Discovering and Deriving Service Variants from Business Process Specifications*

Karthikeyan Ponnalagu and Nanjangud C. Narendra

IBM India Research Lab, Bangalore, India
{karthikeyan.ponnalagu,narendra}@in.ibm.com

Abstract. Software service organizations typically develop custom solutions from scratch in each project engagement. This is not a scalable proposition, since it depends too heavily on labor alone. Rather, creating and reusing software "assets" is more scalable and profitable. One prevalent approach to building software solutions is to use service-oriented architecture (SOA) to compose software services to support business processes. In this context, the key to reusing assets is to discover (from existing assets in a portfolio) or derive service variants to meet the requirements of a stated business process specification. To that end, this paper presents our Variation-Oriented Service Design (VOSD) algorithm for the same. Via IBM's Rational Software Architect modeling tool, we also demonstrate the practical usefulness of our algorithm via a prototype implementation in the insurance domain.

Keywords: Service-oriented Architecture, Business Process, Reuse.

1 Introduction and Motivation

Software service organizations developing custom business solutions are being faced with the increased need to effectively reuse existing assets and thereby enhance profitability. The emergence of service oriented architecture (SOA) [5], with its emphasis on loose coupling and dynamic binding of services, promises to enable more effective reuse by developing business processes as loosely-coupled collections of services modeled as reusable assets. However, one major obstacle against the realization of this vision is the cost involved in modeling and manipulating service variants in order to meet varied business process specifications. Currently this practice is carried out manually, making it cumbersome and error-prone. To that end, in this paper we present an algorithm called Variation Oriented Service Design (VOSD), which automatically discovers (from reusable assets in a portfolio) and/or derives (in case the reusable asset is not available) service variants from a portfolio of existing services to meet a stated business process specification. Our approach leverages from our earlier Variation-Oriented Engineering methodology (VOE) [3], which is an end-to-end approach spanning business processes to their SOA implementation to formally model and develop these variants, so that the reuse of solutions with variants can be facilitated.

* Thanks to Dipayan Gangopadhyay, Biplav Srivastava and Islam Elgedawy for their feedback.

A. Bouguettaya, I. Krueger, and T. Margaria (Eds.): ICSOC 2008, LNCS 5364, pp. 691–707, 2008.

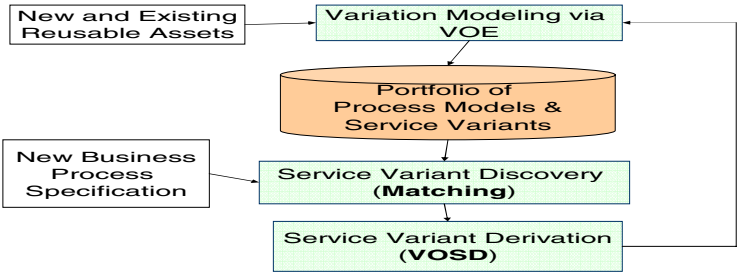

Fig. 1. Integrated Service Variant Discovery and Derivation Approach

This paper is an extension of an earlier paper [7], which introduced our basic VOSD algorithm. That paper, however, only focused on deriving service variants from stated business process specifications; in this paper, we extend the algorithm in [7] to the problem of selecting the appropriate (variant of) services in a portfolio, said variants being stored in a manner consistent with VOE principles. Service variant derivation is resorted to only if the appropriate variant is not available. We also demonstrate that this extension is non-trivial, and show how the meta-modeling approach in VOE is able to assist in automatic service variant discovery. Our overall approach is illustrated in Figure 1. Throughout the paper, we illustrate our algorithm on a simple yet realistic running example (drawn from a real-life project engagement) in the insurance domain. We also demonstrate our algorithm on a prototype implementation using IBM's Rational Software Architect (RSA) modeling tool, thereby demonstrating its practical usefulness (please note, however, that our algorithm is independent of the modeling tool used).

This paper is organized as follows. We will first describe our running example. We then describe some preliminary concepts on which our integrated approach is based. We then describe our integrated VOSD discovery + derivation algorithm. We will then illustrate our algorithm on the running example via the RSA tool (however, our algorithm is not dependent on RSA, and can be implemented on any UML-based modeling tool). The paper will finally conclude with discussions on related & future work.

2 Running Example

Let us assume a basic insurance claims process solution, Sol1, which has been implemented for a customer, as in Figure 2. The net output of the execution of this process is an evaluation of the applicant's claim; if the claim is accepted, the output would also include information on the payment to be made to the applicant. For this paper, we focus on the **Verify Claim** sub-process; in this sub-process, the *DetermineLiability* and *PotentialFraudCheck* services are first executed in parallel, and send their results to Claim Investigation service. A final review of the verified claim is then implemented by *FinalReview* service.

