A Service Pattern Model and its Instantiation Using Rule-Based Reasoning

Chien-Hsiang Lee¹, San-Yih Hwang¹, I-Ling Yen²

¹Department of Information Management, National Sun Yat-sen University
Kaohsiung, Taiwan 80424, syhwang@mis.nsysu.edu.tw

²Computer Science Department, University of Texas at Dallas
Richardson, United States, ilyen@utdallas.edu

ABSTRACT

Although reuse is the main goal of SOA, composing existing services to realize different user requirements is still a difficult and time-consuming task. There have been research works on workflow templates and design patterns that facilitates reuse and assist with the process composition task. Nevertheless, existing workflow templates are too specific, whereas design patterns may be too abstract; their effectiveness in assisting the composition process may be limited. In this paper, we propose a comprehensive service pattern model that is more flexible than existing workflow/service templates while allowing systematic instantiation into the concrete workflows. The service pattern model supports adaptive control flow and incorporates abstract services and prospect services (similar to a node in a design pattern but with formal partial definitions). We also design a reasoning process to automatically instantiate the service pattern and generate the desired workflow by following the rules of data type compatibility and precondition-effect matching. Service pattern model enables the integration of flexible pattern-based reuse and efficient and automated generation of workflows. Experimental study shows that the automated reasoning process is very efficient due to the use of service patterns.
1. Introduction

Web services technology supports lightweight integration of applications by integrating multiple services into a composite service (or a workflow) using the service-oriented architecture (SOA) paradigm [10]. Business process modeling standards, such as BPMN and BPEL which support the specification of service compositions with complex workflows, have been well established. However, composing services into workflows to realize the desired functionality of an application system is still a challenging task. In practice, there have been CASE tools that help developers build service compositions [16, 31, 37], but business process analysts still have to manually plan the workflow and then consult IT staff to deploy workflow-based Web service composition, which demands considerable efforts.

To reduce the effort of service composition, reasoning approaches, such as planning and logical reasoning [8, 23, 34, 38, 47], have been adopted to automatically compose service operations to satisfy a given goal. However, due to the high complexity pertaining to the service composition process and the large number of available services, it is difficult to reason in a realistic environment without additional assistance.

Another approach that has been investigated in the literature to help with service composition is the use of templates or patterns. Generally the workflow templates are in a form of abstract workflows, where services in the workflow can be concrete or abstract. An abstract service has well defined I/O, precondition, and effects. Some works consider QoS-driven service composition based on workflow templates, where different concrete services are selected to ground the abstract services to satisfy different QoS goals or request-specific constraints [11, 13]. These workflows are adaptable in QoS but are still very specific and lack of flexibility in functionality. There is no systematic method to adapt the functionality of workflow patterns.

Design patterns have been widely investigated in software development and the approach is also applicable to service composition. Benetallah et al. proposed several patterns to solve specific technical problems in service composition, such as cross-enterprise data exchange and service discovery [4, 5]. However, design patterns are frequently defined at a high level. Manual efforts are still needed to compose services. Moreover, the human composer needs to have a good understanding of the design patterns and the existing services in order to effectively perform the composition tasks. It is especially difficult when it comes to integrating automated composition tools with reuse techniques to offer augmented help for service composition.

In this paper, we propose a meta-model, called service pattern model, to ease service composition and promote better reuse. The major task is to allow a service pattern be instantiated...
into different functional goals. In order to achieve this objective, a service pattern should support adaptive flow structure. Also, it should be possible for a component service in the workflow to have flexible service specifications, i.e., a service node in a service pattern should be able to be grounded into different concrete services with different functionalities (and/or different QoS). Consequently, a service pattern can have different inputs and outputs and different preconditions and effects.

Based on the concept discussed above, we develop a suite of related techniques as described in the following:

- We extend OWL-S model and formally define the service pattern model. A service pattern achieves flexibility by defining a set of instantiation parameters, which could be flow control parameters, data type parameters, and predicate parameters. The flow control parameters can be configured to generate different workflows with different flow structures. The data type parameters can be flexibly instantiated into different data types. The predicate parameters can be used in the definitions of preconditions and effects. The data type parameters and predicate parameters can be used to define the input/output data types and precondition and effects of the service nodes (refer to as the prospect services) in the workflow. Our service pattern model addresses the functional inflexibility problem in existing workflow template works.

- A systematic instantiation process is outlined in the paper. A designer can step by step define the instantiation parameters for the target application system without needing to fully understanding the internals of the pattern. This instantiation process is much simpler than the instantiation of design patterns.

- We integrate the rule-based reasoning technique into the service pattern model to automate the instantiation, service selection, and grounding process. A set of interference rules are defined to facilitate reasoning. This further reduces the efforts for service composition, which is not possible when using design patterns. Also, due to the use of service patterns, the problem space for reasoning is highly confined, making reasoning a feasible and efficient solution in the composition process.

The rest of this paper is structured as follows. In Section 2, we review previous research works related to automated Web service composition and flexible service composition. We then describe our proposed model and illustrate service pattern example in Section 3. Next, we discuss the design of the automated reasoning process for deducing feasible assignment of instantiation parameters and discuss experimental results in Sections 4 and 5, respectively.
2. Literature Review

This paper is aimed at providing flexible Web service composition solution with considering message mismatch issue in order to improve flexibility of Web service composition and reusability of Web service. To analysis the gap between our solution and previous researches, we survey existing literature regarding automated Web service composition to identify the limitations of previous research to flexible Web service composition. Then, the researches about workflow template as well as design patterns and variability management in SOA are also studied to clarify the current research state of reusing Web service composition and modeling function variability in SOA. Although these researches make existing Web service composition be adaptable to changing situations, particular types of Web services and confined range of application domains preclude them improving flexibility of Web service composition for a variety of requirements. Finally, we also collect the papers regarding message mismatch resoution between Web services and summarize them to articulate the requirement of message mismatch resolution between Web services, state-of-the-art techniques, and the gap between our solution and previous solutions. In the following sections, the related works will be described in above sequence.

2.1 Automated Web service composition

The benefits of SOA can be realized by Web service composition techniques. In academic and practice, there have been many Web service composition techniques proposed from different perspectives, such as composition validation and correction, composition enforcement, and automated service composition. Theoretical process meta-models, such as Petri net, finite state machine, π-calculus, and linear temporal logic, describe the invocation order in a process of service composition, which enables subsequent analysis and validation of the system [6, 12, 22, 27, 32, 40, 41]. The emergence of the process language WS-BPEL has allowed for orchestrating the cooperation of component services in service composition, and the plan can be executed in several commercial or open source WS-BPEL engines, e.g., ActiveBPEL or ODE.

To reduce the human efforts on composing Web services, there also have been many works focusing on how to automatically compose Web services to achieve a given goal. Research in this line treats automatic Web service composition as a planning problem. Several planning techniques are proposed, such as rule-based planning [24, 36], hierarchical task network planning [38], planning based on model checking [34], and planning based on Markov decision processes [8, 9]. However, due to the high complexity pertaining to the service composition process and the often large number of available services, it is difficult to reason in a realistic
environment without additional assistance. In addition, I/O messages of Web services in these works are often considered as variables in states or predicates and used for determining compatible Web services; yet the dependency between messages are seldom considered.

On the other hand, some works focus on data dependency between services for composing services [14, 21, 47]. Liang and Su [21] use an AND/OR graph to represent dependency among data and operation nodes, which originate from Web services in some service categories. Accordingly, a service dependency graph is built to show all possible input-output dependencies among Web services. A Web service composition, a complete solution subgraph of the AND/OR graph with minimum cost under a given goal, can be derived by using bottom-up search strategy. Gu et al. [14] extend their work by distinguishing the instance level from the abstract level in an enhanced service dependency graph. They also identify cardinality relationships of messages exchanged between two operations and suggest using XSLT or XPath for the attribute transformation. Zeng et al. [47] define three rules, forward-chaining, backward-chaining, and data flow, to indicate the preconditions of a task, the effects after executing it, and the data dependency among tasks, respectively. Their rules-inference approach constructs qualified execution plans and measures weighted QoS scores of the plans to identify the best plan. However, above studies ignore the message mismatch issue, which is important for Web service composition in the real world because Web services are often developed by different providers without common standards for defining exchanged messages.

### 2.2 Flexible service composition and service design patterns

Another approach to helping with service composition is to use templates or patterns. Generally the workflow templates (or abstract process) are in a form of abstract workflows, where services in the workflow can be concrete or abstract. An abstract service has well defined I/O, precondition, and effects without specifying grounding information. The QoS-driven service composition works are based on workflow templates, where different concrete services are selected to ground the abstract services to satisfy different QoS goals [2], preference [39] or request-specific constraints [11, 13]. Workflow templates allow for adaptable concrete Web services to achieve QoS goals but are still very specific and lack of flexibility in functionality.

Design patterns have been widely investigated in software development and can be applied to service composition. From extant literature, Zdun et al. classify design patterns that can be applied to SOA into different abstraction layers of message processing architecture, from service communication layer to composition layer [45]. For example, on composition layer, Enterprise Service Bus (ESB) is an architectural pattern used for designing and implementing the interaction and communication between mutually interacting services. They then formally define
design patterns used on composition layers to improve communication and understanding of patterns by adopting activity diagrams in UML and Object Constraint Language (OCL) [46]. Benetallah et al. identify several design patterns to address specific technical problems in the life cycle of activities related to composite services, such as wrapping native services, assembling composite services, and monitoring service executions [4, 5]. For instance, service wrapper pattern is applied to wrapping native services. It includes four elements: communication manager that supports the translation of messages between heterogeneous communication protocols; security manager that handles security issues including authentication and authorization; content manager that is in charge of the conversion between different internal and external document format; and, conversation manager that is concerned with the conversational interactions among Web services. However, design patterns are frequently defined at a high level. Dramatic manual efforts are still needed to compose services. Moreover, the human composer needs to have a good understanding of the design patterns and the existing services in order to effectively perform the composition tasks. It is especially difficult when it comes to integrating automated composition tools with reuse techniques to offer augmented help for service composition.

Based on the idea of variability management in software product line domain [35], some recent works propose to accommodate several variation points in the workflow of composite Web service, and these variation points allow users to adapt workflow or replace services to build customized composite Web services. In addition to common functionality, the customized composite services provide assorted services for different users. Therefore, variability modeling is crucial in this area and variability types differ at different abstraction levels. From the feature level, customizable function items for different user requirements are major consideration and they are categorized as common, optional, and alternative features. Common features are required function items for all users, optional features were either included or excluded, and alternative features have several variants for choice [1, 30]. Abu-Matar and Gomaa adopt 4+1 views, which include service contract, business process, service interface, and service coordination views for representing specification and cooperation of services, and feature view for presenting the functionality of composite service, to model SOA systems as service families [1]. On the other hand, Nguyen and Colman focus on the customization framework in which the feature model including variation points and their variants links to service model and process model templates. The customized models are specialized by removing elements in model templates, which are associated with unselected features [30].

To represent flexible workflow, some works consider variability at the model level to instantiate assorted workflows from a workflow template [15, 25]. In addition to feature variability mentioned before, Mietzner and Leymann model variation point dependency by
which the available alternative variants of target variation point are bound by the relationship between target and its source variation point. They also separate variability descriptor from model templates, so it can be independent from the type of application documents. Finally, a customization process was proposed to generate customized workflow of composite Web service in WS-BPEL by referring to variability descriptor in XML document and users’ choice for each variation point [25]. Hadaytullah et al. proposed another variability type, flow order, to accommodate different requirements of service invoking order in a workflow. They adopt activity diagram to model workflow of composite service and incorporate optional variant, alternative variant, and alternative flow order. In particular, an optional variant is represented by a service role whose cardinality is 0..1, and XOR operator can be used to describe alternative service list and alternative execution sequence list [15]. In contrast, Narendra et al. model the adaptation of variation points using primitive operators: add and delete. These operators are applied in service and process level to describe variation actions for a variant service derived from an initial service [28, 29]. Although the works of Narendra et al. provide more flexibility in adaptation, the user has to be familiar with the structure of model, which is akin to software framework in object-oriented systems (e.g. OWL or MFC). More importantly, while previous works about variability modeling in service computing consider different types of variability, there are some limitations. First, the reuse of existing workflow presented in previous works is confined to component services having concrete interface. Indeed, the improvement of flexibility of Web service composition not only requires that flexibly compose concrete services but also needs the ability to change service interface for responding dynamic market places [17]. Second, they only model variant points with different types and consume human efforts to select variants. Finally, validation of customized process is not considered in the model, and message mismatch between services in customized process is still not well addressed.

3. Service Pattern Model

A service pattern is intended to encompass a variety of applications to facilitate reuse and enable users to flexibly build the workflow-based Web service composition. A workflow-based Web service composition involves a number of cooperative Web services, data exchanges between services, and a variety of control flows. A service pattern can be regarded as a general workflow, from which different workflow-based Web service compositions can be derived. Similar to design patterns, service patterns can assist with service composition to meet different requirements. As a result, the reusability can be extended to the composition level, not just the services level, and the applicability of service compositions can be broadened.
Some components in a service pattern are well-defined, while others are only partially specified and need to be provided during instantiation time. The well-defined part of a service pattern contains concrete services, abstract services (i.e., services with concrete interface yet no implementation), and well-established workflow prescribing legal execution order of services. These components comprise the common functions of different applications; in contrast, the partially specified components in a service pattern include prospect services (to be explained later), adaptable workflow template, and instantiation parameters. While the common functions can be realized by the well-defined part of a service pattern, the special requirements for different applications are embodied by prospect services, whose interfaces, including input, output, precondition, and effect (IOPE), are only partially specified via instantiation parameters. During instantiation time, the instantiation parameters are decided and the IOPE specifications of prospect services become clear, and existing Web services can be selected to realize the specific functions. In addition, different applications may have different arrangement on the same set of services in their workflows, e.g., skipping services that are not needed in the application, reverse service order, etc. A service pattern therefore also provides adaptable control flows, where the actual flow is decided during instantiation to allow customized service composition.

In the following sections, at first, we will introduce three examples to illustrate the design of service patterns, followed by the formal definition of service patterns. Finally, four examples of instantiating a service pattern are described.

