IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 30, 2017, accepted February 20, 2018, date of publication March 1, 2018, date of current version April 4, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2810848

D3L-Based Service Runtime Self-Adaptation Using Replanning

XIANGHUI WANG[1,2], (Member, IEEE), ZHIYONG FENG[1], (Member, IEEE), AND KEMAN HUANG[3], (Member, IEEE)

[1]Department of Computer Science and Technology, Tianjin University, Tianjin 300072, China
[2]Department of Computer Science and Technology, Shandong Jianzhu University, Jinan 250101, China
[3]Sloan School of Management, MIT, Cambridge 02142, USA

Corresponding author: Keman Huang (keman@mit.edu)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB1401201 and in part by the National Natural Science Foundation of China under Grant 61502333, Grant 61572350, and Grant 61503220.

ABSTRACT For business processes based on micro service architecture in an enterprise, failures often occur because of modified business rules or goals, change of service availability, and dynamical running environment. Based on dynamical replanning technologies, these failures can be repaired at runtime. However, semantic conflicts among services from different providers can greatly decrease response efficiency and response success rate of a business goal. In this paper, we propose a novel service runtime self-adaptation framework to decrease response time and raise success rate. Distributed dynamic description logic is utilized to eliminate semantic conflicts among services and provide basic models for carrying out planning among services. Considering inputs, outputs, preconditions, and effects properties of services, local and global planning algorithms based on artificial intelligence graph planning are designed. Local planning can rapidly search a service-based path only including services from a provider, and global planning can try to explore a path including services from multiple providers. Based on these two algorithms, local and global replanning strategies are designed to handle runtime exceptions at service level and path level. We implement a prototype system by means of workflow engine Activiti and business process language BPMN2.0. Experiments show that compared with previous works, our framework can guarantee higher efficiency and success rate.

INDEX TERMS Running self-adaptation, software service, graph planning, D3L, workflow.

I. INTRODUCTION

With the popularity of micro service architecture [1], in an enterprise, more and more services related to business operations emerge. These services may be developed by different departments of this enterprise, or directly obtained from different service platforms[1,2,3] on Internet. Based on them, various business processes in the enterprise either are manually created by domain experts by means of special tools [2], [3], or automatically generated with the help of service composition technologies [4]–[8]. Thus, a service ecosystem [9] within the enterprise is formed, including services, service providers, and business processes. However, a business process tends to encounter failures under dynamical running environment. For example, business rules or goals are modified; running environment is interrupted; or various exceptions over services (unavailable, unexpected effects, execution exceptions, etc.). To guarantee robustness of the service ecosystem, self-adaptation mechanisms for failures at runtime are urgently required.

Currently, based on artificial intelligence (AI) planning technology [10], some service runtime self-adaptation approaches are proposed [11]–[15]. In these approaches, when a failure occurs at runtime, an adaptation requirement is determined immediately according to current runtime context; then, a planning operation is utilized to search a currently feasible path for the requirement from all available services; at last, the new path is executed by means of off-the-shelf workflow engines [14], [16], or special process execution procedures [17], [18]. If the new path succeeds to run, the failure is repaired successfully.

However, off-the-shelf service runtime self-adaptation approaches always assume that all services are annotated by one domain ontology. And a service ecosystem within an enterprise may adopt distributed ontologies to annotate

[1]https://www.programmableweb.com 2017-12-23
[2]https://www.juhe.cn/ 2017-12-23
[3]https://www.jisuapi.com/ 2017-12-23

TABLE 1. Related works about service runtime self-adaptation using replanning.

Literature	planning	Running mechanism	Adaptation mechanism		
			Exception	Strategy	Context
[14], [15]	UCPOP planner	YAWL	UnPre& UnExe&UnEff	Local	Predicate facts
[23]	RuG planner	Special	Variable changed	Local	Variables in dependency scopes
[12], [17]	Classical planner	Special	UnPre	Local	State about domain objects
[6]	HTN and CSP	Workflow	UnExe	Global	No
[18]	Simplanner	Special	UnPre& UnExe	Local& global	Predicate facts
[13]	process-fragment	Apache ODE	Feature changed	Local& global	Features in variability models
[16]	IOPE-based planner	Apache ODE	UnPre& UnExe& UnEff LocalAdaptFail	Local& global	Parameter and predicate facts, execution histories, and available services
D3LSRAF	Distributed Planners	Activiti	UnPre& UnExe& UnEff LocalAdaptFail	Local& global	Parameter and predicate facts and available services, execution histories, and they are organized by SKBs

service semantics, because services may be provided by different providers (departments or platforms). This brings two new problems for these off-the-shelf approaches. The first is that unhandled semantic conflicts among different ontologies would hinder the judgment about interoperability among services. For example, *Person* and *Passenger* express the same semantics, although they are from different domain ontologies; effect fact *personAtAddress(Tom, Jinan West)* is true in an domain ontology, and then *personAtStation(Tom, Jinan West)* also is true in another domain ontology, because that *Jinan West* is not only an instance of *Address*, but also of *Station*. If no measure is adopted for these conflicts, the response success rate of business goals would decrease. The second is, for each self-adaptation, the searching space of these approaches always is created by all services, and this is time-consuming when only a few of services from one provider are used in the self-adaptation.

To solve the problems above, we propose a D3L-based service runtime self-adaptation framework (D3LSRAF) that can provide higher response success rate and efficiency. Here, all services annotated by the same domain ontology can be modeled as a DDL (Dynamic Description Logic) system [19], [20], namely *Service Knowledge Base* (SKB). In each SKB, existing automatic service composition approaches [8] can be used to plan a service-based path for a given request. Multiple SKBs are modeled as a D3L (Distributed Dynamic Description Logic) system [21], [22], namely *Distributed Service Knowledge Base* (DSKB), where various bridge rules are created to solve semantic conflicts among concepts, relations among SKBs. For a service request that annotated by distributed ontologies, we design local planning and global planning algorithms to search a service-based path from DSKB. Based on these algorithms, two self-adaptation strategies are created: local and global replanning strategies. Both strategies can play an important role to guarantee higher response success rate and efficiency of a service ecosystem. The main contributions of this paper are in the following folds:

1) D3L-based local and global planning algorithms are provided. For a service request annotated by distributed ontologies, local planning can rapidly search a service-based path only including services from a SKB, and global planning can plan a path including services from multiple SKBs to the best extent possible.
2) A novel service runtime self-adaptation framework is proposed, and it can dynamically repair various failures at runtime by means of local and global replanning strategies.
3) Based on Activiti engine and BPMN2.0 language, a prototype system is developed. A series of experiments show that, compared with previous self-adaptation approaches, our framework has better response efficiency and higher response success rate.

The remainder of this paper is organized as follows. Section 2 describes related works. Section 3 proposes an overview about D3LSRAF. Section 4 presents some formal definitions, describes two D3L-based planning algorithms, and illustrates the self-adaptation mechanism based on replanning technology. Section 5 presents implementation details about a prototype system of D3LSRAF. Section 6 reports the empirical results. Section 7 concludes the paper.

II. RELATED WORKS

A. SERVICE RUNTIME SELF-ADAPTATION USING REPLANNING

Recently, there are some approaches related to service runtime self-adaptation using replanning. The core operations in these approaches are planning, running and self-adaptation. When a failure occurs over a service-based process at runtime, the self-adaptation operation would invoke the planning operation to replan a new path, and then invoke the running operation to execute the path. And execution of the path can repair the failure and guarantee that the final goal of this process is achieved. These approaches adopted different planning approaches, running mechanism and self-adaptation mechanism, as shown in Table 1.

Based on a process execution engine YAWL, references [14] and [15] provided a business process runtime self-adaptation mechanism. Various exceptions, which needed to be adapted, were defined in a business process definition. When an exception occurred at runtime, the monitor component would create a planning problem according to the physical facts in the context and precondition/effects of fault

task, and then a planning service was invoked to repair the exception. The planning service would employ an AI planner based on a partial order planning algorithm to find a plan fulfilling the adaptation goal. And then the plan was converted to a recovery process and ran over the YAWL. It is noticed that these exceptions could be set before or after a service task in the process. Thus *UnPre* (preconditons unsatisfied), *UnExe* (excution Failure), and *UnEff* (unexpected effects) [16] for each service can be repaired at runtime.

Reference [23] used dependency scopes (DSs) to specify the correct execution for critical parts of business processes. And when a volatile process variable was modified, an intervention process was generated by a domain-independent RuG planner according the adaptation goal specified in corresponding DS declaration. Just as the approach in [14], self-adaptation operations are triggered by the predefined exception monitoring points (volatile variables) in process definitions. However, exceptions in adaptation or intervention processes can't be adapted, because these processes are generated dynamically and no any exception monitoring point is set in their definitions. Once these exceptions occur, the main process would fail to terminate.

References [12] and [17] provided a self-adaptation approach using context-aware replanning. The approach could continuously monitor the abnormal situation according current context and automatically adapt the main process at runtime. The context is defined by a set of domain objects. Before an activity in a process ran, the current context was checked. When some domain objects had abnormal status or an abstract activity runs, the process would terminate, and then an AI planning would be invoked to compose existing process fragments into an adaptation process. In this approach, the concrete monitoring and process running mechanism can't be introduced. Different with previous two approaches, the approach didn't predefine any exception monitoring points in the original process definitions, and it also could handle abnormal situations of generated adaptation processes. However, when no adaptation process was found, or an adaptation process failed to run, no further adaptation measure for the final business goal was adopted.

In the approaches above, the adaptation goals are preconditions or effects of faulted services, and when corresponding adaptation operations fail, the final business goals can't be achieved. Here, we call these adaptation strategies as local adaptation. Accordingly, if an adaptation strategy targets the final business goals as adaptation goals, it is called global adaptation. There are some approaches to provide global adaptation strategies [6], [13], [18].

Reference [13] proposed a framework that used variability models to support the runtime adaptation of service compositions. The variability models recorded various variants for a business process, and feature models were created for these variants. An exception at runtime could be monitored according the changes of features in the running context, and then was solved through removing and adding some process fragments in the main process. These fragments could be searched by means of SQL-like language from predefined variability models. And the workflow engine Apache ODE was employed to execute corresponding process fragments. The variability models constructed a whole solution space for a business goal, and the approach could theoretically support local and global adaptation. However, the construction of variability models is complexity and closely depends on domain specialists.

Traditional automatic service composition (ASC) approaches generally included four phases: planning, discovery, selection, and execution. Reference [6] presented a scalable architecture for ASC by means of nested workflow management. When an exception occurred at runtime, the nested workflow management module would invoke HTN and CSP-based planners to generate a new path for the final goal, and the path was automatically deployed to a workflow engine to run. Each repair in this approach targeted the final business goal, and didn't consider any local repair, such as local replacement etc. Therefore, this approach only provides global adaptation strategies, and each repair is time-consuming.

Reference [18] used a Simplanner to generate an initial path rapidly, and could solve information lose and service unavailable problem during the path ran. When a user couldn't provide necessary input parameters before a service in the initial path ran, the approach would plan a new path that could produce these parameters. After the new path succeeded to run, the service continued to run. When the service was unavailable, the approach would plan and run a new path for the final goal. Therefore, local and global adaptation strategies both could be supported by this approach. However, just like the approach in [6], for service unavailable problem, it still did't try any local adaptation strategy. This decreased the adaptation efficiency of this approach.

In order to overcome the shortcomings of approaches above, we provided a service runtime self-adaptation framework (ASAF) for local and global adaptation in our previous works [16]. We presented the formal definition for four local self-adaptation exceptions: *UnPre*, *UnExe*, *UnEff*, and a global exception *LocalAdaptFail*. Given a service-based process, the framework firstly converted it into a process for self-adaptation automatically, where local and global self-adaptation exception monitoring points are respectively set at service level and at process level automatically. When an exception occurs at runtime, corresponding exception units would invoke an IOPE-based AI planner [8] to search an adaptation process, and then ran the process by means of the workflow engine Apache ODE. The adaptation process in the local exception unit would replace the execution of the faulted service and didn't affect the running of other parallel services. And the adaptation process in the global exception would replace the execution of the main process. Specially, all adaptation processes also could run self-adaptively.

Although ASAF can solve the service runtime self-adaptation problems, it assumes that all services are annotated by one domain ontology, and the searching space for each

invocation of the AI planner includes all services regardless of local and global adaptation. However, in practice, available services may be from different providers, and ontologies adopted by different providers may be different. Semantic conflicts among different ontologies can affect the interoperation of services and decrease the response success rate. Moreover, most local exceptions at service level can be solved by few services from one provider. If the searching space for local exceptions is limit as all services from different providers, the response is time-consuming. Therefore, distributed semantic service composition technology is needed.

D3LSRAF in this paper improve ASAF through considering distributed domain ontologies. For local adaptation, a local planner is adopted to rapidly search a service-based path respectively in each SKB. And for global adaptation, a global planner is used to search a path from services in all SKB. Thus, compared with other approaches, the local planner can greatly reduce the adaptation time when a local adaptation runs, because it has smaller searching space; meanwhile, when the local adaptation fails to run, the global planner can improve the response success rate through attempting all possible paths for the original business goal. These improvements are implemented by means of D3L. Related works about D3L will be discussed in the next section.

B. D3L AND SEMANTIC SERVICE COMPOSITION

Dynamic description logic (DDL) combines description logic and action theory, and is a model to depict real world from dynamic and static respects [19]. It gives unified and logical syntax and semantics for concept, formula, and action. DDL has three components: *TBox*, *ABox*, *ActBox* [20], where, *TBox* is a set of assertions related to concepts and relationships, *ABox* is a set of individual assertions, *ActBox* is a set of atomic actions. Each atomic action in *ActBox* is modeled a tuple $\langle Pa, Ea\rangle$, *Pa* and *Ea* respectively represent satisfied preconditions before execution and effects after execution. Elements in *Pa* and *Ea* come from *ABox*.

A semantic service can be modeled into an atomic action in DDL. Thus service composition problem can be converted into a DDL-based planning problem $P=\langle T, Ac, A, G\rangle$ [24]. Here, *T* is a *TBox* expressing concepts and relationships related to this problem; *Ac* is a *ActBox* describing atomic actions related to web services; *A* and *G* both are *ABox* respectively describing initial state and goal state of the planning problem. Therefore, finding a service composition solution is converted into finding planning path for the planning problem. Here, semantics in a DDL are from one domain ontology.

Distributed dynamic description logic (D3L) is expansion of DDL to deal with distributed and heterogeneous knowledge [21], [22]. It has four components. The first is a set of independent DDL systems, and each system has its own knowledge representation form and reasoning mechanism. The second is distributed *TBox (DTB)* to record link rules between concepts, relationships and actions. The third is distributed *ABox (DAB)* to record link rules between individuals. The fourth is the reasoning mechanism based on *DTB* and *DAB*. The reasoning on D3L system has two ways: local reasoning and global reasoning. The former is to reason in each independent DDL system. The latter links all DDL systems as a whole by link rules, and then carries on the reasoning considering all DDL systems.

