WANG Xianghui (Orcid ID: 0000-0002-5887-7775)

Self-adaptation and Distributed Knowledge-based Service Ecosystem Evolution

Xianghui Wang[1], Zhiyong Feng[2], Keman Huang[3,4,*], Shizhan Chen[2]

1 Department of Computer Science and Technology, Shandong Jianzhu University, Jinan 250101, China

2 Department of Computer Science and Technology, Tianjin University, Tianjin 300072, China

3 School of Information, Renmin University of China, Beijing 100086, China

4 Sloan School of Management, MIT, Cambridge 02142, USA

wxh_225@163.com, zyfeng@tju.edu.cn , keman@ruc.edu.cn, Shizhan@tju.edu.cn

Abstract: Web services (or Web APIs) on the Internet tends to encounter various unexpected runtime failures because of their dynamicity and distribution. Self-adaptation technologies for the service-based business process can effectively repair runtime failures and improve its success rate. However, the same failures may occur on subsequent invocations because relevant processes do not evolve after failures. This makes the response time of the business processes too long. We proposed a self-adaptation and distributed knowledge-based evolution model (SDKEM) to guarantee business processes' stabilities, that is, low failure rates and stable response time. SDKEM adopts a service knowledge base (SKB) to organize services from a provider and uses bridge rules to eliminate semantic conflicts among multiple distributed SKBs. It can automatically trigger the evolution of a service ecosystem through the designed self-adaptation mechanism. We adopt the "survival of the fittest" principle for crucial elements in the ecosystem during evolution so that ultimately, service-based processes and services with high stability remain. Experiments show that, with the developed evolution mechanism, runtime failures of business processes significantly reduce. In most cases, their response time and success rates are comparable to those under the running situation where no runtime failure occurs, meaning the runtime failures within a service-based process are automatically repaired.

Key words: service ecosystem, runtime self-adaptation, distributed knowledge, service evolution, stability evaluation.

1 Introduction

Under cloud computing and micro-service architectures, more and more low-cost services emerge on the Internet[1][2]. They are provided by different providers through open services or API platforms,

such as programmableweb [1]. Meanwhile, various business operations and human tasks[3] in an enterprise also are encapsulated as services, and they are implemented through real software applications or employees from different departments. A developer can utilize off-the-shelf process model tools[4][5] and service composition technologies[6][7] to create various business processes for specific business goals. Thus, in an enterprise, a service ecosystem naturally consists of users (providers and consumers), services and service-based business processes, and infrastructure[8].

Notably, an existing business process can be used as a service and becomes a component service of other business processes. Under a dynamic and distributed environment, business processes tend to encounter runtime failures because their component services may fail when being invoked due to various situations. For example, a service is removed, its interface description is changed, or some network exception occurs. To reduce the failure rates of business processes, self-adaptation technologies for service-based processes[9][10] are adopted to repair faulted processes at runtime. However, after self-adaptation, faulted services and original processes don't be evolved. This means that the same failures will occur when retuning the process. Therefore, it is necessary to provide an evolution model to evolve services and business processes in a service ecosystem continuously. During evolution, "survival of the fittest" occurs among these elements. That is, "bad" services are eliminated in the course of competition; and, relevant processes are updated into new versions through using "good" services to replace those "bad" ones. Thus, in a service ecosystem, all business processes will have a high success rate and stable response time.

In practice, it will encounter three main difficulties in the following to implement the evolution model.

1) It is difficult to evaluate how "bad" of web services and business processes. In practice, providers offer quality information about their services, such as user number, success rate etc. However, different providers provide different quality indexes and evaluation criteria. Also, providers, services, and business processes can influence the evaluation of each other.

2) There is no holistic and continuous evolution approach for a service ecosystem. Existing approaches either off-line evolve services and relevant service-based processes, or online adjust the execution path of a process only for the current run. And they ignore the holistic and continuous evolution of a service ecosystem. Hence, services and providers' changes cannot immediately be reflected in relevant processes and increase the failure rate and response time of business processes.

[1] https://www.programmableweb.com 2021-01-26

3) Conflicts among semantics of services hamper the evolution of a service ecosystem. The semantics of services can play an important role in determining their competition and cooperative relationships. However, semantic vocabularies may be from different ontologies, so that some conflicts may exist. For example, two vocabularies with the same name have different semantics, or two vocabularies with different names have the same semantics.

This paper proposes an evolution model that can overcome the difficulties above and guarantee holistic and continuous healthy evolution of a service ecosystem. The model consists of a stability evaluation model and various evolution mechanisms. Based on the self-adaptation mechanism and distributed knowledge among different providers, it can automatically catch evolution opportunities and evolve various elements in a service ecosystem.

The main contributions of this paper are in the following folds:

1) A stability evaluation model is designed to evaluate the stabilities of services, providers, and business processes.
2) A self-adaptation and distributed knowledge-based evolution model for a service ecosystem is proposed.
3) Experiments show that our evolution model can continuously evolve a service ecosystem, which significantly decreases the runtime failure rate of services in running business processes and guarantees the stability of business processes.

The remainder of this paper is organized as follows. Section 2 introduces related works and section 3 presents preliminary work of our evolution model. Section 4 describes an overview and architecture of our evolution model. Section 5 presents a stability evaluation model in a service ecosystem, and then section 6 provides approaches about competition and cooperation considering stability and distributed knowledge. Section 7 describes evolution mechanisms in the model. Section 8 shows a prototype system and reports the empirical results. Section 9 concludes the paper.

2 Related Work

Like the natural world ecosystem, a service ecosystem is a dynamic system with a continuous evolution in the digital world. Web services (or Web APIs) are considered as species, and they are produced by special providers or through integrating multiple other services[8]. Users can utilize these services to achieve their own functional or non-functional requirements. Because of dynamism, a service

ecosystem is always changing. For example, new services emerge, old services are removed, or services change due to general network exceptions.

Recently, there are some researches on the evolution of service ecosystems from different perspectives. Some studies focus on the user-centric service ecosystem. Literature[11] constructed service ecosystem with time feature according to the current user's API usage history and aimed to recommend appropriate APIs to users. Some investigate the global service ecosystem from a macro-economic perspective. Literature [12][13] adopted a complex social network model to illustrate the structure of service ecosystems and predict the evolution trend of service ecosystems from the view of a business. The prediction results are used to provide decision support for providers or market regulators. Others focus on a developer-centric service ecosystem and help developers build and iterate service-based software quickly. Our approach is related to the evolution of the developer-centric service ecosystem. According to the difference of information used in evolution, these researches are divided into four categories: interface-document-based, running-log-based, and complex-network-based and runtime-self-adaptation-based.