### 3.1 The Motivating Examples

Consider the following three typical workflows that are commonly used in business organizations [20]. The replenishment workflow (RPW) computes the quantity of products to be ordered based on the sales quantity, stock quantity, and safety stock. The profit computation workflow (PCW) calculates net profit from sales amount and store expenses in a given time period. Also, an order fulfillment workflow (OFW) prepares and delivers ordered products to the customer, and charges her for the sales transaction. The three workflows are shown in Figure 3-1, in which data manipulation operations, as introduced in [19], are represented by pure square boxes, and services are enclosed in square boxes with double gears.

All of the three workflows, shown in Figure 3-1, have sales data and product profile as inputs and process them in a similar pattern. First, they compute the joined table. Then, they filter the data using different selection criteria (by product category in RPW and by transaction state in PCW) and/or perform data conversion (ConvertByUnit in RPW and ConvertByCurrency in PCW). Second, RPW and PCW aggregate sales in multi-dimensions (AggregateSales). Third, they obtain data from external sources and combine them into the data processed so far.
(CombineExtendedData). On the other hand, OFW simply uses original sales data processed so far. In the final stage, three workflows seem to have different tasks, but they still have common processes: accepting partial sales data (e.g. sales amount and quantity), generating new attributes (e.g. order quantity in RPW and store profit in PCW), and aggregating original attribute (e.g. total up sales amount for each customer in MakePayment service of OFW).

Because there are common features and variant flows between the workflows in Figure 3-1, a general workflow can be derived from them as shown in Figure 3-2, which consists of four stages. In the first stage, the workflow includes obtaining master data, namely sales of multiple stores and product profile, and a data join node to merge the data. Then, we represent the data filtering task by a selection data service with the filtering conditions as a configuration parameter. Above tasks are common among the three example workflows. In the second stage, ConvertByCurrency and ConvertByUnit are the main data conversion tasks needed for many global corporations which possess multiple currencies or multiple measuring units. Instead of having a very general data conversion service, we specifically provide two choices and use a flow control parameter to allow the developers to make flow selections. Both choices can be skipped, as in the case of OFW. Similarly, in the third stage, the developers can choose one from the two possible execution sequences, (AggregateSales, CombineExtendedData, ExtractData) or (ExtractData) only, where the former will be chosen by RPW and PCW, and the later is selected by OFW. Note that these services may accept different input to achieve different purposes after instantiation. For example, AggregateSales aggregates sales attributes in multiple dimensions according to the selected attributes and dimensions specified by developers. In RPW, the task, AggregateSales, can be realized by this common service through specifying a total attribute (i.e., sales quantity) and two dimensions, namely SKU and store. Similarly, CombineExtendedData provides customizable data type for combining sales data with other extended data, such as inventory in RPW and store expenses in PCW; ExtractData prepares a request message from the output of the previous service for invoking the next service according to some selective attributes, e.g., store sales amount and expenses in PCW. In the final stage, the data obtained from preceding services will be processed to produce final result. Because three workflows support different functionalities, their final services are different from each other. Therefore, the workflow contains an exclusive choice to select adequate services according to functionality.
Figure 3-1. Three workflow-based service compositions
Figure 3-2. The general workflow for three workflow-based Web service compositions in Figure 3-1.

Although the general workflow in Figure 3-2 can realize any of the three service compositions by overriding data manipulation operations and supplying appropriate input to services, it complicates the design of workflow and causes worse performance because many judgments have to be made at runtime. To maintain the design flexibility and improve the execution performance, we proposed service pattern model to substitute the general workflow as the one shown in Figure 3-2 and provide instantiation parameters to allow dynamic service composition. We demonstrate the requirements of modeling flexible workflow in the following paragraph.
There are several variation points in Figure 3-2 that have to be adequately modeled to provide workflow flexibility at design time, and they are shown in Figure 3-3. The first type of variation point is optional features, which indicate that some services in workflow may be either required or skipped for different requirements, such as variation point 1 in Figure 3-3. For example, `ConvertByCurrency` service is needed by PCW, but it is not required for RPW and OFW. The second type is flow variations by which there are different execution orders of services for different requirements. In Figure 3-3, variation points 2 and 3 belong to this kind of variations. For instance, PCW and RPW demand the execution sequence of `(AggregateSales, CombineExtendedData, ExtractData)`, whereas OFW only need `ExtractData` service. The third type of variation point is customizable services that support customizable data types of I/O parameters and preconditions/effects to accommodate different types of services for different requirements. In this paper, we call this kind of services as prospect services. Variation point 4 indicates a prospect service that derives new attributes from existing attributes, and variation point 5 shows a prospect service that aggregates multiple attributes in the table. The former will produce equal number of output records as its input, whereas the latter will produce less or equal number of output records than its input, as used at the service of `MakePayment` in OFW, where sales amounts of the same customer are added up. On the other hand, for the example of the former, the service of `Replenish` in RPW derives order quantity for each requested product from sales quantity, stock quantity, and safety stock, and is an instantiation of `Generate new attributes`. These two prospect services are of general purpose as they can take database tables of any type and perform the computation in row-based data aggregation or column-based data derivation, to restrict the range of service candidates to fit the concretized preconditions/effects. Furthermore, they provide customizable data types for I/O parameters and/or customizable predicates as their preconditions/effects to accommodate a variety of inputs and outputs.
Figure 3-3. The variation points of general workflow in Figure 3-2

3.2 The definition of service pattern model

Various standards, such as OWL-S and BPMN, have been developed for service and process specification. BPMN provides strong support in workflow specification, and OWL-S has a well defined upper ontology for service specifications. A service in OWL-S can be a concrete or an abstract one, where an abstract service can be grounded to different concrete services. A service can also be atomic or composite and a workflow can be specified as a process in a composite service. A service pattern goes beyond the concept of abstract services. As mentioned before, a service pattern should support the concepts of “prospect services” and an “adaptable workflow”. These service pattern elements have specific properties and can be controlled by instantiation parameters. We use a combination of OWL-S and BPMN models and some extended features to specify the service pattern examples. However, to avoid the unnecessary
details, we use some mathematical notations for service pattern related definitions. In the following two sections, we firstly introduce prospect service and adaptable workflow with their corresponding instantiation parameters. Finally, we formally define service pattern.

### 3.2.1 Prospect service

In OWL-S, input, output, precondition, and effects (IOPE) are defined in the upper ontology to describe the behavior of a service. A concrete or an abstract service has well defined IOPE. A prospect services is a partially-defined service with IOPE constraints, as opposed to abstract services that need to have full IOPE specifications. The differences among concrete services, abstract services, and prospect services are summarized in Table 3-1.

<table>
<thead>
<tr>
<th>Service</th>
<th>IOPE Specification</th>
<th>Grounding Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concrete service</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Abstract service</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Prospect service</td>
<td>Partial</td>
<td>No</td>
</tr>
</tbody>
</table>

A prospect service has customizable data types for I/O parameters and/or customizable predicates for preconditions/ effects in its non-specific specification. A customizable data type includes a data type parameter that is defined at instantiation time, while a customizable predicate is flexible and has a predicate parameter, whose value is determined during instantiation. Data type parameters may be used in the I/O specification of prospect services and the service pattern itself.

We call data type and predicate parameters the instantiation parameters because their values are assigned when service pattern is instantiated. In this section, a variable name with prefix “?” denotes an instantiation parameter. The following is an example of a prospect service.

Name: DeriveData
In: ?TypeOfDrvIn DrvIn
Out: ?TypeOfDrvOut DrvOut
Precondition:

\[
?\text{DeriveDataPrecond}(?:\text{TypeOfDrvIn: DrvIn})
\]

Effect:

\[
(\forall e_1 (e_1 \in \text{DrvOut} \Rightarrow \exists e_2 (e_2 \in \text{DrvIn}), \text{Key}(e_2) = \text{Key}(e_1) \land \\
\forall a \in \text{Attributes}(?:\text{TypeOfDrvOut}), \text{ComputeFrom}(e_1.a,e_2)) \land \\
(|\text{DrvIn}| = |\text{DrvOut}|) \land ?\text{DeriveDataEffect}(?:\text{TypeOfDrvOut: DrvOut})
\]

The input and output parameters of the prospect service DeriveData have customizable data types, ?TypeOfDrvIn and ?TypeOfDrvOut, respectively. There could be restrictions on customizable data types. We apply first-order logic to specify the restriction rules. Restrictions
on a data type parameter limit the actual values that can be bound during instantiation. In other words, a data type parameter may have to conform to a particular class of domain ontology or the output schema has a specific relationship with the input schema. For example, the following rules mandates that ?TypeOfDrvIn and ?TypeOfDrvOut must map to Table concept of domain ontology [3].

\[
\begin{align*}
\text{ModelRef}(?TypeofDrvIn) &= \text{Ont#Table} \\
\text{ModelRef}(?TypeofDrvOut) &= \text{Ont#Table}
\end{align*}
\]

On the other hand, predicate parameters can appear in the precondition and/or effect of a prospect service to provide flexibility while enforcing certain restrictions in service selection. For example, to fulfill the requirement of replenishment in B2B environment, developers may want to match prospect service DeriveData with the concrete service that not only derives order quantity but also automatically sends orders to B2B partners. Correspondingly, the developer can do the following assignment for predicate parameter \(?cpDeriveDataEffect\), which is the effect of DeriveData service.

\[
?cpDeriveDataEffect(?TypeOfDrvOut: DrvOut) = (\forall e_3 \in \text{DrvOut}, \text{Sent2Partner}(e_3))
\]

The effect of DeriveData service therefore includes (1) each output tuple must be computed from one input tuple, (2) each input tuple has a corresponding output tuple, and (3) each output tuple must be transmitted to a B2B partner.

### 3.2.2 Adaptable workflow template

An adaptable workflow template is based on workflow standards, such as BPMN, and extended by some adaptable control constructs for specifying flexible flow structures. There are many types of control flows described in the literature [42, 43]. To avoid reinventing the wheel, we do not assume any particular control flows, and any control flow can be potentially used in the proposed workflow template model. An adaptable workflow template can be considered as a generic workflow, from which an executable Workflow-based Web service composition can be realized by binding instantiation parameters defined in the service pattern. The basic blocks for constructing an adaptable workflow template are services. There are four kinds of services: concrete services, abstract services, prospect services, and data manipulation services. Data manipulation services are represented by WS-data expressions as described in [19], whereas the other kinds of services realize specific functionalities. After instantiating a service pattern, its prospect services will be concretized into abstract services, and its adaptable controls will be
specialized, and the final workflow will only include conventional control flows and become executable. For brevity, we only describe the extensions in the following.

There are two types of adaptable controls in an adaptable workflows template: the alternative selection and the skippable toggle. The definitions of the two types of parameterized control constructs are described as follows:

**Definition 4.1: Service skippable toggle**
A service skippable toggle is an ordered pair $<s, a_b>$, where $s$ is a service and $a_b$ is a workflow adapting parameter with Boolean type. A toggle $<s, \text{TRUE}>$ indicates that the service specified in the workflow template can be skipped at instantiation time.

**Definition 4.2: Alternative selection**
An alternative selection is an ordered pair $<a_e, SW>$, where $a_e$ is a workflow adapting parameter of an enumerated type and $SW$ is a set of sub-workflow alternatives, each of which is associated with a case value. For example, $<\text{?ChooseFormat}, \{(\text{‘Cube’, } sw_1), (\text{‘Flat’, } sw_2)\}>$ indicates that when instantiation parameter $\text{?ChooseFormat} = \text{‘Cube’ (‘Flat’)}$, only the subworkflow $sw_1 (sw_2)$ remains in the workflow.

The control flow in a workflow defines the execution paths, but not the message exchange between component services. Generally, the output of a preceding service is fed to the input of the succeeding service. However, a service may have inputs from different preceding services and it is necessary to have a clear definition of how the data flow, i.e., source parameter and destination parameter for a message exchange must be precisely connected. This is especially important in a service pattern to ensure the compatibility of the customizable data types. Due to the lack of such feature in existing models, we define a new construct, message alignment, to connect the I/O parameters of interacting services.

**Definition 4.3: Message alignment**
A message alignment is a 4-tuple, $<s_s, s_d, v_s, v_d>$, where $s_s$ and $s_d$ are source service and destination service respectively, and $v_s$ and $v_d$ are the formal parameters of source and destination services respectively.
3.2.3 Service pattern model

A service pattern is specified by an adaptive workflow template and flexible IOPE that are partially defined using instantiation parameters. To support instantiation, a service pattern also includes the specification of its instantiation parameters.

Definition 4.4: Service pattern
A service pattern is a 6-tuple, $< I, O, P, E, IP, W_T >$, where $I$ ($O$) is the set of input (output) parameters, each including a name and a data type, $P$ and $E$ are sets of preconditions and effects, respectively, $IP$ is a set of instantiation parameters, and $W_T$ is an adaptable workflow template.

We employ BPMN to model adaptable workflow template because it is prevail in business process modeling and provides the mapping from BPMN orchestration process models to WS-BPEL executable process models, where WS-BPEL is a de-facto standard for Web service composition. We also develop five new symbols for extended features of service pattern model. They are shown in Figure 3-4 and can be applied to BPMN process models to prescribe adaptable workflow template of service patterns.

![Symbols for extended features of service pattern model](image)

Figure 3-4. New symbols for extended features of service pattern model

An example service pattern, SalesCube, is shown in Figure 3-5. Note that SalesCube can be instantiated to any of the three example workflow in Figure 3-1. Its adaptable workflow template of SalesCube is shown in Figure 3-6 using the notation of BPMN with new symbols in Figure 3-4; and, the IOPE specification of involved prospect services are shown in Figure 3-7. In the
following, we use RPW as the example to demonstrate the assignment of instantiation parameters.

