Hindawi Publishing Corporation
ISRN Software Engineering
Volume 2013, Article ID 408267, 26 pages
http://dx.doi.org/10.1155/2013/408267

Research Article

Web Services Conversation Adaptation Using Conditional Substitution Semantics of Application Domain Concepts

Islam Elgedawy

Computer Engineering Department, Middle East Technical University, Northern Cyprus Campus, Guzelyurt, Mersin 10, Turkey

Correspondence should be addressed to Islam Elgedawy; elgedawy@metu.edu.tr

Received 5 June 2013; Accepted 12 July 2013

Academic Editors: A. Lastovetsky, G. Petrone, and G. Saake

Internet of Services (IoS) vision allows users to allocate and consume different web services on the fly without any prior knowledge regarding the chosen services. Such chosen services should automatically interact with one another in a transparent manner to accomplish the required users' goals. As services are chosen on the fly, service conversations are not necessarily compatible due to incompatibilities between services signatures and/or conversation protocols, creating obstacles for realizing the IoS vision. One approach for overcoming this problem is to use conversation adapters. However, such conversion adapters must be automatically created on the fly as chosen services are only known at run time. Existing approaches for automatic adapter generation are syntactic and very limited; hence they cannot be adopted in such dynamic environments. To overcome such limitation, this paper proposes a novel approach for automatic adapter generation that uses conditional substitution semantics between application domain concepts and operations to automatically generate the adapter conversion functions. Such conditional substitution semantics are captured using a concepts substitutability enhanced graph required to be part of application domain ontologies. Experiments results show that the proposed approach provides more accurate conversation adaptation results when compared against existing syntactic adapter generation approaches.

1. Introduction

Internet of Services (IoS) vision enables users (i.e. people, businesses, and systems) to allocate and consume the required computing services whenever and wherever they want in a context-aware seamless transparent manner. Hence, chosen services automatically interact with one another in a transparent manner to accomplish the required users' goals. Middleware software plays an essential role in supporting such interactions, as it hides services heterogeneity and ensures their interoperability. Middleware enables services to locate one another without a priori knowledge of their existences and enables them to interact with one another even though they are running on different devices and platforms [1]. Services interactions are conducted via exchanging messages. A conversation message indicates the operation to be performed by the service receiving the message. A sequence of messages exchanged between services to achieve a common goal constitutes what is known by a conversation pattern. A set of conversation patterns is referred to as a service conversation. However, services may use different concepts, vocabularies, and semantics to generate their conversation messages, raising the possibility for having conversation incompatibilities. Such incompatibilities must be automatically resolved in order to enable services conversations on the fly. This should be handled by a conversation adapter created on the fly by the middleware, please refer to Section 2.2 for more information about conversation adapters.

In general, in order to create a conversation adapter, first we have to identify the possible conversation incompatibilities and then try to resolve the incompatibilities using the available conversation semantics, which are constituted from service semantics (such as service external behavior, encapsulated business logic, and adopted vocabulary) and application domain semantics (such as concepts relations and domain rules). If solutions are found, the adapter can be created; otherwise the conversations are labelled as unadaptable, and the corresponding services cannot work together. Hence, we argue that in order to automatically generate conversation adapters the following prerequisites must be fulfilled.

(i) First, we require substitution semantics of application domain concepts and operations to be captured in application domain ontologies in a context-sensitive manner, as such semantics differ from one context to another in the same application domain, for example, the concepts `Hotel` and `Resort` could be substitutable in some contexts and not substitutable in others. Hence, capturing substitution semantics and its corresponding conversion semantics in a finite context-sensitive manner is mandatory to guarantee the adapter functional correctness, as these conversion semantics provide the basic building blocks for generating converters needed for building the required adapters.

(ii) Second, we require services descriptions to provide details about the supported conversation patterns (that is the exchanged messages sequences), such as the conversation context, the supported operations, and the supported invocation sequences. Such information must be captured in a machine-understandable format and must be based on the adopted application-domain ontology vocabulary.

(iii) Finally, as different conversation patterns could be used to accomplish the same business objective, different types of mappings between the conversation patterns operations must be automatically determined (whether it is many-to-many, or one-to-one, and etc). Such operations mappings are essential for determining the required adapter structure.

Unfortunately, existing approaches for adapter generation (such as the ones discussed in [2–9]) do not fulfill the mentioned prerequisites; hence they are strictly limited and cannot be adopted in dynamic environments implied by the IoS vision. More details and discussion about these approaches are given in the related work section (Section 3).

To overcome the limitations of the existing adaptation approaches, this paper proposes a novel approach for automatic adapter generation that is able to fulfill the above prerequisites by adopting and integrating different solutions from our previous research endeavors discussed in [10–16]. The proposed approach successfully adapts both signature and protocol conversation incompatibilities in a context-sensitive manner. First, we adopt the metaontology proposed in [13, 14, 16] to capture the conversion semantics between application domain concepts in a context-sensitive manner using the Concepts Substitutability Enhanced Graph (CSEG) (details are given in Section 4). Second, we adopt the G^+ model [10, 16] to semantically capture the supported service conversation patterns using concepts and operations defined in CSEG (details are given in Section 5). Third, we adopt the context matching approach proposed in [12] to match conversation contexts and adopt a Sequence Mediation Procedure (SMP) proposed in [10, 15] to mediate between different exchanged messages sequences (details are given in Section 2). Fourth, the proposed approach generates the conversation patterns from the services G^+ model then matches these patterns using context matching and SMP procedures to find the operations mappings, which determine the required adapter structure, and then generates converters between different operations using the concepts substitution semantics captured in the CSEG. Finally, it builds the required adapter from the generated converters between conversation operations (i.e. messages). Each couple of conversation patterns should have their own corresponding adapter. Experiments results show that the proposed approach provides more accurate conversation adaptation results when compared against existing syntactic adapter generation approaches. We believe the proposed automated approach helps in improving business agility and responsiveness and of course establishes a solid step towards achieving the IoS vision.

1.1. Contributions Summary. We summarize paper contributions as follows.

(i) We propose a novel approach for automatic service conversation adapter generation that uses conditional substitution semantics between application domain concepts and operations in order to resolve conversation conflicts on the fly, in a context-aware manner.

(ii) We propose to use a complex graph data structure, known as the Concepts Substitutability Enhanced Graph (CSEG), which is able to capture the aggregate concept substitution semantics of application domain concepts in a context-sensitive manner. We believe CSEG should be the metaontology for every application domain.

(iii) We propose a new way for representing a service behavior state that helps us to improve the matching accuracy and propose a new behavior matching procedure known as Sequence Mediator Procedure (SMP) that can match states in a many-to-many fashion.

(iv) We propose an approach for service operation signature adaptation using CSEG semantics.

(v) We propose an approach for service conversation adaptation using CSEG semantics and SMP.

The rest of the paper is organized as follows. Section 2 provides some background regarding service conversation management, conversation adaptation, application domain representation, and concepts substitutability graph. Section 3 provides the related work discussions in the areas of conversation adaptation and ontology mapping. Section 4 provides an overview on the adopted metaontology and its evolution. Section 5 proposes the adopted conversation model and describes how to extract the corresponding behavior model. Section 6 proposes the adopted approach for signature adaptation, while Section 7 proposes the adopted approach for conversation protocol adaptation. Section 8 proposes the adopted approach and algorithms for automatic adapter generation. Section 9 shows the various verification experiment and depicts results. Finally, Section 11 concludes the paper and discusses future work.

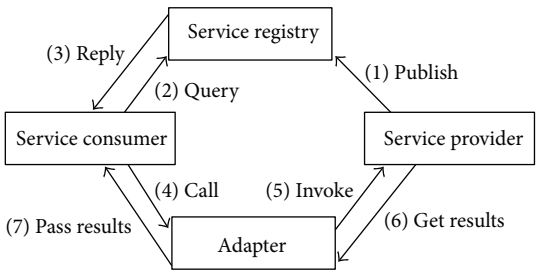

Figure 1: Service-Oriented Architecture with Adapters.

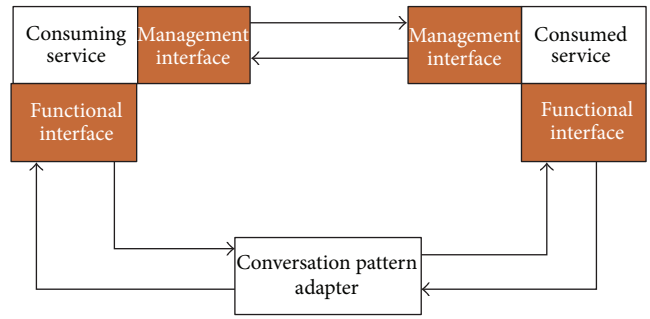

Figure 2: Conversation Management Architecture.

2. Background

This section provides some basic principles regarding conversation management and application domain representation needed to understand the proposed automatic adapter generation approach.

2.1. Service Conversation Management. Basically, we require each web service to have two types of interfaces: a functional interface and a management interface [17, 18]. The functional interface provides the operations others services can invoke to accomplish different business objectives, while the management interface provides operations to other services in order to get some information about the service internal state that enables other services to synchronize their conversations with the service, such as operations for conversation start and end, supported conversation patterns, and supported application domain ontologies. However, to generate the right adapters, we need first to know which conversation patterns will be used before the conversation started. Therefore, we require from the consuming service to specify which conversation pattern it will use and which conversation pattern required from the consumed service before starting the conversation. This information is provided to the consuming service via the matchmaker or the service discovery agent, as depicted in Figure 1. Figure 1 indicates consuming services call conversation adapters to accomplish the required operations, and in turn the adapter invokes the corresponding operations from the service providers side and does the suitable conversions between exchanged messages.

Once the consuming service knows which conversation patterns are required, it needs to communicate this information with the consumed service in order to build the suitable adapter. This could be achieved by invoking the conversation management operations defined in the management interface of the consumed service, as depicted in Figure 2. The figure indicates that the consuming service calls the consumed service management interface to specify the required conversation pattern; once it gets the confirmation, it starts the conversation and performs the conversation interactions via the conversation adapter. The adapter in turn will invoke the needed operations from the functional interface of the consumed service. This forms what we define as the *conversation management architecture*, in which each service is capable of monitoring and controling its conversations and can synchronize with other services via management interfaces.

Specifying the required conversation patterns in advance has another benefit that services could determine the correctness of the interactions during the conversation and that if a service received any operation invocation request not in the specified conversation pattern or even in the wrong order, it could reject the request and reset the conversation. It is important to note that management interfaces should be created according to a common standard such as Web Services Choreography Interface (WSCI) [19].

2.2. Conversation Adaptation. Conversations incompatibilities are classified into signature incompatibilities and protocol incompatibilities [20, 21]. Signature incompatibilities arise when the operation to be performed by the receiving service is either not supported or not described using the required messaging schema (such as using a different number of input and output concepts, different concepts names, and different concepts types). On the other hand, protocol incompatibilities arise when interacting services expect a different ordering for the exchanged message sequences. An example for signature incompatibility occurs when one service needs to perform an online payment via operation `PayOnline` that has input concepts `CreditCard`, `Amount`, and `Currency`, and the output concept `Receipt`. The `CreditCard` concept contains the card information such as card holder name, card number, card type, and card expiration date while the `Receipt` concept contains the successful transaction number. Continuing with our example, another service performs online payment by invoking operation `PaymentRequest` that has one input concept `Payment` (which contains all the payment details) and one output concept `Confirmation`, which contains the transaction number. With purely syntactical matching between operations signatures, the first service cannot invoke the second service in spite of its ability to perform the required payment operation. An example for protocol incompatibility occurs when one service needs to perform a purchase operation and expects to send a message containing user details first and then another message containing purchase-order details, while the other interacting service is receiving the purchase-order details first and then the user details. One well-known approach for handling conversation incompatibilities is through the use of conversation

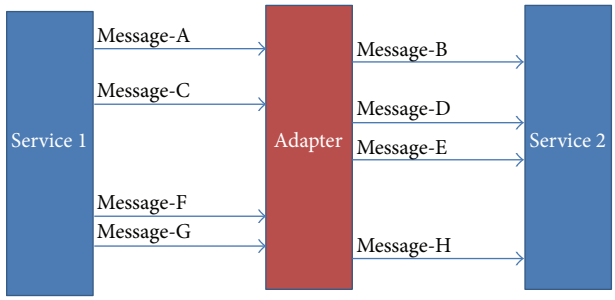

Figure 3: Conversation customization via an adapter.

adapters [2–6, 11]. A conversation adapter is the intermediate component between the interacting services that facilitates service conversations by converting the exchanged messages into messages "understandable" by the interacting services, as indicated in Figure 3.

Figure 3 shows an example of interactions between two incompatible services via a conversation adapter. Figure 3 shows that mapping between messages could be of different types (i.e. one-to-one, one-to-many, many-to-one, and many-to-many). For example, the adapter converts `Message-A` into `Message-B`, while it converts `Message-C` into the sequence consisting of `Message-D` and `Message-E`, and finally it converts the sequence consisting of `Message-F` and `Message-G` into `Message-H`. In other words, the adapter performs conversation customization and translation. The adapter can convert one message into another message or into another sequence of messages. It can also convert a sequence of messages into a single message or into another sequence of messages. Creating conversation adapters manually is a very time-consuming and costly process, especially when business frequently changes its consumed services, as in the IoS vision. This creates a need for automatic generation for web services conversations adapters, in order to increase business agility and responsiveness, as most of the business services will be discovered on the fly.