2.1 Interface Document based

Interface description documents are open in the form of WSDL, WADL[14], or text documents etc., and also are a unique basis of invoking related services. A service's functional and structural changes can be reflected in its interface document and can be identified through comparison between new and old interface documents. WSDarwin[15] is an evolution framework for a service-oriented system. When a service-based system failed to run, it compared interface description documents. Compared results were used to update old client stub of this service. The old client stub invoked the new client stub. Meanwhile, it still received old inputs and responded to old outputs. Literature[16] automatically checked the change of description documents through notification management architecture for service evolution. They used an evolution agent (EVA) to manage a group of RESTful services, and maintain client lists for each service. Once changes occur over a service, the EVA automatically informs all clients of this service, and then it deployed corresponding new service implementation. This approach only evolves services in the ecosystem, and doesn't specify evolution approaches for related service-based processes.

2.2 Running Log based

The running logs of service-based systems can reflect running details of related services, including success or failure, session-level data among different services etc. Adalberto etc.[17] proposed a

service evolution model. Through retrospectively and prospectively analyzing these logs, past model and future model are obtained, respectively. And these can offer important suggestions about system architecture improvement and deployment trade-offs for developers. The analysis work for running logs is time-consuming and can't provide an evolution strategy for the service ecosystem in a real-time style.

2.3 Complex Network based

Elements in a service ecosystem have various correlations, such as service-service, service-provider, and service (service composition)-consumer, and these correlations can be modeled by a complex network[18].

Through analyzing the complex network, Xia etc.[19] identified the feature of perishing services, and used a statistic machine learning algorithm to forecast potential perishing services. This can provide key information for service ecosystem evolution. Based on a complex network model for a service ecosystem, Liu etc.[20] predicted failure services, and they were replaced from existing service-based processes through special replacement strategies. This approach can update all service-based processes related to those failure services. Like the interface-document-based approach, it can reduce service failure rate when a service-based process runs and guarantees the ecosystem's response efficiency. However, it can't immediately evolve running processes when runtime failures occur.

2.4 Runtime Self-adaptation based

By means of runtime self-adaptation, a runtime failure over a service-based process can be automatically repaired by adjusting the running process's execution paths in time. Existing runtime self-adaptation approaches mainly include three categories: exception mechanism based[21][22][23], ECA (event-condition-action) rules[24] or variability models[25][26], and goal-based[27][28]. Those approaches above only updated the current execution of a process and didn't update the process definition. Failures caught may occur again in the next execution. Thus, response time could not be reduced when the same failures occurred again.

In summary, none of the existing approaches automatically implements holistic and continuous evolution of a service ecosystem because of the various problems above. They can't guarantee the sustainable stability of business processes.

Therefore, we provided a service ecosystem evolution model DKEM in our previous work[29]. Based on the proposed evolution mechanism, the model could guarantee holistic and continuous evolution of

a service ecosystem. However, the work only provided an idea and was short of implementation details. In this paper, we add implementation details and improve the model.

3 Preliminary Work

In this paper, our evolution model is implemented on the basis of self-adaptation and distributed knowledge and is called SDKEM. It employs a self-adaption mechanism and distributed knowledge conception in our previous self-adaptation framework D3LSRAF[10]. D3LSRAF is implemented by combining exception mechanisms and goal-based approaches. Here, preliminary works will be presented in the following.

3.1 Self-adaptation Mechanism

In D3LSRAF, a service-based process was described by BPMN2.0[21]. Each service in the process was encapsulated as a *serviceTask* in BPMN2.0. In the *serviceTask*, there was a built-in exception handling unit for the current service. The exception unit could catch three types of unexpected runtime failures over the current service, and invoke corresponding adaptation strategies for these failures to repair the service. The three types of failures were called local failures, include *UnPre* (preconditions unsatisfied), *UnExe* (execution failure), and *UnEff* (effects unachieved). Meanwhile, if the service failed to be repaired, a global failure over the current process was thrown and would be caught and handled by a built-in exception handling unit at the process level. The global failure was called *LocalAdaptFail*. A service-based process following previous definition specifications could run on an existing workflow engine for BPMN2.0, and automatically repair various failures through preconfigured adaptation strategies in the corresponding exception-handling unit.

3.2 Service Knowledge Base

Services from the same provider and business domain are organized as an individual service knowledge base (SKB). An SKB includes four components: *D*, *TP*, *Facts*, and *Actions*. Here, *D* represents an ontology; *TP* represents services where their IOPE features are annotated by the ontology in *D*; *Facts* represents facts related to IOPEs of these services, and the facts may be known or produced by existing service instances; and *Actions* represents existing service instances corresponding to services in *TP*, and their input parameters and preconditions are instantiated by facts in *Facts*.

For instance, *SK* is an SKB and its component *TP* includes various services about a train. Component *D* provides vocabularies about a train, such as **City**, **Station**, **Date**, **Train** and so on.*ProposeTrain* is a service in *TP* can provide relevant train information according to given cities and date information.

To make services from different knowledge bases cooperate with each other, various bridge rules between any two different ontologies are introduced to eliminate semantic conflicts among different knowledge bases, and they can describe *equivalence*, *subclass* or *sameAs* relations between two vocabularies in different ontologies. For example, semantic vocabularies *Station* and *TrainStation* are two concepts respectively in ontology *o1* and *o2*, but they have the same semantics. Thus, a bridge rule $< o1, Station, equalc, o2, TrainStation >$ is introduced and means that *Station* and *TrainStation* have the same semantics. Here, the rule type is *equalc*. In D3LSRAF, six types of bridge rules are introduced: *intoc*, *ontoc*, *equalc*, *intor*, *ontor* and *equalr*. The first three are used for concepts, and the last three are for object properties. Rules with the prefix *into*, *onto* and *equal* respectively represent included, including and equality relationships between two concepts or properties. Especially, $equalr_F$ is the bridge rule type with the condition, and when the condition in *F* is true, the corresponding rule will hold.