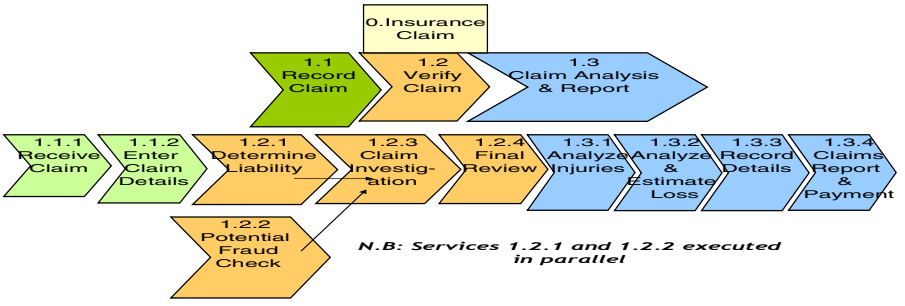

Fig. 2. Basic Insurance Claims Process – Solution Sol1

Let us assume that the same solution should now be tailored for a second customer to produce Sol2, with the following changed requirements:

- Improve cycle time for **Verify Claim** sub-process – for a new class of "high priority" applicants not previously served
- Improve fraud checking – a new and improved fraud checking module to be incorporated, given the fact these are now high-value claims

Based on these inputs, the following variations to Sol1 are anticipated:

- *DetermineLiability* & *PotentialFraudCheck* services to be outsourced for improved speed
- Also, *PotentialFraudCheck* service should be modified to take into account extent of liability – this would eventually involve changes in its business logic
- New *Liability+FraudChecks* service to be added

Let us assume that these variations would result in solution Sol2, as shown in Figure 3.

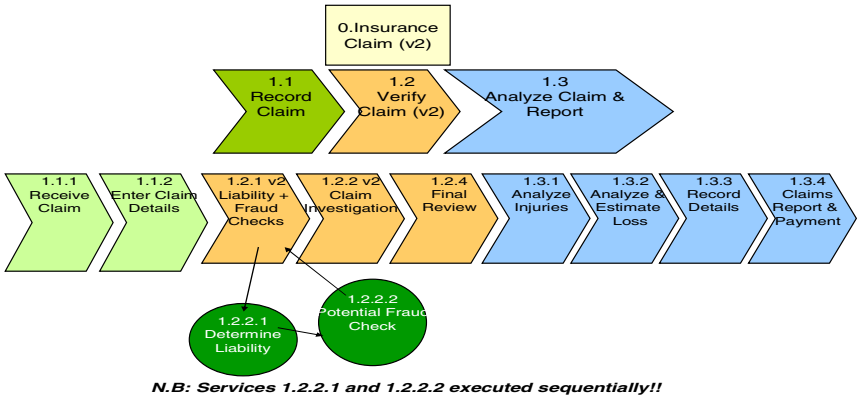

Fig. 3. Modified Insurance Claims Process – Solution Sol2

The original inputs and outputs for the services to be modified in Sol1 are as in Figure 4.

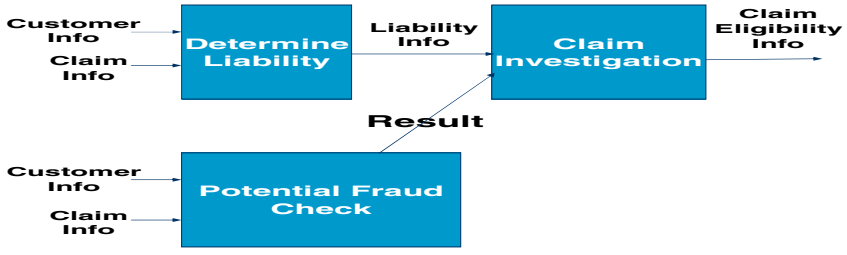

Fig. 4. Original Services from Sol1

The modifications for Sol2 are shown in Figure 5.

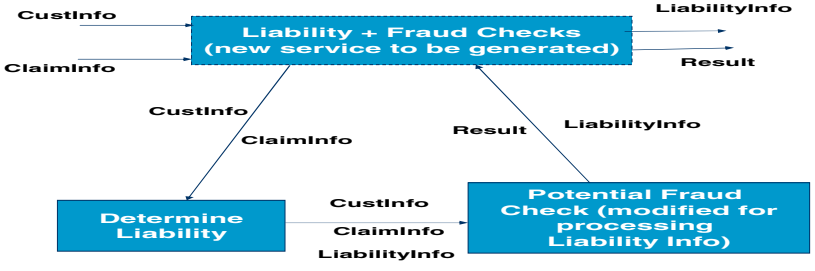

Fig. 5. Modified Services for Sol2

Upon examination of Figures 4 and 5, the following question arises: how do we decide that the *DetermineLiability* and *PotentialFraudCheck* services need to be sequentialized while they were originally executing in parallel? Also, how do we determine what the inputs and outputs of *Liability+FraudChecks* service should be? Current practice would be to determine the answers to these questions based on a combination of visual inspection (perhaps based on modeling the business processes in a process modeling tool) and manual calculation.

From a software engineering perspective, however, such an approach is cumbersome and time-consuming, and hence does not scale to large business processes. Additionally, such a calculation is in general non-trivial. For example, if a service S_k were to be sequentially inserted between S_i and S_j, this may involve the introduction of new control and data dependencies among S_i, S_j and S_k. The question that then arises, is how should the functionality of S_i and S_j be modified? That is, some outputs of S_i may need to be redirected to S_j; also, some inputs of S_j may have to be obtained from S_k. The introduction of S_k may also create new outputs, which would have to be redirected to S_j, thereby necessitating an enhancement of the functionality of S_j, i.e., the source code of the service S_j will need to be modified. This can be observed from

the example of inserting the *Liability+FraudChecks* service between *DetermineLiability* and *PotentialFraudCheck* services. Now, when the number of such business process-level changes grows, determining all the sevice-level code changes becomes a major design exercise that can only be eased and speeded up via automation. Hence the need for an algorithm such as VOSD.