It is noticed that the link rules in *DTB* and *DAB* are core to eliminate semantic conflicts among different domain ontologies, also called bridge rules. These rules can be generated by means of the state-of-the-art ontology alignment approaches [25], [26], which can be utilized to produce *equivalence*, *subclass* or *sameAs* relations among concepts and individuals in domain ontologies.

Based on D3L, a semantic service composition problem under distributed domain ontologies can be modeled as a local or global planning problem on multiple DDL systems. However, existing local and global reasoning algorithms [27] for D3L are not usable, because there are some differences between actions in DDL and services. For example, except for preconditions and effects, inputs and outputs also are needed for a service; some effects only describe the world permanent state and can't be altered, such as *addressAtCity(sdjzu,Jinan)*, but others present the temporary state and can be canceled by another service, such as *trainIsBooked(D404, Tom, 2017-11-15)*. Therefore, special reasoning algorithms are needed for distributed service composition problem. In this paper, we model each planning is a D3L-based reasoning problem, and design special local and global planning algorithms to generate a service-based path through improving classical AI graph planning algorithm [10].

C. RUNTIME CONTEXT AND HUMAN SERVICE

In previous works, runtime context provided necessary information for solving runtime self-adaptation problem, and they can be used to determine abnormal running situations and various adaptation requirements. In [17], the context is composed by various domain objects, and status of all objects can affect current running state of a business process. In [13] and [23], process variables related to various dependence scopes and process features are respectively put in the context. Their values can affect the main process structure changes. In most works [14]–[16] and [18], the execution effects of each service in a business process are recorded in the context. Through continuously comparing the physical context with expected preconditions and effects of services, *UnPre* and *UnEff* exceptions can be automatically determined. And the initial state of various adaptation requirements can be directly obtained from current context.

In order to support various adaptation strategies, ASAF in our previous work utilized the context to record the running state, current available services, and execution histories of a service-based process. The running state includes output parameter facts and effects facts of completed services, and they play an important role in determining abnormal situation and the initial state of adaptation requirements.

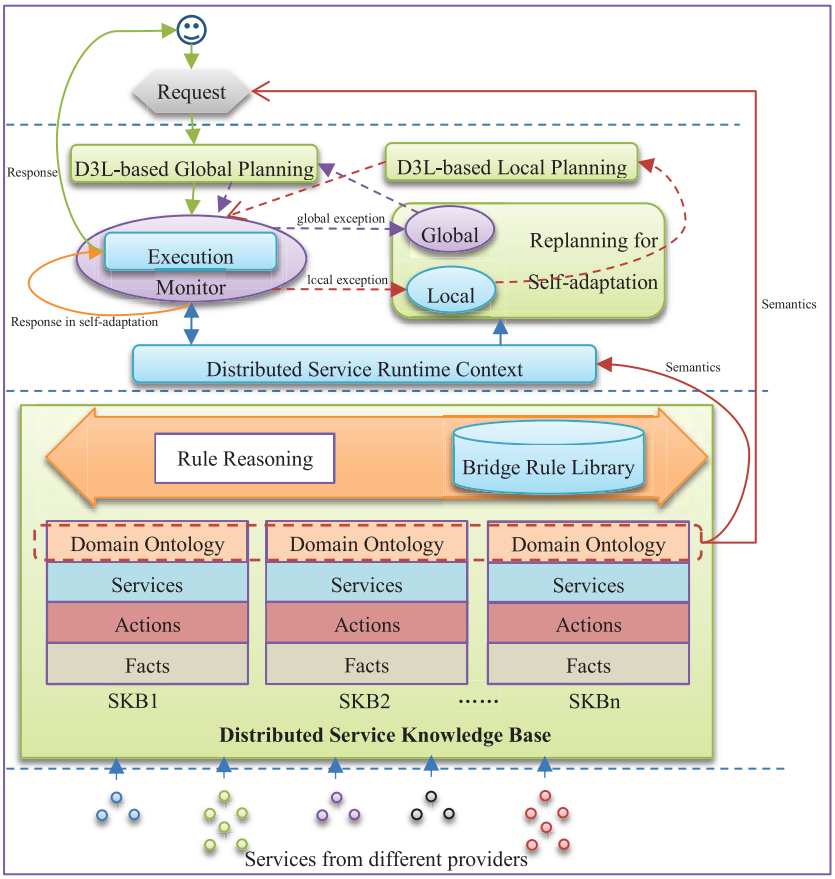

FIGURE 1. D3LSRAF at concept level.

Current available services can be used to define the problem domain of planning for corresponding adaptation requirement. And execution histories can be used in a global adaptation to cancel the completed effects.

However, under distributed ontologies, the context may include objects or state literals annotated by multiple domain ontologies. And it can't be consider in previous work. Therefore, in this paper, we propose *Distributed Service Runtime Context* concept to support context with distributed semantics.

Furthermore, in practice, there are some tasks that can't be replaced by software services, such as *taking taxi*, *checking in hotel*, etc. And these tasks generally can play an important role during planning a complex service-based path. For example, in a whole travel process, *searching* and *booking train*, and *calling a taxi* are main services that can be obtained from the Internet, but, *taking taxi* and *taking train* also are the key human tasks in the travel. In order to guarantee the wholeness of the travel process, these human tasks also should be choreographed together with those real services. From the view of SOA [28], these tasks can be considered as services (human services), and can be described by unified interface description language just as services on Internet [29]. In this paper, we consider a human service as a common software service, and also use IOPE to describe its function.

III. D3LSRAF OVERVIEW

D3LSRAF is a service runtime self-adaptation framework, and uses D3L-based replanning technology to self-adaptively respond a service request on a distributed service knowledge base. Fig. 1 shows an overview of D3LSRAF at concept level.

Distributed service knowledge base (DSKB) is the basis of D3LSRAF, and it organizes services from different providers into multiple service knowledge bases. Those services from the same domain ontology are expressed as a service knowledge base (SKB). And each SKB records not only static description including semantics for IOPE and invocation details at syntactic level, but also dynamic features about known facts (individuals and their relations) and available service instances (actions for planning). To eliminate semantic conflicts among different SKBs, various bridge rules are created before the framework runs, mainly including *subclass* and *equivalence* rules among concepts and predicates in different domain ontologies. Rule reasoning is needed to obtain new rules according to created rules when a new SKB is added.

A request handled by D3LSRAF is composed of known facts, desired parameters, and desired effects. Semantics of these components are from multiple domain ontologies in DSKB. When a request is received, D3LSRAF firstly uses D3L-based global planning to search a path including

services from multiple SKBs (global path for short) for this request from DSKB; then, it executes the path according known facts in the request and automatically monitors various exceptions at runtime.

During the running, the path would encounter three running situations, and D3LSRAF would adopt various strategies to deal with them. The first is that no exception occurs. Thus, D3LSRAF would response the user successfully, that is, return the values for desired parameters and achieve the desired effects. The second is that an exception at system level occurs, such as process executor breaks down. D3LSRAF would response the user with corresponding failure information. The third running situation is that a service fails to run because an exception that may be repaired occurs, such as the preconditions not satisfied, unavailable etc. Thus, D3LSRAF uses local replanning to repair the faulted service at runtime. The local replanning firstly creates an adaptation request for the repair according current distributed service runtime context; then, it uses D3L-based local planning to rapidly search a path only including services from single SKB (local path for short) for the adaption request; next, it executes the local path. If the local replanning succeeds, the running of the original path doesn′t be affected; Otherwise, D3LSRAF terminates all running services in the path, and adopts global replanning to repair the whole path. The global replanning firstly creates an adaptation request that can achieve the desired parameters and effects of the main path according the runtime context; secondly, it searches a global path through exploring all possible paths for the original service request; thirdly, it executes and monitors the global path. If the global replanning succeeds, D3LSRAF would response the user successfully. Otherwise, it would response the user with fault information. Specially, all new paths during replanning also run self-adaptively.

Distributed service runtime context provides key information for the self-adaptively running of service requests. It records running status, running histories of each request, and current available services. The running state includes known facts produced by completed service instances. These facts are continuously updated and monitored during the running of these requests. Various exceptions can be immediately thrown when the physical running state is inconsistent with the expected state. Specially, semantics of these facts may be from distributed domain ontologies because service instances that produce them may be from different SKBs. The running histories of a path record which fact is produced by which service instance, and they are used in the global replanning to cancel those redundant effects in current running state.

IV. DETAILS FOR D3LSRAF

A. PROBLEM MODELING

In D3LSRAF, various services are organized into SKBs according to domain ontologies that they use, and their *IOPE* functional properties are annotated. Considering the dynamic respects of services, we model each a SKB as dynamic description logic (DDL) system that combines description logic and action theory, shown in Definition 1.

Definition 1 (Service Knowledge Base (SKB)): A service knowledge base is a DDL system, and is expressed as a tuple $skb = \langle D, TP, Facts, Actions \rangle$, where,

- D is a *TBox* of this DDL system, and includes semantic concepts and their relation assertions;
- TP is a set of services, and their *IOPE* are annotated by D;
- $Facts$ is an *ABox* of this DDL system and includes two types of known facts: parameter facts(*IOFacts*) and predicate facts (*PEFacts*) [16], and these facts also are annotated by D;
- $Actions$ is an *ActBox* of this DDL system, and they can be generated by services in TP according to different assignments for input parameters. That is, there is a many-to-one mapping from $Actions$ to TP.

According to Definition 1, one domain ontology can be used by multiple SKBs, but, one SKB only uses one domain ontology. In practice, the cooperation between services from different SKBs with different domain ontologies is difficult, because there are many semantic conflicts among these domain ontologies. For example, *skb1*, *skb2*, *skb3* are three SKBs, an instance of concept *Station* in *skb1* can′t be considered as the instance of *Address* in skb3, but, *Station* naturally is a subclass of *Address*; concept *Person* in *skb1* is not equal to *Passenger* in *skb2*, but, both represent a human.

To achieve the reasoning between services from different SKBs, we use D3L to describe distributed and heterogeneous knowledge in multiple SKBs. In D3L, various bridge rules are the key to eliminate semantic conflicts among DDL systems. Basic bridge rules among concepts, relations, actions, and individuals in different DDL systems had been defined and are used in reasoning [21]. However, there are two main differences between a DDL system and a SKB. The first is that all individuals and actions in the DDL systems are known before a reasoning runs, however, *Facts* (corresponding to individuals in DDL) and *Actions* (corresponding to actions in DDL) components in SKBs are unknown before a reasoning runs. In SKBs, the reasoning is used to plan a service-based path according the cooperation relation among services, and it mainly depends on the concepts and relations in the component D of SKBs. The second is that equivalence and equivalence with conditions relationship among concepts and relations in different SKBs often are used during the reasoning among SKBs. For an instance, *personAtAddress(Person, Address)* and *personAtStation(Person, Station)* are relation predicates respectively in the SKB *skb1* and *skb2*. *Person* in the two SKBs are equal concepts, and *Station* in *skb2* is the subclass of Address in *skb1*. If the fact *personAtStation(Person: Tom, Station:s1)* in *skb2* is *true*, then it is reasonable that the fact *personAtAddress(Person: Tom, Address:s1)* in *skb1* also is true. Here, *Tom* and *s1* respectively are instances of *Person* and *Station*. Meanwhile, if *personAtAddress(Person: Tom, Address:s1)* in *skb1* is *true* and *s1* is the instance of *Station*, and then we can assert that

personAtStation(Person: Tom, Station:s1) is *true* in *skb2*. That is, the literals of *personAtAddress(Person, Address)* are equal to *personAtStation(Person, Station)* when its second variable is an instance of *Station* except for the same parameter assignment. Considering the two differences above, we improve old bridge rules to make them be used for the reasoning among SKBs. The rules only specify the links among concepts and relations in different SKBs, shown in Definition 2.

Definition 2: Let KB_i and KB_j be two SKBs, satisfying $KB_i.D \neq KB_j.D$, concept $a \in KB_i.D$ and $b \in KB_j.D$, relation $c \in KB_i.D$ and $d \in KB_i.D$, and then a bridge rule from KB_i to KB_j is one of the followings:

- $KB_i : a \xrightarrow{\subseteq} KB_j : b$ represents a is the subset of b, that is, a can replace b at semantic level;
- $KB_i : a \xrightarrow{\supseteq} KB_j : b$ represents a is the superset of b, that is, a can be replaced by b at semantic level;
- $KB_i : a \xrightarrow{=} KB_j : b$ represents a is the equivalent of b;
- $KB_i : c \xrightarrow{\subseteq} KB_j : d$ represents that if c is *true* in KB_i, and then d is *true* in KB_j when they have the same assignments;
- $KB_i : c \xrightarrow{\supseteq} KB_j : d$ represents that if d is *true* in KB_j, and then c is *true* in KB_i when they have the same assignments;
- $KB_i : c \xrightarrow{=F} KB_j : d$ represents c in KB_i is *true* if and only if d is *true* in KB_j when they have the same assignments and the condition F is *true*, where F is a limited condition for semantics of variable values in c.

In the following, we call the first and fourth rules *into* bridge rules; the second and fifth rules *onto* bridge rules; the third and sixth rules *equivalence* bridge rule. For simplicity, given two literals a, b, and a rule x, we use $a \xrightarrow{x} b$ to represent that corresponding relation predicates *a.pred* and *b.pred* satisfy $a.pred \xrightarrow{x} b.pred$, and that a and b have the same assignment.

In D3LSRAF, bridge rules are created by an offline rule reasoning operation. The operation firstly adopt existing ontology alignment technologies [25], [26] to determine *equivalence*, *into*, *onto* rules according to *equivalence* and *subclass* relations between concepts and predicates in two SKBs; then adjust the reasoning results manually to guarantee the correctness. When a new SKB is added, all rules between the SKB and each existing SKB are created according to the process above. Ultimately, all rules are obtained between any two SKBs. The rule reasoning operation is time-consuming and semi-automatically, therefore, it is completed before D3LSRAF responds service requests.

Definition 3 (Distributed Service Knowledge Base (DSKB)): A distributed service knowledge base is a D3L system, and is expressed as a tuple $dskb = \langle SKBs, BRs \rangle$, where,

- *SKBs* is a set of SKBs, and each SKB conforms to Definition 1;
- *BRs* is a *DTB* of this D3L system, and includes all bridge rules among *SKBs*.

Generally, a service request is expressed as a tuple: $rq = \langle In, Init, Out, Goal \rangle$,where *In* represents known parameter facts, *Init* represents known predicate facts, *Out* represents desired output parameters, and *Goal* represents desired effects. All facts and their semantics in the request are from one SKB. Here, for convenience, we call the service request *Single Domain Service Request* (*Single-SR*). However, if we want to search a path from a DSKB, a service request with distributed semantics is expected. Here, it is called *D3L-based Service Request* (Definition 4).