3.3 Automatic Service Composition Considering Distributed Knowledge

In D3LSRAF, an automatic service composition planner is the core operation to implement self-adaptation of service-based processes, and it is used to find suitable alternatives for faulted services or business processes from available services. The planner can find composition results from single SKB or from multiple SKBs by means of two planning algorithms: *LocalD3LPlanning* and *GlobalD3LPlanning*. *LocalD3LPlanning* can search for a solution from each SKB, and each solution only consists of services from the corresponding SKB. However, *GlobalD3LPlanning* can search for a solution from multiple SKBs, and services in a solution may be selected from different SKBs.

The two algorithms are both implemented through improving classical AI graph planning algorithms.

Given a service request $< In, Init, Out, Goal >$, where *In* and *Init* represents known input parameters and known preconditions about these inputs, *Out* and *Goal* represents expected output parameters and expected effects. Firstly, the planning algorithms create a planning graph where state layers and action layers alternately according to component *In* and *Init*, and then search composition solution from the graph according to component *Out* and *Goal*. Each state layer includes all facts that can be produced by service instances in the previous action layer, and each action layer includes all service instances that can be satisfied by facts in the previous state layer.

4 Overview of SDKEM

In this paper, a service ecosystem is developer-oriented and consists of various elements related to building software systems: web services, service-based business processes, and stakeholders. And stakeholders have three types: software developers, providers, and users. Generally, software developers can collect available web services from different providers on the Internet and integrate them together to form various business processes that satisfy different requirements of users.

In practice, a service ecosystem continuously changes because of various dynamic factors. For example, old web services are removed or updated, new web services are added, old requirements of users are changed etc. These changes may make those old business processes fail to run or can't produce expected results. To make the ecosystem evolve normally, software developers need to repair those faulted or unsuitable business processes in time. It is time-consuming and trivial for developers to do this repair manually.

SDKEM can assist developers in automatically evolve a service ecosystem and reduce software maintenance costs for developers. In SDKEM, services from one provider are organized as an individual SKB. Here, a SKB may use multiple ontologies to annotate their services semantically. And, bridge rules in D3LSRAF are adopted to eliminate various semantic conflicts among different ontologies. During the evolution of a service ecosystem, they can support competition and cooperation among services from distributed SKBs. Fig. 1 shows the structure of the service ecosystem for SDKEM.

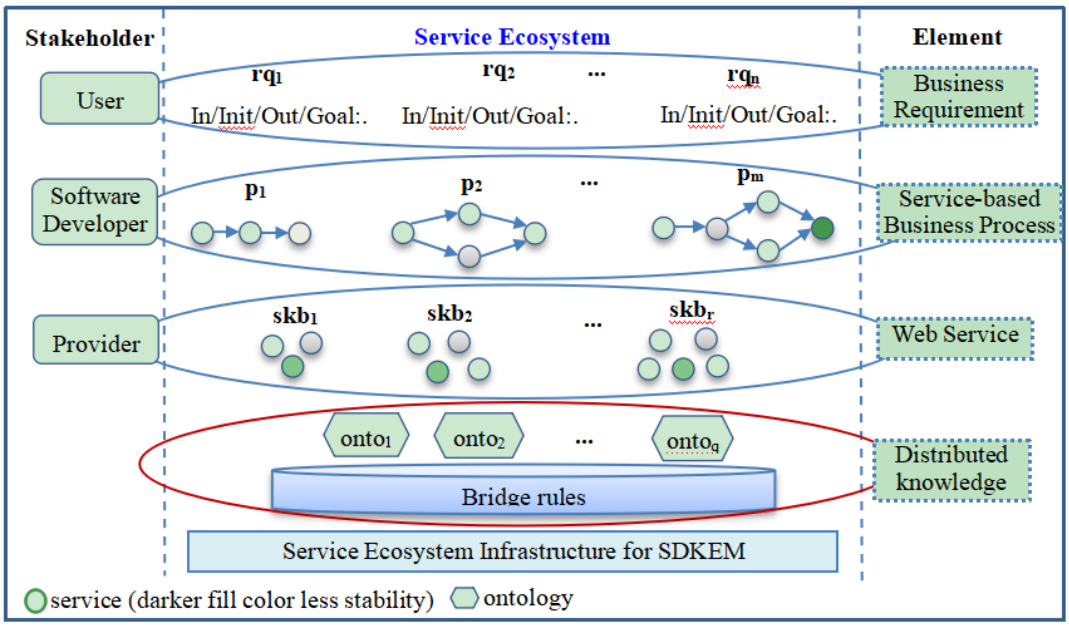


Figure 1. Structure of service ecosystem for SDKEM.

To implementation the evolution of a service ecosystem, SDKEM solves three main problems in the following:

- How to evaluate "good" or "bad" web services or business processes?
- How to evolve business processes?
- How to evolve services and bridge rules?

SDKEM provides a stability evaluation model to compute the stability of business processes, services, and SKBs. Here, an element with higher stability is better, and will win out during evolution. And, the stability is computed according to service running histories, including invocation times, failure type, failure times etc.

SDKEM considers a runtime failure over the current business process as an evolution opportunity and then automatically evolves the process at runtime. After evolution, a faulted process is substituted with a new version with higher stability than the old. Catching failures and invoking evolution operations are implemented through a self-adaptation mechanism in D3LSRAF. Automatic service composition algorithms in D3LSRAF are improved to find an alternative with higher stability for current processes.

In SDKEM, every invocation information(success or failure) for every service will be recorded when related business processes run. Thus, according to this information, the stability of services is computed by the stability evaluation model. Those services with low stability will be automatically marked as perishing services. Developers will further evolve the perishing services through updating service descriptions or removing them. Furthermore, SDKEM also assists developers to semi-automatically evolve bridge rules when new vocabularies are added to existing ontologies or new ontologies are introduced. This can ensure that full competition and cooperation among services from different SKBs occur.

Service ecosystem infrastructure for SDKEM in Fig. 1 provides functions to support the various previous evolution and consists of three databases and four function modules. The architecture of the infrastructure is shown in Fig. 2. And details of its components are shown in the following.

Service DB database stores basic information in a service ecosystem, including service semantic and syntactic information, SKBs, service-based processes.

Service Running Histories database stores service invocation histories during the life cycle of this ecosystem, including invocation times, various exception times, etc.

Bridge Rule DB database stores all domain ontologies that are used in SKBs, and various bridge rules among them.

Manage Process module mainly consists of sub-modules related to service-based processes.