Automatic conversation adaptation is a very challenging task, as it requires understanding of many types of semantics including user semantics, service semantics, and application-domain semantics. All of these types of semantics should be captured in a machine-understandable format so that the middleware can use them to generate the required conversation adapters. One way of capturing different types of semantics in a machine-understandable format is using ontologies. Ontologies represent the semantic web architecture layer concerned with domain conceptualization. They are created to provide a common shared understanding of a given application domain that can be communicated across people, applications, and systems [10]. Ontologies play a very important role in automatic adapter generation, as they provide the common reference for resolving any appearing semantic conflicts. Therefore, we argue that the adopted application domain ontologies must be rich enough to capture different types of semantics in order to be able to resolve different conversation conflicts in a context-aware semantic manner. We argued in our previous work [10, 12, 13] that ontologies defined as a taxonomy style are not rich enough to capture complex types of semantics; hence more complex ontology models must be adopted. Therefore, we proposed in [10, 13, 14] to capture relationships between application domain concepts as a multidimensional hypergraph rather than a simple taxonomy; more details will be given in Section 4.

2.3. Application Domain Representation. Business systems use application domain ontologies in their modelling and design in order to standardize their models and to facilitate systems interaction, integration, evolution, and development. This is because application domain ontologies provide a common shared understanding of application domains that can be communicated across people, applications, and systems. Ontologies represent the semantic web architecture layer concerned with domain conceptualization; hence application domain ontology should include descriptions of the domain entities and their semantics, as well as specify any attributes of domain entities and their corresponding values. An ontology can range from a simple taxonomy to a thesaurus (words and synonyms), to a conceptual model (where more complex relations are defined), or to a logical theory (where formal axioms, rules, theorems, and theories are defined) [22, 23].

It is important to note the difference between application-domain ontologies and service modelling ontologies. In Application-domain ontologies the vocabulary are needed for describing the domain concepts, operations, rules, and so forth. Application-domain ontologies could be represented by existing semantic web standards, such as Web Ontology Language (OWL 2.0) [24]. On the other hand, service modelling ontologies provide constructs to build the service model in a machine-understandable format; such constructs are based on the vocabulary provided by the adopted application domain ontologies. Web Services Modelling Ontology (WSMO) [25], Web Ontology Language for Services (OWL-S) [26], and Semantic Annotations for Web Services Description Language (SAWSDL) [27] are examples of existing service modelling ontologies. The conversation modelling problem has attracted many research efforts in the areas of SOC and agent communication (such as in [28–30]). Additionally, there are some industrial standards for representing service conversations, such as Web Services Choreography Interface (WSCI) [19] for modelling service choreography and Web Services Business Process execution Language (WS-BPEL) [31] for modelling service orchestration. In this paper, we preferred to conceptually describe our conversation and applications-domain models without being restricted to any existing standards. However, any existing standard that is sufficiently rich to capture the information explained below will be suitable to represent our models.

In general, there are two approaches that can be adopted for application domain conceptualization: the single-ontology approach and the multiple-ontology approach. The single-ontology approach requires every application domain to be described using only one single ontology, and everyone in the world has to follow this ontology. The multiple-ontology approach allows the application domain to be

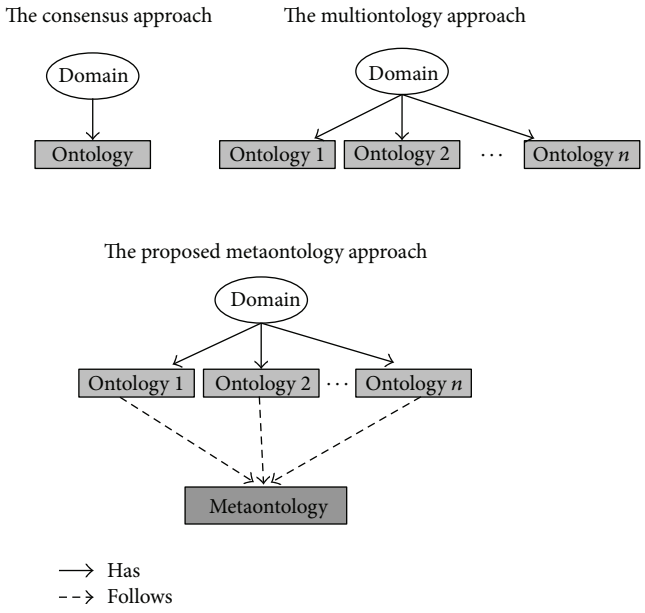

Figure 4: Approaches for application domain conceptualization.

described by different ontologies such that everyone can use a different preferred ontology. As we can see both approaches have serious practicality concerns if adopted, the single-ontology approach requires reaching world consensus for every application domain conceptualization, which is far from feasible. On the other hand, the multiple-ontology approach requires determining the mappings between different ontologies in order to be able to resolve any appearing incompatibilities, which is not feasible approach when the number of ontologies describing a given application domain is big. Ontologies incompatibilities result due to many reasons. For example, two different concepts could be used to describe the same entity, or the same concept could be used to represent different entities. An entity could appear as an attribute in a given ontology and appear as a concept in other ontology, and so forth [32].

The ontology mapping process is very complex, and it requires identification of semantically related entities and then resolving their appearing differences. We argued before in [10] that any appearing conflicts should be resolved according to the defined semantics of involved application domains as well as the semantics of the involved usage contexts. When services in the same domain adopt different ontologies, ontology mapping becomes crucial for resolving conversation incompatibilities. To maintain the flexibility of application domain representation without complicating the ontology-mapping process, we propose to adopt a metaontology approach, which is a compromise between the consensus and multiple-ontology approaches, as depicted in Figure 4. Figure 4 shows the difference between the single-ontology, multiple-ontology, and metaontology approaches. Adopting a metaontology approach for application domain conceptualization provides users with the flexibility to use multiple ontologies exactly as in the multiple-ontology approach, but it requires ontology designers to follow a common structure indicating the entities and the types of semantics to be captured, which indeed simplifies the ontology mapping process. Furthermore, having a common structure ensures that all application domain ontologies capture the same types of semantics; hence we can systematically resolve any appearing conflicts; more details are given in Section 4.

2.4. Concepts Substitutability Graph (CSG). As we indicated before that we adopt a metaontology approach for describing application domain ontologies. Following the separation of concerns design principle, we argue that the metaontology should consist of two layers: a schematic layer and a semantic layer [10, 13, 14]. The schematic layer defines which application domain entities need to be captured in the ontology, which will be used to define the systems models and their interaction messages. The semantic layer defines which entities semantics need to be captured in the ontology.

In the metaontology schematic layer, we propose to capture the application domain concepts and operations. *An application domain concept* is represented as a set of features defined in an attribute-value format. *An application domain operation* is represented as a set of features defined in an attribute-value format. In addition it has a set of input concepts, a set of output concepts, a set of preconditions and a set of postconditions. The preconditions are over the input concepts and must be satisfied before the operation invocation. The postconditions are over the output concepts and guaranteed to be satisfied after the operation finishes its execution.

A *conversation message* is basically represented by an application domain operation. A sequence of conversation messages constitutes a *conversation pattern*, which describes an interaction scenario supported by the service. Each conversation pattern has a corresponding *conversation context* that is represented as a set of preconditions and a set of postconditions. The context preconditions are the conditions that must be satisfied in order to be able to use the conversation pattern, while the context postconditions are the conditions guaranteed to be satisfied after the conversation pattern finishes its execution. A set of conversation patterns constitutes the *service conversation model*. In general, service conversation models are not necessarily linear. However, linear models (in which interactions are described as a sequence of operations) could be extracted from the nonlinear models (in which interactions are described as a graph of operations) by tracing all possible paths in the nonlinear model. During runtime, having linear conversation patterns provides faster performance than subgraph matching approaches, as graph paths are analyzed and enumerated (which could be performed offline) only once when a service is published and not repeated every time a matching process is needed as in subgraph matching approaches; additional details about this approach may be found in [10].

In our previous work [10, 12, 15, 16], we argued that concept substitutability should be used for concept matching that our approach maps a concept `A` to a concept `B` only if the concept `A` can substitute the concept `B` in the involved context without violating any conditions in the involved context

Table 1: A Part of CSG segment for `CargoTransportation` operation, adapted from [10].

From scope	To scope	Conversion function	Substitution constraints
Cargo.Det	Freight.Det	Freight.Det = Cargo.Det	
Freight.Det	Cargo.Det	Cargo.Det = Freight.Det	

Credit.Period	Payment.Type	IF (Credit.Period > 0) THEN
		Payment.Type = Credit
		ELSE
		Payment.Type = Cash
		END IF
			Credit.Period ≥ 0

Payment.Type	Credit.Period	IF (Payment.Type = Credit) THEN
		Credit.Period ∈ {15, 30, 45, 60}
		ELSE
		Credit.Period = 0
		END IF
			Payment.Type ∈ {Credit, Cash}

or any rule defined in the application domain ontology. Matching concepts based on their conditional substitutability is not a straightforward process due to many reasons. First, there exist different types of mappings between concepts such as one-to-one, one-to-many, many-to-one, and many-to-many mappings, which require taking concept aggregation into consideration. For example, the `Address` concept could be substituted by a composite concept constituted from the `Country`, `State`, `City`, and `Street` concepts, as long as the usage context allows such substitution. Second, concept substitution semantics could vary according to the logic of the involved application domain operation; hence substitution semantics should be captured for each operation separately. Third, concept substitutability should be determined in a context-sensitive manner and not via generic schematic relations in order to be able to check if such concept substitution violates the usage context or not. In order to fulfill these requirements and capture the concept conditional substitution semantics in a machine-understandable format, we propose to use a complex graph data structure, known as the Concepts Substitutability Enhanced Graph (CSEG), which is able to capture the aggregate concept substitution semantics in a context-sensitive manner with respect to every application domain operation. Hence, we propose the metaontology semantic layer to include CSEG as one of its basic constructs.

CSEG extends the Concept Substitutability Graph (CSG) previously proposed in [10], which captures only the bilateral conditional substitution semantics between concepts. CSEG captures both bilateral as well as aggregate conditional substitution semantics of application domain concepts. Hence, we first summarize CSG graph depicted in Figure 5 and then discuss CSEG in more details. Figure 5 indicates that CSG consists of segments, where each segment captures the substitution semantics between application domain concepts with respect to a given application domain operation. For every pair of concepts the following are defined: substitutable attributes and their substitution constraints, conversion functions, and operator mapping matrices. The substitution context is represented by a set of substitution constraints that must be satisfied during substitution in order to have a valid substitution. A CSG captures the concepts functional substitution semantics at the scope level (a scope is defined by a combination of concept C_i and attribute $attr_k$ with the form $C_i.attr_k$), and not at the concept level only. This is needed because attributes with similar names could have different semantics when they are used to describe different concepts.

The proposed concept matching approach maps a concept `A` to a concept `B` only if the concept `A` can substitute the concept `B` in the involved context without violating any conditions in the involved context or any rule defined in the application domain ontology. This is done by defining the conditional substitution semantics of application domain concepts in application domain ontologies and then using such conditional semantics to resolve appearing incompatibilities by checking if the conditions representing the involved context satisfy the required substitution conditions between concepts before performing any concepts substitutions. In other words, concept mapping is conditional and not generic that concept mapping will be only valid in the contexts satisfying the required substitution conditions. Table 1 shows an example of a segment of a CSG in the logistics application domain that corresponds to the *CargoTransportation* operation. A row represents an edge in a segment in the substitutability graph. For example, the first row indicates the existence of an edge in the CSG going from the scope `Cargo.Det` (the cargo details) into the scope `Freight.Det` (the freight details). This edge has also the corresponding substitution constraint as well as conversion function. Substitutability semantics defined in CSG can be seen as conditional conversion semantics, as it allows conversion only when the substitution constraints are valid. Also it provides the details of how to perform such conversion via conversion functions and operator mapping matrices.

CSG managed to provide a conditional ontology mapping approach that is able to resolve appearing concepts incompatibilities in a context-sensitive manner (more details will be given later in Section 4.1). Unfortunately, this approach cannot resolve cases requiring concept aggregation, in which one concept can substitute for a group of concepts and vice versa. For example, in the signature incompatibilities example given before, this proposed approach can resolve the conflict between the `Confirmation` and `Receipt` concepts but it cannot resolve the conflict between the input concepts, as the

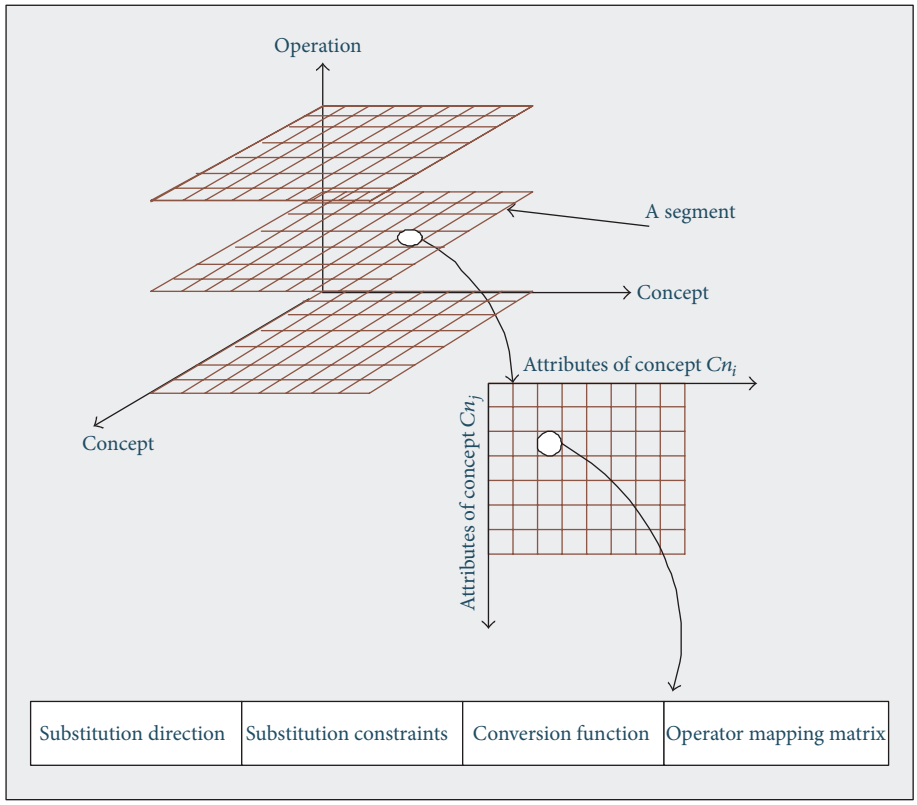

Figure 5: Concepts substitutability graph.