3 Preliminaries

3.1 Meta Model-Based Representation of Variations

We leverage the metamodel introduced in [3] that allows one to separately model the static and variable parts of any software component (service or business process) for our VOSD algorithm. This metamodel consists of the following parts, and is depicted in Figure 6:

- *Variation Points* - these are the points in the component where variations can be introduced. They are in turn of two types. `Implementation variation points` are the points in the component where the implementations of certain methods can be modified, without affecting the externally observable behavior of the component. Whereas, `specification variation points` are the points at the interface of the component which can be modified. This may necessitate changes to the internal implementation of the component, which are specified via implementation variation points. Specification variations could therefore involve adding new input/output data, and/or removing input/output data to the component.
- *Variation Features* – these further refine variation points, by specifying the action semantics of the variation and its specific applicability. The same variation point can admit more than one variation feature, and one variation feature can be applied to many variation points.

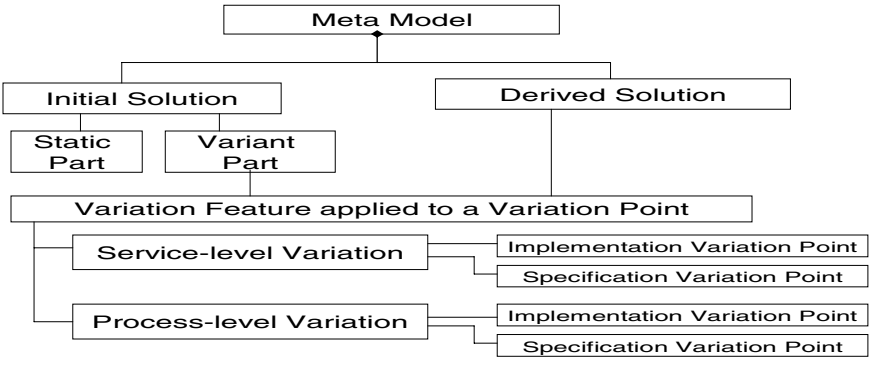

Fig. 6. VOE Meta-Model

This meta model will be further specialized to instantiate conceptual models for modeling service-level and business process-level variations. These conceptual models can then be treated as design templates from which actual variation-oriented design can be accomplished. Hence for our running example, an example of a variation point would be a method in *DetermineLiability* service for calculating insurance liability. A variation feature would be an action to replace that method by a different method. The actual service variant would be the modified *DetermineLiability* service.[1]

3.2 Modeling Business Process and Service Level Variations

Based on the metamodel from [3], we broadly categorize variations in a business process-based solution into two cases. *Service-level Variations* are variations at the level of individual services. They can in turn be classified into two sub-cases.

- Service implementation (ServImpl) variations model changes only to the internal service implementations, without requiring changes to their interfaces, and this is realized via implementation variation points. An example would be to modify the internal fraud checking method in *PotentialFraudCheck* service.
- Service Interface (ServIntf) variations model changes to the interfaces of the services, which will also require implementation changes – this is realized via specification variation points. An example would be to add the outputs custInfo and claimInfo to *DetermineLiability* service. This would also require concomitant ServImpl variations, as follows:
    - Input data received by the service – this could arise due to the following triggers. First, a change in the output data sent by a previously executed service, which is to be consumed by the service in question; second, a change in the input data needed by a service to be executed later - this data may have to be transmitted by the service in question, perhaps after modification.
    - Output data sent by the service - this would be a trigger for modifications to the services to be executed next, i.e., those that are dependent on the service in question.

Process-level variations are variations in the application flow of the business process. These are realized via combinations of ServImpl and ServIntf variations, and can be classified as follows:

- AddSeqSvc: a service S_j is added between S_i and S_k. If this does not cause any modifications to the inputs of S_k and outputs of S_i, then the output methods of S_i and input methods of S_k have to be redirected towards S_j – this can be realized as ServImpl variation, since this will involve modifying the input and output methods for the services S_i and S_k. However, if modifications are required, then this will be realized as a ServIntf variation on S_i and/or S_k, and needs to be modeled as such. An example is to add a suitably modified *PotentialFraudCheck* service between *DetermineLiability* service & *Liability+FraudChecks* service.

[1] Please note that our approach resembles inheritance-based variations from the object-oriented (OO) domain only in the implementation variation points; otherwise, the well-known "open to extension - closed to modification" principle prevalent in the OO domain does not apply for specification variation points.

- DeleteSvc: service S_{i+1} (predecessor is S_i and successor is S_{i+2}) is deleted. If this does not cause any modifications to the outputs of S_i or the inputs of S_{i+2}, then the output methods of S_i and input methods of S_{i+2} need to be redirected towards each other. However, if modifications are required, this will require ServIntf variation on S_i and S_{i+2}, and needs to be modeled as such.
- AddParSvc: Add service in parallel – a service S_j is added between S_i and S_k in parallel – this would require ServIntf variations to S_i and S_k. If this does not cause any modifications to the inputs of S_k and outputs of S_i, then additional methods would need to be added to each service to accommodate the new service S_i – this would be an ServImpl variation. However, if modifications are required, then this will be realized as ServIntf variations on S_i and S_k, and needs to be modeled as such.
- AddFlow: Add dependency between two services – akin to adding an edge in the business process – this will be a ServIntf variation, requiring interface changes to the services.
- DelFlow: Delete dependency between two services – akin to deleting an edge in the business process – this is similar to AddFlow, in that it would require a ServIntf variation.