<table>
<thead>
<tr>
<th>Name: SalesCube pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose: To support the commercial requirements which base on sales data with multiple currencies and units considerations.</td>
</tr>
<tr>
<td>Context: The applications not only require sales data aggregated by multiple dimensions but also extend other data to derive outcome.</td>
</tr>
<tr>
<td>Category: Behavioral pattern, Sales data</td>
</tr>
<tr>
<td>IN: TStore:In_sl, TSalesClass:In_ea, TTimePeriod:In_tp, TMoney:In_cu, TSalesClass:In_ga, ?TypeOfEd In_ed</td>
</tr>
<tr>
<td>OUT: ?TypeOfOut Out</td>
</tr>
</tbody>
</table>

**Constraints:**
- Precondition: $|In Sl|\neq 0 \land Day(In \_tp) < \text{CurrentDate}()-7$
- Effect: $|Out|>0$

**Instantiation parameters:**

**Workflow adapting parameters:**
- ?SkipConvertByCurrency: Boolean;
- ?SkipConvertByUnit: Boolean;
- ?SkipDeriveData: Boolean;
- ?SkipAggregateData: Boolean;
- ?ChooseFormat: {'Cube', 'Flat'};
- ?ChooseComputeOrder: {'AggregateFirst', 'DeriveFirst'};

**Data type parameters:**
- ?TypeOfEd, ?TypeOfJedOut, ?TypeOfExtrIn, ?TypeOfExtrOut,
- ?TypeOfDrvIn, ?TypeOfDrvOut, ?TypeOfAggIn, ?TypeOfAggOut,
- ?TypeOfOut

**Restriction rules:**
- ModelRef(?TypeofEd)=Ont#Table,
- ModelRef(?TypeOfExtrOut)=Ont#Table,
- ModelRef(?TypeOfJed)=Ont#Table,
- ModelRef(?TypeOfJedOut)=Ont#Table,
- ModelRef(?TypeOfDrvIn)=Ont#Table,
- ModelRef(?TypeOfDrvOut)=Ont#Table,
- ModelRef(?TypeOfAggIn)=Ont#Table,
- ModelRef(?TypeOfAggOut)=Ont#Table,
- ModelRef(?TypeOfOut)=Ont#Table,
- KeyAttr(?TypeOfJedIn)=KeyAttr(?TypeOfed)=KeyAttr(?TypeOfJedOut), Attributes(?TypeOfJedOut) ⊆ Attributes(TSalesData) ∪ Attributes(?TypeOfEd)

**Predicate parameters:**
- ?fpSelectData(TSalesData:SelIn),
- ?cpDeriveDataPrecond(?TypeOfDrvIn: DrvIn),
- ?cpDeriveDataEffect(?TypeOfDrvOut: DrvOut),
- ?cpAggregateDataPrecond(?TypeOfAggIn: AggIn),
- ?cpAggregateDataEffect(?TypeOfAggOut: AggOut)

Figure 3-5. IOPE specification and instantiation parameters of SalesCube service pattern
Figure 3-6. The adaptable workflow template of SalesCube service pattern
**Name:** SalesCube pattern

**Prospect Services:**

<table>
<thead>
<tr>
<th>Name</th>
<th>In:</th>
<th>Out:</th>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtractData</td>
<td>?TypeOfExtrIn ExIn, TAttrNames ExAttr</td>
<td>?TypeOfExtrOut ExOut</td>
<td>ExtraAttr \subseteq Attributes(TSalesData)</td>
<td>ExtraAttr = Attributes(?TypeOfExtrOut) \land</td>
</tr>
<tr>
<td>JoinExtendedData</td>
<td>TSalesData JedIn, ?TypeOfEd JedEd</td>
<td>?TypeOfJedOut JedOut</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
| DeriveData          | ?TypeOfDrvIn DrvIn            | ?TypeOfDrvOut DrvOut        | \?cpDeriveDataPrecond(?TypeOfDrvIn: DrvIn)                                   | (\forall e_1 (e_1 \in DrvOut \Rightarrow \exists e_2 (e_2 \in DrvIn), Key(e_2) = Key(e_1), (\forall a \in Attributes(?TypeOfDrvOut), ComputeFrom(e_1.a, e_2.a)) )
|                     |                              |                             | \land |DrvIn|\land DrvOut\land \?cpDeriveDataEffect(?TypeOfDrvOut: DrvOut)                |
| AggregateData       | ?TypeOfAggIn AggIn            | ?TypeOfAggOut AggOut        | \?cpAggregateDataPrecond(?TypeOfAggIn: AggIn)                               | |AggOut|\leq |AggIn\land \?cpAggregateDataEffect(?TypeOfAggOut: AggOut) |

Figure 3-7. The IOPE specification of involved prospect services in SalesCube service pattern

*SalesCube* has two customizable data types for input parameter *In_ed* and output parameter *Out*. The former represents the type of the extended data that can be combined with sales data. The latter indicates the type of the desired output. For PRW in Figure 3-1(a), the extended data is product inventory whose type is TProdInv; the desired output is product order list of type TOrder. As a result, we can specify their bound instantiation parameters as follows:

?TypeOfEd=TProdInv;
?TypeOfOut=TOrder

Similarly, we can also specify data type parameters for prospect services in Figure 3-7 to concretize their specification. In Figure 3-6, there are four skippable services, namely *ConvertByCurrency, ConvertByUnit, DeriveData, and AggregateData*. The values of their binding instantiation parameters will affect the structure of workflow. For example, RPW does not use *ConvertByCurrency* and *AggregateData*. Also, it only requires the *AggregateData*
prospect service. Accordingly, the binding of instantiation parameters can be specified as follows:

```plaintext
?SkipConvertByCurrency=True;
?SkipConvertByUnit=False;
?SkipDeriveData=False;
?SkipAggregateData=True;
```

In addition, RPW has to aggregate sales by store and SKU and combine inventory information. In addition, it uses DeriveData service and does not need AggregateSales service. Therefore, the instantiation parameters of two alternative selections in Figure 3-5 can be specified as follows:

```plaintext
?ChooseFormat='Cube';
?ChooseComputeOrder='DeriveFirst'
```

Message alignments in Figure 3-6 are specified to facilitate compatibility validation of the connected parameters after their types are concretized. Consider mfi0 message alignment; it connects output parameter of ExtractData service to input parameter of DeriveData service. Thus, the data type of ExtractData’s output should be compatible with the data type of DeriveData’s input.

We extend the components of OWL-S to incorporate the proposed constructs, which resulting in the ontology shown in Figure 3-8. The prospect service is a subclass of process class, and both of alternative selection and service skippable toggle constructs are subclasses of control construct class. Furthermore, a service pattern is a subclass of composite process class with the invokable attribute being FALSE because it may contain prospect services and abstract services which have no grounding information.
Figure 3-8. Ontology of the extended workflow components for service pattern model
3.3 The instantiation of service pattern

The instantiation of service pattern includes four steps. The first step is to specify instantiation parameters according to the requirement of the application. Second, the workflow should be converted to the well-defined form by eliminating unselected alternative sub-flows and skipped services. Also, prospect services should be transformed into abstract services by instantiating customizable data types and predicate parameters. The third step is to align flow actual parameter with formal parameter, validate the instantiated data types against the constraints of the corresponding customizable data types, and resolve mismatches of data types between interacting services. Finally, same as the regular matchmaking process, the abstract services are matched with concrete instances to build an executable workflow-based Web service composition.

If a user can specify the customizable data types and predicate parameters in the first step, the following three steps can be automatically realized by using workflow adaptation [26], semantic type matching [18, 33], and service matchmaking approaches [7, 44]. However, users may only have an expected goal and do not know how to completely specify instantiation parameters. In this case, reasoning techniques such as deductive reasoners and planners can be used to automate the instantiation process. Compared to the case of constructing the entire workflow without any blueprint provided by the service pattern, the reasoning process with service pattern can be much more feasible and scalable. We developed the reasoning technique, which will be described in more detail in Section 4. In this section, we introduce the instantiation processes for three workflow-based Web service compositions based on the SalesCube service pattern to demonstrate the feasibility of the proposed model. Furthermore, to prove the flexibility of SalesCube pattern, we use a new business process to confirm that SalesCube pattern can be instantiated into new, and perhaps unexpected, customized processes with different functionalities.

3.3.1 The instantiation for stock replenishment

The instantiation parameters are specified according to the requirement of stock replenishment. Their values are assigned as shown in Figure 3-9.
Composition name: Stock replenishment

Instantiation parameters:
 Workflow adapting parameters:
  ?SkipConvertByCurrency=True;
  ?SkipConvertByUnit=False;
  ?SkipDeriveData=False;
  ?SkipAggregateData=True;
  ?ChooseFormat='Cube';
  ?ChooseComputeOrder='DeriveFirst';

Data type parameters:
  ?TypeOfEd=TProdInv;
  ?TypeOfJedOut=TSalesInvQty;
  ?TypeOfExtrIn=TSalesInvQty;
  ?TypeOfExtrOut=TReplenishReq;
  ?TypeOfDrvIn =TReplenishReq;
  ?TypeOfDrvOut=TOder;
  ?TypeOfAggIn=null;
  ?TypeOfAggOut=null;
  ?TypeOfOut=TOder;

Predicate parameters:
  ?fpSelectData= (Product.Category="LaptopComputer");
  ?cpDeriveDataPrecond=null;
  ?cpDeriveDataEffect=(\( \forall e \in DrvOut, Sent2Partner(e) \));
  ?cpAggregateDataPrecond=null;
  ?cpAggregateDataEffect=null;

Figure 3-9. The assignment of instantiation parameters for stock replenishment

After applying workflow adapting parameters to alternative selection and service skippable toggle constructs, the workflow is tailored as shown in Figure 3-10.
After determining the workflow, the involved prospect service will be instantiated as abstract services by referring to the data type parameters. However, the instantiation parameters must conform to the constraints in the service pattern. For example, the restriction rules must be applied to data type parameters in order to assure that the assignment of data type parameters is valid and data exchange between services is smooth. Specifically, the first nine restriction rules for data type parameters rules in restriction rules of Figure 3-5 are applied to validate each data type assigned in data type parameters of Figure 3-9, and the next two restriction rules in Figure 3-5 further limit the relationships between data types. In addition, each message flow aligns output parameter of its source service with input parameter of its destination service; therefore
their data types need to be equivalent. For instance, in Figure 3-6 the message flow $mf_{10}$ connects the output variable $ExtrOut$ of $ExtractData$ service to the input variable $DrvIn$ of $DeriveData$ service. Since the type of $ExtrOut$ is $TReplenishReq$ that is equivalent to the type of $DrvIn$, it indicates that the data exchange between $ExtractData$ and $DeriveData$ activities is smooth. If all of the restriction rules checking and flow actual parameter alignment are successful, the involved prospect services can be instantiated into abstract services with specific IOPE specification as shown in Figure 3-11.

| Composition name: Stock replenishment  |
| Prospect Services:                      |
| Name: ExtractData                       |
| In: $TSalesInvQty$  ExtrIn, $TAttrNames$ ExtrAttr |
| Out: $TReplenishReq$ ExtrOut            |
| Precondition:                           |
\[
\text{ExtraAttr} \subseteq \text{Attributes} (TSalesData) 
\]
| Effect:                                |
\[
\text{ExtraAttr} = \text{Attributes} (TReplenishReq) \land |\text{ExtrIn}|=|\text{ExtrOut}| \land \\
(\forall e_1 (e_1 \in \text{ExtrOut} \Rightarrow \exists e_2 (e_2 \in \text{ExtrIn}), (\forall a \in \text{ExtrAttr} \ e_1.a = e_2.a)) 
\]

| Name: JoinExtendedData                  |
| In: $TSalesData$ JedIn, $TProdInv$ JedEd |
| Out: $TSalesInvQty$ JedOut              |
| Precondition:                           |
\[
\text{null} 
\]
| Effect:                                |
\[
\text{null} 
\]

| Name: DeriveData                        |
| In: $TReplenishReq$ DrvIn               |
| Out: $TOrder$ DrvOut                    |
| Precondition:                           |
\[
\text{null} 
\]
| Effect:                                |
\[
(\forall e_1 (e_1 \in \text{DrvOut} \Rightarrow \exists e_2 (e_2 \in \text{DrvIn}), \text{Key}(e_2) = \text{Key}(e_1)) \land \forall a \in \text{Attributes}(TOrder), \text{ComputeFrom}(e_1.a, e_2.a)) \\
\land (|\text{DrvIn}|=|\text{DrvOut}|) \land (\forall e \in \text{DrvOut}, \text{Sent2Partner}(e)) 
\]

Figure 3-11. The IOPE specification of instantiated prospect services for stock replenishment

After the matchmaking between abstract service and concrete service, both $JoinExtendedData$ and $ExtractData$ services can be implemented by WS-data operators, namely join and extraction respectively; $DeriveData$ service can be implemented by Replenish() Web service operation. Therefore, the final process of workflow-based Web service for stock replenishment can be shown in Figure 3-12.
Following the above process, the other two examples in Figure 3-1 can be instantiated into workflow-based Web service compositions as well. We illustrate them in brief in the following subsections.
3.3.2 The instantiation for profit computation

At first, the instantiation parameters are specified according to the requirement of profit computation. Their values are assigned as shown in Figure 3-13.

<table>
<thead>
<tr>
<th>Composition name:</th>
<th>Net income computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instantiation parameters:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Workflow adapting parameters:</strong></td>
<td></td>
</tr>
<tr>
<td>?SkipConverByCurrency=False;</td>
<td></td>
</tr>
<tr>
<td>?SkipConvertByUnit=True;</td>
<td></td>
</tr>
<tr>
<td>?SkipDeriveData=False;</td>
<td></td>
</tr>
<tr>
<td>?SkipAggregateData=True;</td>
<td></td>
</tr>
<tr>
<td>?ChooseFormat='Cube';</td>
<td></td>
</tr>
<tr>
<td>?ChooseComputeOrder='DeriveFirst';</td>
<td></td>
</tr>
<tr>
<td><strong>Data type parameters:</strong></td>
<td></td>
</tr>
<tr>
<td>?TypeOfEd=TExpenditure;</td>
<td></td>
</tr>
<tr>
<td>?TypeOfJedOut=TRevenueExpense;</td>
<td></td>
</tr>
<tr>
<td>?TypeOfExtrIn=TRevenueExpense;</td>
<td></td>
</tr>
<tr>
<td>?TypeOfExtrOut=TIncomeStatementReq;</td>
<td></td>
</tr>
<tr>
<td>?TypeOfDrvIn =TIncomeStatementReq;</td>
<td></td>
</tr>
<tr>
<td>?TypeOfDrvOut=TIncomeStatement;</td>
<td></td>
</tr>
<tr>
<td>?TypeOfAggIn=null;</td>
<td></td>
</tr>
<tr>
<td>?TypeOfAggOut=null;</td>
<td></td>
</tr>
<tr>
<td>?TypeOfOut=TIncomeStatement;</td>
<td></td>
</tr>
<tr>
<td><strong>Predicate parameters:</strong></td>
<td></td>
</tr>
<tr>
<td>?fpSelectData= (SalesData.Status=&quot;Confirmed&quot;);</td>
<td></td>
</tr>
<tr>
<td>?cpDeriveDataPrecond=null;</td>
<td></td>
</tr>
<tr>
<td>?cpDeriveDataEffect=null;</td>
<td></td>
</tr>
<tr>
<td>?cpAggregateDataPrecond=null;</td>
<td></td>
</tr>
<tr>
<td>?cpAggregateDataEffect=null;</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-13. The assignment of instantiation parameters for profit computation

Apply workflow adapting parameters to alternative selection and service skippable toggle constructs, the workflow is tailored as shown in Figure 3-14.
Figure 3-14. The adapted workflow for net income computation

The assignment of data type parameters is validated by restriction rules and the involved prospect services are instantiated into abstract services with specific IOPE specification as shown in Figure 3-15.
After the matchmaking between abstract service and concrete service, both of JoinExtendedData and ExtractData services can be implemented by WS-data operators, namely join and extraction respectively, and DeriveData service can be implemented by ComputeProfit() Web service operation. Therefore, the final process of workflow-based Web service for income statement can be shown in Figure 3-16.