Definition 4 (D3L-Based Service Request (D3L-SR)): Let *ds* be a DSKB, a service request based on *ds* is a tuple $rq = \langle In, Out, Init, Goal \rangle$, where,

- *In* is a distributed *ABox*, satisfying

$$In \subseteq \bigcup_{\forall skb \in ds.SKBs} skb.IOFacts. \tag{1}$$

  representing known parameter facts, and each parameter facts is a tuple $\langle skb, semType, value \rangle$ where the components respectively represent the SKB providing semantics, its semantic type, and its real value;
- *Init* is a distributed *ABox*, satisfying

$$Init \subseteq \bigcup_{\forall skb \in ds.SKBs} skb.PEFacts. \tag{2}$$

  representing known predicate facts, and each predicate fact is a tuple $\langle skb, pred, literal \rangle$ where the components respectively represent the SKB providing semantics, corresponding predicate, and the literal that grounded by parameter facts in *In*;
- *Out* is a desired output parameter set, and each parameter is a tuple $\langle skb, semType, paraname \rangle$ where the components respectively represent the SKB providing semantics, its semantics, and unique ID;
- *Goal* represents desired goal state, and each element is a tuple $\langle skb, pred, literal \rangle$, where the components respectively represent the SKB providing semantics, corresponding predicate, and the literal that grounded by parameters *In* and *Out*.

According to Definition 4, if a DSKB only has one SKB, all *D3L-SR* requests on the DSKB are *Single-SR*.

Given a *D3L-SR* $rq = \langle In, Init, Out, Goal \rangle$, if there is a parameter fact set Out' that can provide an assignment for *rq.Out*, and then *rq.Goal* can be grounded by Out' as predicate facts $Goal'$. Thus, a new service request $rq' = \langle In, Init, Out', Goal' \rangle$ is called a concrete *D3L-SR*.

Given a *Single-SR* based on a SKB, considering known parameter and predicate facts as an initial state, desired outputs and effects as a goal, and the *Actions* in the SKB as a problem domain, then searching a solution for the request is converted into a classical AI planning problem in the SKB. Thus, existing planning algorithms for service composition can be used to obtain a solution [8]. The solution is a service execution path that can be defined

as a sequence $\langle a_{11}, \cdots, a_{1n}, \cdots, a_{k1}, \cdots, a_{kp}\rangle$ where each element is an action set and called an execution step [8]. However, for a *D3L-SR* based on a DSKB, existing planning algorithms can't be used, because they can't consider distributed domain ontologies and can't solve the semantics conflicts among SKBs. Here, we call this planning problem *D3L-based Service Planning Problem* (Definition 5).

Definition 5 (D3L-Based Service Planning Problem (D3L-SPlan)): Given a DSKB *ds*, and a *D3L-SR rq*, A D3L-based service planning problem is to search an action execution path $p = \langle a_{11}, \cdots, a_{1n}, \cdots, a_{k1}, \cdots, a_{kp}\rangle$ from *ds.SKBs*, that can achieve *rq* with the help of *ds.BRs*.

In the following, we use a tuple $dp = \langle ds, rq\rangle$ to represent a *D3L-SPlan*. An action execution path for a *D3L-SPlan* is composed of service instances, and also is called *service-based path*. Specially, the path is called *local path* if it only includes services from one SKB, and is called *global path* if it includes services from different SKBs.

When a global path runs, it may produce facts annotated by different domain ontologies, because it may include services from different SKBs. To support various runtime self-adaptation, the service runtime context of a request will record these facts and services producing them. To distinguish with service runtime context under single ontology [16], we call this context *Distributed Service Runtime Context* (Definition 6).

Definition 6 (Distributed Service Runtime Context (DSRunContext)): Given a DSKB *dskb*, *sr* is a *D3L-SR* for *dskb*, a distributed service runtime context for *sr* at moment *t* is a tuple $\varepsilon_{sr(t)} = \langle Ds, Fs, As, DF, CFs, AF\rangle$, where,

- Ds is a set of domain ontologies in *dskb*, satisfying $Ds = \bigcup_{skb \in dskb.SKBs} skb.D$;
- Fs is a set of known facts currently, satisfying $Ds = \bigcup_{skb \in dskb.SKBs} skb.Facts$;
- As is a set of service instances, satisfying $Ds = \bigcup_{skb \in dskb.SKBs} skb.Actions$;
- DF is a many to one mapping $DF : Fs \rightarrow Ds$, and represents which facts in Fs are annotated by which ontology in Ds;
- CFs is a set of revocable facts, satisfying $CFs \subseteq Fs$;
- AF is a many to one mapping $AF : CFs \rightarrow As$, and represents which facts are produced by which service instances.

In D3LSRAF, *DSRunContext* will be monitored continuously to determine abnormal running situations, and is utilized to generate various adaptation requests for these abnormal situations. Specially, the revocable facts are those world-altering effect facts that can be canceled, such as *trainIsBooked(Tom, D404, 2017-01-02)* etc. And it is specified manually in the effect description of services, as in [16].

B. D3L-BASED LOCAL PLANNING

In D3LSRAF, the D3L-based local planning is used in a local replanning strategy, and it can rapidly search an adaptation path for a *D3L-SR* request. To raise planning efficiency, it should search the path from as few as possible services. Here, we design a *D3L-based local planning algorithm*(*LocalD3LPlanning*), which can concurrently search a local path for given request in each SKB. Once a path is found from one of SKBs, the algorithm terminates. Given a *Single-SR* based on a SKB, the searching in the SKB can be achieved by existing planning approaches [8]. However, requests in D3L-based local planning are *D3L-SR*. Therefore, the algorithm needs to convert a *D3L-SR* request into a *Single-SR* before the searching in a SKB runs. Bridge rules play a key role during the conversion. Let *kb* be a SKB, *BRs* be a set of bridge rules, *rq* be a *D3L-SR*, *rq'* be a *single-SR*, the conversion illustration from *rq* to *rq'* in *kb* is shown in the following:

- For $\forall p \in rq.In$, if $\exists sem \in kb.D$, satisfying $p.semType \overset{\subseteq/=}{\rightarrow} sem$, and then put $\langle kb, sem, p.value\rangle$ into *rq'.In*;
- For $\forall p \in rq.Init$, if $\exists pred \in kb.D$, satisfying $p.pred \overset{\subseteq/=/=_F}{\rightarrow} pred$, and then put $\langle kb, pred, replace(p.literal, p.pred, pred)\rangle$ into *rq'.Init*, where *replace()* represents replacing *p.pred* in *p.literal* with *pred*;
- For $\forall p \in rq.Out$, if $\exists sem \in kb.D$, satisfying $p.semType \overset{\supseteq/=}{\rightarrow} sem$, and then put $\langle kb, sem, p.paraname\rangle$ into *rq'.Out*;
- For $\forall p \in rq.Goal$, if $\exists pred \in kb.D$, $p.pred \overset{\supseteq/=}{\rightarrow} pred$, and then put $\langle kb, pred, replace(p.literal, p.pred, pred)\rangle$ into *rq'.Goal*.

Here, $a \overset{x/y}{\rightarrow} b \Leftrightarrow a \overset{x}{\rightarrow} b \vee a \overset{y}{\rightarrow} b$.

Algorithm 1 D3L-based local planning (LocalD3LPlanning)

Inputs: *ds*: a DSKB, *rq*: a D3L-SR

Outputs: a local service-based path in *ds* and it can achieve *rq*

```
01. c=null; //the solution is initialized by null
02. FOR all skb∈ds.SKBs DO //concurrently
03.    convert rq to a Single-SR rq' in skb according to ds.BRs
04.    invoke the IOPE-based graph planner to search a path c satisfying rq' from skb
05.    IF c is not null THEN
06.        break; // a path is found and the algorithm terminates
07.    ENDIF
08. ENDFOR
09. RETURN c;
```

FIGURE 2. D3L-based local planning algorithm.

Algorithm 1(Fig. 2) shows details of D3L-based local planning. In each SKB, given *D3L-SR* firstly is converted into a *Single-SR* according to the illustration above (row 03), and then, based on the *Single-SR*, an IOPE-based graph planner is invoked to search a local path (row 04). Once a path is found, the algorithm would terminate and return the path (rows 05-07). Otherwise, the algorithm would terminate until all SKBs have been searched. Ultimately, *null* will be returned. Specially, for all SKBs, the searching process runs concurrently (row 02).

An example is shown here to describe execution procedure of Algorithm 1. Let *skbs* be a set of SKBs,

BRs be a set of bridge rules, *rq1* be a *D3L-SR*, then $dp1 =<< skbs, BRs >, rq1 >$ is a *D3L-SPlan* problem, where,

$skbs = \{skb1, skb2\}$, here, $skb1.TP = \{TakeTaxi\}$, $skb2.TP = \{ProposeTrain, BookTrain, TakeTrain\}$,

$BRs = \{skb2 : Person \overset{=}{\rightarrow} skb1:Person, skb1:Vehicle \overset{\sqsupseteq}{\rightarrow} skb2:Train, skb1: vehicleCrossCity \overset{\sqsupseteq}{\rightarrow} skb2:trainCrossCity, skb1:vehicleIsBooked \overset{\sqsupseteq}{\rightarrow} skb2:trainIsBooked, \cdots\}$,

$rq1.In = \{\langle skb1, Person, Tom\rangle, \langle skb1, City, JN\rangle, \langle skb1, City, TJ\rangle, \langle skb1, Date, d\rangle\}$,

$rq1.Init = \{\langle skb1, personAtCity(Tom, JN)\rangle\}$ where the component *pred* in each fact is omitted,

$rq1.Out = \{\langle skb1, Vehicle, v\rangle, \langle skb1, Address, s1\rangle\}$,

$rq1.Goal = \{\langle skb1, vehicleFrom(v, s1)\rangle, \langle skb1, vehicleCrossCity(v, JN, TJ)\rangle, \langle skb1, vehicleIsBooked(Tom, v, d)\rangle\}$.

According to the algorithm above, the searching operations carry out in *skb1* and *skb2* concurrently. When in *skb1*, *rq1* doesn′t be converted because semantics of all parameters and literals are from *skb1*. Thus, no solution for *rq1* is found in *skb1*. When in *skb2*, *rq1* would be converted into *rq1'*:

$rq1'.In=\{\langle skb2, Person, Tom\rangle, \langle skb2, City, JN\rangle, \langle skb2, City, TJ\rangle, \langle skb2, Date, d\rangle\}$,

$rq1'.Init=\{\langle skb2, personAtCity(Tom, JN)\rangle\}$,

$rq1'.Out=\{\langle skb2, Train, v\rangle, \langle skb2, Station, s1\rangle\}$,

$rq1'.Goal=\{\langle skb2, trainFrom(v, s1)\rangle, \langle skb2, trainCrossCity(v, JN, TJ)\rangle, \langle skb2, trainIsBooked(Tom, v, d)\rangle\}$.

Finally, a path $\langle\{ProposeTrain\}, \{BookTrain\}\rangle$ is returned from *skb2*, and the algorithm terminates.

The advantage of *LocalD3LPlanning* is that it can concurrently search a local path from multiple SKBs. And compared with the searching at once from all services in all SKBs, the planning time can greatly decrease because there are fewer services in each SKB.

C. D3L-BASED GLOBAL PLANNING

LocalD3LPlanning only finds a local path. However, in some cases, to achieve a *D3L-SR*, services from different SKBs are needed to cooperate with each other. That is, a global path is needed for a *D3L-SPlan* problem. Unfortunately, classical service composition algorithm [10] only can be used when a request is a *Single-SR* and all services are annotated by one domain ontology. To obtain a global path, we improve the classical algorithm to support the planning in a DSKB, and the improved algorithm is called *D3L-based global planning algorithm* (*GlobalD3LPlanning*). Fig. 3 shows the detailed process of this algorithm.

GlobalD3LPlaning firstly constructs a planning graph according to a DSKB, and then searches a global path for a *D3L-SR* based on the graph. The graph would be extended if no path is found, and the searching operation continues until it levels off or its number of layers already reaches a specified max value. Just like the IOPE-based graph planning algorithm [8], the graph in *GlobalD3LPlanning* also includes two types of layers: state and action, and they are alternated. A state layer is composed of facts, and they may

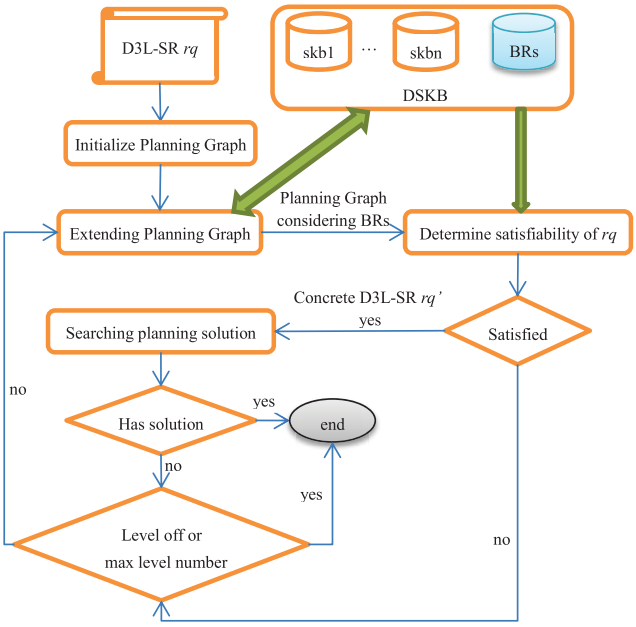

FIGURE 3. The detailed process of GlobalD3LPlaning algorithm.

be a known parameter (*parameter fact*), such as an address, a train, etc. or a predicate literal (*predicate fact*), such as *personAtAddress(Tom, sdjzu)*, *addressAtCity(sdjzu, JiNan)*, etc. These facts are generated by actions in the previous action layer. The actions include two types: persistence action [10] and service instance. A persistence action corresponds to a fact in the previous state layer, and its precondition and effect both are the fact. A service instance is an invocation for a service where its input parameters are instantiated. When the invocation really occurs, new facts would be produced in the next state layer. Different with the classical planning graph, the facts and actions may be from different SKBs. Specially, each service instance must be satisfied by facts in the previous state layer according to BRs (Definition 7).

Definition 7 (State Layer Satisfying Service Instance): Let *S* be a state layer, *a* be a service instance, *BRs* be a bridge rule set, we say *S* satisfies *a* when and only when *a.I* and *a.P* are included by *S* directly or through *BRs*:

$$\begin{aligned} S \overset{satisfy}{\rightarrow} a \Leftrightarrow\ & \forall c \exists l (c \in (a.I \sqcup a.P) \wedge l \in S) \\ & \wedge \{[(skb_l = skb_c) \rightarrow (l = c)] \\ & \vee \left[(skb_l \neq skb_c) \rightarrow (l \overset{\subseteq/=/=_F}{\rightarrow} c)\right]\} \end{aligned} \quad (3)$$

Fig. 4 shows a planning graph in *GlobalD3LPlanning*, where service instances in A_0 can be satisfied by facts in S_0 according Definition 7. It is noticed that instances in skb_1 are satisfied by facts in skb_1, and instances in skb_2 are satisfied by skb_2 and skb_1. Here, bridge rules between skb_1 and skb_2 are used when inputs and preconditions of instances in skb_2 are matched with facts in skb_1. Similarly, rules between skb_2 and skb_3 are used for matching facts in

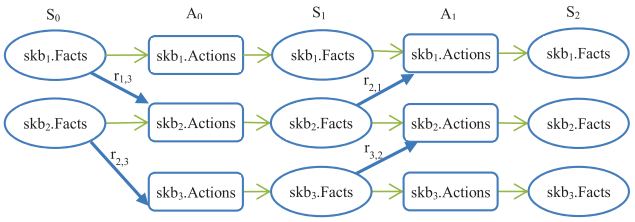

FIGURE 4. A planning graph in *GlobalD3LSearch*. Note. $r_{i,j}$ represents bridge rules between skb_i and skb_j.

skb_2 with inputs and preconditions of instances in skb_3. And, for $i \geq 1$, S_i is generated by service instances and persistence actions in A_{i-1}.