`CreditCard`, `Amount`, and `Currency` concepts need to be aggregated in order to substitute the Payment concept. To overcome such a limitation, our work in [13, 14] extended CSG graph to capture aggregate conditional substitution semantics of application domain concepts. The new graph is known as the Concepts Substitutability Enhanced Graph (CSEG). CSEG uses the notion of substitution patterns that indicate the mapping types (such as one-to-one, one-to-many, many-to-one, and many-to-many) between application domain concepts with respect to every application domain operation. More details about CSEG are given in Section 4.

3. Related Work

This section discusses two main related areas for our work. First, we discuss related work in the area of conversation adaptation and then discuss the related work in the area of ontology mapping that shows different approaches for resolving conflicts.

3.1. Conversation Adaptation. The problem of synthesizing adapters for incompatible conversations has been studied by many researchers in the area of SOC such as the work described in [2–8, 11] and earlier in the area of component-based software engineering such as the work described in [9]. We can broadly classify these efforts into three categories: manual such as work in [2, 3, 7, 8], semiautomated such as work in [4, 9], and fully automated solutions such as work in [5, 6, 11].

The manual approaches provide users with guidelines to identify conversation incompatibilities and propose templates to resolve identified mismatches. for example, work in [7] tries to mediate between services based on signatures without taking into consideration services behavior, while work in [8] requires adapter specification to be defined manually. On the other hand, work in [3] proposes a method for creating adapters based on mismatch patterns in service composition; however they adopt a syntactic approach for comparing patterns operations, which of course cannot work if different operations sequences or different operation signatures are used.

The semiautomated approaches generate the adapters after receiving some inputs from the users regarding conversation incompatibilities resolution. The fully automated approaches generate the adapters without human intervention provided that conversation models are created according to some restrictions to avoid having signature incompatibilities and protocol deadlocks. Manual and semiautomated approaches are not suitable for dynamic environments due to the following reasons. First, they require experts to analyze the conversation models and to design solutions for incompatibilities resolution, resulting in high financial costs and time barriers for adapter development. This creates obstacles for achieving on-demand customizations and

minimizes users' flexibility and agility, especially when users tend to use services for a short term and to change services frequently. Second, the number of services and users in dynamic environments is rapidly growing, which diminishes any chances for having predefined manual customizations policies. Therefore, to have on-demand conversation customizations, adapters should be created automatically. To achieve such a vision, we argue that the middleware should be enabled to automatically create such adapters to avoid any human intervention and to ensure smooth services interoperability. Unfortunately, existing automatic adapter generation approaches are strictly limited [11, 20] as they require no mismatch at the services interface level; otherwise the conversations are considered unadaptable. We argue that such syntactic approaches are not suitable for dynamic environments as service heterogeneity is totally expected in dynamic environments. Hence, conversation incompatibilities should be semantically resolved without any human intervention. Therefore, in this paper, we capture both service conversations and application domain semantics in a machine-understandable format such that we can automatically resolve appearing conflicts without human intervention; more details are given in Sections 5, 6, 7, and 8.

3.2. Ontology Mapping. Concepts incompatibilities arise when business systems adopt different application domain ontologies during their interactions. One approach for resolving such incompatibilities is using an intermediate ontology mapping approach that transforms the exchanged concepts into concepts understandable by the interacting systems. Unfortunately, existing approaches for ontology mapping are known for having limited accuracy. This is because such approaches are basically based on generic schematic relations (such as Is-a and Part-of) and ignore the involved usage context as well as the logic of the involved operation.

We argue that the ontology mapping process could be tolerated if the number of ontologies representing a given application domain is small and if there exists a systematic straightforward approach in finding the mappings between semantically related entities. Indeed, in real life, we are expecting the number of ontologies describing a given application domain to be small, as people tend to cluster and unify their understanding. Of course, we are not expecting them to cluster into one group that uses a single ontology; however it is more likely they will cluster into few groups using different ontologies. To fulfil the second requirement, many research efforts have been proposed to provide systematic straightforward approaches for ontology mapping such as [33–37]. A good survey about existing ontology mapping approaches could be found in [22]. For example, work in [33] proposed a language for specifying correspondence rules between data elements adopting a general structure consisting of general ordered labelled trees. Work in [34] developed a translation system for symbolic knowledge. It provides a language to represent complex syntactic transformations and uses syntactic rewriting (via pattern-directed rewrite rules) and semantic rewriting (via partial semantic models and some supported logical inferences) to translate different statements. Its inferences are based on generic taxonomic relationships. Work in [35] provides an ontology mapping approach based on tree structure grammar. They try to combine between internal concept structure information and rules provided by similarity languages. Work in [36] proposed a metric for determining objects similarity using hierarchical domain structure (i.e. Is-a relations) in order to produce more intuitive similarity scores. work in [37] determines the mapping between different models without translating the models into a common language. Such mapping is defined as a set of relationships between expressions over the given model, where syntactical inferences are used to find matching elements. As we can see, existing ontology mapping approaches try to provide a general translation model that can fit in all contexts using generic schematic relations (such as Is-a and Part-of relations), or depending on linguistic similarities to resolve conflicts. We argue that such approaches cannot guarantee high accuracy mapping results in all contexts [10]. Simply because such generic relations and linguistic rules could be sources of ambiguities, which are resulting from the actual domain semantics themselves. For example, the concept `Islam` could be a name of a religion or a name of a person and could be applied for both males and females. Another example, the `Resort` concept could be related to the `Hotel` concept using the *Is-a* relation, however, we cannot substitute the concept `Resort` by the concept `Hotel` in all context. Such ambiguities can be resolved only by taking the involved contexts into consideration. Hence, we argue that in order to guarantee the correctness of the mapping results, ontology mappings should be determined in a customized manner according to the usage context as well as the logic of the involved application domain operation (i.e. the transaction needs to be accomplished by interacting systems or users). Next section provides our approach for fulfilling these requirements.

4. A Context-Sensitive Metaontology for Applications Domains

Unlike CSG only capturing bilateral substitution semantics between application domain concepts, CSEG is able to capture the aggregate concept conditional substitution semantics in a context-sensitive manner to allow a concept to be substituted by a group of concepts and vice versa. This is achieved by introducing the notion of substitution patterns. CSEG consists of a collection of segments, such that each segment is corresponding to one of the application domain operations. Each segment consists of a collection of substitution patterns corresponding to the operation input and output concepts. Each substitution pattern consists of a scope, a set of substitution conditions, and a conversion function, as depicted in Figure 6.

Figure 6 indicates the substitution patterns corresponding to a given operation input and output concepts. For example, the input concept `C1` has three substitution patterns. The first pattern indicates that the concepts `C5`, `C6`, and `C7` can substitute the concept.

Table 2: An example for operation substitution patterns.

Operation	Concepts	Scope	Conversion function	Substitution condition
PayOnline	Input: CreditCard Input: Amount Input: Currency	Payment	Payment.Method = Credit Payment.Details = CreditCard.Details Payment.Currency = Currency Payment.CreditAmt = Amount	CreditCard.Details ≠ NULL Amount >0 Currency ≠ NULL
	Output: Receipt	Confirmation	Receipt = Confirmation	Confirmation ≠ NULL

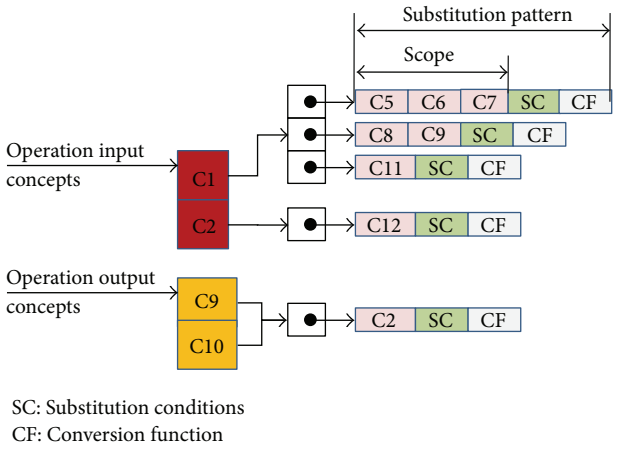

Figure 6: An example for a CSEG segment.

A substitution pattern scope is a set of concepts that contains at least one application domain concept. A substitution condition is a condition that must be satisfied by the conversation context in order to consider such substitution as valid. A conversion function indicates the logic needed to convert the scope into the corresponding operation concepts or vice versa. Of course, instead of writing the conversion function code, we could refer to a service or a function that realizes it using its corresponding Uniform Resource Identifier (URI). A substitution pattern could correspond to a subset of concepts. For example, a substitution pattern for a subset of input concepts represents the set of concepts (i.e. the pattern scope) that can substitute such subset of input concepts, while a substitution pattern for a subset of output concepts represents the set of concepts that can be substituted by such subset of output concepts.

Table 2 shows an example of an input and an output substitution patterns for `PayOnline` operation. The input pattern indicates that `CreditCard`, `Amount`, and `Currency` concepts can be replaced by the `Payment` concept only if credit card details and the currency are not null and the amount is greater than zero. The output pattern indicates we can substitute the concept `Confirmation` by the concept `Receipt` only when conformation is not null. As we can see, substitution patterns are valid only in the contexts satisfying their substitution conditions. Of course instead of writing the conversion function code, we could refer to the URI of its realizing web service. Another advantage of using CSEG is that it systemizes the ontology mapping process, as all that needs to be done is to add the suitable substitution patterns between the ontologies concepts with respect to every domain operation. The mappings between the operations will be automatically determined based on the satisfiability of their pre- and postconditions (details are given later). In the next section, we will show how CSEG substitution patterns are used to resolve concepts incompatibilities.

Indeed CSEG could be represented in many different ways differing in their efficiency. However, we prefer to represent it in an XML format as XML is the industrial de facto standard for sharing information. In case the XML file becomes very large, it should be compressed with a query-aware XML compressor and then accessed in its compressed format; more details about this approach could be found in [38]. For example, the substitution patterns depicted in Table 2 could be represented in XML format as shown in Listing 1.

4.1. Resolving Concepts Conflicts via Substitutability Semantics. CSEG contains the information indicating which concepts are substitutable with respect to every application domain and also indicates the corresponding conversion functions. Hence, concepts mapping is determined by checking if there exists a sequence of transformations (i.e. substitution patterns) that can be carried out to transform a given concept or a group of concepts into another concept or group of concepts. This is done by checking if there exists a path between the different concepts in the CSEG segment corresponding to the involved application domain operation. Having no path indicates there is no mapping between such concepts according to the logic of the involved operation. We identify the concepts as reachable if such path is found. However, in order to consider reachable concepts as substitutable, we have to make sure that the usage context is not violated by such transformations. This is done by checking if the conditions of the usage context satisfy the substitution conditions defined along the identified path between the concepts. The concepts are considered substitutable only when the usage context satisfies such substitution conditions. Determining condition satisfiability is a tricky process, as conditions could have different scopes (i.e. concepts appearing in the conditions) and yet could be satisfiable; for example, the condition (`Capital.Name = Cairo`) satisfies the condition (`Country.Name = Egypt`) in spite of having a different scope. Unfortunately, such cases cannot be resolved by existing condition satisfiability approaches [39, 40] as they

```
<Root>
  <Operation name = "PayOnline">
    <Inputs>
      <Concepts names = {"CreditCard, Amount, Currency"} >
        <SubstitutionPattern>
          <Scope>
            <Concepts names = {"Payment"}/>
          </Scope>
          <Condition>
            (CreditCard.Details ≠ NULL) and (Amount >0)
             and (Currency ≠ NULL)
          </Condition>
          <ConversionFunction>
              "http://example.org/URI/path/convert1.java"
          </ConversionFunction>
        </SubstitutionPattern>
      </Concepts>
    </Inputs>
    <Outputs>
      <Concepts names = {"Receipt"} >
        <SubstitutionPattern>
          <Scope>
            <Concepts names = {"Confirmation"}/>
          </Scope>
          <Condition>
            (Confirmation ≠ NULL)
          </Condition>
          <ConversionFunction>
              "http://example.org/URI/path/convert2.java"
          </ConversionFunction>
        </SubstitutionPattern>
      </Concepts>
    </Outputs>
  </Operation>
  -
</Root>
```

Listing 1: An XML representation for a CSEG segment.

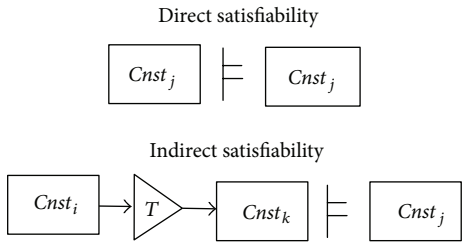

Figure 7: Direct versus indirect condition satisfiability.

are syntactic and require the conditions to have the same scope in order to be examined.