4 Integrated VOSD Algorithm

Before we describe our algorithm, we first informally introduce a process and its associated services, before giving a formal definition. Briefly, a process is defined as consisting of four parts: (a) a set of associated service models, (b) data dependencies between the services based on the execution of preceding services (also called *produced data dependencies*), (c) data dependencies between the services based on the input model of preceding services (also called *received data dependencies*), and (d) control flow dependencies that provides the choreography of the services. Please note that this definition is quite realistic from a business perspective. Most business analysts in software organizations (who are not expected to possess programming expertise) would typically represent a business process as a collection of services with their respective inputs and outputs, and then augment it with control flow and data dependencies among pairs of services that do not share the predecessor/successor relationship. Therefore, such business analysts would also find it easy to distinguish between received and produced data dependencies, which is crucial for our algorithm (as we will see later in this section).

4.1 Formal Process and Service Model

We define a **process** $P = \{S, E, D, C\}$, wherein

$S = \{S_1.. S_n)$ is the set of services that participate in P

$S \subset U$, where U is the total portfolio of services (and their associated variations) for that particular domain of P.

$E = \forall ij \{E_{ij}\}$, {iff $S_i \xrightarrow{eij} S_j$ = true}, is the set of all *produced data dependencies* of service S_j with S_i, where $i \neq j$

E_{ij}, with the value of true, lets us know that the data produced by the execution of S_i, irrespective of S_i's input is part of the data d_{ij} that needs to be passed from S_i to S_j.

$D = \forall ij \{D_{ij}\}$, {iff $S_i \xrightarrow{dij} S_j$ = true }, is the set of all *received data dependencies* of service S_j on S_i, where $i \neq j$. Here Service S_i needs to pass its input data *without modifications* to Service S_j. This is represented by D_{ij} with the value of true.

The distinction between produced and received data dependencies is *absolutely crucial*, since this will help in the case of removal of S_i from the process flow P or sequential inclusion of a new Service S_x between S_i and S_j (i.e., DelSvc- and AddSeqSvc-type variations – see Section 3.2), as we will illustrate later in this Section with the help of the running example of Section 2.

$C = \forall ij \{C_{ij}\}$, {iff Si $\xrightarrow{Cij}$ S_j = true}, where $i \neq j$, is the set of *control flow* dependencies between S_i and S_j, where C_{ij} is either true or false, based on whether S_i controls the execution of S_j, i.e., iff Si precedes S_j in terms of control flow.

Only with the value of C_{ij} being true, can we know that a service S_i precedes the Service S_j in the business process. Only with this information, can we expect Service S_i to pass on the received data D_j for Service S_j from Service S_j's predecessor services. On the other hand, C_{ij} having a value of false means S_j would execute in parallel with respect to S_j and S_i is not expected to carry any additional data for S_j.

We also define a **service** S via its inputs and output sets respectively, i.e.

$S = \{D_{in}, D_{out}\}$, where $\{D_{in}\}$ = Set of input Data required for invoking S, and $\{D_{out}\}$ = Set of Output Data expected after invoking S.

To illustrate via our running example, from Figure 2, let Start Service = S0, which is the initial service that provides all the inputs for subsequent services; let *DetermineLiability* = S1, *PotentialFraudCheck* = S2, *ClaimInvestigation* = S3, *Liability+FraudChecks* = S4. The service definitions for our running example are listed here below.

$S0 = \{D_{in} = D_{out} = \{CustInfo, ClaimInfo\}\}$
$S1 = \{D_{in} = \{CustInfo, ClaimInfo\}, D_{out} = \{LiabilityInfo\}\}$
$S2 = \{D_{in} = \{CustInfo, ClaimInfo\}, D_{out} = \{Result\}\}$
$S3 = \{D_{in} = \{LiabilityInfo, Result\}, D_{out} = \{LiabilityInfo, Result\}\}$

Let us consider Figure 4 (for original process) and Figure 5 (for variant process) and only those services highlighted in those two diagrams for illustration. Let us refer to the processes, respectively, as P_{old} and P_{new}. Hence the process in Figure 4 can be formally modeled as:

P_{old} = {S, E, D, C}
Where S = {S0, S1, S2, S3) //used to derive new services or contains variants
E = $\{e_{13},e_{23}\}$, where $e_{13}= e_{23}$ = true
D = $\{d_{01},d_{02}, \}$, where $d_{01}=d_{02}$ = {Custinfo, Claiminfo}
C = $\{c_{01},c_{02},c_{13},c_{23}\}$
$c_{01} = c_{02} = c_{13} = c_{13}$ = true

With respect to D in the original process (i.e., received data dependencies), Services S1 and S2 are dependent on S0 via received data and control dependencies (i.e., S1 and S2 can execute only upon successful execution of S0). Similarly Service S3 is dependent on both S1 and S2 via received data, produced data and control dependencies.

Now let us look at the process depicted in Figure 5, i.e., the variant process. It can be represented as below.