Figure 3-15. The IOPE specification of instantiated prospect services for net income computation

<table>
<thead>
<tr>
<th>Composition name: Net income computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prospect Services:</strong></td>
</tr>
<tr>
<td>Name: ExtractData</td>
</tr>
<tr>
<td>In: TRevenueExpense ExtrIn, TAttrNames ExtrAttr</td>
</tr>
<tr>
<td>Out: TIncomeStatementReq ExtrOut</td>
</tr>
<tr>
<td>Precondition:</td>
</tr>
<tr>
<td>ExtraAttr⊆Attributes(TSalesData)</td>
</tr>
<tr>
<td>Effect:</td>
</tr>
<tr>
<td>ExtraAttr=Attributes(TIncomeStatementReq)∧</td>
</tr>
<tr>
<td>(∀ e₁ ∈ ExtrOut ⇒∃e₂ ∈ ExtrIn) (</td>
</tr>
<tr>
<td>(∀ a ∈ ExtrAttr e₁.a=e₂.a))</td>
</tr>
<tr>
<td>Name: JoinExtendedData</td>
</tr>
<tr>
<td>In: TSalesData JedIn, TExpenses JedEd</td>
</tr>
<tr>
<td>Out: TRevenueExpense JedOut</td>
</tr>
<tr>
<td>Precondition:</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td>Effect: null</td>
</tr>
<tr>
<td>Name: DeriveData</td>
</tr>
<tr>
<td>In: TIncomeStatementReq DrvIn</td>
</tr>
<tr>
<td>Out: TIncomeStatement DrvOut</td>
</tr>
<tr>
<td>Precondition:</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td>Effect:</td>
</tr>
<tr>
<td>(∀ e₁ ∈ DrvOut ⇒∃e₂ ∈ DrvIn) Key(e₂)=Key(e₁)∧</td>
</tr>
<tr>
<td>(∀ a ∈ Attributes(TIncomeStatement), ComputeFrom(e₁.a, e₂))</td>
</tr>
<tr>
<td>(</td>
</tr>
</tbody>
</table>
3.3.3 The instantiation for order fulfillment

At first, the instantiation parameters are specified according to the requirement of order fulfillment. Their values are assigned as shown in Figure 3-17.
Composition name: Order fulfillment

Instantiation parameters:

Workflow adapting parameters:
- ?SkipConvertByCurrency=True;
- ?SkipConvertByUnit=True;
- ?SkipDeriveData=False;
- ?SkipAggregateData=False;
- ?ChooseFormat='Flat';
- ?ChooseComputeOrder='AggregateFirst';

Data type parameters:
- ?TypeOfEd= null;
- ?TypeOfJedOut= null;
- ?TypeOfExtrIn= TSalesData;
- ?TypeOfExtrOut= TTransData;
- ?TypeOfDrvIn = TTransData;
- ?TypeOfDrvOut= TShippingPlan;
- ?TypeOfAggIn= TTransData;
- ?TypeOfAggOut= TTransData;
- ?TypeOfOut= TShippingPlan;

Predicate parameters:
- ?fpSelectData= (SalesData.Status="Confirmed");
- ?cpDeriveDataPrecond=(∀ e ∈ DrvIn, e.Status="Charged");
- ?cpDeriveDataEffect=(∀ e ∈ DrvOut, e.Status="Shipped");
- ?cpAggregateDataPrecond=(∀ e ∈ AggIn, e.Status="Confirmed");
- ?cpAggregateDataEffect=(∀ e ∈ AggOut, e.Status="Charged");

Figure 3-17. The assignment of instantiation parameters for order fulfillment

After applying workflow adapting parameters to alternative selection and service skippable toggle constructs, the workflow is tailored as shown in Figure 3-18.
Figure 3-18. The adapted workflow for order fulfillment
The assignment of data type parameters is validated by restriction and the involved prospect services are instantiated into abstract services with specific IOPE specification as shown in Figure 3-19.

<table>
<thead>
<tr>
<th>Composition name:</th>
<th>Order fulfillment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prospect Services:</strong></td>
<td></td>
</tr>
<tr>
<td>Name: ExtractData</td>
<td></td>
</tr>
<tr>
<td>In: TSalesData ExrIn, TAttrNames ExtrAttr</td>
<td></td>
</tr>
<tr>
<td>Out: TTransData ExtrOut</td>
<td></td>
</tr>
<tr>
<td>Precondition:</td>
<td></td>
</tr>
<tr>
<td>ExtraAttr ⊆ Attributes(TSalesData)</td>
<td></td>
</tr>
<tr>
<td>Effect:</td>
<td></td>
</tr>
<tr>
<td>ExtraAttr = Attributes(TTransData) (</td>
<td>\text{ExrIn}</td>
</tr>
</tbody>
</table>

Name: DeriveData
In: TTransData DrvIn
Out: TShippingPlan DrvOut
Precondition:
(\forall e \in \text{DrvIn}, e.\text{Status} = "Charged")
Effect:
(\forall e_1 (e_1 \in \text{DrvOut} \Rightarrow \exists e_2 (e_2 \in \text{DrvIn}), \text{Key}(e_2) = \text{Key}(e_1) \land 
 \forall a \in \text{Attributes(TShippingPlan)}, \text{ComputeFrom}(e_1.a, e_2.a) ) \land (|\text{DrvIn}| = |\text{DrvOut}|) \land (\forall e \in \text{DrvOut}, e.\text{Status} = "Shipped")

Name: AggregateData
In: TTransData AggIn
Out: TTransData AggOut
Precondition:
(\forall e \in \text{AggIn}, e.\text{Status} = "Confirmed")
Effect:
(|\text{AggOut}| \leq |\text{AggIn}|) \land (\forall e \in \text{AggIn}, e.\text{Status} = "Charged")

Figure 3-19. The IOPE specification of instantiated prospect services for order fulfillment

After the matchmaking between abstract service and concrete service, ExtractData service can be implemented by WS-data operator extraction, and AggregateData and DeriveData services can be implemented by MakePayment() and Shpping() Web service operations respectively. Therefore, the final process of workflow-based Web service for order fulfillment can be shown in Figure 3-20.
3.3.4 The instantiation for a new business process

This example, namely merchandise performance evaluation, is a prevalent process in retailing management to compute gross margin return on inventory investment (abbreviated as GMROI). It measures how many gross margin dollars are earned on every dollar of inventory
investment made by the buyer [20]. It needs aggregated sales amount and cost, which can be derived from sales transactions directly, so we can skip `ConvertByCurrency` and `ConvertByUnit` services and select the execution sequence (`AggregateSales`, `JoinExtendedData`, `ExtractData`) by specifying the alternative selection `ChooseFormat="Cube"`. After joining average inventory at cost, the process can derive a new attribute, namely GMROI, from gross margin and average inventory at cost. Therefore, the compute order is `DeriveFirst` and `AggregateData` service is skipped. The assignment of instantiation parameters is shown in Figure 3-21.

<table>
<thead>
<tr>
<th>Composition name: Merchandise performance evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Workflow adapting parameters:</strong></td>
</tr>
<tr>
<td>?SkipConvertByCurrency=True;</td>
</tr>
<tr>
<td>?SkipConvertByUnit=True;</td>
</tr>
<tr>
<td>?SkipDeriveData=False;</td>
</tr>
<tr>
<td>?SkipAggregateData=True;</td>
</tr>
<tr>
<td>?ChooseFormat=&quot;Cube&quot;;</td>
</tr>
<tr>
<td>?ChooseComputeOrder=&quot;DeriveFirst&quot;;</td>
</tr>
<tr>
<td><strong>Data type parameters:</strong></td>
</tr>
<tr>
<td>?TypeOfEd=TAvgInvCost;</td>
</tr>
<tr>
<td>?TypeOfJedOut=TSalesInvCost;</td>
</tr>
<tr>
<td>?TypeOfExtrIn=TSalesInvCost;</td>
</tr>
<tr>
<td>?TypeOfExtrOut=TMGROIReq;</td>
</tr>
<tr>
<td>?TypeOfDrvIn=TMGROIReq;</td>
</tr>
<tr>
<td>?TypeOfDrvOut=TMGROI;</td>
</tr>
<tr>
<td>?TypeOfAggIn=null;</td>
</tr>
<tr>
<td>?TypeOfAggOut=null;</td>
</tr>
<tr>
<td>?TypeOfOut=TMGROI;</td>
</tr>
<tr>
<td><strong>Predicate parameters:</strong></td>
</tr>
<tr>
<td>?fpSelectData=(State=&quot;Confirmed&quot;);</td>
</tr>
<tr>
<td>?cpDeriveDataPrecond=null;</td>
</tr>
<tr>
<td>?cpDeriveDataEffect=null;</td>
</tr>
<tr>
<td>?cpAggregateDataPrecond=null;</td>
</tr>
<tr>
<td>?cpAggregateDataEffect=null;</td>
</tr>
</tbody>
</table>

Figure 3-21. The assignment of instantiation parameters for merchandise performance evaluation

Apply workflow adapting parameters to alternative selection and service skippable toggle constructs, the workflow is tailored as shown in Figure 3-22.
Figure 3-22. The adapted workflow for merchandise performance evaluation

The assignment of data type parameters validated by restriction and the involved prospect services are instantiated into abstract services with specific IOPE specification as shown in Figure 3-23.
The IOPE specification of instantiated prospect services for merchandise performance evaluation

After the matchmaking between abstract service and concrete service, both of JoinExtendedData and ExtractData services can be implemented by WS-data operators join and extraction respectively, and DeriveData service can be implemented by ComputeGMROI() Web service operation. Therefore, the final process of workflow-based Web service for order fulfillment can be shown in Figure 3-24.
4. Automated Instantiation Parameters Assignment

As mentioned earlier, the instantiation parameters have to be specified for instantiating a service pattern. However, the user may not be able to properly specify all instantiation parameters. The values of instantiation parameters depend on the business requirements and software developing environment, such as existing Web services and message schemas. In other
words, business requirements define the goal, including given input types and expected output types, and the service pattern achieves the goal by determining the values of instantiation parameters which consider the existing Web services and message schemas. For example, a data type parameter can be assigned a message type only if all the prospect services that involve the data type parameters have corresponding concrete Web services that use the message type. Therefore, an assignment for instantiation parameters is feasible only if the developing environment supports all instantiation parameters. In this section, we will introduce the system architecture that aims to automatically infer a feasible assignment of instantiation parameters for a given goal from the developing environment. Then, the reasoning rules will be described in detail.

4.1 Overview of system architecture

Consider the instantiation of a service pattern sp into a target composite service ts. The first step for the developer is to specify the IOPE of ts denoted ts.IOPE. Next, the developer can either manually build ts based on sp and ts.IOPE or use the automated tool to reason and derive ts from sp and ts.IOPE. Since the derivation is based on the blueprint provided by sp, the search space for reasoning is greatly reduced, making it a feasible solution to significantly reduce the composition efforts.

Figure 4-1 shows the architecture of our automated instantiation system. The specifications of both service patterns and concrete services in the system are represented as facts and stored in SP repository. When a developer wants to build a target service ts, she/he first specifies ts.IOPE, which is converted into facts on the fly and stored in fact working memory.

We used Jess v7.1p2 as the rule engine and employ the forward-chaining method in the automated instantiation reasoning process. In Jess, some rules have to consider the related facts as a whole but each fact can only trigger a rule once. Therefore, we divide the reasoning process into three stages to gradually build the needed facts.

First, the matchmaking rules are applied and the set of concrete services that matches each prospect service in the pattern is identified, resulting in a number of facts added into the fact working memory. Second, for each potential set of service selections, all the customizable data type parameters declared in message alignment constraints are examined against the data consistency rule. If it is confirmed, a valid message flow assertion will be put into the working memory. Finally, the condition of feasible assignment is checked to verify whether an assignment is feasible. The three rules and corresponding inference processes are introduced in the following sections.
4.2 Facts creation

For IOPE of a concrete Web service can be specified by one fact for the interface and several others for the parameters in the interface. The following shows the facts for the IOPE of GetStoreSales concrete service in Jess:

(MAIN::CS (service GetStoreSales) (Inputs TStore TTimePeriod) (Outputs TSalesTrans) (Preconditions ) (Effects ))
(MAIN::Parameter (Service GetStoreSales) (IO Input) (Type TStore) (Variable StoreList))
(MAIN::Parameter (Service GetStoreSales) (IO Input) (Type TTimePeriod) (Variable SalesPeriod))
(MAIN::Parameter (Service GetStoreSales) (IO Output) (Type TSalesTrans) (Variable Trans))

The inference rule of services matchmaking in first stage uses this kind of facts to find out matched services. Parameter list details information for each parameter, including I/O mode, data type, and variable name. The inference rule validating message alignment draws on it to verify data types of connected parameters.

The facts of a prospect service can be similarly constructed, but a prospect service has additional elements to represent customizable information. The following shows the facts of JoinExtendedData prospect service. In contrast to the IOPE facts of a concrete service, there are four customizable elements for a prospect service, namely customizable inputs, customizable
outputs, customizable preconditions, and customizable effects. In our example, two customizable I/O parameter facts with type parameters, TypeOfED and TypeOfJedOut in boldface, are added.