- *Manage Process Definition* is used to create a service-based process. It can invoke *Service Composition with Stability* sub-module to implement automatic creation of the process.
- *Service Composition with Stability* can automatically generate a service-based process with higher stability for a service request according to the current running context. It implements two automatic service composition algorithms to separately obtain a composition result with high stability from one SKB or all knowledge bases. The module also provides the automatic conversion operation from a semantic composition solution to a syntactic one to execute the service-based process.
- *Process Executor* provides the running environment for a defined process, and can directly employ an existing workflow engine, such as Activiti[2].
- *Self-adaptation for service-based Process* can automatically execute and monitor a service-based process. When a failure occurs at runtime, this module immediately executes the automatic repair for the failure through *Service Composition with Stability* and *Process Executor* module. Here, we achieve this module by improving the self-adaptation framework in D3LSRAF.
- *Evolve Service-based Process* can invoke *Self-adaptation for service-based Process* and *Service Composition with Stability* module when a process is evolved.

Evaluate Stability module can compute stability of services, SKBs, and service-based processes according to following stability evaluation model and running data in **Service Running Histories** database. The module will be invoked while evolving a process.

Manage Service module can manage service definitions from providers, manage various SKB base information, and assist developers in evolving those perishing services and SKBs.

Manage Distributed Knowledge module can manage ontologies about semantics, manually manage bridge rules, and evolve bridge rules. Here, *Manage Ontology* sub-module can employ an existing ontology management tool, such as Protege[3].

[2] https://www.activiti.org/

[3] https://protege.stanford.edu/

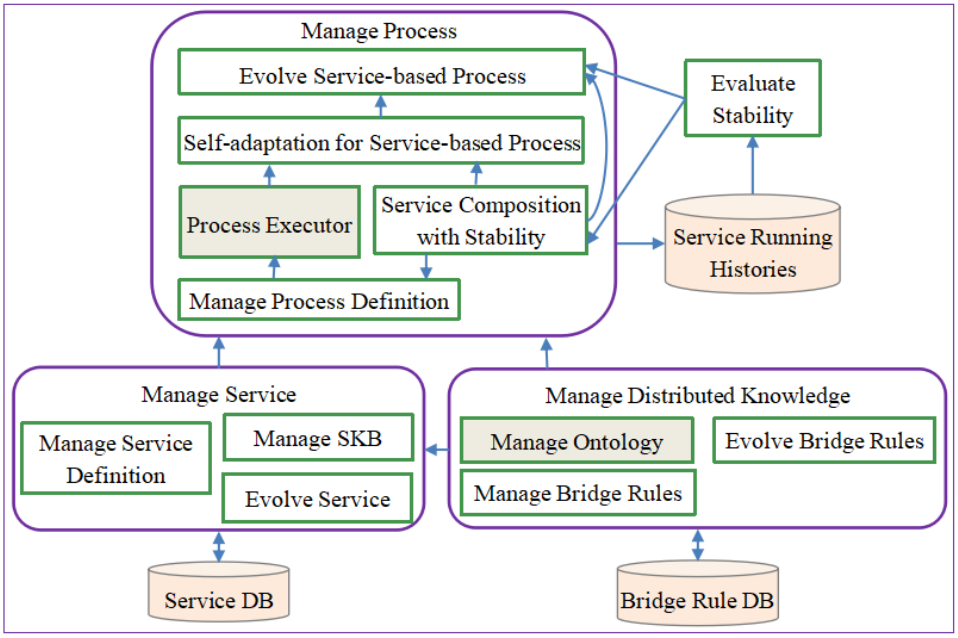


Figure 2. Architecture of Service Ecosystem Infrastructure for SDKEM.

The main implementation details of SDKEM will be shown in the following sections.

5 Stability Evaluation of Service Ecosystem

This section firstly presents stability concepts and formal definitions of elements in a service ecosystem, then illustrates a stability evaluation model.

5.1 Stability Concept View

In a service ecosystem, a provider can provide multiple services, and a service-based process can compose of multiple services from different providers. Here, we model services from a provider as a SKB. The stability concept view is shown in Fig. 3.

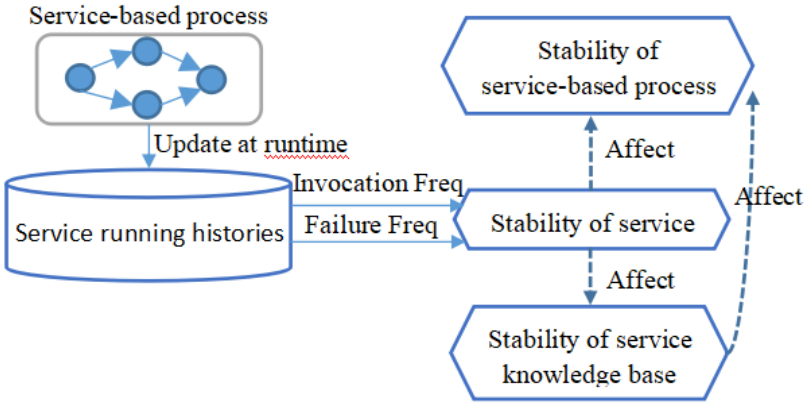


Figure 3. Stability concept view.

The stability of service, SKB, and the service-based process is the basis of competition during an evolution. In our paper, three assumptions are made: 1) a service with high stability always can succeed in to respond a request during its life cycle; 2) a highly stable service-based process always not only succeed in responding but also has optimal response time for each invocation; 3) all services in a highly stable SKB have high stability. These assumptions are consistent with the actual situation in reality.

Service is the key element in a service ecosystem, and its stability can be predicted from running histories, such as invocation and failure frequency etc. And the stability can affect the stability of other related service-based processes and SKBs. Meanwhile, SKBs can also affect the stability of corresponding service-based processes. For example, a SKB with lower stability can be considered that services in the base may be lower stable; a service-based process including services with lower stability has a lower stability.

5.2 Basic Concepts

Service is a basic business operator unit, and it may be a software service from the Internet, or a human service from some business department of an enterprise. Generally, a service comprises multiple operators, where each operator can implement a special function, and also is called API. Here, we consider an operator with a special function as a service for convenience.

Definition 1: *Service*. A service can be expressed as a tuple $\langle FunSem, InvSyn, StabInd \rangle$, where,

- *FunSem* describes functional semantics including inputs (*I*), outputs (*O*), preconditions (*P*), and effects (*E*)[30];
- *InvSyn* describes invocation details at the syntactic level, including input/output parameters, access address, request methods, etc.;
- *StabInd* describes indicators related to its stability, including total invocation frequency (*IF*) and failure frequency (*FF*).