To handle such cases, first we differentiate between the two cases as follows. When satisfiable conditions have the same scope, we identify this case as "*condition direct satisfiability*" which should be determined using existing condition satisfiability approaches. When satisfiable conditions have different scopes, we identify such case as "*condition indirect satisfiability*" which should be determined via generation of intermediate condition, as depicted in Figure 7. The figure indicates that conditions indirect satisfiability implies transforming the first condition into another intermediate condition via a transformation (T) such that the intermediate condition directly satisfies the second condition. Transformation (T) must not violate any condition in the usage context. We determine conditions indirect satisfiability between two different conditions as follows. First, we check if the conditions scopes are reachable. Second, if the scopes are reachable, we use the conversion functions defined along the path to convert the first scope into the second scope and use the obtained values to generate another intermediate condition with the same scope of the second condition. Third, we check if the intermediate condition satisfies the second condition using existing syntactic condition satisfiability approaches. Finally, if the intermediate condition satisfies the second condition, we check if the conditions of the usage context

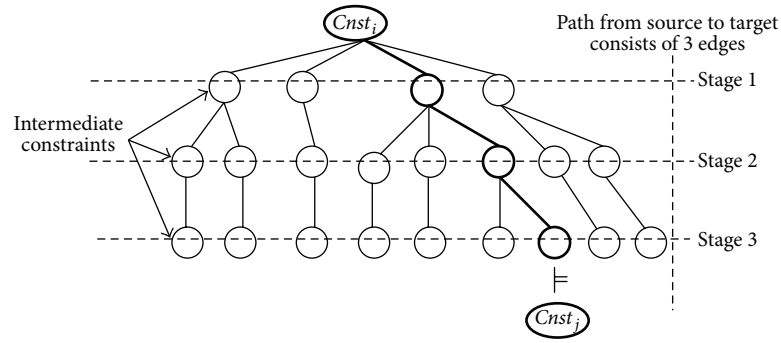

Figure 8: Generated intermediate conditions.

Table 3: Interaction context.

Goal	Freight movement
Preconstraints	{Freight.Det ≠ Null, Origin.Det ≠ Null, Dest.Det ≠ Null, IncoTerm.Type ∈ {FOB, EXW, CIF}}
Desc-Constraints	{Credit.Period = 15, Speciality.Type ⊆ {Motor-Vehicles}}
Postconstraints	{ShippingOrder.Status = Fulfilled, Payment.Status = Received}

satisfy the substitution conditions defined along the path to accept such transformation. More theoretical details and proofs regarding indirect satisfiability could be found in [10]. As a conversion function could have multiple finite output values, the first condition could be transformed into a finite number of intermediate constraints at a given stage (i.e., a path edge). This forms a finite tree of the possible intermediate constraints that can be obtained from the first condition using the defined finite conversion function. When one of the intermediate constraints of the final stage directly satisfies the second condition, this implies that the first condition can indirectly satisfy the second condition, as indicated in Figure 8. More details about the condition indirect satisfiability approach and the techniques for intermediate conditions generation as well as the corresponding theoretical proofs could be found in [10].

5. Service Conversation Model: G^+ Model

Services interactions are captured via the G^+ model [10, 12, 16, 41]. G^+ model captures services goals and interaction contexts as well as the expected interaction scenarios (depicted in Figure 9). A goal is represented by an application domain operation, a scenario is represented by a sequence of application domain operations, and a context is represented by different sets of constraints over application domain concepts (that is pre, post, and capability describing constraints), as in Table 3.

A Goal Achievement Pattern (GAP) is a global (end-to-end) snapshot of how the service's goal is expected to be accomplished, representing one given way to achieve a goal. A GAP is determined by following the path from the goal node to a leaf operation node, as depicted in Figure 9.

At the point where a branch starts, a group of constraints must be valid in order to visit that branch. This group of constraints acts as a *subcontext* for the GAP. This subcontext will be added to the preconstraints of the context of the G^+ model to form the GAP interaction context, forming what we define as a *conversation context*, and the GAP formulates what we define as a *conversation pattern*. In order to be able to semantically match conversation patterns, we need to generate their corresponding behavior models. A behavior model corresponding to a given conversation pattern is a sequence of conversation states representing the transition point between its operations. The first transition point is the point before invoking the first operation in the pattern, and the final transition point is the point after finishing the execution of the last operation in the pattern. Intermediate transition points are the points located between each pair of consecutive operation. A conversation state is represented by a set of conditions that are guaranteed to be satisfied at the corresponding transition point. For example, the conditions at the first transition point are the preconditions of the conversation context, while the conditions at a given transition point x are the ones constituted from the postconditions of the preceding operations as well as the preconditions of the conversation context that are still satisfied at x. Table 4 shows a simplified example for a sequence of operations and its corresponding state sequence. We propose a new way for representing a behavior state that helps us to improve the matching accuracy. Instead of representing the state as a set of conditions or constraints holding at a given transition point, we differentiate between these constraints based on their effect on the next operation to be executed. As we can see in Table 4, we classify state conditions in two classes: effective conditions and idle conditions. Effective conditions are the minimal subset of the state conditions that satisfies the preconditions of the following operation, while the idle conditions are the maximal subset of the state conditions that are independent from the preconditions of the following operation. This differentiation is important as states will be matched according to their effective conditions only, as including idle conditions in the state matching process just

Table 4: An example of a conversation pattern and its corresponding state sequence.

Conversation pattern	Preconditions	Postconditions
Conversation context	$C.a = 10$ $C.b = 20$	
OP1	$C.a \neq$ NULL	$C.a < 0$ $C.x = 5$
OP2	$C.b \neq$ NULL $C.x \neq$ NULL	$C.b > 0$
State	Effective conditions	Idle conditions
S_0	$C.a = 10$	$C.b = 20$
S_1	$C.b = 20, C.x = 5$	$C.a < 0$
S_2	$C.a < 0, C.b > 0$	

adds unnecessary restrictions as idle conditions have no effect on the invocation of the following operation [10].

The first row in Table 4 contains the conversation context. The preconditions of the conversation context are divided into an effective condition ($C.a = 10$) and an idle condition ($C.b = 20$) to form the first state S_0, as only the condition ($C.a = 10$) is satisfying the pre-condition of operation *OP1*. After *OP1* finishes its execution, three conditions are still satisfied ($C.b = 20$), ($C.x = 5$), and ($C.a < 0$), which in turn are divided into effective and idle conditions according to the preconditions of *OP2* to form the state S_1. The process is repeated at every transition point to compute the corresponding state. We consider all the conditions of the final state as effective. Such behavior models could be constructed offline as well as on the fly, and they will be used to determine the mappings between conversation patterns to create the conversation adapter.

6. Signature Adaptation

This section discusses the proposed approach for signature adaptation. It is based on the context-sensitive conditional concept substitutability approach discussed before to resolve concepts conflicts using CSEG semantics. As a conversation message is formulated according to the vocabulary of the sending service, a chance for signature incompatibility may arise if such a vocabulary is not supported by the receiving service or the receiving service is adopting a different messaging schema. It is fortuitous that a signature incompatibility may be resolved using converters if the operations are substitutable with respect to the involved conversation context [10].

Operations mapping is determined based on their substitutability status. Operations substitutability is determined according to the satisfiability status between their pre- and postconditions, respectively, that an operation *OP1* can be substituted by an operation *OP2* when the preconditions of *OP1* satisfy the preconditions of *OP2* and the postconditions of *OP2* satisfy the postconditions of *OP1*, as indicated in Figure 10. The figure shows that operation *OP2* can substitute operation *OP1* with respect to a given conversation context. *OP2* is adapted to *OP1* by generating an input converter (which converts *OP1* inputs to *OP2* inputs) and an output converter (which converts *OP2* outputs to *OP1* outputs). Converters consist of a set of conversion functions determined according to the mapping types between involved concepts. Operations substitutability is determined according to the satisfiability status between their pre- and postconditions, respectively. An operation *OP1* can be substituted by an operation *OP2* when the preconditions of *OP1* satisfy the preconditions of *OP2* and the postconditions of *OP2* satisfy the postconditions of *OP1*. Operations substitutability is not necessarily bidirectional, as it depends on the satisfiability directions between their conditions. When we have two operations *OP1* and *OP2* with different signatures, we check if the preconditions of *OP1* satisfy the preconditions of *OP2* and the postconditions of *OP2* satisfy the postconditions of *OP1* with respect to the conversation context as discussed above. When such conditions are satisfied, the input and output converters are generated from the conversation functions defined along the identified paths. We summarize the steps needed to generate a converter that transforms a set of concept A to a set of concepts B in Algorithm 1. Generating concepts converters is not a trivial task, as it requires to capture the conversion semantics between application domain concepts, in a context-based finite manner and requires use of these semantic to determine conversion validation with respect to the conversation context. Luckily, concept substitutability graph captures concepts functional substitutability semantics in a context-based manner and provides the conversion semantics and the substitutability constraints that must be satisfied by the conversation context, in order to have a valid conversion. It is important to note that one concept can be converted to another concept in one context, and the same two concepts cannot be converted in other contexts. In order to determine whether two concepts are convertible or not, first we check if the there is a path between the two concepts in the CSEG. If there is no path this means that they cannot be convertible; otherwise, we check the satisfiability of the substitution constraints along the path with respect to the conversation context. If all the constraints are satisfied, this means that the concepts are convertible; otherwise, they are not. Details about this process are in given [10, 16].

To convert a list of concepts to another list, first we construct a concepts mapping matrix (Γ) between the two lists (one list is represented by the columns, and the other is represented by the rows). A matrix cell has the value 1 if the corresponding concepts are convertible in the direction needed otherwise the cell will have the value 0. When concepts are convertible, we perform the conversion process by invoking the conversion functions defined along with the edges of the path between them. So the invocation code of such conversion functions forms the source code of the needed converter. Steps of generating such converter are indicated in Algorithm 1.

The converter class will have a `CONVERT` method to be invoked to perform the conversion process. Of course conversion functions along the path are cascaded, so there is no need for adaptation. The converter is represented as a class with different methods corresponding to conversion functions to be invoked. Algorithm 1 requires the converter class to have a CONVERT method, which is invoked to apply

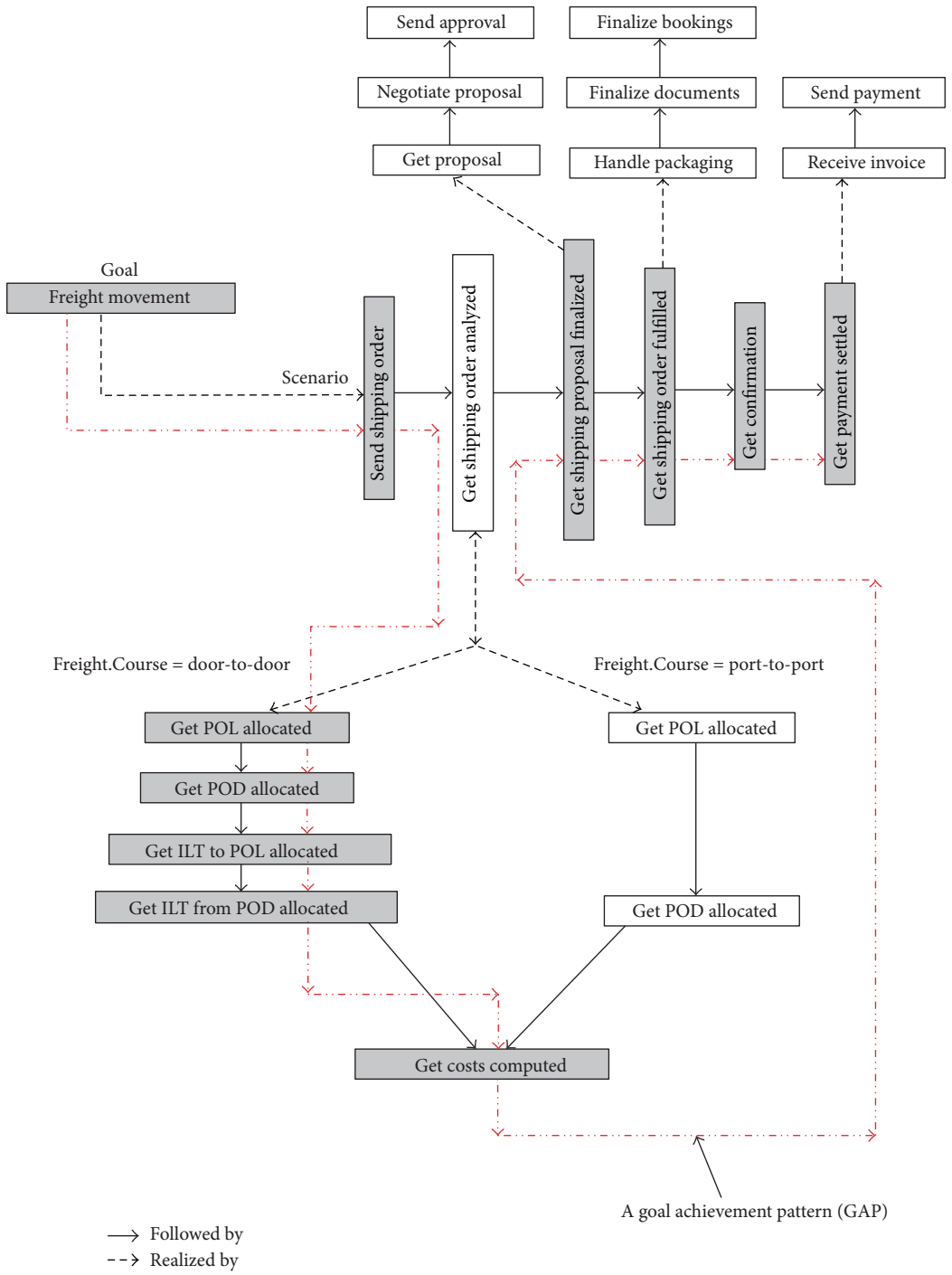

Figure 9: Interaction scenarios.

the conversions. The algorithm indicates that each element in B should be reachable to a subset of A (i.e., the subset appeared as a scope in a given substitution pattern) and also indicates that the conversation context should satisfy all the substitution conditions defined along the identified path; otherwise such concept substitution is considered invalid and cannot be used. Once substitutions validity is confirmed, the determined concepts mappings are accepted, and the converter is generated. Figure 11 shows an example for a converter consisting of six conversion functions resulting from different types of concept mappings. For example, the conversion function *CF4* is responsible for converting the concepts `C6`

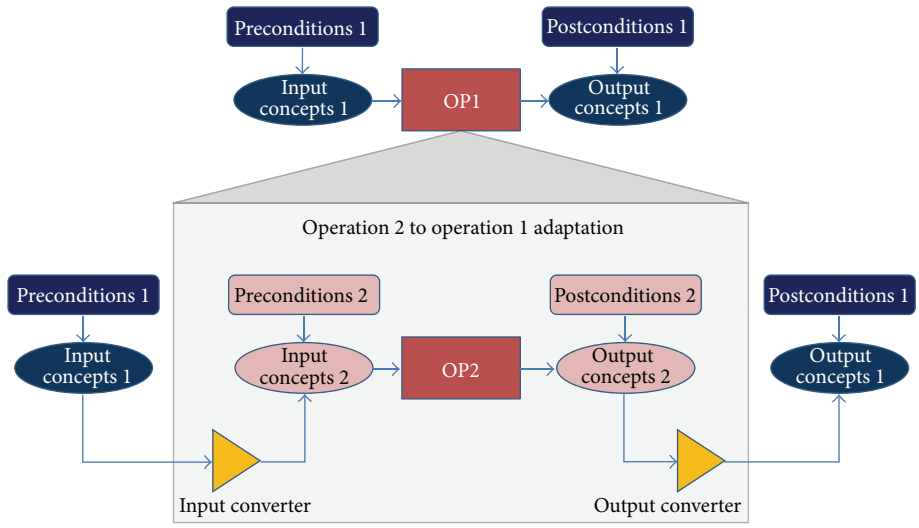

Figure 10: Signature adaptation.

and C7 into the concept C8. In the next section, we show how the substitutability between two different sequences of operations (conversation patterns) is determined. More details about adapter generation will be given later.