(Main::PS (service JoinExtendedData) (Inputs TSalesData) (CustomizableInputs TypeOfED) (Outputs TypeOfJedOut) (CustomizableOutputs TypeOfJedOut) (Preconditions CustomizablePreconditions) (Effects CustomizableEffects))
(Main::Parameter (Service JoinExtendedData) (IO Input) (Type TSalesData) (Variable JedIn))
(Main::CustomizableParameter (Service JoinExtendedData) (IO Input) (TypeParameter TypeOfED) (Variable JedED))
(Main::CustomizableParameter (Service JoinExtendedData) (IO Output) (TypeParameter TypeOfJedOut) (Variable JedOut))

The other types of facts for a service pattern include message alignment, alternative selection, and skippable toggle facts. For example, the following message alignment fact connects parameters of ExtractData and DeriveData services, which will be used in the second stage to validate message flow between services.

(Main::MF (name mf10) (SourceService ExtractData) (DestinationService DeriveData) (SourceVariable ExtrOut) (DestinationVariable DrvIn))

The following facts describe the specification of an alternative selection and a skippable toggle. The first and last facts show basic information for these adapting controls, especially, their name slots correspond to bound instantiation parameters, where ChooseFormat and SkipConvertByCurrency are marked in boldface for clarity. Furthermore, alternative selection enumerates possible choices, Flat and Cube, whereas skippable toggle is attached to a service, ConvertByCurrency. The second and third facts list the associated message alignments and services for each choice of alternative selection. For example, in case of ‘Cube’ choice, AggregateSales and JoinExtendedData services will be included in final workflow.

(Main::AlternativeSelection (name ChooseFormat) (choices Flat Cube))
(Main::Choice (name ChooseFormat) (value Flat) (validmfs mf6) (services ))
(Main::Choice (name ChooseFormat) (value Cube) (validmfs mf7 mf8 mf9) (services AggregateSales JoinExtendedData))
(Main::SkippableToggle (name SkipConvertByCurrency) (service ConvertByCurrency))

The above facts can be converted before instantiating a service pattern. When a developer needs to instantiate a service pattern for a new requirement, they have to specify the goal for the requirement, given input and desired output, and make decisions on adapting controls, namely alternative selections and skippable toggles. The following facts show an example assignment. The first and second facts assign concrete data types, TProdInv and TOrder, to instantiation parameters, TypeOfED and TypeOfOut respectively, for customizable input and output. The third fact indicates that the service attaching with SkipConvertByCurrency instantiation parameter (i.e., ConvertByCurrency) will be removed in final workflow, whereas the associated service (i.e., ConvertByUnit) in the fourth fact remains. Finally, ChooseFormat instantiation parameter is set as Cube, which means AggregateSales and JoinExtendedData services will be included as mentioned before.
4.3 Inference rules

As mentioned before, the reasoning procedure is divided into three stages. The reason is that some rules have to consider the related facts as a whole, such as message alignment checking and final goal testing, but each fact can only trigger the same rule once in Jess.

4.3.1 Concretizing rules in the first stage

The purpose of the first stage is to concretize IOPE of the prospect services, conforming to restriction rules on data type parameters and their partial precondition/effect specifications. We define two rules to achieve the goal. The first rule is to match the interface of a prospect service with that of concrete services and assign specific data type to bound instantiation parameters of customizable data types and preconditions/effects. When this rule is fired, several new facts will be asserted for each bound instantiation parameter with specific data type from matched concrete services. The first rule in Jess is shown as follows:

```
(defrule Concretize-Prospect-Service
 (test (confirmNoSkip ?psname)) =>
 (assert (PSAssignment (ProspectService ?psname) (service ?csname)))
 (assertInputIP ?ps ?cs)
 (assertOutputIP ?ps ?cs)
 (if (and (> (length$ (fact-slot-value ?ps CustomizablePreconditions)) 0) (> (length$ (complement$ (fact-slot-value ?ps Preconditions)) 0)) then
  (assert (IPAssignment (ProspectService ?psname) (service ?csname) (Type Predicate) (Parameter (fact-slot-value ?ps CustomizablePreconditions)) (Assignment (complements$ (fact-slot-value ?ps Preconditions) (fact-slot-value ?cs Preconditions))))))
 (if (and (> (length$ (fact-slot-value ?ps CustomizableEffects)) 0) (> (length$ (complement$ (fact-slot-value ?ps Effects) (fact-slot-value ?cs Effects))))) then
)
```

The second rule, shown in the following, is to confirm the satisfaction of the restriction rules given in the service pattern. If the conditions of the second rule are met, the target fact will be asserted as a type assignment fact.

```
(defrule Check-Prospect-service
 ?psaf<-<PSAssignment (ProspectService ?psname))
 (test (ValidPSDataType ?psaf)) =>
 (assert (Hold (type ProspectService) (name (fact-slot-value ?psaf ProspectService)) (value (fact-slot-value ?psaf service)) (checked 1)))
)
```
In addition to the above rules, we also defined other rules that consider the facts converted from the goal, such as the values of alternative selections and skippable toggles specified by developer. Those rules assert new facts that will affect the trigger of the first rule by employing the prospect services according to developers’ choice. In other words, the prospect services, which are skipped by skippable toggles or not chosen in alternative selections, will not trigger the first rule.

4.3.2 Alignment rules in the second stage

In second stage, we want to confirm that the message exchange between component services is smooth by checking whether data types of connected parameters are compatible. We define three rules for this task: testing connected parameters of message alignment for matched type, checking a sequence of message flow facts for consistent message alignments, and removing false message flow facts. A prospect service may have several type assignment facts for its customizable data types because there could be several matching concrete services with different data types. The following rule (check-message-flow) considers the message flow facts converted from message alignments of service pattern and the type assignment facts asserted in the first stage. If data types of source and destination parameters in message flow fact are compatible, the rule will be triggered and create a valid message flow fact that asserts the target message flow connecting two cooperative service instances.

(defrule check-message-flow
  ?mf<- (MF (name ?mfname) (SourceService ?s1) (DestinationService ?s2) (SourceVariable ?v1) (DestinationVariable ?v2))
  ?ss<- (or (CS (service ?s1)) (PSAssignment (ProspectService ?s1))))
  ?ds<- (or (CS (service ?s2)) (PSAssignment (ProspectService ?s2)))
=>
  (assert (validMF (name ?mfname) (SourceService ?s1) (SourceInstance (fact-slot-value ?ss service)) (DestinationService ?s2)
    (DestinationInstance (fact-slot-value ?ds service))))
)

Even though a valid message flow fact is asserted, it may become false when it is inconsistent with its preceding or succeeding valid message flow facts. For example, consider the execution of a sequence of prospect services, JoinExtendedData, ExtractData, and DeriveData in SalesCube service pattern in Figure 3-5; there are two message alignments, mf9 and mf10, that connect source and destination parameters between two immediate prospect services. Suppose that each of ExtractData and DeriveData prospect services has two matched concrete Web services: PrepareReplenish and PrepareReorder for ExtractData and Replenish and Reorder for DeriveData; and JoinExtendedData prospect service maps to JoinProdInv concrete Web service. After applying the first rule as mentioned before, for message alignment mf10, there are two valid message flows: (PrepareReplenish, Replenish) and (PrepareReorder, Reorder), where in each message flow the first element is the source service instance and the second element is the
destination service instance. On the other hand, for message alignment $mf_9$, there is only one valid message flow, ($JoinProdInv$, $PrepareReplenish$), because the output parameter $JedOut$ of $JoinProdInv$ service is incompatible with the input parameter $ExtrIn$ of $PrepareReorder$ service. Their relationships are graphically displayed in Figure 4-2; the corresponding concrete services are shown in parentheses; and, three valid message flow facts are shown in red box beside the message alignments. In Figure 4-2, the sequence of valid message flows, ($JoinProdInv$, $PrepareReplenish$)-($PrepareReplenish$, $Replenish$), is the only consistent message flows when considering both $mf_9$ and $mf_{10}$ message alignments, whereas, the sequence of message flows, ($JoinProdInv$, $PrepareReplenish$)-($PrepareReorder$, $Reorder$), is not consistent because the destination service instance of the first message flow is different from the source service instance of the second message flow. In other words, in Figure 4-2 (a), output message of $JoinProdInv$ service can be fed to input of $PrepareReplenish$ service, and in turn output message of $PrepareReplenish$ service can be transferred to input of $Replenish$ service. In contrast, in case of Figure 4-2 (b), the output message of $JoinProdInv$ service cannot be fed to input of $PrepareReorder$ service due to type mismatch. Therefore, consistency checking between message flows can be conducted by examining the consistency of destination service instance and source service instance between every adjacent immediate message flows.

Figure 4-2. The graphic presentation of $mf_9$ and $mf_{10}$ message flow facts
The next two rules in the second stage identify consistent message flows and remove inconsistent ones. In the following code, the first rule (check-parameter-alignment-between-message-flows) assigns 1 to the target message flow fact (i.e. ?vmf2 in boldface) when the target is consistent with its preceding and succeeding message flows. We develop two functions, checkPrecedingMF() and checkSucceedingMF(), to check the consistency between target message flow and its preceding and succeeding message flows.

```
(defrule check-parameter-alignment-between-message-flows
  (test (checkPrecedingMF ?ss1 ?si1))
  (test (checkSucceedingMF ?ds1 ?di1))
  ?vmf2<- (validMF (name ?mf2) (SourceService ?ss1) (SourceInstance ?si1) (DestinationService ?ds1) (DestinationInstance ?di1) (checked 0))
  =>
  (modify ?vmf2 (checked 1))
)
```

As mentioned before, the consistency checking between two adjacent message flows can be carried out by inspecting their service instances in message flow facts. However, a complete consistency checking has to consider all of preceding and succeeding message flows for each valid message flow fact. Consider the example of stock replenishment, its adapted workflow is shown in Figure 4-3. Suppose that a valid message flow for message alignment mf9 is confirmed as a target message flow in red box in Figure 4-3. To confirm that the target message flow is consistent with all of its preceding valid message flows, a backward traverse until top valid message flows, inf1 and inf2, is required in order to check each valid message flow alone the traversing path. Consider Figure 4-3; firstly, the source service instance of a target message flow for mf9 must be consistent to the destination service instance of valid message flow fact for mf8. In turn, the valid message flow for mf8 must also be consistent with valid message flow fact for mf5. Similar checking is iterated until top valid message flows are reached. In the following, we take the function, checkPrecedingMF(), as the example to describe the implementation of a complete consistency checking.
Figure 4-3. The checking path of preceding valid message flows for message alignment \( mf_0 \) in stock replenishment

For preceding message flows checking, the checking function repeatedly traverses preceding valid message flows facts until some stop condition is met. There are two stop conditions: reaching top valid message flow or no further preceding valid message flow existed. Top valid message flow indicates that the source message comes from users’ input, so it is a success condition. Similarly, if there is no further preceding valid message flow, the check is complete and successful. In contrast, if there exists some preceding valid message flow along yet no compatible message alignment, there is no compatible message flow conforming to preceding message alignment. As a result, it is a fail condition. The codes of preceding message flows
checking is shown as follows. The succeeding message flows checking is similar to preceding message flows checking, but it traverses valid message flows by forward direction.

(defun checkPrecedingMF (?ds ?di)
  ;; Traverse preceding valid message flows
  (bind ?r (run-query* search-ValidMF-by-dst ?ds ?di))
  (bind ?stk (create$))
  (while (not (?r next))
    (bind ?vmf (?r getObject vmf))
    (if (eq (str-index in (fact-slot-value ?vmf name)) 1) then
      (return TRUE)
    else
      (bind ?stk (insert$ ?stk 1 (fact-slot-value ?vmf SourceService) (fact-slot-value ?vmf SourceInstance))))
  )
  (if (not (length$ ?stk)) then
    (while (not (length$ ?stk))
      (bind ss (nth$ 1 ?stk))
      (bind si (nth$ 2 ?stk))
      (bind lst (delete$ ?stk 1 2))
      ;; Traverse preceding valid message flows and push into stack
      (bind ?r2 (run-query* search-ValidMF-by-dst ?ss ?si))
      (if (?r2 next) then
        (bind ?vmf (?r2 getObject vmf))
        ;; checking stop condition of top message flow
        (if (eq (str-index in (fact-slot-value ?vmf name)) 1) then
          (return TRUE)
        else
          (bind ?stk (insert$ ?stk 1 (fact-slot-value ?vmf SourceService) (fact-slot-value ?vmf SourceInstance))))
      )
    )
  )
  else
    ;; checking stop condition of no further preceding message flow
    (bind ?r3 (run-query* search-MF-by-dst ?ss))
    (if (not (?r3 next)) then (return TRUE))
  )
  else
    ;; checking stop condition of no preceding message flow
    (bind ?r3 (run-query* search-MF-by-dst ?ds))
    (if (not (?r3 next)) then (return TRUE))
  )
  (return FALSE)
)

4.3.3 Goal checking rule in the third stage

An assignment is infeasible if any specified data types violate restriction rules, parameters in message alignment are incompatible, or any prospect services have no matched concrete services. Therefore, in final stage, we define a rule by which all facts deduced in previous stages are checked for a feasible assignment. The antecedent of this rule includes the validity checking of all of message alignments, alternative selections, and skippable toggle and the assurance that
all of prospect services are concretized with valid data types. The rule of inspecting feasible assignment is shown as follows:

(defrule check-feasible-assignment
  (forall (PS (service ?psn)) (Hold (type ProspectService) (name ?psn) (value ?csn) (checked 1)))
  (forall (MF (name ?mfn)) (validMF (name ?mfn) (checked 1)))
  (forall (AlternativeSelection (name ?asn)) (Hold (type AlternativeSelection) (name ?asn)))
  (forall (SkippableToggle (name ?stn)) (Hold (type SkippableToggle) (name ?stn) (checked 1)))
=>
  (printout t "There is a feasible assignment" crlf)
  (showFeasibleAssignment 1))
)

In case all of the conditions fit, the assignment for instantiation parameters is pronounced to be feasible. However, the complete assignment of instantiation parameters cannot be directly extracted from matched concrete services because there may be many matched concrete services for a prospect service, but only the matched instances conforming to the message alignments is a correct binding. Therefore, we develop the extraction function, namely showCompleteAssignment(), to print out only a feasible, complete assignment for instantiation parameters. The function codes are shown as follows.