A SKB may include multiple services, and may use multiple domain ontologies to annotate semantics of services. In practice, a service ecosystem generally includes multiple SKBs.

Definition 2: *Service Knowledge Base*. A service knowledge base(SKB) is modeled as a tuple $\langle TP, Ds, Facts, Actions, Stab \rangle$, where, *TP*, *Facts,* and *Actions* are the same as in D3LSRAF (Section); *Ds* is a set of ontologies that are used to annotate services in *TP*; *Stab* represents stability coefficient of the knowledge base, and it relates to the stability of services in*TP*.

A service-based process includes multiple services from different SKBs.

Definition 3: *Service-based Process*. A service-based process is expressed as a tuple $\langle Sset, SKBset, SKBMap, Seq, Stab\rangle$, where,

- *Sset* is a set of services that are used in the process;
- *SKBset* is a set of SKBs that include services in *Sset*;
- *SKBMap* is a one-to-many mapping from *SKBset* to *Sset*;
- *Seq* describes cooperative relationships among services, and is expressed as a service instance set sequence $\langle\{a_1, \cdots, a_n\}, \cdots, \{b_1, \cdots, b_m\}\rangle$, and each set in *Seq* is called an execution step[31] . Service instances in an execution step are run in parallel, and execution steps in the sequence are sequentially run;
- *Stab* is the stability coefficient of the process, and it relates to the services in *Seq*.

Definition 4: *Local process*. A local process is a service-based process where its component *SKBset* only includes one SKB.

Definition 5: *Global process*. A global process is a service-based process where its component *SKBset* includes all SKBs in a service ecosystem.

5.3 Stability Evaluation Model

In a service ecosystem, service change is a key factor in changing the stability of related elements, such as removing an old service, or updating function or interface invocation syntax of an old service etc. These changes can cause permanent invocation failures of related services. Meanwhile, a dynamic network environment also can make temporary invocation failures, such as temporary connect failure, or running context disturbed. In subsequent invocation, a temporary failure may occur few times, and a permanent failure always occurs. Therefore, the failure rate is a key factor distinguishing the two types of failures and reflecting the stability of service.

Definition 6: *Stability of Service*. Let *s* be a service, stability of *s* is expressed as a function *stable(s)*:

$$stable(s) = \begin{cases} 1 & s.IF = 0 \\ \frac{s.IF - s.FF}{s.IF} & s.IF \neq 0 \end{cases} \qquad (1)$$

For a service *s*, it would go through three statuses during its life cycle: *new*, *active*, and *perishing*. A *new* service means that it never is invoked by any consumer; An *active* service means that it ever is invoked by some consumers successfully, and it expects to be invoked in the future; A *perishing* service means that its stability is poor and has been abandoned. We use a function *status(s*) to represent the status of service *s*, and its definition is shown in the following:

$$status(s) = \begin{cases} new & stable(s) = 1 \wedge s.IF = 0 \\ active & stable(s) > P \wedge s.IF \neq 0 \\ perishing & stable(s) \leq P \end{cases} \quad (2)$$

Here, *P* is a threshold value to distinguish active and perishing services.

In practice, except for running histories, a provider's reputation can also affect the future stability of services provided. We can consider that services from a provider with a better reputation would be with higher stability. Here, we use the stability of SKB to represent the reputation of the related provider.

Definition 7: *Stability of SKB.* Let *skb* be a SKB, its stability is expressed as the following:

$$skb.Stab = \begin{cases} 0 & PN = TN \\ \frac{NN}{TN-PN} & AN = 0 \\ \frac{\sum_{s \in skb.TP \wedge s\ is\ active}(s.IF - s.FF)}{\sum_{s \in skb.TP \wedge s\ is\ active} s.IF} \times \frac{AN}{TN-PN} + \frac{NN}{TN-PN} & AN > 0 \end{cases} \quad (3)$$

Here, *AN* is the number of active services, *NN* is the number of new services, *PN* is the number of perishing services, *TN* is the total number of services.

It can be seen that the stability of a SKB is related to the status of included services and invocation histories, and reflects the reputation of a provider from a global view. A provider will have a higher reputation when the total failure rate of all services is lower. If all services from a provider are perishing services, the reputation of the provider is worst. Especially if two services from different SKBs have similar stability, the one from a SKB with higher stability will win in the competition.

A service-based process is constructed based on mutual cooperation among services from different SKBs. In this paper, we assume that all processes don't have redundant services. Therefore, the stability of a cooperative relationship among multiple services can become lower as the stability of any of them lower. Moreover, to make the stability of a service-based process more reasonable, the stability of SKB should also be considered.

Definition 8: *Stability of service-based process*. Let *sp* be a service-based process, its stability is expressed as the following:

$$sp.Stab = \prod_{skb \in sp.SKBSet} [\,(\prod_{s \in sp.SKBMap(skb)} s\ table(s)) \times skb.Stab] \quad (4)$$

6 Competition and Cooperation Considering Stability and Distributed Knowledge

Competition and cooperation among services are key activities to promote the evolution of a service ecosystem. Automatic service composition can make the activities occur automatically. Here, stability is competition basis among services, and can affect cooperation result among multiple services. Meanwhile, bridge rules can be used to eliminate semantic conflicts among knowledge bases to improve interoperability.

In SDKEM, we design two automatic service composition algorithms to promote competition and cooperation, respectively called local planning with stability (*LPlanWithStab*) and global planning with stability (*GPlanWithStab*). Both of them can respond to a service request with distributed knowledge (short for *DK-SR*). Its semantics over components may be from different SKBs. Compared with *DK-SR*, a service request, which semantics are from only one SKB, is called *Single-SR*. Based on all SKBs in a service ecosystem, *LPlanWithStab* tries to search the most stable local process from all knowledge bases, and *GPlanWithStab* tries to search the most stable global process.

6.1 LPlanWithStab

LPlanWithStab uses local reasoning with stability in each SKB to search for the most stable local process. And the reasoning is implemented through introducing stability factors in *LocalD3LPlanning* in D3LSRAF, shown in Fig. 4. It firstly converts a *DK-SR* into a *Single-SR* for each knowledge base according to various bridge rules. Then, it concurrently searches multiple most stable local processes from all bases through local reasoning with stability. Lastly, it picks the most stable one from these local processes.