7. Conversation Protocol Adaptation

One approach for semantically resolving conversation incompatibilities involves the use of the substitutability rule [10, 42] in which two conversation patterns are considered compatible when one pattern can substitute for the other without violating any condition in the corresponding conversation context. In order to determine the substitutability between two conversation patterns, we must check the substitutability of their messages (representing the operations to be performed), which in turn requires checking the substitutability of their input and output concepts. Hence, the first step needed to resolve conversation incompatibilities involves the ability to automatically determine concepts substitutability, as indicated before.

Every service supports a specific number of conversation patterns and requires other services to follow the supported patterns during their interactions. However, protocol incompatibilities could arise when the interacting services expect different ordering for the exchanged message sequences. Protocol incompatibilities may be resolved if there exists a mapping pattern between the operations appeared in the conversation patterns [10]. Conversation adapter structure is decided according to the determined operations mappings, as they specify which messages should be generated by the adapter when a given message or a sequence of messages is received. Operations mappings could be of different types such as one-to-one, one-to-many, many-to-one, and many-to-many mappings and guaranteed to exist if the conversation patterns are substitutable with respect to the conversation context [10]. Hence, to resolve protocol incompatibilities, first we must check the substitutability of the involved conversation patterns, and then find their corresponding operations mappings. Conversation patterns substitutability is determined according to the satisfiability status between their pre and postconditions corresponding to the pre and postconditions of their contexts, respectively, that a conversation pattern *CP1* can be substituted by a conversation pattern *CP2* when the preconditions of *CP1* satisfy the preconditions of *CP2* and the postconditions of *CP2* satisfy the postconditions of *CP1*. Conversation patterns substitutability is not necessarily bidirectional, as it depends on the satisfiability directions between their conditions. To find the operation mappings between two substitutable conversation patterns, we must analyze their corresponding behavior models as operations are matched semantically not syntactically. To find the operation mappings between two substitutable conversation patterns, we must find the mappings between their corresponding behavior states by grouping adjacent states in both models into matching clusters. A state S_x matches a state S_y only when the effective conditions of S_x satisfy the effective conditions of S_y. A cluster CL_x matches another cluster CL_y when the state resulting from merging CL_x states matches the state resulting from merging CL_y states, as depicted in Figure 12.

The figure shows the initial state sequences, the state clusters, and the final state sequences. Merging two consecutive states S_x, S_{x+1} in a given behavior model to form a new expanded state S_m means that we performed a virtual operation merge between OP_{x+1}, OP_{x+2} to obtain a coarser operation OP_m, as depicted in Figure 13. The figure indicates that the input of OP_m is formulated from the sets of concepts A and B, and its output is formulated from the sets of concepts C and E. As we can see, the set of concepts D does not appear in OP_m signature and consequently will not appear in S_m conditions. Such information hiding provides a chance for having matching states. S_m is computed by reclassifying the effective and idle conditions of S_x into new sets of effective and idle conditions according to the preconditions of OP_m. For example, by merging states S_0, S_1 shown in

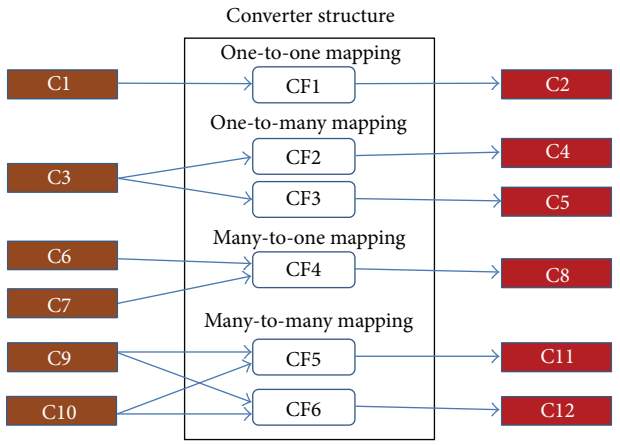

Figure 11: Converter structure.

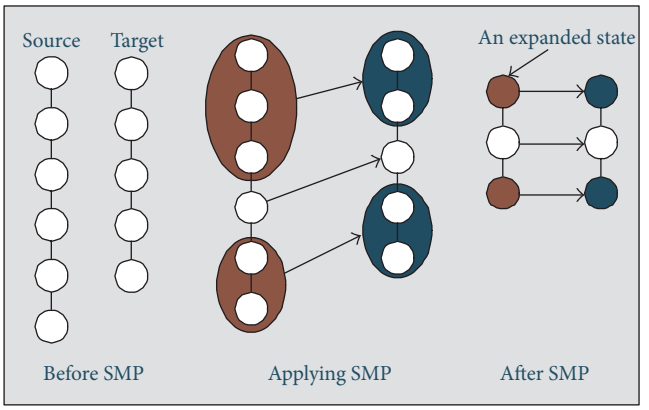

Figure 12: State clustering effect.

Table 4, the resulting S_m will have the set ($C.a = 10$), ($C.b = 20$) as its effective conditions, and the set ($C.a < 0$), ($C.b > 0$) as its idle conditions. As we can see, conditions on $C.x$ do not appear in S_m. We use a Sequence Mediation Procedure (SMP) (discussed in the next subsection) to find such matching clusters. SMP starts by examining the initial states in both sequences, then moves forward and backward along the state sequences until matching clusters are formed and the corresponding operations mappings are determined. The highest level of abstraction that could be reached occurs when all the conversation pattern operations are merged into one operation. As the number of the states is quite small, the backtracking approach does not diminish the performance.

7.1. Conversation Pattern Matching. Sequence Mediator Procedure (SMP) is a procedure used to match different state sequences. Such state sequences are generated from the GAPs (conversation patterns) to be matched. Each transition point x between two consecutive operations Op_x and Op_{x+1} in a given GAP is represented by a behavior state. Such state is captured via constraints active at this transition point x. A constraint at a transition point x is considered effective if it needs to be true in order to invoke Op_{x+1}. A state S_x matches a state S_y when its effective constraints subsume the effective constraints of S_y (theoretical models and proofs could be found in [10]). SMP does not require the state sequences to have the same number of states in order to be matched; however, it applies different state expansion operations to reach to a matching case if possible. When a state is expanded, it could be merged with either its successor states (known as *Down Expansion* and denoted as $\Downarrow_G$) or its predecessor states (known as *Reverse Expansion* and denoted as $\Uparrow_G$), where G is the conversation goal, setting the conversation context. SMP uses these different types of state expansions to recluster unmatched state sequences to reach a matching case. This reclustering operation could happen on both state sequences, as indicated in Figure 12. Merging two consecutive states in a given state sequence means that their successor operations are merged to form a new operation, as depicted in Figure 13.

Figure 13 shows that the states S_x and S_{x+1} are merged forming a new state S_m, which is computed as if there is a new operation Op_m in the sequence replacing the operations Op_{x+1} and Op_{x+2}. The input of Op_m is the union between the sets of concepts A and B, its output is the union between the sets of concepts C and E, while the set of concepts D will not appear neither in Op_m input nor in Op_m output.

SMP tries to recluster both state sequences until it reached into an organization that has both sequences matched; if such organization is reached, SMP announces that it found a match and provides the mappings between the resulting clusters. Such mappings are provided in the form of an Operations Mapping Matrix (denoted as Θ) that indicated which operations in a source sequence are mapped to which operations in a target sequence, as indicated in Table 5.

Once obtaining the operations mapping matrix from SMP, only matched GAPs that require no change in the requested conversation pattern will be chosen, and therefore their corresponding adapters could be generated. SMP starts by examining the first state of the "source target" against the first state of the "target sequence." When the source state matches the target state, SMP applies Algorithm 2 to handle the matching case. When a source state matches a target state, SMP checks the target down expansion to match as many target states as possible with the source state (lines 2 and 3).

In Algorithm 3, SMP aims to find a matching source cluster for every target state. However, when a source state fails to match a target state, SMP checks if the source state could be down expandable (lines 5–7). If this checking fails too, SMP checks whether the source state could be reverse expanded with respect to the target state (lines 9–11). When a source state cannot be expanded in either directions, SMP tries the successor source states to match the target state using the down and reverse source expansion scenarios (line 16). It stores the unmatched source state for backtracking purposes (line 13). When a target state cannot be matched to any source state, SMP tries reverse expanding the target state to find a match for it (lines 18–20); when that fails this target state is considered unmatched, and the next target state will be examined (lines 22-23). The algorithm continues even if unmatched target state is reached, as this unmatched state

Input: Set of concepts A, B, Conversation Context CC, and The CSEG segment S.
Output: Source code of converting A to B with respect to CC.
Begin
Get Concepts Mapping Matrix Γ between A and B.
Create an empty class CONVERTER.
Create a method CONVERT with A as its input, and B as its output.
for each concept c_i in A **do**
  Get corresponding mapping concept c_j using Γ.
  if There exists a path from c_i to c_j in S **then**
    for each edge $e_{(w,z)}$ in the path **do**
      Create a method in CONVERTER with same signature and body of the edge conversion function $\Psi_{(w,z)}$.
      Add $\Psi_{(w,z)}$ invocation code to CONVERT body.
    end for
  else
    Return Error.
  end if
end for
Return CONVERTER generated code.
End

Algorithm 1: Converter generator.

could be merged with any of its successors if they are going to be reversely expanded.

8. Automatic Adapter Generation

Each service has different conversation patterns (generated from its G^+ model) that could use to interact with other services. Such conversation patterns could be matched by one service or by many different services, as depicted in Figure 14.

Figure 14 indicates that each conversation pattern should have its own adapter. Once the required conversation patterns are specified via the management interfaces (as indicated in Section 2.1), the adapter generation process is started. The outcome of the adapter generation process is the source code for the adapter class that consists of the methods to be invoked by the consuming services. The body of these methods consists of the invocation code for the consumed service operations and the invocation code for the corresponding converters. First, we determine the required adapter structure then generate the source code for the adapter and the needed converters. Once the class adapter is generated, it is compiled, and the corresponding WSDL file is generated, in order to expose the adapter class as a service, which could be easily invoked by the consuming service. The details are discussed in the following subsections.

Once two services "decide" to interact with each other, they notify the middleware such that it identifies their substitutable conversation patterns and generates the corresponding conversation adapters. The middleware notifies back the services with the identified substitutable patterns such that each service knows which patterns should be used during the conversation [11]. Once a conversation pattern CP_x is identified as substitutable with a conversation pattern CP_y, the middleware performs the following steps (similar to Algorithm 1) to generate their corresponding conversation adapter, which transforms CP_x incoming messages into CP_y outgoing messages. First, it generates an adapter class with methods corresponding to CP_x operations (incoming messages), such that each method consists of a signature (similar to the signature of the corresponding incoming message) and an empty body (which will be later containining the code for generating the corresponding CP_y outgoing messages). Second, it determines the operations mappings between CP_x and CP_y and then uses these mappings to construct the generation code for the outgoing message. Table 5 provides an example for a CP_x conversation pattern that is substituted by a conversation pattern CP_y, showing the corresponding operations mappings.

Table 5: Example of a conversation patterns mapping matrix Θ.

Mapping type	CP_x	CP_y
1 to 1	$OP_{(x,1)}$	$OP_{(y,1)}$
1 to many	$OP_{(x,2)}$	$OP_{(y,2)}, OP_{(y,3)}$
Many to 1	$OP_{(x,3)}, OP_{(x,4)}$	$OP_{(y,4)}$
Many to many	$OP_{(x,5)}, OP_{(x,6)}$	$OP_{(y,5)}, OP_{(y,6)}, OP_{(y,7)}$

Figure 15 shows the corresponding adapter structure. Signature incompatibilities are handled by generating the suitable input and output converters. The outgoing message generation code is constructed as follows.