In *GlobalD3LPlanning*, the planning graph would be extended layer by layer, and after it is extended every time, two new layers are added: one is an action layer; another is a state layer. In each extending, the key operation in the following is to determine whether or not the original *D3L-SR* is satisfied by the last state layer (Definition 8).

Definition 8 (State Layer Satisfying D3L-SR): Let S be a state layer, rq be a *D3L-SR*, *BRs* be a bridge rule set, and we say S satisfies rq when and only when there is a subset S' of S and a mapping $f : rq.Out \cup rq.Goal \rightarrow S'$, satisfying the four situations in the following:

- If $x \in rq.out$ and $skb_x = skb_{f(x)}$, then $f(x) \in S'.ParaFacts \wedge x.semType = f(x).semType$;
- If $x \in rq.out$ and $skb_x \neq skb_{f(x)}$, then $f(x) \in S'.ParaFacts \wedge x.semType \overset{\supseteq/=}{\rightarrow} f(x).semType$;
- If $x \in rq.Goal$ and $skb_x = skb_{f(x)}$, then $f(x) \in S'.PEFacts \wedge x_{grounded} = f(x)$;
- If $x \in rq.Goal$ and $skb_x \neq skb_{f(x)}$, then $f(x) \in S'.PEFacts \wedge x_{grounded} \overset{\supseteq/=/=_F}{\rightarrow} f(x)$.

Here, skb_x represents the skb where element x is; $x_{grounded}$ is a literal that are grounded by the mapping values of $rq.Out$. We use $S \overset{satisfy}{\rightarrow} rq$ simply to denote rq is satisfied by S.

It is noticed that, if $S \overset{satisfy}{\rightarrow} rq$, we can obtain a concrete *D3L-SR* rq' for rq: $rq' = \langle rq.In, \{f(x) | x \in rq.Out\}, rq.Init, \{f(x) | x \in rq.Goal\}\rangle$. Specially, there may be multiple subsets of S that conform to Definition 8. Thus concrete *D3L-SR* for rq may have more than one, and Algorithm 2(Fig. 5) shows the generation process of concrete *D3L-SR* for rq.

Firstly, the algorithm computes all assignments for $rq.Out$ (rows 02-05). Then it uses these assignments respectively to instantiate $rq.Goal$, and multiple goal instances are produced (rows 06). At last, it checks each goal instance whether or not satisfied by S, and a concrete *D3L-SR* request is generated when the goal instance is satisfied (rows 07-10).

Specially, finding semantically matching parameter facts from S is a key operation in Algorithm 2. Here, for a parameter fact p in S, and an output parameter g in $rq.Out$, we say p matching g when one of two conditions in the following are satisfied:

- $p.semType \subseteq g.semType$, when $skb_p = skb_g$;
- $p.semType \overset{\subseteq}{\rightarrow} g.semType$, when $skb_p \neq skb_g$.

Once concrete *D3L-SR* requests are produced, the searching for each request from current planning graph would be carried out. *GlobalD3LPlanning* would be ended as long as one solution is founded. Otherwise the graph would continue to be extended when it doesn't level off and not reach specified max layer number. *GlobalD3LPlanning* is shown in Algorithm 3(Fig. 6).

Here, we illustrate *GlobalD3LPlanning* through an example. Given a *D3L-SPlan* problem $dp2 = \langle \langle SKBs, BRs\rangle, rq2\rangle$, where, a DKBS $\langle SKBs, BRs\rangle$ is the same with *dp1* in previous section, and *rq2* is described in the following:

rq2.In = {⟨ *skb1, Person, Tom*⟩, ⟨ *skb1, City, JN*⟩, ⟨ *skb1, City, TJ*⟩, ⟨ *skb1, Date, d*⟩, ⟨ *skb1, Address, sdjzu*⟩,⟨ *skb1, Address, tju*⟩},

rq2.Init = {⟨ *skb1,personAtAddress(Tom, sdjzu)*⟩, ⟨ *skb1, addressAtCity(sdjzu, JN)* ⟩, ⟨*skb1, addressAtCity(tju, TJ)*⟩},

rq2.Out = {⟨*skb1, Vehicle, v*⟩},

rq2.Goal = {⟨*skb1, vehicleCrossCity(v, JN,)*⟩, ⟨*skb1, personAtAddress(Tom, tju)*⟩}.

According to Algorithm 3, state and action layers in a planning graph for *dp2* are shown in the following:

S_0 = {⟨*skb1, Person, Tom*⟩, ⟨*skb1, City, JN*⟩, ⟨*skb1, City, TJ*⟩, ⟨*skb1, Date, d*⟩, ⟨*skb1, Address, sdjzu*⟩, ⟨*skb1, Address, tju*⟩},

A_0 = {*ProposeTrain(JN, TJ, d)*, ⋯ },

$S_1 = S_0 \cup$ {⟨*skb2, Train, t*⟩, ⟨*skb2, Station, s1*⟩, ⟨*skb2, Station, s2*⟩, ⟨*skb2, trainFrom(t, s1)*⟩, ⟨*skb2, trainTo(t, s2)*⟩, ⟨*skb2, trainCrossCity(t, JN, TJ)*⟩, ⟨*skb2, stationAtCity(s1, JN)*⟩, ⟨*skb2, stationAtCity(s2, TJ)*⟩, ⟨*skb2, trainIsValid(t, d)*⟩, ⋯ },

$A_1 = A_0 cup$ {*BookTrain(Tom, t, d), TakeTaxi(Tom, sdjzu, s1, JN)*, ⋯ }∪*persistenAction*(S_1),

$S_2 = S_1 \cup$ {⟨*skb2, trainIsBooked(Tom, t, d)*⟩, ⟨*skb1, personAtAddress(Tom, s1)*⟩, ⋯ },

$A_2 = A_1 \cup$ {*TakeTrain(Tom, t, d, JN, TJ, s1, s2)*,⋯ }∪ *persistenAction*(S_2),

$S_3 = S_2 \cup$ {⟨*skb2,personAtStation(Tom, s2)*⟩,⋯ },

$A_3 = A_2 \cup$ {*TakeTaxi(Tom, s2, tju, TJ)*,⋯ },

$S_4 = S_3 \cup$ {⟨*skb1,personAtAddress(Tom, tju)*⟩}.

At state layer S_4, we can find an parameter fact ⟨*skb2, Train, t*⟩ that can instantiate the parameter ⟨*skb1, Vehicle, v*⟩ in *rq2.Out*, and a concrete request that can be satisfied by S_4 is obtained: *rq2'* = ⟨*rq2.In,rq2.Init*, {⟨*skb2, Train, t*⟩}, {⟨*skb2, trainCrossCity(t, JN, TJ)*⟩, ⟨*skb1, personAtAddress(Tom, tju)*⟩}⟩. The rules are used from S_i to A_i are shown in the following:

From S_0 to A_0: *skb1: City* $\overset{=}{\rightarrow}$ *skb2: City*, *skb1: Date* $\overset{=}{\rightarrow}$ *skb2: Date*;

From S_1 to A_1: *skb2: Station* $\overset{\subseteq}{\rightarrow}$ *skb1:Address*;

From S_2 to A_2: *skb1: personAtAddress* $\overset{=_{para2:Station}}{\rightarrow}$ *skb2: personAtStation*, where the condition *para2:Station* represents the second variable value in the *personAtAddress* is an instance of *Station* in *skb2*;

From S_3 to A_3: *skb2: personAtStation* $\overset{\subseteq}{\rightarrow}$ *skb1: personAtAddress*.

Algorithm 2 Generate Concrete D3L-SR Request (GenConcreteReq)

Inputs: *rq:* a *D3L-SR* request, *S:* a state layer, *BRs: bridge rule set*

Outputs: *crqset*: concrete *D3L-SR Requests* for *rq* satisfied by *S*

01. *crqset*← Ø, assign←{Ø} //initialize *crqset* and output parameter assignment set
02. **FOR** each out parameter *g* in *rq.Out* **DO**
03. $fact_g$←getParaFactsMatched(*g,S,BRs*); //get parameter facts in *S* that are semantically matched with *g*
04. *assign*←*assign*$\times$ $fact_g$
05. **ENDFOR**
06. according different assignments in *assign* ground *rq.Goal*, and generate a goal instance set *goalinsts*
07. **FOR** each *gi* in *goalinsts* **DO**
08. Find replacement in *S* for each literal in *gi according BRs*, and produce the replacement set *gi'*.
09. **IF** *gi'*⊆*S* **THEN** create a concrete request *rq'*←<*rq.In,assignment for gi, rq.Init, gi'*>, and add *rq'* into *crqset*
10. **ENDFOR**
11. **RETURN** *crqset*

FIGURE 5. **Generate concrete *D3L-SR* requests algorithm.**

Algorithm 3 D3L-based global planning (GlobalD3LPlanning)

Inputs: *ds*: a DSKB, *rq*: a D3L-SR, *MaxLevel*：max layer number of in a planning graph

Outputs: a global service-based path for *rq*

01. *PG*= Ø; //initialize a empty planning grap, and state and action layers will be stored in a list
02. *StateLayer*←*new StateLayer(0, rq.Init);* // *0* is the order number of layer
03. adding *StateLayer in PG*
04. initialize the order number of the layer *level=0*
05. **WHILE** *level>=MaxLevel-1* DO
06. S_{level} ← the last state layer in *PG*
07. *crqset*← *GenConcreteReq (rq,* S_{level}*, ds.BRs)*//Algorithm 2
08. **IF** *crqset* is not null, that is, at least one concrete D3L-SR is produced **THEN**
09. **BREAK**; //exit and then searching planning solution in *PG*
10. **ELSE** A_{level}←the last action layers in *PG*
11. **ENDIF**
12. **FOR** ∀*skb*∈*ds.SKBs* **DO**
13. *S'*← facts in *skb* that are converted from S_{level} according to *into/equal* in *ds.BRs*
14. *ActInsSet*← service instances in *skb* that satisfied by *S'*
15. A_{level}←A_{level} ∪*ActInsSet*
16. **ENDFOR**
17. A_{level}←A_{level} ∪*noop*(S_{level}) and add A_{leve+1} in *PG* // *noop*(S_{level}) are persistence actions corresponding to facts in S_{level}
18. $S_{level+1}$←S_{level} ∪ $\{a.0 \cup a.E | a \in A_{level}\}$ and add S_{leve+1} in *PG*
19. **IF** *PG level off* **THEN BREAK** //$S_{level+1}$ *equal* S_{level}
20. *level*←*level+1*;
21. **ENDWHILE**
22. **FOR any** *g*∈*crqset* **DO**
23. search a path *c* for *g* from *PG* using backward searching
24. **IF** *c* exist **THEN BREAK**
25. **ENDFOR**
26. **RETURN** *c*

FIGURE 6. **D3L-based global planning algorithm.**

At last, an execution path with 4 execution steps is generated by a backward searching algorithm [8]: ⟨{*ProposeTrain(JN, TJ, d)*}, {*TakeTaxi(Tom, sdjzu, s1, JN)*, *BookTrain(Tom, t, d)*}, {*TakeTrain(Tom, t, d, JN, TJ, s1, s2)*}, {*TakeTaxi(Tom, s2, tju)*}⟩.

D. REPLANNING FOR RUNTIME SELF-ADAPTATION

In our previous work [16], we identified three local self-exception exceptions that may occur when a service is invoked: *UnPre*, *UnExe*, *UnEff*. *UnPre* occurs before invocation, and represents preconditions of this service can't

be satisfied by current runtime context. *UnExe* occurs on invocation, and means the service fails to run. *UnEff* occurs after invocation, and indicates this service doesn′ t obtain the desired output parameters or effects. These exceptions have different adaptation requests, and need different adaptation strategies. The adaptation request for *UnPre* is to achieve those unsatisfied preconditions according to current running state; the request of *UnExe* is to achieve desired effects and output parameters of this service according to its pre-conditions; and the request of *UnEff* is to also achieve the desired effects and output parameters according to current runtime status. All requests can be handled by means of an IOPE-based planner. We wrapped checking and handling operations for the three exceptions and a service invocation into a self-adaptation service activity (SSA).

Except for the three exceptions, a global self-adaptation exception *LocalAdaptFail* also was identified, and it can catch the running failures of SSA, and would be further handled by a global adaptation strategy at path level. The strategy could sequentially execute cancel and adaptation operations step by step, until the goal was achieved or no effects could be canceled.

In D3LSRAF, we also monitor and handle the four exceptions above: *UnPre*, *UnExe*, *UnEff* and *LocalAdaptFail*, and improve previous adaptation mechanism through introducing D3L-based planning algorithms above. The improvement can guarantee that a *D3L-SR* request is received by D3LSRAF, and that high response efficiency and response success rate are obtained. Local replanning strategy at SSA level and global replanning strategy at path level play an important role for the improvement.

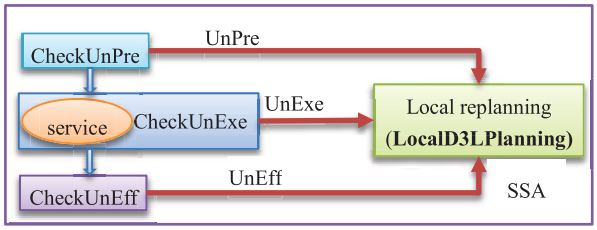

FIGURE 7. Concept view of local self-adaptation using local replanning at SSA level.

1) Local Replanning at SSA Level

In D3LSRAF, SSA in our previous work also is used as a self-adaptive invocation unit for a service, and it can automatically monitor and handle *UnPre*, *UnExe*, and *UnEff*. These exceptions can be monitored by special procedures that are set before, on and after service invocation. When an exception occurs, a local replanning strategy is adopted to repair the exception. The strategy utilizes *LocalD3LPlanning* algorithm to rapidly generate an adaptation path and continues to self-adaptively execute the path. A concept view about the local strategy is shown in Fig. 7.