P_{new} = {S, E, D, C}
Here, S = {S0, S1, S2, S3, S4} // we need to derive or discover these
E = { e_{12}, e_{24},} Where e_{12}, e_{24}= true
D = $\{d_{04}, d_{41},d_{12},d_{24},d_{43}\}$
$d_{04} = d_{41}$ ={custinfo, claiminfo}
d_{12} = {custinfo,Claiminfo,liabilityinfo}
$d_{24} = d_{43}$ = {liabilityinfo, result}
Similarly C = $\{ c_{04},c_{41},c_{12},c_{24},c_{43}\}$
Where $c_{04},c_{41},c_{12},c_{24},c_{43}$ = true

4.2 Algorithm Description

We assume the following inputs to our VOSD algorithm: New Process Model = NP, Functionally closer, existing process Model from portfolio = OP, Associated set of services for OP from the portfolio = S, set of process tasks mapped from NP to services in S = S_{mod}, set of process tasks *not* mapped from NP = S_{new}, and finally, variation models for all the services in the portfolio (modeled as per VOE principles from Section 3.2).

Before we present the algorithm, however, we first describe a function called **Matching** which implements service variant discovery from existing service variants in the portfolio. This function will be invoked by the VOSD algorithm as its first step. The inputs to **Matching** are: (i) the selected task in NP for which the appropriate service variant needs to be determined, (ii) a selected service variant S' in OP (which belongs to the set S_{mod}) which needs to be matched against (i). The **Matching** function returns one of the following outputs: either (a) FALSE, i.e., no match occurs; or (b) TRUE, along with the appropriate (variation point, variation feature) pair of the selected service variant.

```
BEGIN Matching
Variation Model Vm ← new VariationModel (S');
//Lists all variation points and variation features applicable to S'
For each Variation Point
{
        For each Variation Feature applicable at that Variation Point
        {
                Vm.getVariant(VariationPoint);
//Choose the variation point corresponding to the service variant.
                If (No such variation point exists)
                {
                        return ("FALSE");
                        exit(-1);
                }
                Vm.getVariantFeature(VariationPoint, Variant);
// Check whether there is an actual service variant available in the portfolio of reusable assets. If so, this function returns the actual variation feature, or exits with failure.
                If (No such variant is available)
                {
                        return ("FALSE");
                        exit(-1);
                }
return (ServiceVariant, VariationPoint, VariationFeature);
//Finally returns the service variant along with its variation point and variation feature
END Matching
```

Algorithm: VOSD

1. Pick a task from NP. Verify and compare the input and outputs for it and its corresponding identified Service in S_{mod}
2. If **Matching,** go to Step 1 for next task. If this is the Last Task then Exit. Otherwise, go to Step 3.
3. Get the service Variation Model. Find the associated Variation Points and Variants for that Service based on the conflict.
    a. Ensure the service is declared Variant (VP exists)
    b. Ensure the associated Operation is declared Variant (VP exists).
    c. If neither a nor b occurs, exit with message ("Variation Failed"). If no Variation model available (this will be the case for new services in Snew), go to step 5.
4. Select the Variant from the selected list in step 3, that obeys the following properties.
    a. it addresses the change identified from NP
    b. it matches on the listed Variation Features

    If Variant exists, return ("Variant Available") ; else go to step 5
5. Invoke **DeriveServices** **//**This is the service variant derivation step

At the end of execution of this algorithm, either we will get an implemented service variant from the portfolio or a skeleton variant related to an existing implementation through the algorithm **DeriveServices**.

Algorithm: DeriveServices
(i) Get P_{new},P_{old},
Service S1[] = P_{new}.getS();
// Abstract Services , links to S2.
Service S2[] = P_{old}.getS();
// Returns Actual Services
Service S_{new}[] = S2[] – S1[]
// Services to be created for P_{new}
Service S_{old}[] = S1[] $\bigcap$ S2[] // Services to be modified
(ii) createNewServices(P_{new}, S_{new})
(iii) deriveServiceVariants(P_{new},P_{old})
(iv) InstantiateServices(S_{new}[])

We will elaborate steps **(ii), (iii) and (iv)** of the **DeriveServices** algorithm here. **CreateNewServices()** first creates all the new services that are required in the new process. By creation of services, we mean a consolidated list of references with respect to other service's inputs and outputs. This approach is required, as at this point of time, we expect many of the services to go through a variation phase which will impact their input/output interface models to suit the new process. Hence once all the variations are identified for all the services, the input/output model can be used for actual instantiation of the corresponding services, which we will see later in this section. The **CreateNewServices()** algorithm is given below.

createNewServices(Process P_{new}, Service[] S_{new})
//Get Each S_{new}[i] for I = 1 .. n.,
// where n is the total no. of services
Service S= S_{new}[]. getNext() //
S = CreateService(S,P_{new})
CreateService(S,Pnew) //Creates new services
Service Ssource[] = P_{new},getE().getAllSource(S);
//returns all services, whose execution, Service S depends on
for all Services Ssource i = 1..n.
If(Ssource[i] ! in S_{old}[]) continue;
AddProducedinputs(S_{new}[i],Ssource[i])
Service Ssource[] = P_{new}.getD().getAllSource(S)
for all Services Ssource i = 1..n. {
If (Ssource[i] ! in S_{old}[]) continue;
AddReceivedinputs(S_{new}[i],Ssource[i])};