In one-to-one operations mappings, one CP_x operation matches one CP_y operation. The input converter is created

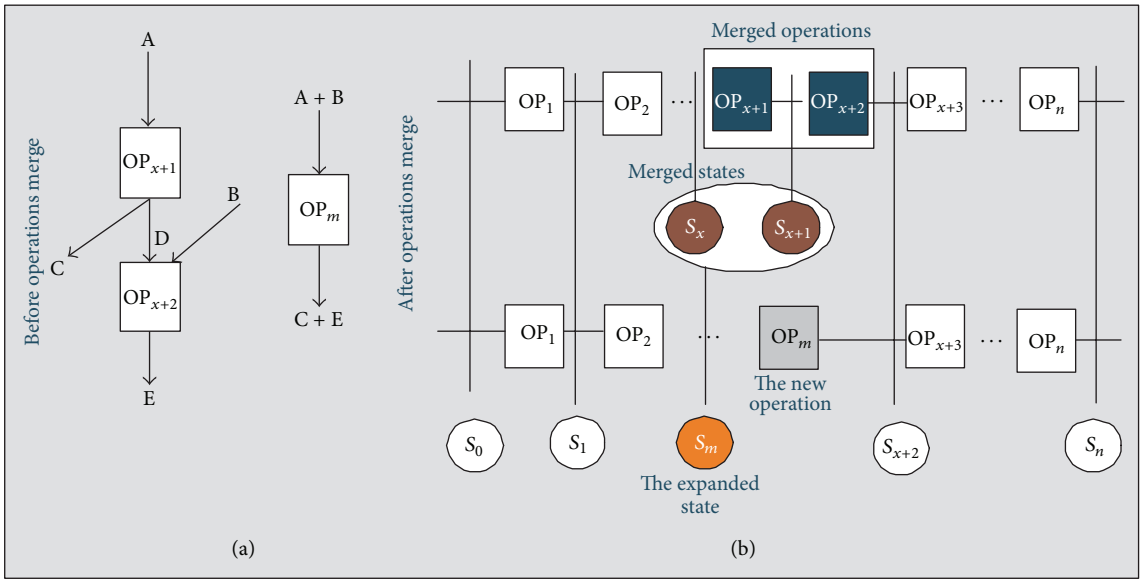

Figure 13: Consecutive states merge.

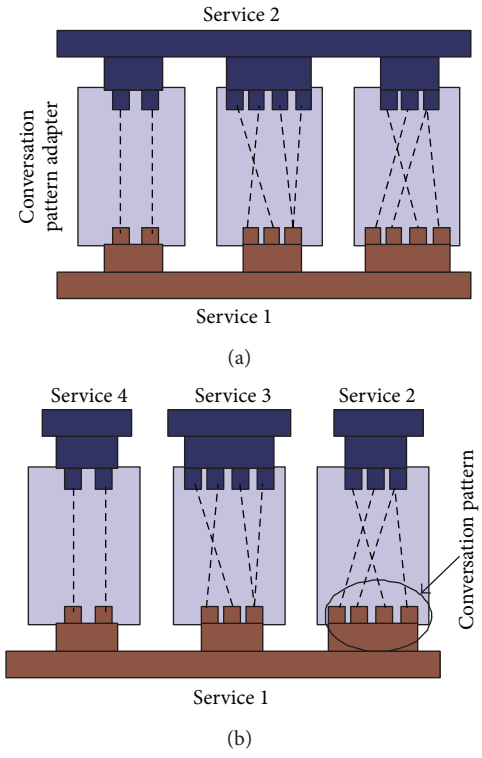

Figure 14: Service conversation patterns adapters.

between the inputs of the CP_x operation and the inputs of CP_y operation. The output converter is created between the outputs of the CP_y operation and the outputs of CP_x operation. The outgoing message generation code consists of the invocation code for the input converter, the CP_y operation, and the output converter, as depicted for OP_{x+1} in Figure 15.

In one-to-many operations mappings, one CP_x operation matches subsequence of CP_y operations. An input converter is created between the inputs of the CP_x operation and the inputs of the OP_{my} operation (resulting from merging the CP_y subsequence). An output converter is created between the outputs of the OP_{my} operation and the outputs of the CP_x operation. The outgoing message generation code consists of the invocation code for the input converter, the CP_y subsequence (multiple messages), and the output converter, as depicted for $OP_{(x,2)}$ in Figure 15.

In many-to-one operation mapping, a subsequence of CP_x operations matches one CP_y operation. The outgoing message cannot be generated unless all the operations of the CP_x subsequence are received. Hence, before generating the outgoing message, all the incoming messages should be buffered until the last message is received. This is achieved by using a message buffer handler. An input converter is created between the inputs of OP_{mx} (resulting from merging the CP_x subsequence) and the inputs of the CP_x operation. An output converter is created between the outputs of CP_y operation and the outputs of the OP_{mx} operation. The outgoing message generation code consists of the invocation code for the input converter, the CP_y operation, and the output converter, as depicted for $OP_{(x,3)}, OP_{(x,4)}$ in Figure 15.

In many-to-many operation mapping, a subsequence of CP_x operations matches a subsequence of CP_y operations. Incoming messages are buffered as indicated earlier. An input converter is created between the inputs of OP_{mx} and the inputs of OP_{my}. An output converter is created between the outputs of OP_{my} and the outputs of OP_{mx}. The outgoing message generation code consists of the invocation code for the input converter, the CP_y subsequence (multiple messages), and the output converter, as depicted for $OP_{(x,5)}, OP_{(x,6)}$ in Figure 15.

Once the adapter class is successfully generated, the middleware can reroute the conversation messages to the

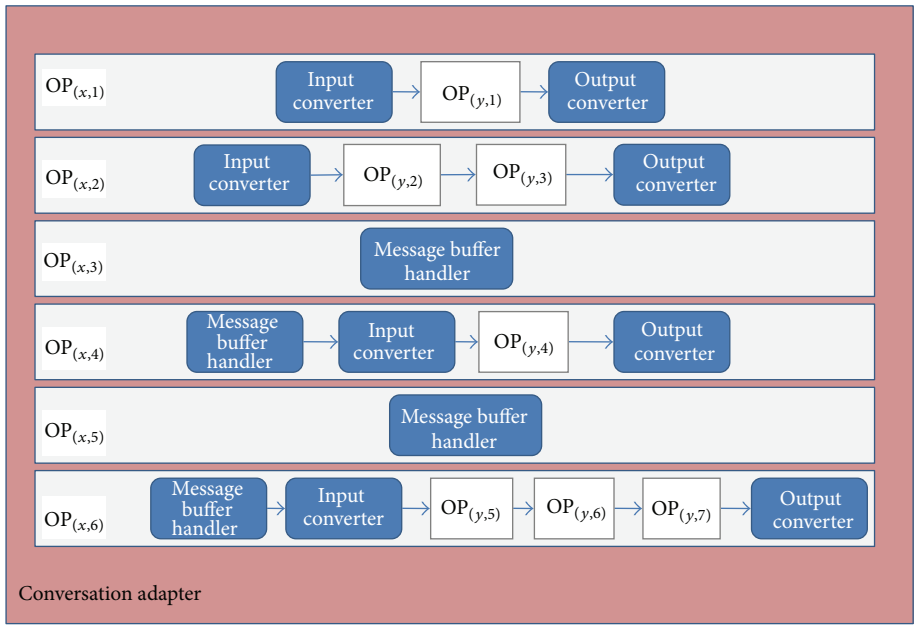

Figure 15: Conversation adapter structure for patterns in Table 5.

Input: $S[i]$ (the ith source state), $T[j]$ (the jth target state), G (the required goal)
Output: Return true when sequences re-clustering is finished.
(1) **Begin**
(2) **if** $(T[j] \Downarrow_G S[i])$ **then**
(3) Expand $T[j]$
(4) **end if**
(5) Mark $T[j]$, $S[i]$ as matching peers
(6) **if** $(j < \|T[]\|)$ **then**
(7) **if** $(i < \|S[]\|)$ **then**
(8) Apply Algorithm 4 over $(S[i+1], T[j+1], G)$
(9) **else**
(10) Apply Algorithm 4 over $(S[i], T[j+1], G)$
(11) **end if**
(12) **else**
(13) Return true
(14) **end if**
(15) **End**

Algorithm 2: SMP matching case handling.

adapter service (corresponding to the generated class) to perform the needed conversation customizations. Invoking operations from existing services is a straightforward simple task, however generating the inputs and outputs converters is not, as we need to find the mappings between the concepts and their conversion functions. Steps of generating such adapter class are indicated in Algorithm 4. Algorithm 4 simply starts by creating an empty class then adds methods to this class with the same signatures of the consuming service conversation pattern. For each created method, it gets the sequence of operations realizing the method with the help of the operation mapping matrix(Θ). Then, it creates the concepts converters by calling the `ConverterGenerator` function (depicted in Algorithm 1) with the proper parameters. Finally, it adds the converter generated code to the adapter if no error resulted during the generation.

In case the algorithm returns error, this means conversation adaptation cannot be performed; therefore, these services cannot talk to each other on the fly, and a manual adapter needs to be created to enable such conversation.

9. Experiments

This section provides simulation experiments used for verifying the proposed approaches. First, we start by the verifying

Input: $S[i]$ (the ith source state), $T[j]$ (the jth target state), G (the required goal)
Output: Return true when sequences re-clustering is finished.

```
(1)  Begin
(2)  if (S[i] ⊵_G T[j]) then
(3)     Apply Algorithm 3 over (S[i], T[j], G)
(4)  else
(5)     if (S[i] ⇓_G T[j]) then
(6)        Expand S[i]
(7)        Apply Algorithm 3 over (S[i], T[j], G)
(8)     else
(9)        if (S[i] ⇑_G T[j]) then
(10)          Apply CRO over S[i]
(11)          Apply Algorithm 3 over (S[i], T[j], G)
(12)       else
(13)          BackTrack = S[i]
(14)          if (j < ‖T[]‖) then
(15)             if (i < ‖S[]‖) then
(16)                Apply Algorithm 4 over (S[i + 1], T[j], G)
(17)             else
(18)                if (T[j] ⇑_G S[i]) then
(19)                   Apply CRO over T[j]
(20)                   Apply Algorithm 3 over (S[i], T[j], G)
(21)                else
(22)                   Mark T[j] as Unmatched
(23)                   Apply Algorithm 4 over (BackTrack,
                       T[j + 1], G)
(24)                end if
(25)             end if
(26)          else
(27)             Merge unmatched source states with their
                 predecessors.
(28)             Return true
(29)          end if
(30)       end if
(31)    end if
(32) end if
(33) End
```

Algorithm 3: Sequence mediator procedure (SMP).

experiments for the proposed signature adaptation approach; then we introduce the verifying experiments for the proposed conversation adaptation approach.

9.1. Signature Adaptation. To verify the proposed signature adaptation using conditional ontology mapping approach, we use a simulation approach to compare between the proposed approach and the generic mapping approach that adopts only Is-a relations to match signature concepts (both input and output concepts). The used comparison metric is the F-measure metric.

F-measure metric combines between the retrieval precision and recall metrics and is used as an indicator for accuracy that approaches with higher values which means that they are more accurate. F-measure is computed as $(2 * Precision * Recall)/(Recall + Precision)$. The experiment starts by generating two random sets of independent concepts (representing two different ontologies). One set will be used as the original dataset, and the second one will be used as a query set. For each concept in the query set, we randomly generate an Is-a relation to a corresponding concept in the original dataset (i.e., mapping using Is-a relation). For each pair of concepts having an Is-a relation, we generate a corresponding substitution pattern in the CSEG. For simplicity, the substitution pattern is generated as follows. The scope is equal to the original dataset concept. The substitution condition is generated as greater than condition with a randomly generated integer number (`e.g., C1 >10`). The conversion function is just an equality function (`e.g., C1 = C2`). From the generated set of concepts, we generate a random signature (i.e., a random operation) by randomly choosing a set of input concept and a set of output concepts. For each generated signature in the query set, we generate a corresponding context. For simplicity, the context will consist of one equality condition with a randomly generated integer number (`e.g., C1 = 20`).

Input: CP_x (consuming GAP), CP_y, Operations Mapping Matrix Θ.
Output: Source code of the adapter between CP_x and CP_y.

```
Begin
Create an empty class ADAPTER.
for each operation Op(x,i) in GAP_x do
    Create a method in ADAPTER with same signature
    as Op(x,i).
    Get realizing operations subsequence OPSeq from
    CP_y using Θ.
    Call ConverterGenerator (Op(x,i) Inputs, OPSeq Inputs).
    // Algorithm 2
    if Converter Code generated successfully then
        Add Inputs Converter Code to method body.
        Add invocation Code for OPSeq to method body.
        Call ConverterGenerator (OPSeq Output, Op(x,i)
        Outputs).
        if Converter Code generated successfully then
            Add Outputs Converter Code to method body.
        else
            Return Error.
        end if
    else
        Return Error.
    end if
end for
Return ADAPTER generated code.
End
```

Algorithm 4: Adapter automatic generator.

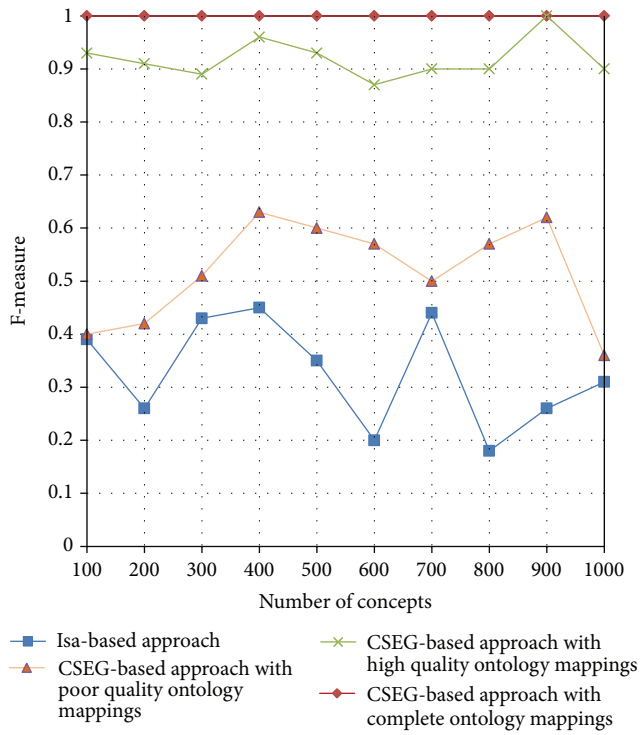

Figure 16: Signature adaptation approaches comparison.