Fig. 8 shows the local self-adaptation exception checking and handling logic in SSA. When a SSA is invoked,

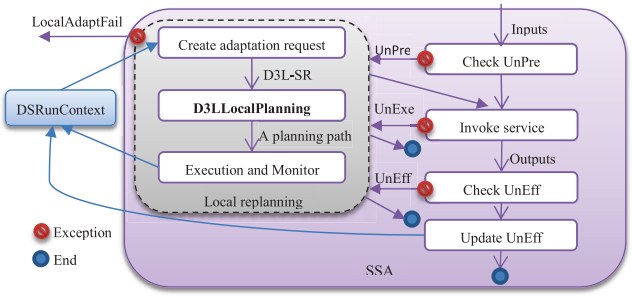

FIGURE 8. Local self-adaptation exception checking and handling process at SSA level.

there are four main operations (*Check UnPre*, *Invoke service*, *Check UnEff*, *Update UnEff*) are executed orderly. And three local exception monitoring points are respectively set in *Check UnPre*, *Invoke service*, and *Check UnEff* operations. If no exception occurs, the SSA can update effects of corresponding service instance into current *DSRunContext*. Otherwise, when an exception occurs, the Local replanning unit can immediately catch and deal with it. In the unit, firstly, an adaptation request is created according to current *DSRunContext* and exception; secondly, *D3LLocalPlanning* algorithm is invoked to generate a local path for the request; at last, the path is executed according the inputs and initial states in the request, and meanwhile, the exceptions during its execution also is monitored and handled self-adaptively. During the replanning, if no path is found in *D3LLocalPlanning*, or it fails to run, a global exception *LocalAdaptFail* is thrown, that is, the replanning fails to run. Otherwise, the replanning succeeds; and corresponding SSA will terminate when handled exception is *UnExe* or *UnEff*, or the *Invoke service* operation in the SSA continues to run when handled exception is *UnPre*.

TABLE 2. Adaptation requests and available services for local self-adaptation on faulted service instance *si*.

Exc	Adaptation request				Available services
	In	Init	Out	Goal	
UnPre	parameter facts in $\varepsilon_{sr(t)}.Fs$	predicate facts in $\varepsilon_{sr(t)}.Fs$	*si.I*	*si.P*	All
UnExe	*si.I*	*si.P*	*si.O*	*si.E*	Except *s*
UnEff	parameter facts in $\varepsilon_{sr(t)}.Fs$	predicate facts in $\varepsilon_{sr(t)}.Fs$	*si.O*	*si.E*	Except *s*

Specially, all adaptation requests during local replanning are generated according to current *DSRunContext*, and all are *D3L-SR* requests. Given a service request *sr*, a service instance *si* (corresponding to service *s*) that is running for achieving *sr* at moment *t*, then when one local exception occurs, its adaptation request and available services is created according to Table 2. These are the inputs of the following *D3LLocalPlanning* algorithm. Meanwhile, during the response process of these adaptation requests, the *DSRunContext* of *sr* ε_{sr} also are updated.

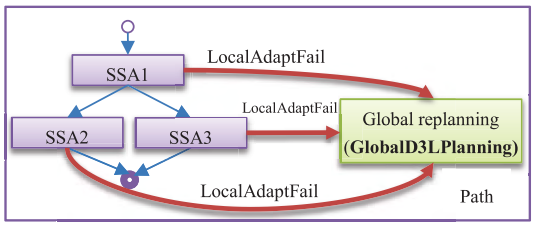

FIGURE 9. Concept view of global self-adaptation using global replanning at path level.

2) Global Replanning at Path Level

When local exceptions can't be handled at SSA, corresponding SSA can throw *LocalAdaptFail*. And a global replanning strategy using *GlobalD3LPlanning* would be adopted, as shown in Fig. 9. With the help of *GlobalD3LPlanning* algorithm, the algorithm tries to search all possible adaptation paths for the original request among all SKBs. If the global replanning strategy fails, then the original request can't be achieved.

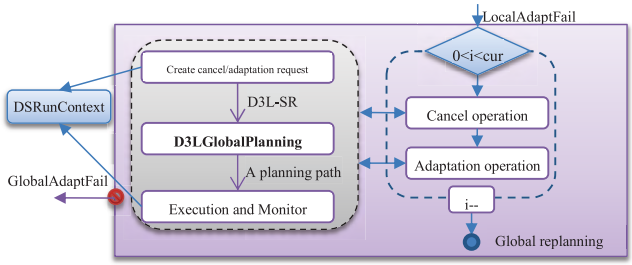

FIGURE 10. The implementation of global replanning strategy at path level. Note. *cur* is the execution step sequence number including faulted service.

Fig. 10 shows the implementation logic of the global replanning strategy. The global replanning includes two main operations: cancel and adaptation. The two operations will be successively executed in each execution step from the execution step including faulted service to the first. In each execution step, the cancel operation firstly is invoked to cancel those revocable effects [16] in the execution step; then, the adaptation operation is used to search an adaptation path for the final goal of the main path. Once an adaptation path is found and executed successfully, the global replanning will terminate with success and the main path runs successfully. Otherwise, the two operations continue to successively run in the next execution step. Specially, if no path is found in the first execution step, or any cancel operation fails to run, then the global replanning strategy fails to run and the main path terminates with the failure *GlobalAdaptFail*.

During the global replanning, some redundant effects may be produced. And they may affect the exploring of new possible paths or may produce unnecessary cost. For example, the effect *trainIsBooked(Tom,D404,20170101)* is produced by *BookTrain* service. However, if the train is unavailable, a flight should be picked during the replanning. Thus, *trainIsBooked(Tom,D404,20170101)* is a redundant effect and should be eliminated before a flight is booked. Cancel operations in the global replanning can guarantee that those redundant effects are canceled no matter whether the global planning succeeds or not.

Furthermore, in a cancel or an adaptation operation, three main modules are invoked orderly: *Create cancel/adaptation request*, *D3LGlobalPlanning*, and *Execution and Monitor*. Firstly, a *D3L-SR* request is created according to current *DSRunContext*. Table 3 shows the requests for the two operations, where *sr* is the original request of the main path, $\varepsilon_{sr(t)}$ is the *DSRunContext* of *sr* at moment *t*, *si* is the faulted service instance and *s* is the service related to *si*, *cur* is the execution step including *si*. Secondly, the *D3LGlobalPlanning* algorithm is utilized to search a global path for the request. At last, the path is executed self-adaptively.

TABLE 3. Requests for cancel and adaptation operations in an execution step cur.

Operations	Adaptation request				Available services
	In	Init	Out	Goal	
Cancel	parameter facts in $\varepsilon_{sr(t)}.Fs$	predicate facts in $\varepsilon_{sr(t)}.Fs$	null	negation of facts in $\varepsilon_{sr(t)}.CFs$ that are produced by service instances in *cur*	Except faulted service
Adaptation	parameter facts in $\varepsilon_{sr(t)}.Fs$	predicate facts in $\varepsilon_{sr(t)}.Fs$	*sr.Out*	*sr.Goal*	Except faulted service

Specially, a new path in local and global replanning also is composed of SSAs, and also includes local and global replanning unit at path level. Therefore, when a self-adaptation exception occurs during the path runs, local and global replanning would work just like the handling process above. This mechanism can guarantee that all possible paths are attempted during the repair. Thus, high response success rate can be provided. Meanwhile, the use of local replanning can rapidly carry out the local self-adaption, and effectively decrease the adaptation time.

V. PROTOTYPE IMPLEMENTATION OF D3LSRAF

According to the architecture of D3LSRAF in Fig. 1, a prototype system of D3LSRAF (D3LSRAFS) mainly implements four core modules: *DSKB management*, *D3LPlanners*, *Execution and Monitor*, *Self-adaptation Management*, shown as Fig. 11. Here, *DSKB management* module is used to organize semantic and syntactic information of available services into various SKBs, and to create various bridge rules among them; *D3LPlanners* module includes a local planner for *LocalD3LPlanning* algorithm (Algorithm 1) and a global planner for *GlobalD3LPlanning* algorithm (Algorithm 3), and they can be invoked to respectively generate local and global paths in a DSKB; *Execution and Monitor* module can provide necessary running environment for these paths, and automatically catch various self-adaptation exceptions

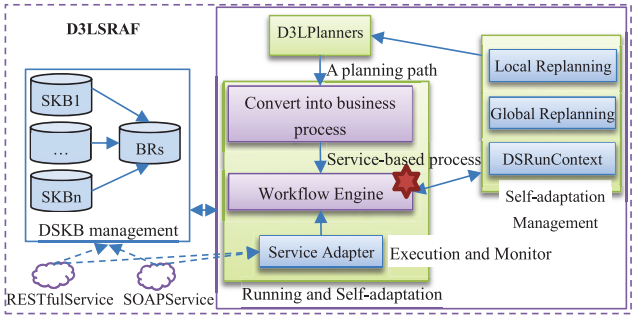

FIGURE 11. The architecture of D3LSRAFS.

and invoke corresponding self-adaptation strategies; Self-adaptation Management module would implement operations related to self-adaptation, including local and global replanning strategies and *DSRunContext* management.

Obviously, the *D3LPlanners* module can be implemented easily through packaging the two D3L-based algorithms into two planners (*LocalD3LPlanner* and *GlobalD3LPlanner*). Therefore, the following subsections will present the implementation details of those modules except D3LPlanners.

A. DSKB MANAGEMENT

DSKB management module provides basic information for the running of other modules. Services are the core elements in a DSKB. In practice, there are two main implementation ways of services: SOAP-based and Restful. They have different interface description formats and semantic annotation formats.

SOAP-based service is a traditional function-oriented web service, and use WSDL (Web Service Definition Language) specification to describe functional operation interface and invocation details. All input/output parameters in operations conform to XML format [30], and they are encapsulated into SOAP (Simple Object Access Protocol) messages when interacting with outside. RESTful services are resource-oriented, and mainly describe resources, access method, and access URI. The interaction with RESTful services is generally through HTTP Protocol.

RESTful services have two types, one is used to represent resources and doesn't provide any functions, and another can provide operations like SOAP-based services through links between resources and is also called Web API. The RESTful services in this paper just are the latter. An API in RESTful service includes URI or parameterized URI template, access HTTP method (*GET, POST, PUT, DELETE*), and input/output parameters. Generally, these APIs are described by common text document. This text has no machine-understandable semantics. To find and compose these APIs easily, some machine-understandable description formats emerge, such as WADL [31], WSDL2.0 [32], and more lightweight hRESTS [33] etc.

From the view of semantics, existing OWL-S [34], WSMO [35], SAWSDL [36] etc. can be used to annotate SOAP-based services. Meanwhile, the semantic annotation for RESTful services also can be achieved by extending annotation methods for SOAP-based services [37], [38]. In theory, it is feasible to compose mixed type services together if we use unified ontology annotation framework for two types of services [38].

Therefore, it is easy to automatically or manually extract IOPE of each service from their semantic and syntactic description documents. According features of various services in real world, in the *DSKB management* module, we abstract 6 basic entities to create a DSKB: *Param*, *Predicate*, *Service*, *SKB*, *BridgeRule* and *DSKB*. The first three are used to collect semantic and syntactic details about services with uniform formats; *SKB* and *DSKB* are consistent with Definition 1 and 3; *BridgeRule* records bridge rules among SKBs. All of them provide necessary information for replanning operations in the following. Furthermore, the module also provides the basic operations that can modify these entities. Fig. 12 describes the relationship among these entities.

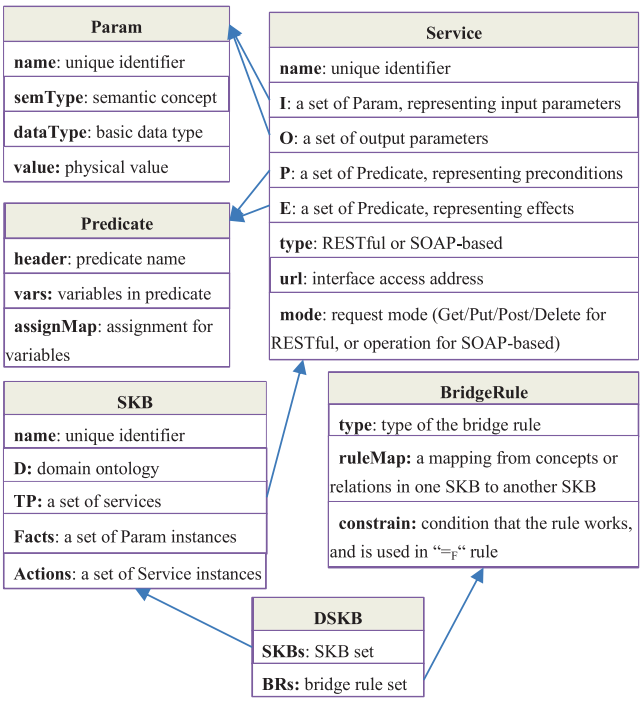

FIGURE 12. Relationship among basic entities related to DSKB.

D3LSRAFS adopts multiple *SKB* databases and a *BridgeRule* database to store information in a *DSKB*. Each *SKB* database includes all services from one provider, and the *BridgeRule* database includes all bridge rules between any two *SKBs*. After a *SKB* is created, by means of ontology alignment technologies, new bridge rules (Definition 2) would be generated through comparing semantics in the new *SKB* and old *SKBs*.

B. EXECUTION AND MONITOR

Execution and Monitor module mainly includes three submodules: *Convert into business process*, *Workflow Engine*, and *Service Adapter*. The *Convert into business process* can

convert a service-based path into a standard executable business process; the *Workflow Engine* is utilized to deploy, run, and monitor the process; the *Service Adapter* is adopted to physically invoke different types of services in a process. It can be seen that the key for implementation of *Execution and Monitor* module is to pick a suitable business process definition language and a workflow engine.

Generally, WS-BPEL2.0 is used to describe SOAP-based service processes. It conforms to OASIS standard and is adopted widely in industrial world [39]. WS-BPEL2.0 supports common control flow structures, such as sequence (element ⟨*sequence*⟩), parallel (element ⟨*flow*⟩) etc. And definitions of data flow and invocation for web services respectively are expressed by element ⟨*assign*⟩ and ⟨*invoke*⟩. Meanwhile, WS-BPEL2.0 has mature exception handling mechanism, and a user can set flexible exception handling strategy in corresponding failure unit. A WS-BEPL2.0 business process can run over an existing business process engine, such as Apache ODE [40]. Unfortunately, it can't support RESTful services to date.

Some researches achieved RESTful service process definition through extending existing BPEL language. Reference [13] added some activities related to invoke RESTful services in WS-BPEL2.0: ⟨*GET*⟩, ⟨*PUT*⟩, ⟨*DELETE*⟩, ⟨*POST*⟩. This makes it possible to execute a process with two types of services. However, few mature business process engines can support such extending in WS-BPEL2.0.