Now let us discuss the step **DeriveServiceVariants**, in which all the existing services of old process that are required in the new process are first verified for need of variation and appropriately their input/output model is defined accordingly. For this also, we consider the values of E and D of the process $P_{new.}$ One thing worth mentioning is, that an iteration on E, will affect only the dependent services, while an iteration on D can affect both the services corresponding to the variable $D_{ij.}$ This step is elaborated here.

```
    Int n = E_new.getSize();  Int j = D_new.getSize();
    For (int i = 0; i<n;i++)
    {
          Service S = E_new[i].getSource();
          Service t = E_new[i].getTarget();
          // Each value of Enew[i], represents a class that supports methods that returns the source
          and target services
          If (S_new.contains(t)) continue;  //Already created
          Else                              // ModifyService
          AddProducedInputs(t,s);
          // Add data that are produced by service s to the input list of service t
    }
    For (int i = 0; i<j;i++)
    {
          Service S =D_new[i].getSource();
          Service t = D_new[i].getTarget();
          If (S_new.contains(t)) continue;  //Already created
          Else                              // ModifyService
          AddReceivedInputs(t,s);
          // Add data that are received as Inputs by s to the input list of t and output list of d
}
```

The final step of the algorithm is InstantiateServices(), which actually instantiates the services of the new process P_{new}. This is illustrated below, along with two methods for adding received and produced inputs as per E and D dependencies, respectively.

```
    addreceivedinputs(t,s)
    {
          s.addoutputs(s.getinputs());
// This method  in  the source service Structure s - adds its  input data  into its Output Model.
          t.addinputs(s.getinputs(t));
// This method gets only those inputs that the target service t does  not have
// for example S4 just wants liabilityinfo from S2 and not Custinfo,cliaminfo
    }
    addproducedinputs(t,s)
    {
          t.addinputs(s.getoutputs(s));
// add only those data that are produced by s; for example liabilityinfo produced by S1 is
added but not custinfo and claiminfo.
          }
InstantiateServices(Service S_new[])
          {
// Here we actually derive service variants based on input output list consolidated with above steps
            int n = s2.getSize();
            for (int k = 0;, k <n; k++)
            {
               create(S2[k]);
                 // This method retrieves the Input and Output Model for each of the Service
                 S2[k] and instantiates the references based on other services
               }
}
```

5 Illustration on the Running Example

Referring to our running example, this Section describes a prototype implementation to the illustration of our algorithm. The prototype was implemented as a transformation plugin in IBM's Rational Software Architect v7.0.

Recall that we have declared earlier the following for the associated services for Verify claim: Start Service = S0, *DetermineLiability* = S1, *PotentialFraudCheck* = S2, *ClaimInvestigation* = S3, *Liability+FraudChecks* = S4. Let us first start with the illustration on Service S4. From step 1,3 and 5 in **VOSD** and from step **(i)** in **DeriveServices** , we know that S_{new} = {S4}. As we mentioned in Section 4, we can define S4 in terms of (D_{in}, D_{out})

With respect to Step **(ii)** in **DeriveServices** algorithm, this needs to be created based on the execution/data dependencies: E = { e_{12}, e_{24},} Where e_{12}, e_{24}= true. D = {d_{04}, d_{41},d_{12},d_{24},d_{43}}. d_{04} = d_{41} ={custinfo, claiminfo}. d_{12} = {custinfo,Claiminfo,liabilityinfo}. d_{24} = d_{43} = {liabilityinfo, result}. Hence for S4 as target, the Set E of P_{new} provides S2, whose execution provides a part of the inputs for S4. The Set D provides two sets of received inputs from Services S0 and S2, which at this point are just existing services. Hence D_{in} **of S4 = (Received Input of S0, Received Input of S2, Processed data from S2 }**

As can be seen here, we do not actually store the values, but only the references of the above variables, as S0 and S2 are still to be modified for the new prcess flow P_{new} and the context of S4s creation is for P_{new} and not for P_{old}. But as one completes the algorithm and instantiates S4, it will contain the following: D_{in} of S4 = {**Custinfo, Claiminfo, Liability Info, Result**} that is instantiated from the above equation. Similarly, one can derive the following: D_{out} of S4 = {**Custinfo, Claiminfo, Liability Info, Result**}. The first two data derived from d_{41} and last two data derived from $d_{43.}$ With this definition, P_{old} consists of the following implemented services.

S0 = {Min, Mout, Mproc}, where Min= Mout = {SetCustinfo(), SetClaimInfo()}, and Mproc = null

S1 = {Min,Mout,Mproc}, where Min = {SetCustinfo(). setClaimInfo()}, Mout = {getLiabilityInfo()}, and Mproc = {processLiabilityInf()}

S2 = {Min, Mout, Mproc}, where Min = {SetCustInfo(),SetClaimInfo()}, Mout = {getResult()}, and Mproc = {fraudcheck()}

S3 = {Min, Mout, Mproc}, where Min = {setliabilityInfo(),SetResult()}, Mout = {getLiabilityInfo(), GetResult()}, and Mproc = {investigateClaim()}

The available Variant Model for the *LiabilityInfo* Service in P_{old} in the prototype implementation is depicted in Figure 7. From step 2 of the **VOSD** algorithm, Service S2 is declared as a variant. From Step 3 of the algorithm, we can see from Figure 7 that the available variation feature matches with the expected output data model that supports creation of two new methods, with the names of the operations and their associated parameter values confirming the matching. Now, in the case of Service S3, from step 3 of the **VOSD** algorithm and this variation model, we could see that there is no variation feature available. (In practical cases, there is possibility that the available variation features even if any, may not match the expected variant required for