Table 6: An example of two matching GAPs

	S_2 GAP	S_1 GAP
Preconstraints	{Cargo.Det = 1000 cars, Cargo.POL = Melbourne-Australia, Cargo.POD = Alexandria-Egypt, Cargo.Course = Port-To-Port, IncoTerm.Type = CIF}	{Freight.Det ≠ Null, Origin.Det ≠ Null, Dest.Det ≠ Null, Freight.Course = Port-To-Port, IncoTerm.Type ∈ {FOB, EXW, CIF}}
Desc-Constraints	{Payment.type = Credit, Speciality.Type = Motor-Vehicles}	{Credit.Period = 15, Speciality.Type ⊆ {Motor-Vehicles, Dangerous-Cargo}}
Postconstraints	{Cargo.Status = Accomplished}	{ShippingOrder.Status = Fulfilled, Payment.Status = Received}
Goal	Cargo transportation	Freight movement
Operation sequence	(1) Send-Cargo-Details (2) Get-Offer (3) Negotiate-Offer (4) Accept-Offer (5) Execute-Offer (6) Send-Payment	(1) Send-Shipping-Order (2) Get-POL-Allocated (3) Get-POD-Allocated (4) Get-Costs-Computed (5) Get-Proposal (6) Negotiate-Proposal (7) Send-Approval (8) Handle-Packaging (9) Finalize-Documents (10) Finalize-Bookings (11) Get-Confirmation (12) Receive-Invoice (13) Send-Payment

Table 7: Part of the ontology operations' definitions adopted by S_2.

Operation	Preconstraints	Postconstraints
`Send-Cargo-Details`	{`Cargo.Det` ≠ Null, `Cargo.POL` ≠ Null, `Cargo.POD` ≠ Null, `IncoTerm.Type` ≠ Null}	{`Cargo.Status` = Received}
`Get-Offer`	{`Cargo.Status` = Received, `Cargo.Course` ≠ Null}	{`Offer.Status` = Sent}
`Negotiate-Offer`	{`Offer.Status` = Sent}	{`Offer.Status` = Approved}
`Accept-Offer`	{`Offer.Status` = Approved}	{`Offer.Status` = Accepted}
`Execute-Offer`	{`Offer.Status` = Accepted}	{`Offer.Status` = Executed}
`Send-payment`	{`Offer.Status` = Executed}	{`Cargo.Status` = Accomplished}

Hence, not all the substitution patterns defined in the CSEG will be valid according to the generated contexts. We submit the query set to the two approaches to find matches in the original dataset, and based on the retrieved concepts the F-measure is computed. Figure 16 depicts the results. As we can see, the generic approach ignores the contexts and retrieves the whole original dataset as answers, which results in low F-measure values, while the proposed approach succeed to reach 100%.

However, this result could be misleading, as the experiment is done with complete CSEG patterns. In practice, an ontology designer may skip some substitution patterns when defining CSEG patterns. Therefore, the proposed approach will not be able to resolve the cases with missing patterns. In other words, the accuracy of the proposed approach mainly depends on the quality of the defined ontology mappings. To show such effect, we repeated the experiment except that we store only a portion of the generated substitution patterns. A high-quality ontology mapping means that up to 25% of the generated patterns are missing. A low-quality ontology mapping means that from 50% to 80% of the generated patterns are missing. Then, we compute the F-measure values for each case. Results are depicted in Figure 16. As we can see, when low-quality mappings are used, the proposed approach

Table 8: Part of the ontology operations' definitions adopted by S_1.

Operation	Preconstraints	Postconstraints
Send-Shipping-Order	{Freight.Det ≠ Null, Origin.Det ≠ Null, Dest.Det ≠ Null, Freight.Course ≠ Null, IncoTerm.Type ≠ Null}	{ShippingOrder.Status = Created}
Get-Shipping-Order-Analyzed	{ShippingOrder.Status = Created}	{ShippingOrder.Status = Analyzed}
Get-POL-Allocated	{ShippingOrder.Status = Created}	{POL.Status = Allocated}
Get-POD-Allocated	{POL.Status = Allocated}	{POL.Status = Allocated, POD.Status = Allocated}
Get-ILT-To-POL-Allocated	{POL.Status = Allocated}	ILT.ToStatus = Allocated
Get-ILT-From-POD-Allocated	{POD.Status = Allocated}	ILT.FromStatus = Allocated
Get-Costs-Computed	{POL.Status = Allocated, POD.Status = Allocated}	{ShippingOrder.Status = Analyzed}
Get-Shipping-Proposal-Finalized	{ShippingOrder.Status = Analyzed}	{ShippingOrder.Status = Approved}
Get-Proposal	{ShippingOrder.Status = Analyzed}	{Proposal.Status = Sent}
Negotiate-Proposal	{Proposal.Status = Sent}	{Proposal.Status = Approved}
Send-Proposal	{Proposal.Status = Approved}	{ShippingOrder.Status = Approved}
Get-Shipping-Order-Fulfilled	{ShippingOrder.Status = Approved}	{ShippingOrder.Status = Executed}
Handle-Packaging	{ShippingOrder.Status = Approved}	{Packaging.Status = Accomplished}
Finalize-Documents	{Packaging.Status = Accomplished}	{Documentation.Status = Accomplished}
Finalize-Bookings	{Documentation.Status = Accomplished}	{ShippingOrder.Status = Executed}
Get-Confirmation	{ShippingOrder.Status = Executed}	{ShippingOrder.Status = Confirmed}
Get-Payment-Settled	{ShippingOrder.Status = Confirmed}	{ShippingOrder.Status = Fulfilled, Payment.Status = Received}
Receive-Invoice	{ShippingOrder.Status = Confirmed}	{ShippingOrder.Status = Pending}
Send-Payment	{ShippingOrder.Status = Pending}	{ShippingOrder.Status = Fulfilled, Payment.Status = Received}

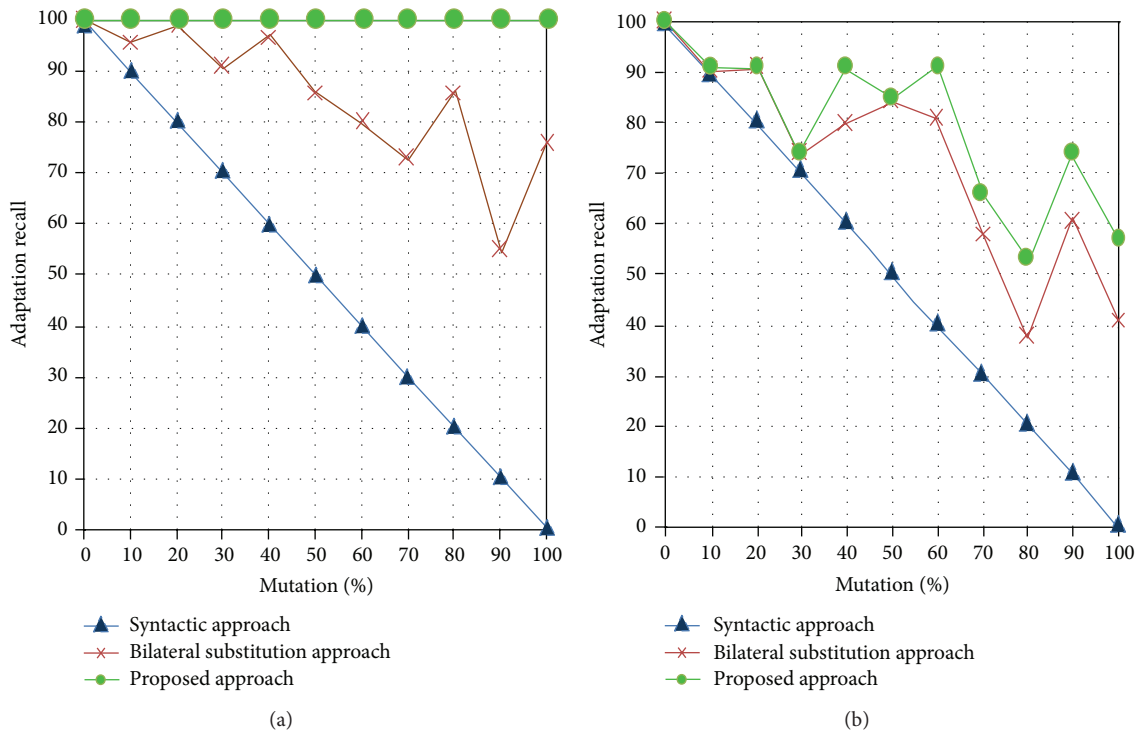

Figure 17: Conversation adaptation approaches comparison: (a) with complete substitution patterns (b) with missing substitution patterns.

Table 9: S_1 and S_2 behavior models.

S_1 Behavior model	
S_0	⟨{*Freight.Det* ≠ Null, *Origin.Det* ≠ Null, *Dest.Det* ≠ Null, *Freight.Course* = Port-to-Port, `IncoTerm.Type` ∈ {FOB, EXW, CIF}}, {}⟩
S_1	⟨{*ShippingOrder.Status* = Created}, {}⟩
S_2	⟨{*POL.Status* = Allocated}, {}⟩
S_3	⟨{*POL.Status* = Allocated, *POD.Status* = Allocated}, {}⟩
S_4	⟨{*ShippingOrder.Status* = Analyzed}, {}⟩
S_5	⟨{*Proposal.Status* = Sent}, {}⟩
S_6	⟨{*Proposal.Status* = Approved}, {}⟩
S_7	⟨{*ShippingOrder.Status* = Approved}, {}⟩
S_8	⟨{*Packaging.Status* = Accomplished}, {}⟩
S_9	⟨{*Documentation.Status* = Accomplished}, {}⟩
S_{10}	⟨{*ShippingOrder.Status* = Executed}, {}⟩
S_{11}	⟨{*ShippingOrder.Status* = Confirmed}, {}⟩
S_{12}	⟨{*ShippingOrder.Status* = Pending}, {}⟩
S_{13}	{*ShippingOrder.Status* = Fulfilled, *Payment.Status* = *Received*}, {}⟩
S_2 Behavior model	
S_0	⟨{*Cargo.Det* = 1000 Cars, *Cargo.POL* = Melbourne-Australia, *Cargo.POD* = Alexandria-Egypt, `IncoTerm.Type` = FOB}, {*Cargo.Course* = Port-to-Port}⟩
S_1	⟨{*Cargo.Course* = Port-to-Port, *Cargo.Status* = Received}, {}⟩
S_2	⟨{*Offer.Status* = Sent}, {}⟩
S_3	⟨{*Offer.Status* = Approved}, {}⟩
S_4	⟨{*Offer.Status* = Accepted}, {}⟩
S_5	⟨{*Offer.Status* = Executed}, {}⟩
S_6	⟨{*Cargo.Status* = Accomplished}, {}⟩

accuracy is negatively affected. The worst case complexity of the proposed approach is $O(n * m^{\|p\|})$, where n is the average number of substitution patterns of domain operations, m is average number of possible outputs generated from conversion functions, and $\|p\|$ is the length of the path p linking between mapped concepts (details could be found in [10]). The factor $m^{\|p\|}$ is the cost endured to find a sequence of generated intermediate conditions to indirectly match two conditions. However, in practice, n, m, and p are expected to be small; hence, we argue that the performance of the proposed approach is acceptable.

9.2. Conversation Adaptation. Currently, there is no standard datasets for service conversations. Hence, to verify the proposed approach for automated adapter generation, we follow a simulation approach similar to the one used in [10]. The proposed simulation approach compares three approaches for automated adapter generation. The first approach is a syntactic approach that requires no changes at the services interface level of the operations. It cannot resolve any semantic differences. We use this approach as a benchmark for our works. The second approach is our approach proposed in [11] that uses bilateral concept substitution to resolve signatures incompatibilities. We use this approach to show the effect of not supporting concept aggregation. The third approach is the approach proposed in this paper that uses aggregate concept conditional substitution semantics to resolve signature incompatibilities. The used comparison metric is the adaptation recall metric. It is similar to the retrieval recall metric and is computed as the percentage of the number of adapted conversation patterns (i.e., the ones that have a successfully generated conversation adapter) with respect to the actual number of the adaptable conversation patterns in the dataset.

The experiment starts by generating a random set of independent conversation patterns, for which each pattern has a unique operation, and each operation has different input and output concepts. A query set is generated as a copy of the original set. The query set is submitted to the three adaptation approaches in order to generate the adapters between the query set patterns and the original set patterns. As the two sets are identical and the conversation patterns are independent, each pattern in the query set will have only one substitutable pattern in the original set (i.e., its copy). The second phase of the experiment involves the gradual mutation of the query set and submission of the mutated query set to the three approaches, and then we check the number of adapters generated by each approach to compute the adaptation recall metric. The mutation process starts by mutating 10% of the query set and then continues increasing the percentage by 10% until the query set is completely mutated. The value of 10% is an arbitrary percentage chosen to show the effect of semantic mutations on the approach. At each step, the adaptation recall metric is computed for the three approaches. The mutation process is performed by changing the signatures of the operations with completely new ones. Then the corresponding substitution patterns are added between the old concepts and the new concepts in the CSEG. The number of concepts in a substitution pattern is randomly chosen between 1 (to ensure having cases of bilateral substitution) and 5 (an arbitrary number for concept aggregation). For simplicity, conversion functions are generated by assigning the old values of the concepts to the new values of the concepts, and the substitution conditions are generated as not null conditions.

The experiment results are depicted in Figure 17(a). The figure shows that the syntactic approach could not handle any mutated cases, as it cannot resolve signature incompatibilities. Hence, its corresponding adaptation recall values drops proportionally to the mutation percentage. The bilateral substitution approach only solved the cases with substitution patterns having one concept in their scopes, while it could not solve the cases with substitution patterns having more than one concept in their scopes (i.e., cases representing concept aggregation). Hence, its corresponding adaptation

Table 10: CSG segment for `CargoTransportation` operation.