BPMN2.0 [41] also is a popular business specification, and it can describe a business process including various tasks, such as *user task*, *script task* etc. And it provides well extending mechanism to add more types of tasks. Generally, an existing business engine for BPMN2.0 can support the extending about its definition to support specific tasks. For example, the Activiti [42] adds a *serviceTask* in BPMN2.0 to custom own business logic. Therefore, through extending two *serviceTasks* respectively to invoke RESTful and SOAP-based services, it is easy to support the service-based process definition with RESTful and SOAP-based services. Moreover, BPMN2.0 also provides exception mechanism that can be used to monitor the exception at runtime.

In D3LSRAFS, we pick BPMN2.0 as the service-based process definition language, and the Activiti as execution engine of BPMN2.0. Meanwhile, in order to support D3LSRAF, we design special business process modeling notation (BPMN) process specification for a service-based path, and the specification mainly includes three aspects in the following.

1) Services in the same execution step are invoked between two *parallel* gateways.
2) Each service invocation includes three process activities: *assemble inputs*, *invoke service*, *update outputs*. Here, the *invoke service* activity is an extended *serviceTask* in BPMN2.0, and it implements a SSA (Fig. 8). That is, it not only includes invocation details about a physical service, but also encapsulates local replanning logic. Other two activities both are *scriptTask*. The *assemble inputs* activity can assemble input parameters of the service according to known process variables, and the *update outputs* activity will put output parameters of the service in a process variable. Specially, for those services without output parameters, *update outputs* activity can be omitted.
3) In a process, two global error sub processes are included respectively to handle *LocalAdaptFail* and *UnKnowFail* at runtime. Here, *UnKnowFail* is used to represent those exceptions except *LocalAdaptFail* at process level. In the sub process for *LocalAdaptFail*, an extended *serviceTask* is invoked to execute global replanning, and is called *GlobalAdaptService*. The *AssignFail* activity in the *UnKnowFail* sub process is a *scripTask*, and is used to return an execution error to the caller.

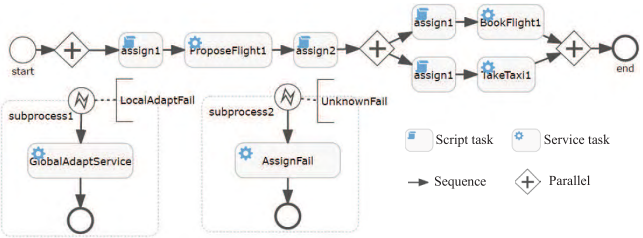

FIGURE 13. BPMN diagram for a service-based process.

Fig. 13 shows a BPMN process diagram for a service-based path ⟨{*ProposeFlight*1}, {*BookFlight*1, *TakeTaxi*1}⟩. Here, *assign1* and *assign2* respectively are *assemble inputs* and *update outputs* activities; *ProposeFlight1*, *BookFlight1*, *TakeTaxi1* are *invoke service* activities; *BookFlight1* and *TakeTaxi1* services have no output parameters.

Considering different types of services may exist in a process, we develop two adapters that respectively are used to invoke SOAP-based and RESTful services: *RESTfulServiceAdapter* and *SOAPServiceAdapter*, and use extended *serviceTask* to implement them. When a SOAP-based service is invoked, its *invoke service* activity is replaced with *SOAPServiceAdapter*, while for a RESTful service, the activity is replaced with *RESTfulServiceAdapter*.

Fig. 14 shows a RESTful service invocation fragment. The first *scriptTask* activity (rows 01-05) is used to assemble input parameters for *ProposeFlight1*, and the second (rows 16-19) is used to set output parameters of this service into corresponding process variables. Rows 06-15 show an extended *serviceTask* that invokes *ProposeFlight1* service with *RESTfulServiceAdapter*.

Fig. 15 shows a SOAP-based service invocation fragment. For service *BookFlight1*, the adapter *SOAPServiceAdapter* is adopted in corresponding *serviceTask* activity (rows 02-03), and the value of property *mode* is replaced by concrete operation name in this service, such as *Book* (row 05). Other properties are the same with *RESTfulServiceAdapter* (rows 06-09).

```
01 <scriptTask id="ProposeFlight1Assign1" scriptFormat="groovy"
02                                          activiti:autoStoreVariables="false">
03     inputs.put("d","20170808"); inputs.put("c1","jn"); inputs.put("c2","tj");
04     execution.setVariable("ProposeFlight1inputs",inputs);
05</scriptTask>
06<serviceTask id="ProposeFlight1" name="ProposeFlight1"
07                              activiti:class="engine.RESTfulServiceAdapter">
08   <extensionElements>
09       <activiti:field name="mode">POST</activiti:field>
10       <activiti:field name="url">
11           http://localhost:8080/jsjxyxt/wskb3/ProposeFlight.do
12       </activiti:field>
13       <activiti:field name="inputs">${ProposeFlight1inputs}</activiti:field>
14   </extensionElements>
15 </serviceTask>
16 <scriptTask id="ProposeFlight1Assign2" scriptFormat="groovy"
17                                          activiti:autoStoreVariables="false">
18   execution.setVariable("ProposeFlight1outputs",result);
19 </scriptTask>
```

FIGURE 14. A RESTful service invocation fragment.

```
01 <scriptTask …></scriptTask>
02<serviceTask id="BookFlight1" name=" BookFlight1"
03                activiti:class="engine.SOAPServiceAdapter">
04   <extensionElements>
05       <activiti:field name="mode">Book</activiti:field>
06       <activiti:field name="url">
07           http://localhost:8080/ services/BookFlight
08       </activiti:field>
09       <activiti:field name="inputs">${ BookFlight1inputs} </activiti:field>
10   </extensionElements>
11 </serviceTask>
```

FIGURE 15. A SOAP-based service invocation fragment.

Specially, we package all local replanning logics in a SSA into each adapter. That is, adapters can automatically monitor and handle local self-adaptation exceptions on corresponding service according to current runtime context. When a local replanning runs in a SSA, other activities paralleled with the SSA in the process can't be affected. If the local replanning fails, then *LocalAdaptFail* is thrown from corresponding *serviceTask*. And it will be handled by the error sub process for *LocalAdaptFail*, meanwhile, other activities in the process will terminate to run. If the sub process fails to run, then *UnknowFail* is thrown, and an execution error is returned through *UnknowFail* error sub process.

In addition, traditional timeout exception and retry strategy are not considered in D3LSRAFS. And, they lie with the underlying engine. If they fail to run, the engine can throw corresponding failure, and D3LSRAFS regards the failure as an *UnExe*.

C. SELF-ADAPTATION MANAGEMENT

Self-adaptation Management module implements the local replanning logics in Fig. 8 and the global replanning logics in Fig. 10. Meanwhile, various management operations for *DSRunContext* also are implemented in this module, including checking the satisfaction of preconditions, updating execution effects and availability of services. All operations in this module would be invoked by the Activiti engine according to the running situation of a process.

Specially, in the replanning logics, the *D3LPlanners* and *Execution and Monitor* modules also are invoked frequently and new service-based business processes would be generated and run on the engine. These new processes also have various self-exception handling logics and also can self-adaptively run. Fig. 16 presents an illustration for the runtime self-adaptation mechanism.

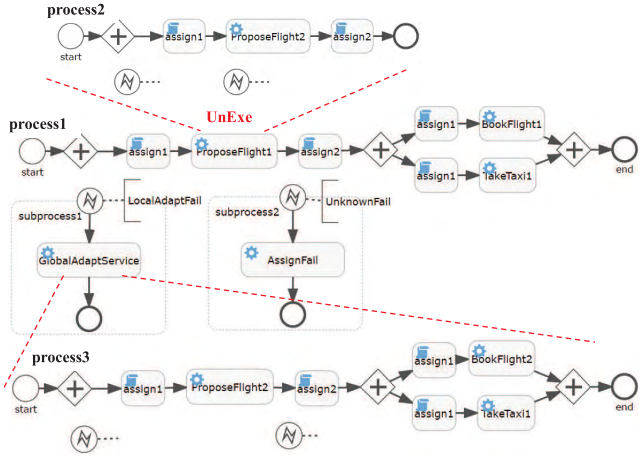

FIGURE 16. An illustration for the runtime self-adaptation mechanism in D3LSRAF.

When *ProposeFlight1* activity in *process1* encounter an exception *UnExe*, the adapter of *ProposeFlight1* firstly would find a new path by means of *LocalD3LPlanner*, and then convert it into a new BPMN process *process2* with various self-adaptation units, at last, self-adaptively run *process2*. If the *LocalD3LPlanner* doesn't find any path for the *UnExe* or *process2* fails to run, and then the adapter of *ProposeFlight1* would throw an exception *LocalAdaptFail*. Next, *GlobalAdaptService* activity in *subprocess1* would invoke *GlobalD3LPlanner* to find a path for the final goal of *process1*, and then self-adaptively run the BPMN process *process3* corresponding to the path. If *process3* succeeds to run, then *process1* also succeeds to run. Otherwise, *process1* also fails to run. During the running of a business process, all self-adaptation operations are automatically completed in the Activiti engine. Therefore, if all exceptions are handled successfully at runtime, the user doesn't feel the occurrence of any exception.

VI. EXPERIMENT EVALUATIONS

A. CASE STUDY AND ENVIRONMENT

Currently, there are no standard test measures for service runtime self-adaptation framework. Here, we design three measures from practicability: *Effectiveness*, *Efficiency*, *Response success rate*.

- *Effectiveness*: it is the most basic measure for a service runtime self-adaptation framework. Given a service request, the framework should plan an initial path for the request and execute the path. If the initial path fails to run, corresponding exception should be caught immediately and be automatically handled at runtime by means of all available services.
- *Efficiency*: It represents the speed that a framework responses a service request. And it includes the time planning an initial path, the time running the path, and the time replanning for self-adaptation. Obviously, the fewer time would make the better interaction experience for a user.
- *Response success rate*: given some service requests, a good framework should succeed to run more requests even if various exceptions occur at runtime.

TABLE 4. Services in the test case.

Domain	Services
TransferInCity	TakeTaxi# ProposeTransitPath1 ProposeTransitPath2 TransferByPath#
Train	ProposeTrain BookTrain1 BookTrain2* TakeTrain# CancelBookTrain
Flight	ProposeFlight BookFlight1 BookFlight2* TakeFlight# CancelBookFlight
Hotel	FindNearHotel CheckInHotel# BookHotel1 BookHotel2* CancelBookHotel
Inn	FindNearInn BookInn1 BookInn2* CancelBookInn CheckInInn#

To evaluate our framework, we create 24 basic services from travel and accommodation domains according to real services on Internet, where 6 human service interfaces are created to make various services be linked into a whole service-based path. Table 4 shows all services in our experiments, and their names imply their functions. It includes *14* RESTful
services, *6* human services ("#" in upper-right corner), *4* SOAP-based services ("*" in upper-right corner). Based on the *24* services, we create *6* SKBs (*SKB1*, *SKB2*,$\cdots$, *SKB6*) and respectively use *6* domain ontologies: *D1*,$\cdots$, *D6* to annotate service semantics. Here, two DSKBs are designed, and the first DSKB *SB1* is composed of the first five SKBs and *102* bridge rules among *D1*,$\cdots$, *D5*, and *24* services in Table 4 respectively are put in corresponding SKB according to the domain that they belong; the second DSKB *SB2* is compose of *SKB6* where the *24* services all are. Specially, if a DSKB only includes one SKB, and then the two D3LPlanning algorithms (Algorithm 1 & 3) based on the DSKB are equal to common graph planning algorithm without D3L-based reasoning (*commonGPlan*). Therefore, in the following experiments, we will simulate the service self-adaptation framework without D3L-based reasoning (*SRAFSnoD3L*) through making all requests run under *SB2*.

We also design *5* service requests and create *5* self-adaptation exceptions respectively for the running of these requests. The detailed illustrations are shown in Table 5.

The running environment of D3LSRAFS includes an application server *Tomcat8.0*, a SOAP-based service engine *Axis2*, a workflow engine *Activiti 5.22*, and a database engine *MySQL5.1*. All software are installed on *DELL 7040MT* (3.41GHz×2, 8GRAM, Win10).

B. EFFECTIVENESS

To evaluate the effectiveness of D3LSRAFS, based on *SB1*, we run the five requests in Table 4 respectively under two situations. In the first situation *s1*, there is no exception at runtime; and in the second situation *s2*, corresponding exception (the last column in Table 4) occurs at runtime. In the experiment, we find that all requests succeed to run under the two situations. And all paths generated by planners in D3LSRAF are successfully converted into an executable BPMN processes with about *500ms*, and these BPMN processes are automatically deployed and executed on the Activiti engine. When an exception occurs at runtime, it is caught and handled successfully with the help of the workflow engine. More details are shown in Table 6.

From the table above, we can see that the local replanning strategy is firstly used when an exception occurs over a service. The strategy uses *LocalD3LPlannning* in local replanning to find an adaptation path for each exception in the first three requests, and then succeeds to run the path by means of the workflow engine. However, the local replanning strategy fails to run for exceptions in *sr4* and *sr5*, because no adaptation path is found in these strategies. Thus, *LocalAdaptFail* is thrown when *sr4* or *sr5* run with exception; and then *GlobalD3LPlannning* in global replanning strategy is used to find an adaptation path for for *sr4* or *sr5*.

Specially, during the global replanning for the exception in *sr4* or *sr5*, a cancel path is found and executed before an adaptation path, because some known effects need to be canceled before a new path is tried. For example, in *sr4*, the effect *trainIsBooked* for the person is canceled by the cancel path before a flight is picked in another path. This is reasonable in real world, because the ticket should be canceled when a train encounters a breakdown.

C. EFFICIENCY

To compare the efficiency of D3LSRAF with SRAFSnoD3L, we run the five requests respectively using *SB1* and *SB2*. For each request, we record the total response time (*TotalTime*), the time planning an initial path (*PlanTime*), and the time replanning for self-adaptation (*ReplanTime*), as shown in Table 7.

From the view of *TotalTime*, compared with SRAFSnoD3L, D3LSRAFS can more rapidly respond each service request no matter whether an exception occurs or not (Fig. 17). The advantage becomes greater when the desired path includes more services, especially when some exception occurs at runtime. It is noticed that the total response time of D3LSRAFS is about *5s* less than SRAFSnoD3L when the exception occurs in *sr4* or *sr5*.

TABLE 5. Request details including expected initial paths and predefined exceptions.

Req	Description	Initial path	Exception
sr1	Get transit path suggest in city *c1* from address *a1* to address *a2*	$\langle\{ProposeTransitPath1\}\rangle$	*UnExe* over *ProposeTransitPath1*
sr2	In city *c1*, a person wants to arrive address *a2* from address *a1*	$\langle\{TakeTaxi\}\rangle$	*UnEff* over *TakeTaxi*
sr3	A person at address *a1* in city *c1* wants to arrive city *c2*	⟨{ProposeTrain}, {BookTrain1, TakeTaxi}, {TakeTrain}⟩	*UnPre* over *TakeTrain*
sr4	A person at address *a1* in city *c1* wants to arrive address *a2* in city *c2*	⟨{ProposeTrain},{BookTrain1, TakeTaxi}, {TakeTrain}, {TakeTaxi}⟩	*UnExe* over *TakeTrain*
sr5	A person at address *a1* in city *c1* wants to arrive address *a2* in city *c2*, and he also wants to check in the accommodation near *a2*	⟨{ProposeTrain, FindNearInn}, {BookTrain1,BookInn1, TakeTaxi}, {TakeTrain},{TakeTaxi,CheckInInn}⟩	*UnExe* over *CheckInInn*

TABLE 6. Running details for various requests.