(deffunction showCompleteAssignment(?p)
  (bind ?pslist (create$ _begin))
  (bind ?mr (run-query* search-all-validMF-facts))
  ;; extract valid message flows and their associated service instances.
  (while (?mr next)
    (bind ?m (?mr getObject vmf))
    (if (not (subsetp (create$ (fact-slot-value ?m SourceService) (fact-slot-value ?m SourceInstance)
                                (fact-slot-value ?m DestinationService) (fact-slot-value ?m DestinationInstance)))
        then
        (bind ?e (create$ (str-cat (fact-slot-value ?m SourceService)  "," (fact-slot-value ?m SourceInstance))))
        (if (not (subsetp ?e ?pslist)) then
          (bind ?pslist (replace$ ?pslist 1 1 (create$ _begin (fact-slot-value ?m SourceService) ?e)))
        )
        (bind ?e (create$ (str-cat (fact-slot-value ?m DestinationService) "," (fact-slot-value ?m DestinationInstance))))
        (if (not (subsetp ?e ?pslist)) then
          (bind ?pslist (replace$ ?pslist 1 1 (create$ _begin (fact-slot-value ?m DestinationService) ?e)))
        ))
    )
  )
  ;; print out prospect service and data type parameter bindings.
  (bind ?r (run-query* search-hold-facts-by-parameters ProspectService))
  (while (?r next)
    (bind ?hrec (?r getObject hld))
    (if (or (and (not (subsetp (create$ (fact-slot-value ?hrec name) ?pslist)) (not (eq (fact-slot-value ?hrec value) nil)))
                        (subsetp (create$ (str-cat (fact-slot-value ?hrec name) "," (fact-slot-value ?hrec value))) ?pslist))
        then
        (printout t "Prospect Service:" (fact-slot-value ?hrec name) " binds " (fact-slot-value ?hrec value) crlf)
        (bind ?ipr (run-query* search-IPAssignments-by-PS-name-and-CS-name (fact-slot-value ?hrec name) (fact-slot-value ?hrec value)))
        (while (?ipr next)
          (bind ?ipe (?ipr getObject ipa))
          (printout t (fact-slot-value ?ipe Parameter) "=" (fact-slot-value ?ipe Assignment) "") crlf)
        )
      )
    )
  )
  ;; print out the assignment for alternative selections
  (bind ?r (run-query* search-hold-facts-by-parameters AlternativeSelection))
  (while (?r next)
    (bind ?hrec (?r getObject hld))
    (printout t "Alternative Selection:" (fact-slot-value ?hrec name) " is " (fact-slot-value ?hrec value) crlf)
  )
  ;; print out the assignment for skippable toggle
  (bind ?r (run-query* search-hold-facts-by-parameters SkippableToggle))
  (while (?r next)
    (bind ?hrec (?r getObject hld))
In this function, we firstly retrieve all valid message flows conforming to the message alignment restriction and put service instances associated with valid message flows into a valid prospect service list, namely ?pslist. Then, the matched concrete services are retrieved and the conditions of correct matched instances are checked. The conditions are either that the matched service exist in the valid prospect service list if they exchange data with other services or that the prospect service has matched concrete service without exchanging data with other services. If either condition is satisfied, the matched services and the bindings of their associated data type parameters will be printed out. Finally, the values of workflow adapting parameters, alternative selection and skippable toggle, can be directly displayed from the inferred facts. In this version, only one complete assignment of instantiation parameters is printed out. Indeed, there may exist several complete assignments that conform to a given (partial) assignment, and how to choose a “good” complete assignment is beyond the scope of this paper. Figure 6-4 shows a screen shot of Jess that displays the result of the instantiation parameter reasoning. The full version of Jess codes is shown in Appendix.
5. Performance evaluations

To evaluate the performance of the automated assignment reasoning, we conducted three experiments to reveal the execution times under different numbers of candidate Web services, message alignments, and prospect services. We use stock replenishment as the example and assume the assignment given by the developer specifies only skippable toggles and alternative selections. Moreover, because message alignment and prospect service create many facts during reasoning, more message alignments or prospect services may consume much time. To recognize the influence of message alignment and prospect service on the performance of automated assignment reasoning, we reduce the numbers of message alignments and prospect services in the second and third scenarios respectively, and compare their reasoning time with the first
scenario by examining reasoning time at different reasoning stages. Therefore, two additional scenarios with different modifications are given to these experiments and they are summarized in Table 5-1. These experiments are conducted on a PC with an Intel Xeon 2.33 GHz CPU. The reasoning process is iterated over 100 times for each case with different numbers of candidate Web services, and the average execution time is reported.

Table 5-1. The modifications and initial setting for three scenarios

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Modifications</th>
<th>Initial Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st scenario:</td>
<td>Use SalesCube service pattern</td>
<td>?TypeOfED=TProdInv; ?TypeOfOut=TOrder; ?SkipConverByCurrency=True;</td>
</tr>
<tr>
<td>17 MFs 3 PSs</td>
<td></td>
<td>?SkipConvertByUnit=False; ?SkipDeriveData=False; ?SkipAggregateData=True;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>?ChooseFormat=&quot;Cube&quot;; ?ChooseComputeOrder=&quot;DeriveFirst&quot;;</td>
</tr>
<tr>
<td>2nd scenario:</td>
<td>Use SalesCube service pattern but remove mf1 to mf8 message alignments</td>
<td>Same as the first scenario</td>
</tr>
<tr>
<td>9 MFs 3 PSs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3rd scenario:</td>
<td>Use SalesCube service pattern but change JoinExtendedData and ExtractData as</td>
<td>Same as the first scenario</td>
</tr>
<tr>
<td>17 MFs 1 PS</td>
<td>two concrete services</td>
<td></td>
</tr>
</tbody>
</table>

In the first experiment, 3 prospect services and 17 message alignments will be concretized and checked in automated reasoning process. The results of the first experiment are shown in Figure 5-1 and Figure 5-2. In Figure 5-1, the execution time has a super-linear trend while the growth of the numbers of facts is linear to the number of candidate Web services in Figure 5-2. It implies that execution performance is highly attributable to number of candidate Web services.
Figure 5-1. The execution time under different numbers of concrete Web services

Figure 5-2. The number of facts under different numbers of concrete Web services

In the second experiment, we reduce the number of message alignments to nine while maintaining the same number of prospect services in SalesCube service pattern. The performance comparison between the first and second scenarios is shown in Figure 5-3. Furthermore, we conduct Welch’s t test, shown in Table 5-2, to confirm that time difference between the first and second scenarios is significant because their variances are heterogeneous in Levene’s test. As can be seen, the execution time of the second scenario is (slightly) less than that of the first scenario under different number of candidate Web services. The reason is that
fewer message alignments result in lower loading of message flow validation and consistency checking. We further analyze the performance differences of the first and second scenarios at different inference stages using the case with 350 candidate Web services. Figure 5-4 shows that the inference stages related with message alignment, namely MF validation and consistency checking, have higher performance gap than other stages. Especially, at consistency checking stage, the second scenario has significantly higher performance improvement because fewer message alignments considerably reduce the depth of traversing message flows. However, in all other stages, the performance differences are really minor. This explains the slight performance improvement with fewer message alignments.

![Figure 5-3. The performance comparison between the first and second scenarios](image)

Table 5-2 The result of Welch’s t test for the first and second scenarios

<table>
<thead>
<tr>
<th>Number of candidate WSs</th>
<th>Degree of freedom for scenarios</th>
<th>Degree of freedom for error</th>
<th>F-value</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1</td>
<td>131.099</td>
<td>169.933</td>
<td>.000</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>162.494</td>
<td>59.241</td>
<td>.000</td>
</tr>
</tbody>
</table>
In the third experiment, we concretize two prospect services and leaves only one prospect service while maintaining seventeen message alignments in SalesCube service pattern. The performance comparison between the first and third scenarios is shown in Figure 5-5. We also conduct Welch’s t test, shown in Table 5-3, to confirm that time difference between the first and third scenarios is significant because their variances are heterogeneous in Levene’s test. In Figure 5-5, the third scenario outperforms the first scenarios. Because third scenario has fewer prospect services, it produces fewer service matchmaking facts and valid message flow facts. As a result, the reasoning load of service matchmaking and message flow validation is reduced. Figure 5-6 provides the evidence for this result. In Figure 5-6, the execution times of third scenario at service matchmaking and MF validation are the lowest.
Figure 5-5. The performance comparison between the first and third scenarios

Table 5-3 The result of Welch’s t test for the first and third scenarios

<table>
<thead>
<tr>
<th>Number of candidate WSs</th>
<th>Degree of freedom for scenarios</th>
<th>Degree of freedom for error</th>
<th>F-value</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1</td>
<td>139.418</td>
<td>212.257</td>
<td>.000</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>215.640</td>
<td>568.136</td>
<td>.000</td>
</tr>
<tr>
<td>150</td>
<td>1</td>
<td>198.657</td>
<td>531.364</td>
<td>.000</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>210.839</td>
<td>941.348</td>
<td>.000</td>
</tr>
<tr>
<td>250</td>
<td>1</td>
<td>215.736</td>
<td>1222.012</td>
<td>.000</td>
</tr>
<tr>
<td>300</td>
<td>1</td>
<td>203.642</td>
<td>845.442</td>
<td>.000</td>
</tr>
<tr>
<td>350</td>
<td>1</td>
<td>211.901</td>
<td>1401.561</td>
<td>.000</td>
</tr>
</tbody>
</table>
Figure 5-6. The performance comparisons between the first and third scenarios at different inference stages for the case with 350 candidate Web services.
Reference


Appendix: The Inference Rules in Jess

The fact template:

(deftemplate CS (slot service) (multislot Inputs) (multislot Outputs) (multislot Preconditions) (multislot Effects))
(deftemplate PS (slot service) (multislot Inputs) (multislot CustomizableInputs) (multislot Outputs) (multislot CustomizableOutputs) (multislot Preconditions) (multislot CustomizableEffects))
(deftemplate MF (slot name) (slot SourceService) (slot DestinationService) (slot SourceVariable) (slot DestinationVariable))
(deftemplate Parameter (slot Service) (slot IO) (slot Type) (slot Variable))
(deftemplate CustomizableParameter (slot Service) (slot IO) (slot TypeParameter) (slot Variable))
(deftemplate AlternativeSelection (slot name) (multislot choices))
(deftemplate Choice (slot name) (slot value) (multislot validmfs) (multislot services))
(deftemplate SkippableToggle (slot name) (slot service))
(deftemplate Hold (slot type) (slot name) (slot value) (slot checked (default 0)))
(deftemplate PSAssignment (slot ProspectService) (slot service))
(deftemplate IPAssignment (slot ProspectService) (slot Type) (slot Parameter) (slot Assignment))
(deftemplate validMF (slot name) (slot SourceService) (slot SourceInstance) (slot DestinationService) (slot DestinationInstance) (slot checked (default 0)))
(deftemplate StrategyforSkip (slot SkippableToggle) (slot msgflow) (slot substituteMF))
(deftemplate STDependecy (slot SkippableToggle) (multislot previousST))

;; Service Pattern IOPE
(assert (PS (service SalesCube) (Inputs TStore TTimePeriod TMoney TSalesClass) (Preconditions SPprecond1) (Effects SPef1) (CustomizableInputs TypeOfED) (CustomizableOutputs TypeOfOut)))
(assert (Parameter (Service SalesCube) (IO Input) (Type TStore) (Variable In_sl)))
(assert (Parameter (Service SalesCube) (IO Input) (Type TSalesClass) (Variable In_ea)))
(assert (Parameter (Service SalesCube) (IO Input) (Type TTimePeriod) (Variable In_tp)))
(assert (Parameter (Service SalesCube) (IO Input) (Type TMoney) (Variable In_cu)))
(assert (Parameter (Service SalesCube) (IO Input) (Type TSalesClass) (Variable In_ga)))
(assert (CustomizableParameter (Service SalesCube) (IO Input) (TypeParameter TypeOfED) (Variable In_ed)))
(assert (CustomizableParameter (Service SalesCube) (IO Output) (TypeParameter TypeOfOut) (Variable Out)))

;; Concrete services
(assert (CS (service GetStoreSales) (Inputs TStore TTimePeriod) (Outputs TSalesTrans)))
(assert (Parameter (Service GetStoreSales) (IO Input) (Type TStore) (Variable StoreList)))
(assert (Parameter (Service GetStoreSales) (IO Input) (Type TTimePeriod) (Variable SalesPeriod)))
(assert (Parameter (Service GetStoreSales) (IO Output) (Type TSalesTrans) (Variable Trans)))
(assert (CS (service GetProductProfile) (Inputs TStore) (Outputs TProdData)))
(assert (Parameter (Service GetProductProfile) (IO Input) (Type TProdData) (Variable StoreList)))
(assert (Parameter (Service GetProductProfile) (IO Output) (Type TProdData) (Variable ProdData)))
(assert (CS (service ConvertByCurrency) (Inputs TSalesData TMoney) (Outputs TSalesData)))
(assert (Parameter (Service ConvertByCurrency) (IO Input) (Type TSalesData) (Variable CvtCurIn)))
(assert (Parameter (Service ConvertByCurrency) (IO Input) (Type TMoney) (Variable CvtCur)))
(assert (Parameter (Service ConvertByCurrency) (IO Output) (Type TSalesData) (Variable CvtCurOut)))
(assert (CS (service ConvertByUnit) (Inputs TSalesData) (Outputs TSalesData)))
(assert (Parameter (Service ConvertByUnit) (IO Input) (Type TSalesData) (Variable CvtUnitIn)))
(assert (Parameter (Service ConvertByUnit) (IO Output) (Type TSalesData) (Variable CvtUnitOut)))

(assert (CS (service DataCube) (Inputs TSalesData) (Outputs TSalesData)))
(assert (Parameter (Service DataCube) (IO Input) (Type TSalesData) (Variable CubeIn)))
(assert (Parameter (Service DataCube) (IO Output) (Type TSalesData) (Variable CubeOut)))

(assert (CS (service JoinProductData) (Inputs TSalesTrans TProdData) (Outputs TSalesData)))
(assert (Parameter (Service JoinProductData) (IO Input) (Type TSalesTrans) (Variable JoinIn1)))
(assert (Parameter (Service JoinProductData) (IO Input) (Type TProdData) (Variable JoinIn2)))
(assert (Parameter (Service JoinProductData) (IO Output) (Type TSalesData) (Variable JoinOut)))

(assert (CS (service SelectData) (Inputs TSalesData) (Outputs TSalesData) (Effects Selelf1)))
(assert (Parameter (Service SelectData) (IO Input) (Type TSalesData) (Variable SelIn)))
(assert (Parameter (Service SelectData) (IO Output) (Type TSalesData) (Variable SelOut)))