The local reasoning firstly creates a relaxed planning graph through forwarding search according to components *In* and *Init* in the *Single-SR*. Then, it searches a service-based process from the graph through backward search according to component *Out* and *Goal* in the *Single-SR*. During the backward search, stability heuristic strategy is adopted to ensure the searching goal for each action layer is most stable, and, for each fact in a goal, the most stable action will be picked. The stability of a goal is evaluated by the formula in (5):

$$stable(g) = \prod_{f \in g} max_{a \in actions_f} stable(a) \quad (5)$$

, where $actions_f$ represents the action set that produces the fact f, and if s is the service related to action a, then $stable(a) = stable(s)$. Obviously, a local process obtained by the reasoning is locally optimal.

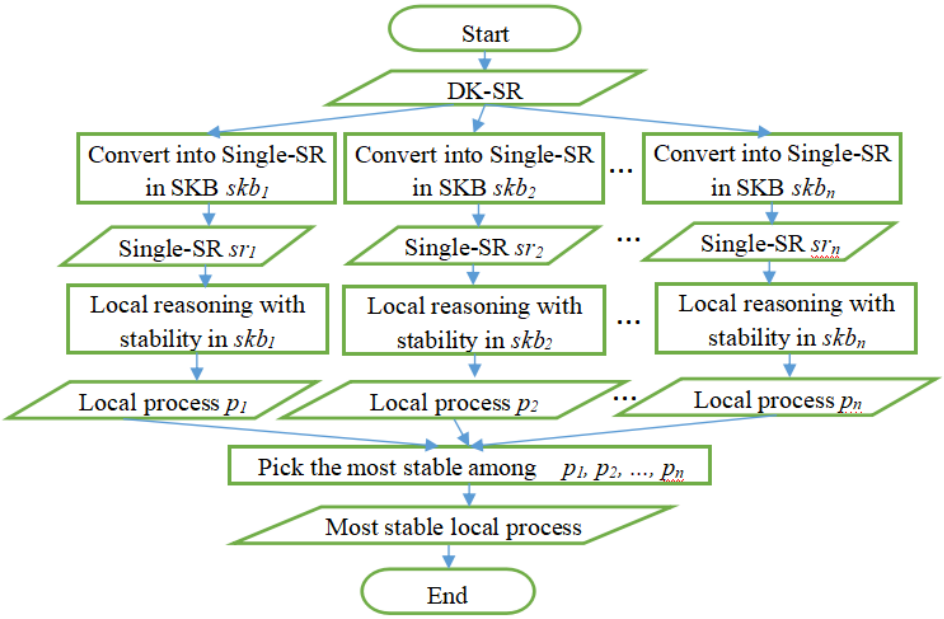


Figure 4. The implementation procedure of LPlanWithStab.

6.2 GPlanWithStab

GPlanWithStab utilizes bridge rules to make multiple SKBs into a whole distributed knowledge base, and executes global reasoning to obtain the most stable global process(Fig. 5).

The reasoning is implemented through introducing bridge rules and stability factors in *GlobalD3LPlanning* in D3LSRAF. Global reasoning firstly creates a relaxed planning graph where services generate actions in action layers with new and active status, and semantics of facts in state layers are from multiple knowledge bases. Then, it determines whether given *DK-SR* is satisfied by the last state layer or not. Here, bridge rules are used to eliminate semantics conflicts among different knowledge bases. If the request is satisfied, and then some concrete goals, where all facts are instantiated, can be generated according to the last state layer. Thus, it picks the most stable goal from all ones, and then searches the locally optimal solution using a stability Heuristic strategy.

For each goal in the state layer, its stability is evaluated by the formula in (6):

$$stable(g) = \prod_{f \in g} max_{a \in actions_f} stable(a) \times skb_a.Stab \quad (6)$$

, where skb_a represents a SKB including service related to the action a. Facts in a goal and actions generating these facts may be from different knowledge bases. Generally, a more stable provider can provide more stable services. And this tendency is reflected by the formula through introducing stability of the corresponding knowledge base. Thus, the global reasoning will pick those more stable actions from more stable knowledge bases for a given goal.

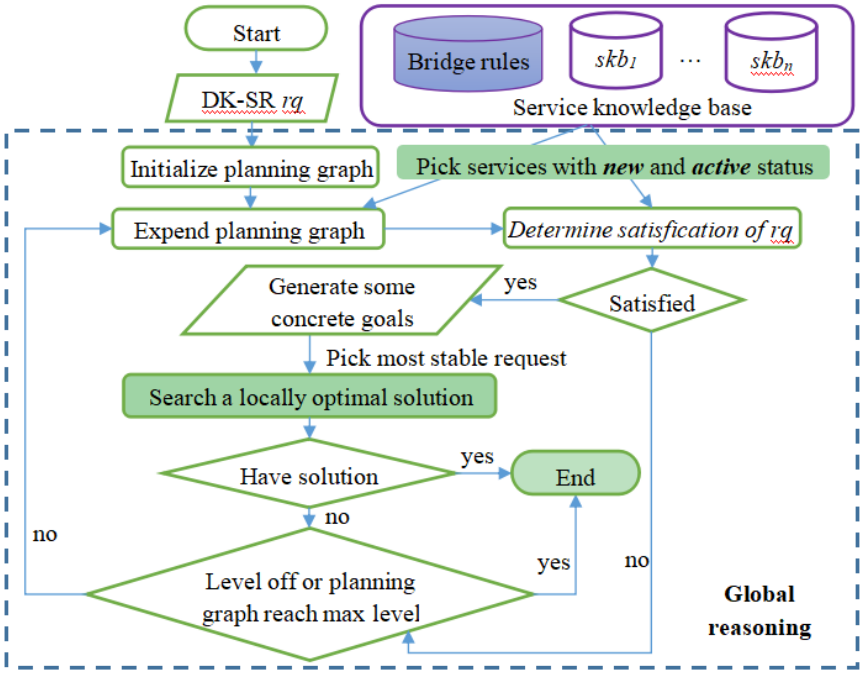


Figure 5. The implementation procedure of GPlanWithStab.

7 Evolution Mechanisms in SDKEM

Evolution mechanisms in SDKEM can implement the evolution of various elements, including bridge rules, service-based process, service, and SKB. Its implementation procedure is shown in Fig. 6.