Req	Exceptions caught	Adaptation strategies	Adaptation/cancel process
sr1	UnExe	Local replanning	An adaptation path ⟨{ProposeTransitPath2}⟩ is found
sr2	UnEff	Local replanning	An adaptation path ⟨{ProposeTransitPath1}, {TransferByPath}⟩ is found
sr3	UnPre	Local replanning	An adaptation path ⟨{TakeTaxi}⟩ is found
sr4	UnExe& LocalAdaptFail	Local&Global replanning	a cancel path⟨{CancelBookTrain}⟩ and an adaptation path ⟨{ProposeFlight}, {BookFlight1, TakeTaxi}, {TakeFlight}, {TakeTaxi}⟩ are found
sr5	UnExe& LocalAdaptFail	Local&Global replanning	a cancel path ⟨{CancelBookInn}⟩ and an adaptation path ⟨{FindNearHotel}, {BookHotel1}, {CheckInHotel}⟩ are found

TABLE 7. Running time details for various requests(ms). Column *L* represents *Local*, and *G* represents *Global*.

	SB1					SB2				
	TotalTime			ReplanTime		TotalTime			Replan	
Req	Normal	Exc	Plan	L	G	Normal	Exc	Plan	L	G
sr1	609	1376	3	579	/	615	1439	7	617	/
sr2	573	1559	6	670	/	629	1623	11	786	/
sr3	1020	1807	92	555	/	1400	2134	300	582	/
sr4	1290	4127	264	30	1615	1562	9189	604	4033	2894
sr5	1891	4197	458	15	1149	2453	9155	1050	3994	2061

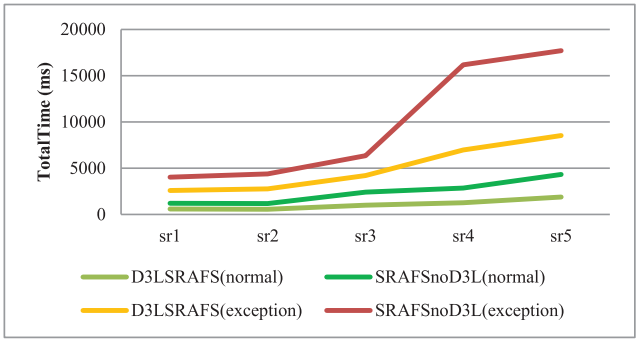

FIGURE 17. Comparison of TotalTime between D3LSRAFS and SRAFSnoD3L.

The response time for a service request mainly is composed of the time producing an initial service-based path, the time converting a path to a BPMN process and the time running the BPMN process. In our experiment, two frameworks spend almost the same time on the conversion and running for the same service request. Thus, the planning time becomes a key factor to affect the total time of two frameworks.

When a request runs normally, a planning algorithm is invoked only once, and it can generate a path for the request. *D3LGlobalPlanning* and *commonGPlan* respectively are used in D3LSRAF and SRAFSnoD3L. They both firstly create a planning graph and then search a path from the graph. However, the scale of the planning graph in *D3LGlobalPlanning* is smaller than in *commonGPlan*, because *D3LGlobalPlanning* searches available actions from each SKB according bridge rules but not searching from all services just like in *commonGPlan*. This can avoid redundant actions and facts to be added in the graph. For example, there are services related to train in a SKB *skb*, and *Station* is used to annotate an address for train stop. However, in *commonGPlan*, *Address* is used to annotate the train stop. Thus, the aim for train stop is ignored. Assume that service *ProposeStationTransit* in *skb* has two input parameters: *depart station* and *arrive station*. When *D3LGlobalPlanning* generates new actions from *skb*, the instances of *Station* in *skb* can be used to assign the two parameters. However, *commonGPlan* can use any two common instances of *Address* for the assignment, which may be not real stations. Thus, redundant actions are generated, and this result in redundant facts in the next state layer.

Fig. 18 and Fig. 19 shows the comparison for number of facts and actions between *D3LGlobalPlanning* and *commonGPlan*. It can be seen that, for each request, more facts and actions are generated in *commonGPlan* than *D3LGlobalPlanning*. Ultimately, less time is spent

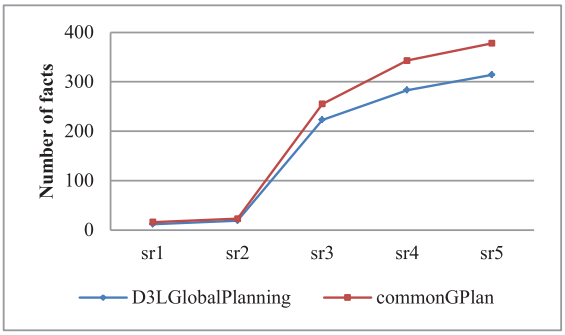

FIGURE 18. Comparison of facts in planning graphs between *D3LGlobalPlanning* and *commonGPlan*.

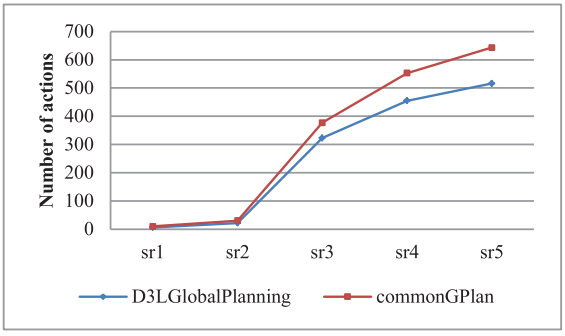

FIGURE 19. Comparison of actions in planning graphs between *D3LGlobalPlanning* and *commonGPlan*.

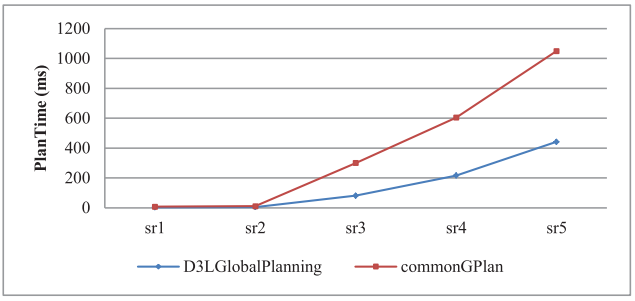

FIGURE 20. Comparison of *PlanTime* between *D3LGlobalPlanning* and *commonGPlan*.

in *D3LGlobalPlanning* than *commonGPlan* for a service request (Fig. 20).

Different with normal running situation, when a request runs abnormally, a planning algorithm is invoked more than once. The first is to generate an initial path, and others are used in local and global replanning. Obviously, more exceptions at runtime, more invocations are needed. According to the description above, D3LSRAF spends less time in the initial path generation than SRAFSnoD3L. For each exception at runtime, D3LSRAF would adopt *D3LLocalPlanning* for a local adaptation request. When no path is found, *D3LGlobalPlanning* would be adopted for a global adaptation request. *D3LLocalPlanning* can search a path concurrently from each SKB. Once a path is obtained, the searching terminates. Compared with *D3LGlobalPlanning*, the searching space of *D3LLocalPlanning* is smaller. Therefore, for an adaptation request, less plan time is spent in *D3LLocalPlanning* than *D3LGlobalPlanning*. However, SRAFSnoD3L always adopts *commonGPlan* to search a path for local and global adaptation requests. Therefore, in each local replanning operation, SRAFSnoD3L would always spend more time than D3LSRAFS. Fig. 21 shows the local replanning time for the requests in Table 5.

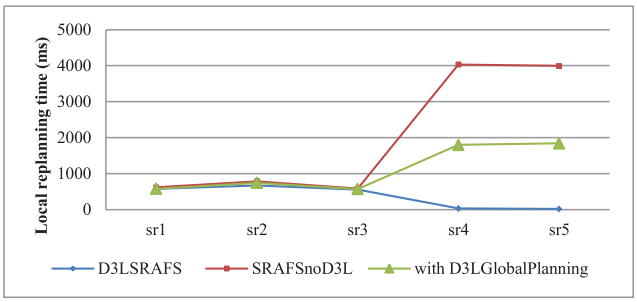

FIGURE 21. Comparison of Local replanning time between D3LSRAFS and SRAFSnoD3L.

Specially, in *sr4* and *sr5*, D3LSRAFS determines the running failure of a local replanning with tens of milliseconds, however, SRAFSnoD3L costs thousands of milliseconds. Even if we replace the *D3LLocalPlanning* with *D3LGlobalPlanning* in the local replanning strategy of our framework (green line in Fig. 21), the determination time is about *1000* milliseconds, and also far less than SRAFSnoD3L. Obviously, for a service request, the times to invoke the planner are more, the time difference between two frameworks is greater, just like in *sr4* and *sr5*. Therefore, through the runtime self-adaptation mechanism at two levels, D3LSRAF can substantially decreases the response time for various service requests no matter whether they succeed or not (Fig. 17).

D. RESPONSE SUCCESS RATE EVALUATION

Given a service request, its single run is called a request running instance. Assume that n is the number of request running instances, m is the number of those instances that are responded successfully, then the response success rate $r=m/n$. Obviously, the rate is higher, and the quality of the self-adaptation system is better. In D3LSRAF, bridge rules can improve the response success rate, because they can effectively eliminate the semantic conflicts among multiple SKBs to make their services cooperate with each other.

To prove this, we design *45* request running instances. And each instance will encounter at least on one local self-adaptation exception at runtime. Fig. 22 shows the number of predetermined exceptions in these instances, and the number of exceptions that are expected to be handled successfully under *SB1*. It is noticed that the ideal response success rate is *73.33%*.

In *SB1*, we design *5* sets of bridge rules, and respectively create *5* running situations *run1*,$\cdots$,*run5*. In each situations,

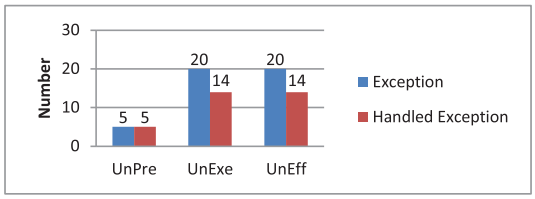

FIGURE 22. Number of exceptions in various running situations.

TABLE 8. Bridge rules in various runnning situations.

Run No.	Bridge rules among SKBs
run1	{}
run2	{SKB1, SKB2}
run3	{SKB1, SKB2, SKB3}
run4	{SKB1, SKB2, SKB3, SKB4}
run5	{SKB1, SKB2, SKB3, SKB4, SKB5}

only a set of bridge rules works. Table 8 shows available bridge rules in each running situation. In *run1*, no bridge rule is adopted for the five SKBs. In *run2*, bridge rules between *SKB1* and *SKB2* are adopted. And, from *run2* to *run5*, available bridge rules become more and more.

The response success rates in the five situations are shown in Fig. 23. It can be seen that the response success rate becomes higher as the number of bridge rules increase.

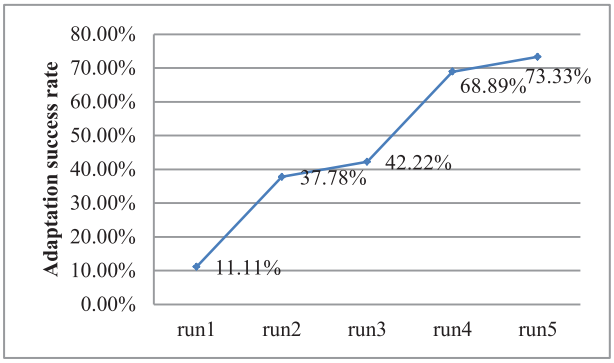

FIGURE 23. Adaptation success rate comparison among various runs.

Specially, in *run5*, all bridge rules are adopted, and the response success rate of D3LSRAF reaches ideal *73.33%*. During the running, *63* exceptions are caught, including *45* local self-adaptation exceptions and *18 LocalAdaptFail* global exceptions. And there are *6 LocalAdaptFail* exceptions that are handled successfully. Other self-adaptation details, including invocation frequency of the planner (*PlanFreq*), number of adaptation processes (*APNum*), number of exceptions caught (*ExcNum*), number of successful request running instances, are recorded in Fig. 24. D3LSRAF automatically and successfully repair *33* of *45* process instances. It invokes the planner *71* times and generates *41* adaptation processes at runtime, where *8* adaptation processes in global self-adaptation strategies are used to cancel completed effects for repairing the main process further. Meanwhile, during repairing each exception, the planner is invoked *1* times least and *3* times most.

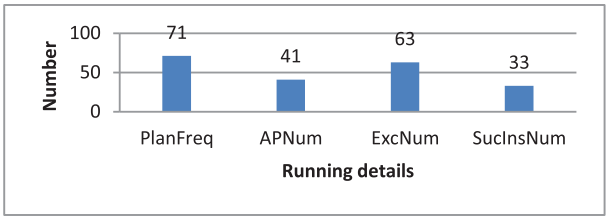

FIGURE 24. Self-adaptation running details in *run5*.

Furthermore, we also record the running status and total response time under various running situations for all service requests in Table 5, shown in Table 9. Specially, during the running, these requests also encounter those predefined exceptions when their expected initial paths can be generated.

The *sr1* and *sr2* both run successfully by means of local replanning strategies under all running situations. And more bridge rules would lead to more total response time. For *sr1*, total response time under *run5* is *148ms* more than *run1*. And for *sr2*, this difference is *344ms*. This is because, during the response, bridge rules can be utilized to convert a *D3L-SR* into a *Single-SR* in *D3LLocalPlanning*, or to create planning graph in *D3LGlobalPlanning* when an initial path is generated.

The *sr3* fails to search an initial path, and only spends *8s* in *run1*. This is because that the request can't be achieved by services from a SKB, and that no bridge rule can be used to explore cooperation among services from multiple SKBs. However, in other situations, bridge rules between *SKB1* and *SKB2* are utilized to support this exploration. Therefore, *sr3* runs successfully under from *run2* to *run5*. And *UnPre* exception over *TakeTrain* is solved by a local replanning strategy. The maximum difference of total time when *sr3* runs successfully is *196ms*.

Just as *sr3*, *sr4* also can't search an initial path under *run1*, and less time is spent. In other situations, an initial path is found with the help of bridge rules among *SKB1* and *SKB2*. However, *UnExe* over *TakeTrain* occurs at runtime, and it doesn't be solved by local and global replanning strategies under *run2*. In *run3*, *run4*, and *run5*, the global replanning strategy succeeds to run by means of bridge rules related to *SKB3*, and *sr4* runs successfully. The maximum difference of total time when *sr4* runs successfully is *735ms*.