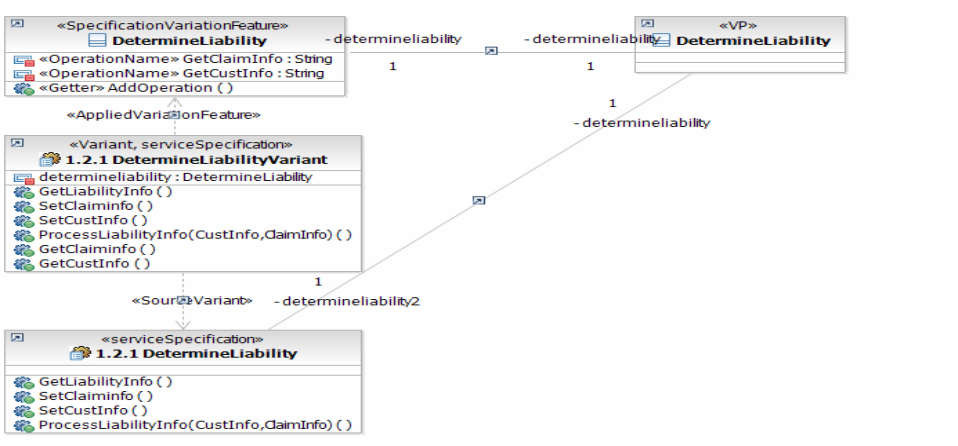

Fig. 7. Variant Model for P_{old}

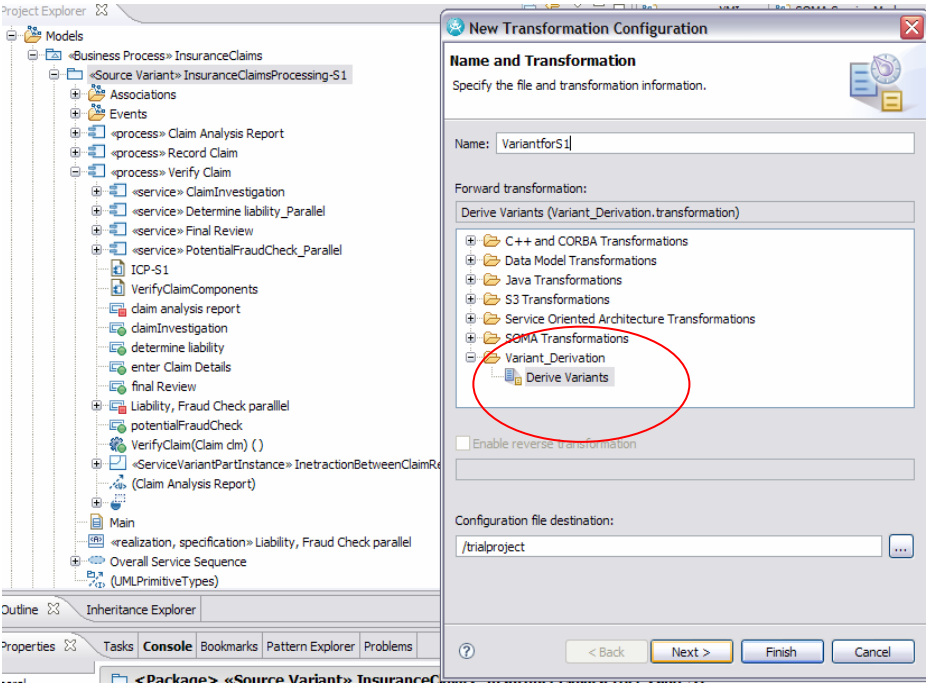

Fig. 8. Transformation Plugin

P_{new}) Hence from step 4 of the algorithm, we invoke the **DerviceServices** algorithm and derive the new variant for S3.

Our transformation plugin is depicted in Figure 8, where the user can invoke the service derivation transformation, and subsequently provide the process model P_{new} as the input to the transformation.

We summarize the services for P_{new}, below, with variations listed in ***bold-italic*** font:

S0 = {Min, Mout, Mproc}; where Min = Mout = {SetCustinfo(), SetClaimInfo()} and Mproc = null

S1 = {Min,Mout,Mproc}, where Min = {SetCustinfo(). setClaimInfo()}, Mout = {getLiabilityInfo(),***getCustinfo()***,***GetClaiminfo***}, and Mproc = {processLiabilityInf()}

S2={Min,Mout,Mproc}, where Min = {SetCustInfo(),SetClaimInfo(),***setLiabilityInfo()***}, Mout = {***getLiabilityInfo()***,,getResult()}, and Mproc = {***fraudcheck()***}

S3 = {Min, Mout, Mproc}, where Min = {setliabilityInfo(),SetResult()}, Mout = {getLiabilityInfo(), GetResult()}, and Mproc = {investigateClaim()}

Figure 9 depicts the output of our integrated VOSD algorithm, displaying P_{old} and P_{new}, with P_{new} (circled in red) named as a variant of P_{old}.