Source	Destination	Conversion Fn	Substitution Cond.
Cargo.Det	Freight.Det	Freight.Det = Cargo.Det	
Freight.Det	Cargo.Det	Cargo.Det = Freight.Det	
Cargo.POL	Origin.Det	Origin.Det = Cargo.POL	
Origin.Det	Cargo.POL	Cargo.POL = Origin.Det	
Cargo.POD	Dest.Det	Dest.Det = Cargo.POD	
Dest.Det	Cargo.POD	Cargo.POD = Dest.Det	
Cargo.Type	Freight.Type	Freight.Type = Cargo.Type	
Freight.Type	Cargo.Type	Cargo.Type = Freight.Type	

Credit.Period	Payment.Type	IF (Credit.Period > 0) THEN
		Payment.Type = Credit
		ELSE
		Payment.Type = Cash
		END IF
			Credit.Period ≥ 0

Payment.Type	Credit.Period	IF (Payment.Type = Credit) THEN
		Credit.Period ∈ {15, 30, 45, 60}
		ELSE
		Credit.Period = 0
		END IF
			Payment.Type ∈ {Credit, Cash}

Order.Stat	Cargo.Stat	SWITCH (Order.Stat) CASE Fulfilled: Cargo.Stat = Done CASE Created: Cargo.Stat = Received END CASE	Order.Stat ∈ {Fulfilled, Created}
Cargo.Stat	Order.Stat	SWITCH (Cargo.Stat) CASE Done: Order.Stat = Fulfilled CASE Received: Order.Stat = Created END CASE	Cargo.Stat ∈ {Done, Received}
Proposal.Stat	Offer.Stat	Offer.Stat = Proposal.Stat	Proposal.Stat ∈ {Sent,Approved}
Offer.Stat	Proposal.Stat	Proposal.Stat = Offer.Stat	Offer.Stat ∈ {Sent, Approved}

Order.Stat	Offer.Stat	IF (Order.Stat = Approved) THEN
		Offer.Stat = Accepted
		ELSE
		Offer.Stat = Executed
		END IF
			Order.Stat ∈ {Approved, Executed}

Offer.Stat	Order.Stat	IF (Offer.Stat = Accepted) THEN
		Order.Stat = Approved
		ELSE
		Order.Stat = Executed
		END IF
			Offer.Stat ∈ {Accepted, Executed}

Payment.Stat	Cargo.Stat	IF (Payment.Stat = Received) THEN
		Cargo.Stat = Done
		END IF
			Payment.Stat = Received

Cargo.Stat	Payment.Stat	IF (Cargo.Stat = Done) THEN
		Payment.Stat = Received
		END IF
			Cargo.Stat = Done

recall values are higher than the values of the syntactic approach (as it solved bilateral substitution cases) and lower than the values of the proposed approach (as it could not resolve cases require concept aggregation). On the other hand, the proposed approach managed to generate adapters for all the mutated cases, providing a stable adaptation recall value of one.

However, these results could be misleading, as the experiment is performed with complete CSEG patterns. In practice, an ontology designer may skip some substitution patterns when defining CSEG patterns, depending on his/her domain knowledge and modelling skills. Therefore, the proposed approach will not be able to resolve the cases with missing patterns. In other words, the accuracy of the proposed

Table 11: Matching behavior models using SMP.

S_1 behavior model	S_2 behavior model
S_0, S_1, S_2, S_3, S_4	S_0, S_1
S_5	S_2
S_6	S_3
S_7, S_8, S_9	S_4
S_{10}, S_{11}, S_{12}	S_5
S_{13}	S_6

approach mainly depends on the quality of the defined ontology mappings. To show such effects, we repeated the previous experiment except that we store only a random portion (0%–100%) of the generated substitution patterns. The results are depicted in Figure 17(b). The figure shows that the proposed approach could not resolve all the mutation cases due to missing substitution patterns; however, it succeeds in adapting more cases than the other approaches.

The worst case complexity of the proposed approach is $O(n^3)$, where n is the number of operations in a conversation pattern (a theoretical proof can be found in [10]). In practice, n is expected to be small; hence, we argue that the performance of the proposed approach is acceptable, especially when compared to the time needed for manually developing conversation adapters (which could require several days). We will focus our future research efforts to optimize the proposed algorithms and apply them to real-life application domains, which require involvement of application-domain experts to precisely define the needed CSEG.

10. Case Study

Given a service S_1 and S_2 with GAPs depicted in Table 6. In order to find wether these GAPs are matching or not, we have to extract the behavior models of each GAP. Let us assume the operations definitions as in Tables 7 and 8. Hence, extracted behavior models will be as listed in Table 9. Assuming that we have a CSG segment as depicted in Table 10, and applying the SMP procedure, we will find the matching behavior models states as indicated in Table 11. We can see from the table that S_1 operations *Send—Shipping—Order*, *Get—POL—Allocated*, *Get—POD—Allocated*, *and Get—Costs—Computed* are matching the operation *Send—Cargo—Details* of S_2. Hence, the corresponding adapter method is created; accordingly, the rest of the adapter methods is created by the mappings given in Table 11.

11. Conclusion

In this paper, we have proposed an automated approach for generating service conversation adapters on the fly in dynamic smart environments, where services interact with each other in seamless transparent manner without human intervention. The proposed approach customizes service conversations in a context-sensitive manner by resolving conversation conflicts (signature and/or protocol) using aggregate concept conditional substitution semantics captured by the proposed concepts substitutability extended graph (CSEG) that required to be a part of the adopted application domain ontology. We illustrated how such semantics are used to resolve signature and protocol incompatibilities. We provided the algorithms needed for automatic adapter generation and presented the verifying simulation experiments. Finally, we indicated how the adapter structure is determined and provided the algorithms needed for adapter source code generation. The proposed approach enables services in dynamic environments to smoothly interact with one another without having semantic interoperability concerns, thus increasing the chances for service reuse, and consequently improving the efficiency of dynamic environments. We believe that the proposed approach helps in improving business agility and responsiveness and of course resembles an important step toward achieving the IoS vision.

References

[1] M. Papazoglou and D. Georgakopoulos, "Service oriented computing," *Communications of the ACM*, vol. 46, no. 10, pp. 24–28, 2003.

[2] M. Dumas, M. Spork, and K. Wang, "Adapt or perish: algebra and visual notation for service interface adaptation," in *Business Process Management*, vol. 4102 of *Lecture Notes in Computer Science*, pp. 65–80, 2006.

[3] B. Benatallah, F. Casati, D. Grigori, H. R. Motahari Nezhad, and F. Toumani, "Developing adapters for web services integration," in *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE '05)*, pp. 415–429, June 2005.

[4] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-automated adaptation of service interactions," in *Proceedings of the 16th International World Wide Web Conference (WWW '07)*, pp. 993–1002, May 2007.

[5] R. Mateescu, P. Poizat, and G. Salaün, "Behavioral adaptation of component compositions based on process algebra encodings," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, pp. 385–388, November 2007.

[6] A. Brogi and R. Popescu, "Automated generation of BPEL adapters," in *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC '06)*, vol. 4294 of *Lecture Notes in Computer Science*, pp. 27–39, 2006.

[7] J. Hau, W. Lee, and S. Newhouse, "The ICENI semantic service adaptation framework," in *UK e-Science All Hands Meeting*, pp. 79–86, 2003.

[8] A. Brogi and R. Popescu, "Service adaptation through trace inspection," *International Journal of Business Process Integration and Management*, vol. 2, no. 1, pp. 9–16, 2007.

[9] D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 2, pp. 292–333, 1997.

[10] I. Elgedawy, Z. Tari, and J. A. Thom, "Correctness-aware high-level functional matching approaches for semantic Web services," *ACM Transactions on the Web*, vol. 2, no. 2, article 12, 2008.

[11] I. Elgedawy, "Automatic generation for web services conversations adapters," in *Proceedings of the 24th International Symposium on Computer and Information Sciences (ISCIS '09)*, pp. 616–621, Guzelyurt, Turkey, September 2009.

[12] I. Elgedawy, Z. Tari, and M. Winikoff, "Exact functional context matching for Web services," in *Proceedings of the Second International Conference on Service Oriented Computing (ICSOC '04)*, pp. 143–152, New York, NY, USA, November 2004.

[13] I. Elgedawy, "A context-sensitive approach for ontology mapping using concepts substitution semantics," in *Proceedings of the 25th International Symposium on Computer and Information Sciences (ISCIS '10)*, vol. 62 of *Lecture Notes in Electrical Engineering*, pp. 323–328, London, UK, 2010.

[14] I. Elgedawy, "Conditional ontology mapping," in *Proceedings of the 36th IEEE International Conference on Computer Software and Applications (COMPSAC '12), the 7th IEEE International Workshop on Engineering Semantic Agent Systems (ESAS '12)*, Izmir, Turkey, 2012.

[15] I. Elgedawy, Z. Tari, and M. Winikoff, "Scenario matching using functional substitutability in web services," in *Proceedings of the 5th International Conference on Web Information Systems Engineering (WISE '04)*, Brisbane, Australia, 2004.

[16] I. Elgedawy, Z. Tari, and M. Winikoff, "Exact functional context matching for Web services," in *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC '04)*, pp. 143–152, Amsterdam, Netherlands, November 2004.

[17] F. Casati, E. Shan, U. Dayal, and M.-C. Shan, "Business—oriented management of Web services," *Communications of the ACM*, vol. 46, no. 10, pp. 55–60, 2003.

[18] M. P. Papazoglou and W.-J. van den Heuvel, "Web services management: a survey," *IEEE Internet Computing*, vol. 9, no. 6, pp. 58–64, 2005.

[19] W3C, "Web service choreography interface," 2002, http://www.w3.org/TR/wsci/.

[20] M. Dumas, B. Benatallah, and H. R. M. Nezhad, "Web service protocols: compatibility and adaptation," *IEEE Data Engineering Bulletin*, vol. 31, no. 3, pp. 40–44, 2008.

[21] M. Nagarajan, K. Verma, A. P. Sheth, J. Miller, and J. Lathem, "Semantic interoperability of Web services—challenges and experiences," in *Proceedings of the 4th IEEE International Conference on Web Services (ICWS '06)*, pp. 373–380, September 2006.

[22] Y. Kalfoglou and M. Schorlemmer, "Ontology mapping: the state of the art," *Knowledge Engineering Review*, vol. 18, no. 1, pp. 1–31, 2003.

[23] N. Shadbolt, W. Hall, and T. Berners-Lee, "The semantic web revisited," *IEEE Intelligent Systems*, vol. 21, no. 3, pp. 96–101, 2006.

[24] B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, and U. Sattler, "OWL 2: the next step for OWL," *Web Semantics*, vol. 6, no. 4, pp. 309–322, 2008.

[25] D. Roman, U. Keller, and H. Lausen, "Web service modeling ontology (WSMO)," Feburary 2005, http://www.wsmo.org/TR/d2/v1.1/20050210/.

[26] "OWL-Services-Coalition, OWL-S: semantic markup for web services," 2003, http://www.daml.org/services/owl-s/1.0/owl-s.pdf.

[27] J. Kopecký, T. Vitvar, C. Bournez, and J. Farrell, "SAWSDL: semantic annotations for WSDL and XML schema," *IEEE Internet Computing*, vol. 11, no. 6, pp. 60–67, 2007.

[28] M. Kova, J. Bentahar, Z. Maamar, and H. Yahyaoui, "A formal verification approach of conversations in composite web services using NuSMV," in *Proceedings of the Conference on New Trends in Software Methodologies, Tools and Techniques*, 2009.

[29] L. Ardissono, A. Goy, and G. Petrone, "Enabling conversations with web services," in *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '03)*, pp. 819–826, July 2003.

[30] M. T. Kone, A. Shimazu, and T. Nakajima, "The state of the art in agent communication languages," *Knowledge and Information Systems*, vol. 2, no. 3, 2000.

[31] M. B. Juric, *Business Process Execution Language for Web Services BPEL and BPEL4WS*, Packt Publishing, Birmingham, UK, 2nd edition, 2006.

[32] V. Kashyap and A. Sheth, "Semantic and schematic similarities between database objects: a context-based approach," *The VLDB Journal*, vol. 5, no. 4, pp. 276–304, 1996.

[33] S. Abiteboul, S. Cluet, and T. Milo, "Correspondence and translation for heterogeneous data," *Theoretical Computer Science*, vol. 275, no. 1-2, pp. 179–213, 2002.

[34] H. Chalupksy, "Ontomorph: a translation system for symbolic knowledge," in *Proceedings of the 17th International Conference on Knowledge Representation and Reasoning*, Breckenridge, Colo, USA, 2000.

[35] S. Li, H. Hu, and X. Hu, "An ontology mapping method based on tree structure," in *Proceedings of the 2nd International Conference on Semantics Knowledge and Grid (SKG '06)*, November 2006.

[36] P. Ganesan, H. Garcia-Molina, and J. Widom, "Exploiting hierarchical domain structure to compute similarity," *ACM Transactions on Information Systems*, vol. 21, no. 1, pp. 64–93, 2003.

[37] J. Madhavan, P. A. Bernstein, P. Domingos, and A. Y. Halevy, "Representing and reasoning about mappings between domain models," in *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI '02)*, pp. 80–86, August 2002.

[38] I. Elgedawy, B. Srivastava, and S. Mittal, "Exploring queriability of encrypted and compressed XML data," in *Proceedings of the 24th International Symposium on Computer and Information Sciences (ISCIS '09)*, pp. 141–146, Guzelyurt, Turkey, September 2009.

[39] J. Pearson and P. Jeavons, A survey of tractable constraint satisfaction problems CSD-TR-97-15, Oxford University, Computing Laboratory, Oxford, UK, 1997, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.9045.

[40] P. G. Jeavons and M. C. Cooper, "Tractable constraints on ordered domains," *Artificial Intelligence*, vol. 79, no. 2, pp. 327–339, 1995.

[41] I. Elgedawy, "A conceptual framework for web services semantic discovery," in *Proceedings of On The Move (OTM) to Meaningful Internet Systems*, Catania, Italy, 2003.

[42] Y. Taher, D. Benslimane, M.-C. Fauvet, and Z. Maamar, "Towards an approach for web services substitution," in *Proceedings of the 10th International Database Engineering and Applications Symposium (IDEAS '06)*, pp. 166–173, December 2006.