" type="tns:endpoint"/>
   <element name="destination"
         type="tns:endpoint"/>
   <element name="message" type="tns:dataType"/>
  </sequence>
 </complexType>

 <complexType name="rulesType">
  <sequence>
   <element name="allowed" type="tns:ruleType"
          minOccurs="0" maxOccurs="unlimited"/>
   <element name="notAllowed" type="tns:ruleType"
         minOccurs="0" maxOccurs="unlimited"/>
  </sequence>
 </complexType>

 <element name="rules" type="tns:rulesType"/>
</schema>
```

The *"Allowed"* and *"NotAllowed"* rules can appear in any relative order subject to the condition that more specific constraints (*i.e.,* catering to a narrower set of sources or destinations) appear first followed by less specific ones.