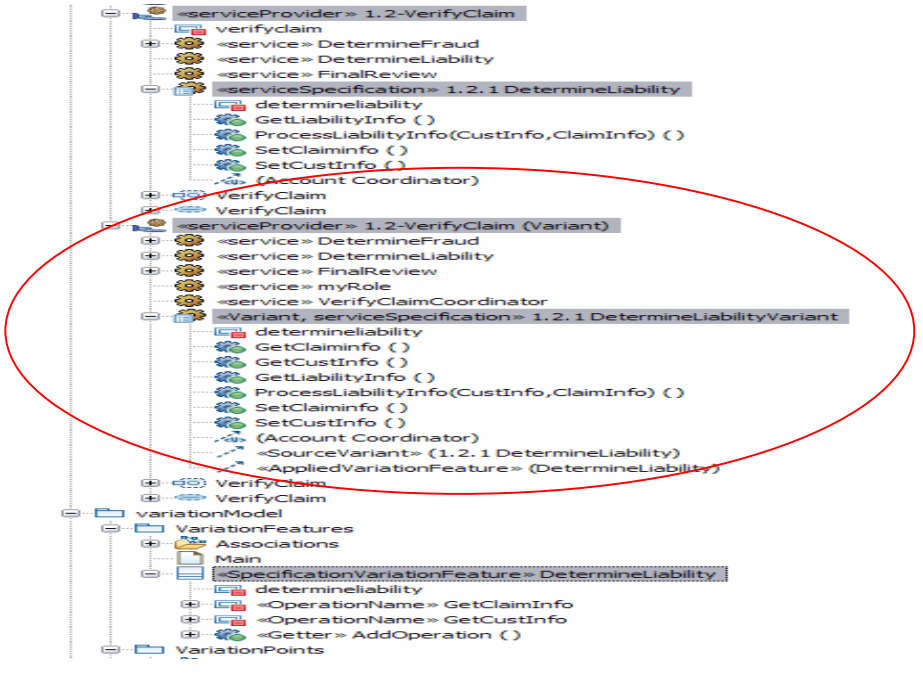

Fig. 9. Output of Integrated VOSD Algorithm

6 Related Work

The importance of a systematic approach towards service identification from business process specifications has been highlighted in [8], which bolsters the need for our VOSD algorithm. One of the early formalizations of the variation-oriented analysis and design (VOAD) principle can be found in [1]. Our paper incorporates and extends

these ideas by applying it to the specific case of business process-based solutions and deriving associated service variations. Another relevant paper is [4], which presents a design method called Grammar-Oriented Object Design (GOOD), which provides a means for the business analyst to declaratively specify the requirements of a dynamically reconfigurable software architecture driven by business-process architectures. Another approach similar to ours is presented in [2]; however, that paper primarily concentrates on detailing various ServImpl-type variations expressed via the object-oriented concept of inheritance (i.e., interfaces are considered to be invariant). In contrast, our approach uses SOA principles to provide a far greater range of variability, as already explained earlier in Section 3. In a similar vein, IBM's Model-driven Business Transformation (MDBT) approach [6] proposes a model-driven approach, based on OMG's MDA (http://www.omg.org/mda/) approach, for transforming business requirements into IT solutions. Our approach is similar, but we focus on transforming business processes into their constituent service specifications.

7 Conclusions and Future Work

In this paper, we have addressed the crucial problem of rapidly comparing an extended template business process (defined for a different context; defined for different customer requirements) with an existing business process that is already implemented as a choreography of collection of services; thereby enhancing reuse of these services by appropriate modification and thereby gaining flexibility with minimal additional effort. As part of this effort, we have described our integrated algorithm for discovering (from a portfolio of reusable assets) and deriving service variants for stated business process specifications, which we have called Variation-Oriented Service Design (VOSD). Via IBM's RSA Modeling Tool, we have also demonstrated our algorithm on a realistic running example in the insurance domain. Our prototype implementation proves the practical usefulness of our algorithm; hence our primary future work is to implement our algorithm along with our industry partners on real-life customer engagements.

References

[1] Arsanjani, A., Zedan, H., Alpigini, J.: Externalizing Component Manners to Achieve Greater Maintainability through a Highly Reconfigurable Architectural Style. In: Proceedings of International Conference on Software Maintenance (ICSM) 2002. IEEE Computer Society, Los Alamitos (2002)

[2] Schneiders, Puhlmann, F.: Variability Mechanisms in E-Business Process Families. In: Proceedings of Business Information Systems, BIS 2006 (2006)

[3] Narendra, N.C., Ponnalagu, K., Srivastava, B., Banavar, G.S.: Variation-Oriented Engineering (VOE): Enhancing Reusability of SOA-based Solutions. In: Proceedings of SCC 2008. IEEE Computer Society, Los Alamitos (to appear, 2008)

[4] Arsanjani, A.: Empowering the Business Analyst for On Demand Computing. IBM Systems Journal 44(1) (2005)

[5] Singh, M.P., Huhns, M.N.: Service Oriented Computing, 1st edn. Wiley-VCH Publishers, Chichester (2004)

[6] Kumaran, S.: Model-driven Enterprise. In: Proceedings of the Global EAI (Enterprise Application Integration) Summit, pp. 166–180 (2004)
[7] Ponnalagu, K.: Deriving service variants from business process specifications. In: Proceedings of ACM Compute (2008), http://portal.acm.org/ft_gateway.cfm?id=1341776&type=pdf&coll=&dl=GUIDE&CFID=25839879&CFTOKEN=86101169
[8] Hubbers, J.-W., Ligthart, A., Terlouw, L.: Ten Ways to Identify Services. SOA Magazine (accessed May 21, 2008), http://www.soamag.com/I13/1207-1.asp