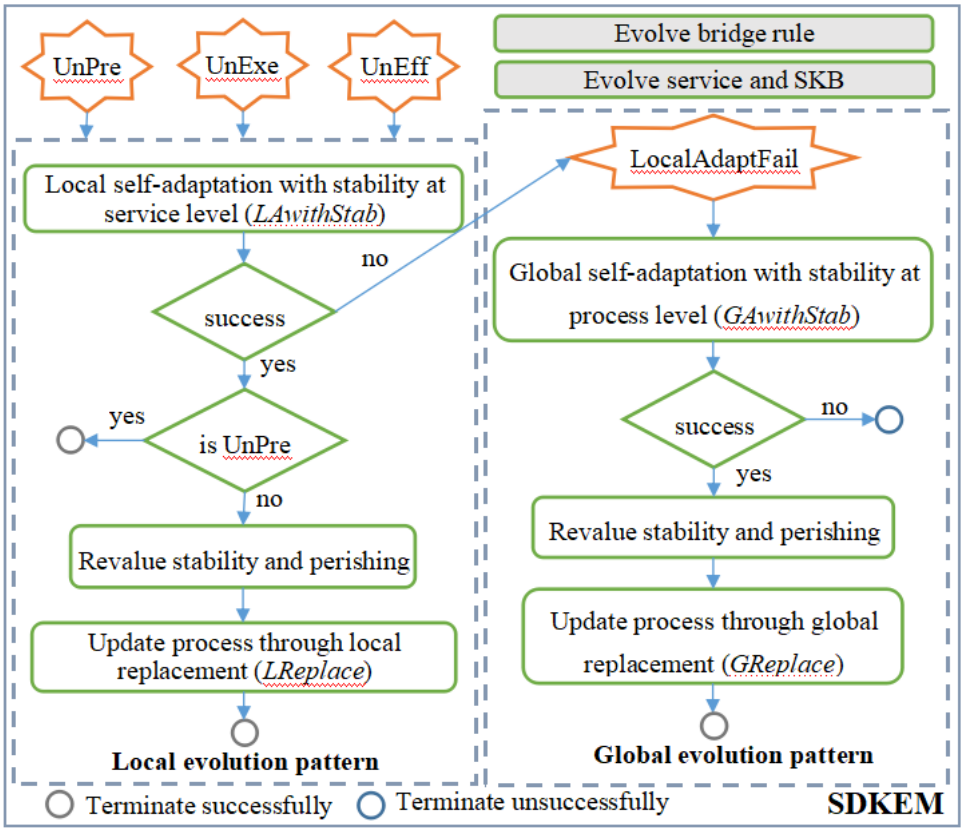


Figure 6. Evolution mechanisms of SDKEM.

7.1 Evolving Bridge Rule

Under a dynamic environment, ontologies tend to change because of changes in available web services. To promote cooperation among all services, it is necessary to evolve bridge rules according to these changes. In SDKEM, a developer triggered the evolution when he introduces new vocabularies to annotate web services. Before evolution, the developer needs to set special bridge rules between new and old vocabularies manually, and specify synonyms of new vocabularies. Then, he invokes a bridge rule evolution algorithm (Fig. 7) to generate new implied bridge rules. After evolution, new bridge rules are added into **Bridge Rule DB**, and will work in the subsequent service-based process evolution.

Here, function *inverse(rtype)* (line 2) will return different rule type according to given *rtype*. If *rtype* is *equalc* or *equalr*, it will return *rtype*; if *rtype* starts with *intoc* or *intor*, it will return *ontoc* or *ontor*; if *rtype* starts with *ontoc* or *ontor*, it will return *intoc* or *intor*. Funtion *synequal(synvoctb)* (line 4) will return bridge rules of equal type according synonym vocabulary table *synvoctb*.

Algorithm 1 EvolveBRLib
Inputs: *brlib*: all bridge rules in current rule library; *newmbrs*: new bridge rules added manually; synvoctb: synonym vocabulary table **Outputs**: updated *brlib*
1. **FOR** each <v_1, *rtype*, v_2> in *newmbrs* **DO**

```
2.  IF rtype is not equalr_F THEN newmbrs←newmbrs∪{ <v2, inverse(rtype), v1>}
3. END FOR
4. brlib=brlib∪synequal(synvoctb)
5. FOR each <v1, rtype, v2> in newmbrs DO
6.  FOR each <v3, rtype', v4> in brlib DO
7.   IF rtype==rtype' and v1==v4 THEN brlib =brlib∪{ <v3, rtype, v2>}
8.   IF rtype==rtype' and v2==v3 THEN brlib =brlib∪{ <v1, rtype, v4>}
9.   IF rtype start with 'equal' and rtype' start with 'into' or 'onto' THEN
10.    IF v1==v3 THEN brlib =brlib∪{ <v2, rtype', v4>}
11.    IF v1==v4 THEN brlib =brlib∪{ <v3, rtype', v2>}
12.    IF v2==v3 THEN brlib =brlib∪{ <v1, rtype', v4>}
13.    IF v2==v4 THEN brlib =brlib∪{ <v3, rtype', v1>}
14.   END IF
15.   IF rtype' start with 'equal' and rtype start with 'into' or 'onto' THEN
16.    IF v1==v3 THEN brlib =brlib∪{ <v4, rtype, v2>}
17.    IF v1==v4 THEN brlib =brlib∪{ <v3, rtype, v2>}
18.    IF v2==v3 THEN brlib =brlib∪{ <v1, rtype, v4>}
19.    IF v2==v4 THEN brlib =brlib∪{ <v1, rtype, v3>}
20.   END IF
21.  END FOR
22. END FOR
23. RETURN brlib∪newmbrs
```

Figure 7. Algorithm of bridge rule evolution.

7.2 Evolving Service-based Process

SDKEM can automatically trigger the evolution of a service-based process through monitoring its running self-adaptation exceptions and automatically evolve the process when online repairing it. During repair and evolution, services from all SKBs compete and cooperate with each other to raise response success rate and stability of updated service-based processes.

Two evolution patterns are designed: local and global, and they may frequently occur during the running of a service-based process. Local evolution is triggered by one of three failures at service level: *UnPre*, *UnExe*, and *UnEff*, and can evolve a process by means of local self-adaptation at service level. Global evolution is triggered by *LocalAdaptFail*, and can achieve the evolution through global self-adaptation at process level.