;; Prospect services
(assert (PS (service DeriveData) (CustomizableInputs TypeOfDrvIn)(CustomizableOutputs TypeOfDrvOut)(Preconditions precond1 precond2) (CustomizablePreconditions cpDeriveDataPrecond)(Effects ef1)(CustomizableEffects cpDeriveDataEffect)))
(assert (CustomizableParameter (Service DeriveData) (IO Input) (TypeParameter TypeOfDrvIn) (Variable DrvIn)))
(assert (CustomizableParameter (Service DeriveData) (IO Output)(TypeParameter TypeOfDrvOut) (Variable DrvOut)))

(assert (PS (service AggregateData) (CustomizableInputs TypeOfAggIn)(CustomizableOutputs TypeOfAggOut)(Preconditions AggPrecond1 AggPrecond2) (CustomizablePreconditions cpAggregateDataPrecond)(Effects AggEf1)(CustomizableEffects cpAggregateDataEffect)))
(assert (CustomizableParameter (Service AggregateData) (IO Input) (TypeParameter TypeOfAggIn) (Variable AggIn)))
(assert (CustomizableParameter (Service AggregateData) (IO Output)(TypeParameter TypeOfAggOut) (Variable AggOut)))

(assert (PS (service DeriveData1) (CustomizableInputs TypeOfDrvIn)(CustomizableOutputs TypeOfDrvOut)(Preconditions precond1 precond2) (CustomizablePreconditions cpDeriveDataPrecond)(Effects ef1)(CustomizableEffects cpDeriveDataEffect)))
(assert (CustomizableParameter (Service DeriveData1) (IO Input) (TypeParameter TypeOfDrvIn) (Variable DrvIn1)))
(assert (CustomizableParameter (Service DeriveData1) (IO Output)(TypeParameter TypeOfDrvOut) (Variable DrvOut1)))

(assert (PS (service AggregateData1) (CustomizableInputs TypeOfAggIn)(CustomizableOutputs TypeOfAggOut)(Preconditions AggPrecond1 AggPrecond2) (CustomizablePreconditions cpAggregateDataPrecond)(Effects AggEf1)(CustomizableEffects cpAggregateDataEffect)))
(assert (CustomizableParameter (Service AggregateData1) (IO Input) (TypeParameter TypeOfAggIn) (Variable AggIn1)))
(assert (CustomizableParameter (Service AggregateData1) (IO Output)(TypeParameter TypeOfAggOut) (Variable AggOut1)))

(assert (PS (service ExtractData) (Inputs TSalesClass)(CustomizableInputs TypeOfExtrIn)(CustomizableOutputs TypeOfExtrOut)(Preconditions ExtractDataPrecond)(Effects ExtractDataEffect1 ExtractDataEffect2)))
(assert (Parameter (Service ExtractData) (IO Input) (Type TSalesClass) (Variable ExtrAttrs)))
(assert (CustomizableParameter (Service ExtractData) (IO Input) (TypeParameter TypeOfExtrIn) (Variable ExtrIn)))
(assert (CustomizableParameter (Service ExtractData) (IO Output) (TypeParameter TypeOfExtrOut) (Variable ExtrOut)))

(assert (PS (service JoinExtendedData) (Inputs TSalesData)(CustomizableInputs TypeOfED)(CustomizableOutputs TypeOfJedOut)))
(assert (Parameter (Service JoinExtendedData) (IO Input) (Type TSalesData) (Variable JedIn)))
(assert (CustomizableParameter (Service JoinExtendedData) (IO Input) (TypeParameter TypeOfED) (Variable JedED)))

63
(assert (CustomizableParameter (Service JoinExtendedData) (IO Output) (TypeParameter
  TypeOfJedOut) (Variable JedOut)))

;; Message alignments
(assert (MF (name in1) (SourceService SalesCube) (DestinationService GetStoreSales)
  (SourceVariable In_sl) (DestinationVariable StoreList)))
(assert (MF (name in2) (SourceService SalesCube) (DestinationService GetProductProfile)
  (SourceVariable In_s1) (DestinationVariable StoreList)))
(assert (MF (name mf1) (SourceService GetStoreSales) (DestinationService JoinProductData)
  (SourceVariable Trans) (DestinationVariable JoinIn1)))
(assert (MF (name mf2) (SourceService GetProductProfile) (DestinationService JoinProductData)
  (SourceVariable ProdData) (DestinationVariable JoinIn1)))
(assert (MF (name mf3) (SourceService JoinProductData) (DestinationService SelectData)
  (SourceVariable JoinOut) (DestinationVariable SelIn)))
(assert (MF (name mf4) (SourceService SelectData) (DestinationService ConvertByCurrency)
  (SourceVariable SelOut) (DestinationVariable CvtCurIn)))
(assert (MF (name mf5) (SourceService ConvertByCurrency) (DestinationService
  ConvertByUnit) (SourceVariable CvtCurOut) (DestinationVariable CvtUnitOut)))
(assert (MF (name mf6) (SourceService ConvertByUnit) (DestinationService ExtractData)
  (SourceVariable CvtUnitOut) (DestinationVariable ExtrIn)))
(assert (MF (name mf7) (SourceService DataCube) (DestinationService JoinExtendedData)
  (SourceVariable CubeOut) (DestinationVariable JedIn)))
(assert (MF (name mf8) (SourceService JoinExtendedData) (DestinationService ExtractData)
  (SourceVariable JedOut) (DestinationVariable ExtrIn)))
(assert (MF (name mf9) (SourceService JoinExtendedData) (DestinationService DeriveData)
  (SourceVariable ExtrOut) (DestinationVariable DrvIn)))
(assert (MF (name mf10) (SourceService DeriveData) (DestinationService AggregateData)
  (SourceVariable DrvOut) (DestinationVariable AggIn)))
(assert (MF (name mf11) (SourceService DeriveData) (DestinationService AggregateData1)
  (SourceVariable DrvOut) (DestinationVariable AggIn1)))
(assert (MF (name mf12) (SourceService ExtractData) (DestinationService AggregateData1)
  (SourceVariable ExtrOut) (DestinationVariable AggIn1)))
(assert (MF (name mf13) (SourceService AggregateData1) (DestinationService DeriveData1)
  (SourceVariable AggOut1) (DestinationVariable DrvIn1)))
(assert (MF (name outf1) (SourceService AggregateData) (DestinationService SalesCube)
  (SourceVariable AggOut) (DestinationVariable Out))))
(assert (MF (name outf2) (SourceService DeriveData1) (DestinationService SalesCube)
  (SourceVariable DrvOut1) (DestinationVariable Out))))

;; Extended control constructs
(assert (AlternativeSelection (name ChooseFormat) (choices Flat Cube)))
(assert (Choice (name ChooseFormat) (value Flat) (validmfs mf6)))
(assert (Choice (name ChooseFormat) (value Cube) (validmfs mf7 mf8 mf9) (services DataCube
  JoinExtendedData)))
(assert (AlternativeSelection (name ChooseComputeOrder) (choices DeriveFirst
  AggregateFirst)))
(assert (Choice (name ChooseComputeOrder) (value DeriveFirst) (validmfs mf10 mf11 outf1
  (services DeriveData AggregateData))))
(assert (Choice (name ChooseComputeOrder) (value AggregateFirst) (validmfs mf12 mf13 outf2
  (services DeriveData1 AggregateData1))))
(assert (SkippableToggle (name SkipConvertByCurrency) (service ConvertByCurrency)))
(assert (SkippableToggle (name SkipConvertByUnit) (service ConvertByUnit)))
(assert (STDependency (SkippableToggle SkipConvertByUnit) (previousST
  SkipConvertByCurrency_end)))
(assert (StrategyforSkip (SkippableToggle SkipConvertByCurrency) (msgflow mf5
  (substitutemf mf4)))

64
Concretizing rules in the first stage:

(defquery search-validMF-by-MF-name
 "Finds validMF with name"
 (declare (variables ?na))
 ?vmf<-(validMF (name ?na))
)

(defquery search-customizable-parameters-by-prospect-service-name
 "Finds all of customizable data types of a prospect service"
 (declare (variables ?psn ?io))
)

(defquery search-customizable-parameter-assignment-by-PS-name-and-parameter-type
 (declare (variables ?psn ?csn ?t))
 ?ipa<-(IPAssignment (ProspectService ?psn) (service ?csn) (Type DataType) (Parameter ?t) (Assignment ?v))
)

(deffunction ValidDataType(?t ?v)
 "Confirm the data type restriction"
 (return TRUE)
)

(deffunction ValidPSDataType(?psaf)
 "Confirm that all of customizable data types of ?psfact are valid"
 (bind ?psn (fact-slot-value ?psaf ProspectService))
 (bind ?csn (fact-slot-value ?psaf service))
 (bind ?plist (create$ Input Output))
 (foreach ?p ?plist
 (while (?res next)
 (bind ?CPrec (?res getObject cp))
 (if (> (count-query-results
 search-customizable-parameter-assignment-by-PS-name-and-parameter-type
 ?psn ?csn (fact-slot-value ?CPrec TypeParameter)) 0) then (return FALSE))
 (bind ?r2 (run-query*
 search-customizable-parameter-assignment-by-PS-name-and-parameter-type
 ?psn ?csn (fact-slot-value ?CPrec TypeParameter)))
 (while (?r2 next)
 (bind ?IPrec (?r2 getObject ipa))
 (if (not (test (ValidDataType (fact-slot-value ?IPrec Parameter)))

(fact-slot-value ?IPrec Assignment)) then (return FALSE))
)
)
(return TRUE)
)

(deffunction assertInputIP(?ps ?cs)
(if (> (length\$ (fact-slot-value ?ps CustomizableInputs)) 0) then
  (bind ?plist (complement\$ (fact-slot-value ?ps Inputs) (fact-slot-value ?cs Inputs)))
  (bind ?rno (count-query-results search-customizable-parameters-by-prospect-service-name
            (fact-slot-value ?ps service) Input))
  (bind ?res (run-query\* search-customizable-parameters-by-prospect-service-name
            (fact-slot-value ?ps service) Input))
  (if (= ?rno (length\$ ?plist)) then
    (bind ?pos 1)
    (while (?res next)
      (bind ?rec (?res getObject cp))
      (assert (IPAssignment (ProspectService (fact-slot-value ?ps service))
                      (service (fact-slot-value ?cs service)) (Type DataType)
                      (Parameter (fact-slot-value ?rec TypeParameter))
                      (Assignment (nth\$ ?pos ?plist))))
    (++ ?pos)
  )
  )
  (return TRUE)
else
  (return FALSE)
) )

(deffunction assertOutputIP(?ps ?cs)
(if (> (length\$ (fact-slot-value ?ps CustomizableOutputs)) 0) then
  (bind ?plist (complement\$ (fact-slot-value ?ps Outputs) (fact-slot-value ?cs Outputs)))
  (bind ?rno (count-query-results search-customizable-parameters-by-prospect-service-name
            (fact-slot-value ?ps service) Output))
  (bind ?res (run-query\* search-customizable-parameters-by-prospect-service-name
            (fact-slot-value ?ps service) Output))
  (if (= ?rno (length\$ ?plist)) then
    (bind ?pos 1)
    (while (?res next)
      (bind ?rec (?res getObject cp))
      (assert (IPAssignment (ProspectService (fact-slot-value ?ps service))
                      (service (fact-slot-value ?cs service)) (Type DataType)
                      (Parameter (fact-slot-value ?rec TypeParameter))
                      (Assignment (nth\$ ?pos ?plist))))
    (++ ?pos)
  )
  )
  (return TRUE)
else
  (return FALSE)
) )

(defquery search-Skippable-Toggle-by-service
(declare (variables ?psn))
 ?st<-(SkippableToggle (name ?skn) (service ?psn))
(defquery search-Held-PS-by-nil
  (declare (variables ?psn))
  ?st<- (Hold (type ProspectService) (name ?psn) (value nil)))
)

(deffunction confirmNoSkip(?psname)
  (bind ?r (run-query* search-Skippable-Toggle-by-service ?psname))
  (while (?r next)
    (bind ?e (?r getObject st))
    (bind ?str (run-query* search-held-Skippable-toggle (fact-slot-value ?e name)))
    (while (?str next)
      (bind ?st (?str getObject hstf))
      (if (eq (fact-slot-value ?st value) TRUE) then (return FALSE))
    )
  )

;; check skipped choice of alternative selection
(if (> (count-query-results search-Held-PS-by-nil ?psname) 0) then (return FALSE))
(return TRUE)
)

(defrule Check-Prospect-service
  ?psaf<-(PSAssignment (ProspectService ?psn))
  (test (ValidPSDataType ?psaf))
=>
  (assert (Hold (type ProspectService) (name (fact-slot-value ?psaf ProspectService))
    (value (fact-slot-value ?psaf service)) (checked 1)))
)

(defrule Concretize-Prospect-Service
    $?psout&:(subsetp ?psout ?out)) (Preconditions
    $?psprecond&:(subsetp ?psprecond ?precond) (Effects
    $?pseffect&:(subsetp ?pseffect ?effect)))
  (test (confirmNoSkip ?psname))
=>
  (assert (PSAssignment (ProspectService ?psname) (service ?csname)))
  (assertInputIP ?ps ?cs)
  (assertOutputIP ?ps ?cs)
  (if (and (> (length$ (fact-slot-value ?ps CustomizablePreconditions)) 0) (> (length$ (complement$ (fact-slot-value ?ps Preconditions) (fact-slot-value ?cs Preconditions))) 0))
    then
    (assert (IPAssignment (ProspectService ?psname) (service ?csname) (Type Predicate)
      (Parameter (fact-slot-value ?ps CustomizablePreconditions)) (Assignment (complement$ (fact-slot-value ?ps Preconditions) (fact-slot-value ?cs Preconditions))))))
  (if (and (> (length$ (fact-slot-value ?ps CustomizableEffects)) 0) (> (length$ (complement$ (fact-slot-value ?ps Effects) (fact-slot-value ?cs Effects))) 0))
    then
    (assert (IPAssignment (ProspectService ?psname) (service ?csname) (Type Predicate)
)

;; Alternative selection handling

(defquery search-MFs-by-choice-of-AlternativeSelection

67
(declare (variables ?na))
?cf<-((Choice (name ?na)))

(defquery search-PS-by-name
  (declare (variables ?na))
  ?psf<-((PS (service ?na)))
)

(defquery search-Held-PS-by-PS-name
  (declare (variables ?psn))
  ?pshf<-((Hold (type ProspectService) (name ?psn)))
)