Compared with other requests, an initial path of *sr5* includes more services from multiple SKBs. In *run1*, *run2*, and *run3*, no initial path is found. However, as bridge rules becomes more, the searching space becomes greater and total response time becomes more. In *run4*, an initial path is found, and it encounters *UnExe* over *CheckInInn*. The exception doesn't be solved by local and global replanning strategies, and *sr5* fails to run ultimately. In *run5*, *sr5* succeeds to run because the global replanning strategy succeeds by means of bridge rules related to *SKB5*. Specially, in *run4*, *D3LGlobalPlanning* in the global replanning strategy can spend a lot of time to determine that no adaptation path can be searched until the planning graph reaches a max level.

TABLE 9. Running status and total response time of various service requests in Table 5 under five running situations in Table 8.

	run1		run2		run3		run4		run5	
Req	State	Time	State	Time	State	Time	State	Time	State	Time
sr1	success	1228	success	1304	success	1337	success	1340	success	1376
sr2	success	1215	success	1300	success	1336	success	1336	success	1559
sr3	failure	9	success	1611	success	1619	success	1681	success	1807
sr4	failure	8	failure	1819	success	3392	success	3447	success	4127
sr5	failure	8	failure	136	failure	918	failure	4395	success	4197

However, in *run5*, before the planning graph reaches a max level, *D3LGlobalPlanning* has found an adaptation path. Therefore, less time is spent in *run5* than *run4*.

From the illustration above, it can be seen that bridge rules can remarkably improve the response success rate of service requests, and that total response time can increase slightly as more bridge rules are utilized. However, when a request is responded successfully, the time of growth, that affected by bridge rules, doesn't exceed *1s*. Comparing with real service invocation time, this can be accepted in practice.

VII. CONCLUSION

In this paper, we model services from the same domain ontologies as a SKB, and multiple SKBs and bridge rules among them construct a DSKB. Based on DSKB, we propose a D3L-based service runtime self-adaptation framework (D3LSRAF). The framework adopts a local replanning strategy to repair *UnPre*, *UnExe*, *UnEff* exceptions at SSA level, and a global replanning strategy to repair the *LocalAdaptFail* at path level. In the local replanning, the D3L-based local planning algorithm is utilized to rapidly generate a local path through concurrently searching a path from each SKB. In the global replanning, the D3L-based global planning algorithm is used to explore all possible paths for the original business goal, and generate a global path ultimately. The global planning algorithm improves classical graph planning algorithm through constructing a planning graph including bridge rules. The cooperation of the two strategies can guarantee that the framework responses a business goal with high response efficiency and response success rate.

Based on the Activiti engine and BPMN2.0 language, we developed a prototype system for D3LSRAF. The system not only provides essential self-adaptively running environment for a *D3L-SR*, but also is compatible with multiple types of services. Through extending *serviceTask* in BPMN2.0, we design two service invocation adapters and a global adaptation activity. The adapters are utilized to respectively invoke RESTful APIs and SOAP-based services, and they also include the local replanning logics for *UnPre*, *UnExe*, and *UnEff* exceptions. And the global adaptation activity encapsulates the global replanning logic, and it is invoked when *LocalAdaptFail* exception occurs.

A series of experiments in previous section show that D3LSRAF can more quickly response various *D3L-SR* requests even if some exceptions occur at runtime, and can guarantee higher response success rate, compared with other self-adaptation approaches.

The aim of D3LSRAF is to achieve the function requirement in a business goal. However, in practice, other requirements, such as timing, QoS constraints [43]–[46], and higher self-adaptation efficiency, also are desired when a self-adaptation carries on. Meanwhile, bridge rules in D3LSRAF are created manually before a request is responded, and this is time-consuming work. Therefore, in the further, we will extend our framework considering self-adaptation under timing and QoS constraints, higher self-adaptation efficiency, and automatic bridge rule generation method.

ACKNOWLEDGMENT

Xianghui Wang thanks her colleagues from Shandong Jianzhu University for their comments.

REFERENCES

[1] N. Dragoni *et al.*, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Cham, Switzerland: Springer, 2017, pp. 195–216.
[2] Y. Alotaibi and F. Liu, "Survey of business process management: Challenges and solutions," *Enterprise Inf. Syst.*, vol. 11, no. 8, pp. 1–35, 2016.
[3] F. Caron and J. Vanthienen, *Exploring Business Process Modelling Paradigms and Design-Time to Run-Time Transitions*. New York, NY, USA: Taylor & Francis, 2016.
[4] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decade's overview," *Inf. Sci.*, vol. 280, pp. 218–238, Oct. 2014.
[5] A. L. Lemos, F. Daniel, and B. Benatallah, "Web service composition: A survey of techniques and tools," *ACM Comput. Surv.*, vol. 48, no. 3, 2015, Art. no. 33.
[6] I. Paik, W. Chen, and M. N. Huhns, "A scalable architecture for automatic service composition," *IEEE Trans. Serv. Comput.*, vol. 7, no. 1, pp. 82–95, Jan./Mar. 2014.
[7] J. Bronsted, K. M. Hansen, and M. Ingstrup, "Service composition issues in pervasive computing," *IEEE Pervasive Comput.*, vol. 9, no. 1, pp. 62–70, Jan. 2010.
[8] X. Wang and Z. Feng, "Semantic Web service composition considering iope matching," *J. Tianjin Univ.*, vol. 50, no. 9, pp. 984–996, 2017.
[9] A. P. Barros and M. Dumas, "The rise of Web service ecosystems," *IT Prof.*, vol. 8, no. 5, pp. 31–37, Sep. 2006.
[10] M. J. Hadley, *Web Application Description Language (WADL)*. London, U.K.: Pearson, 2012, ch. 10.
[11] Y. Liu, Y. Fan, K. Huang, and W. Tan, "Failure analysis and tolerance strategies in Web service ecosystems," *Concurrency Comput. Pract. Experim.*, vol. 27, no. 5, pp. 1355–1374, 2015.
[12] A. Bucchiarone, A. Marconi, M. Pistore, and H. Raik, "Dynamic adaptation of fragment-based and context-aware business processes," in *Proc. IEEE Int. Conf. Web Services*, Jun. 2012, pp. 33–41.
[13] G. H. Alférez, V. Pelechano, R. Mazo, C. Salinesi, and D. Diaz, "Dynamic adaptation of service compositions with variability models," *J. Syst. Softw.*, vol. 91, no. 5, pp. 24–47, 2014.
[14] A. Marrella, M. Mecella, and A. Russo, "Featuring automatic adaptivity through workflow enactment and planning," in *Proc. Int. Conf. Collaborative Comput., Netw., Appl. Worksharing*, Oct. 2011, pp. 372–381.
[15] A. Marrella and M. Mecella, *Continuous Planning for Solving Business Process Adaptivity*. Berlin, Germany: Springer, 2011.

[16] X. Wang, Z. Feng, K. Huang, and W. Tan, "An automatic self-adaptation framework for service-based process based on exception handling," *Concurrency Comput. Pract. Exper.*, vol. 29, no. 5, p. e3984, 2017.
[17] A. Bucchiarone, A. Marconi, M. Pistore, P. Traverso, P. Bertoli, and R. Kazhamiakin, "Domain objects for continuous context-aware adaptation of service-based systems," in *Proc. IEEE Int. Conf. Web Services*, Jun./Jul. 2013, pp. 571–578.
[18] M. Kuzu and N. K. Cicekli, "Dynamic planning approach to automated Web service composition," *Appl. Intell.*, vol. 36, no. 1, pp. 1–28, 2012.
[19] Z. Shi, M. Dong, Y. Jiang, and H. Zhang, "A logical foundation for the semantic Web," *Sci. China Ser. F, Inf. Sci.*, vol. 48, no. 2, pp. 161–178, 2005.
[20] L. Chang, Z.-Z. Shi, L.-R. Qiu, and F. Lin, "A tableau decision algorithm for dynamic description logic," *Chin. J. Comput.*, vol. 31, no. 31, pp. 896–909, 2008.
[21] Y. Jiang, Z. Shi, Y. Tang, and J. Wang, "A distributed dynamic description logic," *J. Comput. Res. Develop.*, vol. 43, no. 9, pp. 1603–1608, 2006.
[22] X. F. Zhao, D. P. Tian, Y. H. Shi, and Z. Z. Shi, "Knowledge propagation and reasoning induced by bridge rule chains in D3L," *Chin. J. Comput.*, vol. 37, no. 12, pp. 2421–2426, 2014.
[23] N. R. T. P. van Beest, E. Kaldeli, P. Bulanov, J. C. Wortmann, and A. Lazovik, "Automated runtime repair of business processes," *Inf. Syst.*, vol. 39, pp. 45–79, Jan. 2014.
[24] C. Liang, J. Liu, T. L. Gu, and Z. Z. Shi, "Semantic Web service composition based on dynamic description logics," *Chin. J. Comput.*, vol. 36, no. 12, pp. 2468–2478, 2013.
[25] X. Hu, Z. Feng, S. Chen, K. Huang, J. Li, and M. Zhou, "Accurate identification of ontology alignments at different granularity levels," *IEEE Access*, vol. 5, pp. 105–120, 2017.
[26] M. Maree and M. Belkhatir, "Addressing semantic heterogeneity through multiple knowledge base assisted merging of domain-specific ontologies," *Knowl.-Based Syst.*, vol. 73, pp. 199–211, Jan. 2015.
[27] Z. Wang, H. Hu, L. Chen, and Z. Shi, "Parallel computation techniques for dynamic description logics reasoning," *J. Comput. Res. Develop.*, vol. 48, no. 12, pp. 2317–2325, 2011.
[28] N. Bieberstein, R. G. Laird, K. Jones, and T. Mitra, *Executing SOA: A Practical Guide for the Service-Oriented Architect*. Indianapolis, IN, USA: IBM Press, 2008.
[29] K. Huang, J. Yao, J. Zhang, and Z. Feng, "Human-as-a-service: Growth in human service ecosystem," in *Proc. IEEE Int. Conf. Services Comput.*, Jun./Jul. 2016, pp. 90–97.
[30] C. Barreto *et al.*, *Web Services Business Process Execution Language Version 2.0*, document wsbpel-specification-draft-01, WS-BPEL TC OASIS, 2007.
[31] M. J. Hadley, *Web Application Description Language (WADL)*. Santa Clara, CA, USA: Sun Microsystems, Inc., 2006.
[32] D. Booth and K. L. Canyang, *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*, document REC-wsdl20-primer-20070626, W3C Rec., 2007.
[33] J. Kopecký, K. Gomadam, and T. Vitvar, "hRESTS: An HTML microformat for describing RESTful Web services," in *Proc. IEEE/WIC/ACM Int. Conf. Web Intell. Intell. Agent Technol. (WI-IAT)*, Dec. 2009, pp. 619–625.
[34] D. Martin *et al.*, "OWL-S: Semantic markup for Web services," in *Proc. Int. Semantic Web Work. Symp. (SWWS)*, 2004, pp. 1–38.
[35] J. Domingue, D. Roman, and M. Stollberg, "Web service modeling ontology (WSMO): An ontology for semantic Web services," in *Proc. W3C Workshop Frameworks Semantics Web Services*, 2005, pp. 9–10.
[36] J. Kopecký, T. Vitvar, C. Bournez, and J. Farrell, "SAWSDL: Semantic annotations for WSDL and XML schema," *IEEE Internet Comput.*, vol. 11, no. 6, pp. 60–67, Nov. 2007.
[37] C. Lira and P. Caetano, *REST-Based Semantic Annotation of Web Services*. Cham, Switzerland: Springer, 2016, pp. 269–279.
[38] D. Roman, J. Kopecký, T. Vitvar, J. Domingue, and D. Fensel, "WSMO-lite and hRESTS: Lightweight semantic annotations for Web services and RESTful APIs," *Web Semantics Sci. Services Agents World Wide Web*, vol. 31, pp. 39–58, Mar. 2015.
[39] D. Jordan *et al.*, *Web Services Business Process Execution Language (WSBPEL) 2.0*, OASIS Standard wsbpel-v2.0-OS, 2007.
[40] S. F. Apache. *Apache Ode (Orchestration Director Engine)*. Accessed: Dec. 12, 2017. [Online]. Available: http://ode.apache.org
[41] S. A. White *et al.*, *BPMN 2.0 Handbook: Methods, Concepts, Case Studies and Standards in Business Process Modeling Notation*, 2nd ed. Lighthouse Point, FL, USA: Future Strategies Inc., 2011.
[42] Z. Laliwala and I. Mansuri, *Activiti 5.x Business Process Management, Beginner's Guide*. Birmingham, U.K.: Packt Publishing, 2014.
[43] K. Huang, Y. Fan, and W. Tan, "Recommendation in an evolving service ecosystem based on network prediction," *IEEE Trans. Autom. Sci. Eng.*, vol. 11, no. 3, pp. 906–920, Jul. 2014.
[44] P. Potena, "Optimization of adaptation plans for a service-oriented architecture with cost, reliability, availability and performance tradeoff," *J. Syst. Softw.*, vol. 86, no. 3, pp. 624–648, 2013.
[45] M.-O. Cordier, R. Micalizio, S. Robin, and L. Roze, "Adapting Web services to maintain qos even when faults occur," in *Proc. IEEE Int. Conf. Web Services*, Jun./Jul. 2013, pp. 403–410.
[46] P. Xiong, Y. Fan, and M. Zhou, "Web service configuration under multiple quality-of-service attributes," *IEEE Trans. Autom. Sci. Eng.*, vol. 6, no. 2, pp. 311–321, Apr. 2009.

XIANGHUI WANG (M'12) received the B.S. and M.S. degrees from the School of Computer Science and Technology, Shandong University, China, in 2002 and 2005, respectively. She is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Tianjin University, China. She is also an Assistant Professor with the School of Computer Science and Technology, Shandong Jianzhu University, China. Her research interests include knowledge engineering and service computing. She is a member of ACM.

ZHIYONG FENG (M'10) received the Ph.D. degree from Tianjin University. He is currently a Full Professor with the School of Computer Science and Technology, Tianjin University, China. He has authored one book, over 130 articles, and 39 patents. His research interests include knowledge engineering, service computing, and security software engineering. He is a member of the IEEE Computer Society and the ACM.

KEMAN HUANG (M'14) received dual B.S. degrees from the Department of Automation and School of Economics and Management from Tsinghua University, China, in 2009, and the Ph.D. degree from the Department of Automation, Tsinghua University, China, in 2014, respectively. He was an Assistant Professor with the School of Computer Science and Technology, Tianjin University, from 2014 to 2016. He is currently a Research Scientist with the Sloan School of Management, MIT, USA. He has authored or co-authored over 40 journal and conference proceedings papers. His research interests include service ecosystem, cyber security behavior, and semantic web. He received the Best Paper Runner-up Award from IEEE SCC 2016 and the Best Student Paper Award from IEEE ICWS 2014 and ICSS 2013. He was on the program committees of many conferences and the publicly Chair of IEEE ICWS/SCC/MS/BIGDATA Congress 2016. He is a member of ACM.