(defquery search-IPAssignments-by-PS-name
  (declare (variables ?psn))
  ?ipa<-((IPAssignment (ProspectService ?psn)))
)

(defquery search-PSAssignments-by-PS-name
  (declare (variables ?psn))
  ?psa<-((PSAssignment (ProspectService ?psn)))
)

(deffunction assertValidMF(?na ?va)
  (while (?r next)
    (bind ?chs (?r getObject cf))
    (if (not (eq ?va (fact-slot-value ?chs value))) then
      (bind ?mflist (fact-slot-value ?chs validmfs))
      (foreach ?m ?mflist
        (if (= (count-query-results search-validMF-by-MF-name ?m) 0) then
          (assert (validMF (name ?m) (SourceService nil) (SourceInstance nil) (DestinationService nil) (DestinationInstance nil) (checked 1)))
        )
      )
    )
    (bind ?slist (fact-slot-value ?chs services))
    (foreach ?m ?slist
      (assert (Hold (type ProspectService) (name ?m) (value nil) (checked 1)))
    )
  ))
)

(defrule force-validMF-when-alternative-selection-specified
  ?asf<-((Hold (type AlternativeSelection) (name ?na) (value ?va))
  =>
    (assertValidMF ?na ?va)
  )
)

(defquery search-Held-AlternativeSelection-by-name
  (declare (variables ?na))
  ?as<-((Hold (type AlternativeSelection) (name ?na) (checked 1)))
)

(deffunction ValidChoice(?c)
  ;; if initial setting exist then return false
  (bind ?na (fact-slot-value ?c name))
)
(bind ?r (run-query* search-Held-AlternativeSelection-by-name ?na))
(if (?r next) then (return FALSE))
;; in case there is no initial setting for alternative selection
(bind ?mflist (fact-slot-value ?c validmfs))
(bind ?n (length$ ?mflist))
(foreach ?m ?mflist
  (bind ?res (run-query* search-validMF-by-MF-name ?m))
  (while (?res next) (bind ?n (- ?n 1))))
)
(if (= ?n 0) then (return TRUE) else (return FALSE))
)

(defrule check-choice
  ?c1<-(Choice (name ?n))
  (test (ValidChoice ?c1))
=>
  (assert (Hold (type AlternativeSelection) (name (fact-slot-value ?c1 name)) (value
           (fact-slot-value ?c1 value)) (checked 1))))
)

(defrule Determine-SkippedProspectService-Assignment
  ?f1<-(SkippableToggle (name ?stn)(service ?psn))
  ?f2<-(Hold (type SkippableToggle) (name ?stn) (value TRUE) (checked 1))
  ?f3<-(PS (service ?psn))
=>
  (assert (Hold (type ProspectService) (name ?psn) (value nil) (checked 1))))
)

;; Handling the message flows related with skipped services
(defquery search-message-flow-by-source-service-name
  (declare (variables ?ssn))
             (DestinationVariable ?dsv))
)

(defquery search-message-flow-by-destination-service-name
  (declare (variables ?dsn))
             (DestinationVariable ?dsv))
)

(defquery search-StrategyforSkip
  (declare (variables ?stn ?mfn))
  ?stg<-(StrategyforSkip (SkippableToggle ?stn) (msgflow ?mfn) (substituteMF ?sbt))
)

(defquery search-message-flow-by-name
  (declare (variables ?mfn))
             (DestinationVariable ?dsv))
)

(deffunction FetchOutMF(?stf)
  (bind ?mf nil)
  (bind ?res (run-query* search-message-flow-by-source-service-name (fact-slot-value ?stf
            service)))
  (while (?res next)
    (bind ?mf (?res getObject mff)))
)
(bind ?r2 (run-query* search-StrategyforSkip (fact-slot-value ?stf name))
 (fact-slot-value ?mf name)))
(while (?r2 next)
 (bind ?tmf (?r2 getObject stg))
 (bind ?r3 (run-query* search-message-flow-by-name (fact-slot-value ?tmf substituteMF)))
(while (?r3 next)
 (bind ?rec (?r3 getObject mff))
 (modify ?mf (SourceService (fact-slot-value ?rec SourceService))
 (SourceVariable (fact-slot-value ?rec SourceVariable)))
))
)

(deffunction FetchInMF(?f)
 (bind ?mf nil)
 (bind ?sn (fact-slot-value ?f service))
 (bind ?res (run-query* search-message-flow-by-destination-service-name ?sn))
(while (?res next)
 (bind ?mf (?res getObject mff))
 (if (= (count-query-results search-validMF-by-MF-name (fact-slot-value ?mf name)) 0) then
 (assert (validMF (name (fact-slot-value ?mf name))
 (SourceService (fact-slot-value ?mf SourceService)) (SourceInstance nil)
 (DestinationService (fact-slot-value ?mf SourceService)) (DestinationInstance nil)
 (checked 1))))
))
)

(defquery search-previous-message-flows
 (declare (variables ?st))
 ?stf<-(STDependency (SkippableToggle ?st))
)

(defquery search-all-previous-message-flows
 ?stf<-(STDependency (SkippableToggle ?st))
)

(defquery search-held-Skippable-toggle-with-checked
 (declare (variables ?st))
 ?hstf<-(Hold (type SkippableToggle) (name ?st) (value ?va) (checked 1))
)

(defquery search-held-Skippable-toggle-with-minus-1
 (declare (variables ?st))
 ?hstf<-(Hold (type SkippableToggle) (name ?st) (value ?va) (checked -1))
)

(deffunction PreviousSTHeld(?hf)
 (bind ?st (fact-slot-value ?hf name))
 (bind ?res (run-query* search-previous-message-flows ?st))
(while (?res next)
 (bind ?stf (?res getObject stf))
 (bind ?plist (fact-slot-value ?stf previousST)))
)

(foreach ?p ?plist
  (if (not (eq ?p _end)) then
    (if (= (count-query-results search-held-Skippable-toggle-with-checked ?p) 0) then
      (modify ?hf (checked -1))
    (return FALSE))
  )
)
(return TRUE)

(deffunction TriggerNextST(?hf)
  (bind ?st (fact-slot-value ?hf name))
  (bind ?res (run-query* search-all-previous-message-flows))
  (while (?res next)
    (bind ?stf (?res getObject stf))
    (if (subsetp (create$ ?st) (fact-slot-value ?stf previousST)) then
      (bind ?r2 (run-query* search-held-Skippable-toggle-with-minus-1
        (fact-slot-value ?stf SkippableToggle)))
      (while (?r2 next)
        (bind ?f (?r2 getObject hstf))
        (modify ?f (checked 0))
      )
    )
  )
  (return 1)
)

(defrule Handle-Skipped-Message-Flow
  ?f1<-(Hold (type SkippableToggle) (name ?stn) (value TRUE) (checked 0))
  ?f2<-(SkippableToggle (name ?stn)(service ?tsn))
  (test (PreviousSTHeld ?f1))
  =>
    (bind ?mf1 (FetchInMF ?f2))
    (bind ?mf2 (FetchOutMF ?f2))
    (modify ?f1 (checked 1))
    (TriggerNextST ?f1)
)

(defquery search-IPAssignments-by-PS-CS-name
  (declare (variables ?psn ?csn))
  ?ipa<-(IPAssignment (ProspectService ?psn) (service ?csn))
)

(deffunction removeIPAssignment(?psaf)
  (bind ?r (run-query* search-IPAssignments-by-PS-CS-name (fact-slot-value ?psaf
    ProspectService) (fact-slot-value ?psaf service)))
  (while (?r next)
    (bind ?f (?r getObject ipa))
    (retract ?f)
  )
  (return 1)
)

(defrule Remove-duplicated-held-prospect
  ?f1<-(Hold (type ProspectService) (name ?psn) (value nil) (checked 1))
Alignment rules in the second stage:

defquery search-customizable-parameters-by-PS-name-and-variable
   (declare (variables ?s ?v))
   ?cpf<-((CustomizableParameter (Service ?s) (TypeParameter ?ip) (Variable ?v)))
)

defquery search-parameters-by-service-name-and-variable
   (declare (variables ?s ?v))
   ?cpf<-((Parameter (Service ?s) (Type ?t) (Variable ?v)))
)

deffunction validateMFVariables(?m ?s1 ?si1 ?s2 ?si2 ?v1 ?v2)
   (bind ?vmf (run-query* search-validMF-by-MF-name ?m))
   (while (?vmf next)
      (return FALSE)
   )
   (bind ?f1 nil)
   (bind ?t1 t1)
   (while (?r1 next)
      (bind ?f1 (?r1 getObject cpf))
   )
   (if (eq ?f1 nil) then
      (bind ?r2 (run-query* search-parameters-by-service-name-and-variable ?s1 ?v1))
      (while (?r2 next)
         (bind ?f1 (?r2 getObject cpf))
         (bind ?t1 (fact-slot-value ?f1 Type))
      )
   )
   else
      (bind ?r3 (run-query* search-customizable-parameter-assignment-by-PS-name-and-parameter-type
         ?s1 ?si1 (fact-slot-value ?f1 TypeParameter)))
      (while (?r3 next)
         (bind ?ipf (?r3 getObject ipa))
         (bind ?t1 (fact-slot-value ?ipf Assignment))
      )
   )
   (if (eq ?f1 nil) then (return FALSE))
   (bind ?f2 nil)
   (bind ?t2 t2)
   (while (?r1 next)
      (bind ?f2 (?r1 getObject cpf))
   )
   (if (eq ?f2 nil) then
      (bind ?r2 (run-query* search-parameters-by-service-name-and-variable ?s2 ?v2))
      (while (?r2 next)
         (bind ?f2 (?r2 getObject cpf))
         (bind ?t2 (fact-slot-value ?f2 Type))
      )
   )
) else
  (bind ?r3 (run-query*
    search-customizable-parameter-assignment-by-PS-name-and-parameter-type
    ?s2 ?si2 (fact-slot-value ?f2 TypeParameter)))
  (while (?r3 next)
    (bind ?ipf (?r3 getObject ipa))
    (bind ?t2 (fact-slot-value ?ipf Assignment)))
)
  (if (eq ?f2 nil) then (return FALSE))
  (if (eq ?t1 ?t2) then (return TRUE) else (return FALSE))
)

(defrule check-message-flow
  ?mf<- (MF (name ?mfname) (SourceService ?s1) (DestinationService ?s2) (SourceVariable ?v1) (DestinationVariable ?v2))
  ?ss<- (or (CS (service ?s1)) (PSAssignment (ProspectService ?s1)))
  ?ds<- (or (CS (service ?s2)) (PSAssignment (ProspectService ?s2)))
  =>
  (assert (validMF (name ?mfname) (SourceService ?s1) (SourceInstance (fact-slot-value ?ss service)) (DestinationService ?s2) (DestinationInstance (fact-slot-value ?ds service))))
)

(run)

Goal checking rule in the third stage:

(defquery search-all-hold-facts
  ?hld<- (Hold (type ?ty) (name ?na) (value ?va))
)

(defquery search-hold-facts-by-parameters
  (declare (variables ?ty))
  ?hld<- (Hold (type ?ty) (name ?na) (value ?va) (checked 1))
)

(defquery search-all-validMF-facts
  ?vmf<- (validMF (name ?na) (checked 1))
)

(defquery search-IPAAssignments-by-PS-name-and-CS-name
  (declare (variables ?psn ?csn))
)

(defrule check-feasible-assignment
  (forall (PS (service ?psn)) (Hold (type ProspectService) (name ?psn) (value ?csn) (checked 1)))
  (forall (MF (name ?mfn)) (validMF (name ?mfn) (checked 1)))
  (forall (AlternativeSelection (name ?asn)) (Hold (type AlternativeSelection) (name ?asn)))
  (forall (SkippableToggle (name ?stn)) (Hold (type SkippableToggle) (name ?stn) (checked 1)))
)
=>
  (printout t "There is a feasible assignment"  crlf)
  (showFeasibleAssignment 1)
)

(deffunction showFeasibleAssignment(?p)
  (bind ?p (create$_begin))
  (bind ?mr (run-query* search-all-validMF-facts))
  (while (?mr next)
    (bind ?m (?mr getObject vmf))
    (if (not (subsetp (create$ nil) (create$ (fact-slot-value ?m SourceService) (fact-slot-value ?m SourceInstance) (fact-slot-value ?m DestinationService) (fact-slot-value ?m DestinationInstance))) then
      (bind ?e (create$ (str-cat (fact-slot-value ?m SourceService) "," (fact-slot-value ?m SourceInstance)))
      (if (not (subsetp ?e ?p)) then
        (bind ?p (replace$ ?p 1 1 (create$ _begin (fact-slot-value ?m SourceService) ?e)))
      )
      (bind ?e (create$ (str-cat (fact-slot-value ?m DestinationService) "," (fact-slot-value ?m DestinationInstance)))
      (if (not (subsetp ?e ?p)) then
        (bind ?p (replace$ ?p 1 1 (create$ _begin (fact-slot-value ?m DestinationService) ?e)))
      )
    )
  )
  (bind ?r (run-query* search-hold-facts-by-parameters ProspectService))
  (while (?r next)
    (bind ?hrec (?r getObject hld))
    (if (or (and (not (subsetp (create$ (fact-slot-value ?hrec name) ?p) list)) (not (eq (fact-slot-value ?hrec name) ?p)))
      (printout t "Prospect Service:" (fact-slot-value ?hrec name) " binds " (fact-slot-value ?hrec value) crlf)
      (bind ?ipr (run-query* search-IPAssignments-by-PS-name-and-CS-name (fact-slot-value ?hrec name) (fact-slot-value ?hrec value)))
      (while (?ipr next)
        (bind ?ipe (?ipr getObject ipa))
        (printout t (fact-slot-value ?ipe Parameter) "=" (fact-slot-value ?ipe Assignment) "")
        crlf)
      )
    )
  )
  (bind ?r (run-query* search-hold-facts-by-parameters AlternativeSelection))
  (while (?r next)
    (bind ?hrec (?r getObject hld))
    (printout t "Alternative Selection:" (fact-slot-value ?hrec name) " is " (fact-slot-value ?hrec value) crlf)
  )
  (bind ?r (run-query* search-hold-facts-by-parameters SkippableToggle))
  (while (?r next)
    (bind ?hrec (?r getObject hld))
    (printout t "Skippable Toggle:" (fact-slot-value ?hrec name) " is " (fact-slot-value ?hrec value) crlf)
  )
  (return 1) )