7.2.1 Local Evolution Pattern

When one of the local self-adaptation failures occurs over a service, a local evolution automatically starts. It firstly carries out local self-adaptation with stability at the service level (*LAwithStab*) for current failure, and then evolves the whole service ecosystem. To raise the adaptation success rate, in *LAwithStab*, *LPlanWithStab* is adopted to obtain a high stable adaptation process. During *LAwithStab*, if there is no adaptation process or current adaptation process fails to run, and then a *LocalAdaptFail* is

thrown and a global evolution pattern will be adopted; otherwise, *LAwithStab* succeeds in running, and further evolution continues.

Specially, we assume that the occurrence of *UnPre* is temporary and corresponding service is always normal. Therefore, when the failure is *UnPre*, the local evolution would terminate after local adaptation succeeds to run. However, when the failure is *UnExe* or *UnEff*, the further local evolution continues. It firstly revalues stability of the faulted service and corresponding SKB according to Definition 6 & 7, and then labels those, which stability is less than a given threshold value as perishing. Subsequently, it updates the original process through local replacement (*LReplace*).

Furthermore, after *LAwithStab* runs successfully for a service s in process p, a local adaptation process *ap* has been generated and runs successfully. However, a successful run of *ap* might be achieved by self-adaptation, because *ap* also could encounter some failures under a dynamical running environment. Therefore, multiple adaptation processes might be produced for a successful run of *LAwithStab*. To guarantee the stability of p, *LReplace* is adopted to attempt the replacement of s with normal services in all adaptation processes by means of *GPlanWithStab*. Local evolution automatically runs in the corresponding service invocation unit, and it can't affect the running of other services in the current original process.

7.2.2 Global Evolution Pattern

When *LocalAdaptFail* occurs for a process, a global evolution automatically starts. It firstly carries out the global self-adaptation with stability at the process level (*GAwithStab*) to repair the original process. If the self-adaptation fails, the global evolution terminates, and then the original process terminates with failure. Otherwise, stability of the faulted service and related SKB are revalued according to current running histories. Just as in local evolution, for a service, if its stability is less than a given threshold value, they will be labeled as perishing. After revaluation, the original process is updated through global replacement (*GReplace*).

GAwithStab adopts *GPlanWithStab* to search a more stable global adaptation process for the goal of the original process. The successful run of *GAwithStab* means at least one global adaptation process is generated and runs successfully. When the adaptation process encounters some failures at runtime, more than one adaptation process would be produced. Furthermore, when *GAwithStab* succeeds in running, the goal of the original process p is achieved, and p will terminate with success. Therefore, those services, that run successfully in all adaptation processes and p, can be enough to plan a new

process for achieving the same function request with p. Based on these adaptation processes, *GReplace* is adopted to update the original process.

Here, we assume that an original process has no redundant services, that is, each service is necessary for the execution of the process. Therefore, in *LReplace* and *GReplace*, we always use adaptation processes to replace the faulted services by means of *GPlanWithStab*.

Specially, the evolution for an adaptation process is meaningless because they are produced during the run of an original process and are temporary. Therefore, when an evolution is triggered by a failure on an adaptation process, it will only invoke other operations excepting *LReplace* or *GReplace* to repair the failure and to perish corresponding faulted service, but not update the adaptation process.

7.3 Evolving Service and SKB

In a service ecosystem, service running histories are recorded during service-based processes run. Therefore, according to definitions 6&7, the stabilities of services and SKBs can be computed. When stabilities of services are lower than a given threshold, these services are automatically annotated as perishing services during evolution. Then, extra evolution operation for perishing services and SKBs is carried out. Algorithm 2 in Fig. 8 shows the implementation procedure.

Algorithm 2 Evolving perishing service and service knowledge base
Inputs: *s*: a perishing service, *skb*: a service knowledge base containing *s* **Outputs**: an updated *skb*
01. *newInvSyn* ← new invocation information of *s* from its official website 02. *oldInvSyn* ← *s.InvSyn*, that is, old invocation information of *s*. 03. **IF** *newInvSyn* doesn't exist or *newInvSyn==oldInvSyn* **THEN** remove *s* from *skb.TP* 04. **ELSE** 05. *s.InvSyn* ← *newInvSyn* 06. *s.FunSem* ← new function semantics manually annotated by developers 07. *s.IF* ← 0 08. **END IF** 09. **IF** *skb.TP==Ø* **THEN** remove *skb* 10. **RETURN** *skb*

Figure 8. The implementation procedure of evolving perishing service and SKB.

In practice, services are perished because of various factors from their providers. For example, they are removed or their invocation information(request/response parameters, access URL, etc.) are updated. To efficiently evolve these services, different evolution strategies should be adapted according to different factors. Those removed services also are removed from its SKB (line 03); those updated services are re-crawled and semantically annotated, and then are set as new services (lines $05-07$). Furthermore, SKBs without any services are meaningless for the current service ecosystem. Therefore,

they will be removed during the evolution (line 09). The evolution operation can be manually carried out by developers of the current service ecosystem.

8 Experiment Evaluation

In this section, we design a series of experiments to evaluate the effectiveness of SDKEM, the influence on business process response efficiency, and the role of stability evaluation. Also, we present a prototype system for SDKEM.

8.1 Test Case Illustration

Currently, there is no standard test case of service ecosystem evolution research. We create a test case through crawling APIs from open API platforms[4], and designing real web APIs by ourselves.

Request and response of web APIs from open platforms are organized loosely. For example, a service for searching train information between two cities returns a list including trains, and each element for a train has more than 20 parameters (*train No.*, *start* and *end* station,*distance* etc.); a service for flight order creation receives more than 10 request parameters including passenger (*name*, *type*, *ID No.*, *mobile* etc.), flight information(*take off port*, *landing port*, *flight No.* etc.), *order ID* and *total amount*. These loose structures make interaction among services difficult. To interact easily, for a service, we use some business objects to wrap those related items, and consider those objects as parameters that are the basis of semantic annotation. Meanwhile, we create an ontology file for each provider to annotate IOPE of services.

Furthermore, most of the services on open platforms are information-providing. However, in an organization, to achieve a complex business goal, a lot of world-altering services are needed. Therefore, according to actual application scenarios, we also design some services to implement specific functions, such as Order food, Payment with different ways, Handle vehicle violations, etc. Also, human services are required for a whole business process. Thus, we design some essential human services, such as Take various vehicles, Send express, etc.

Ultimately, our test case comprises 18 SKBs, 110 services, 13 domain ontology files, and its detailed information is shown in Table 1. Here, only one SKB has no service, because its ontology is $D1$. $D1$ includes abstract concepts and predicates to express abstract service requests.

[4] https://www.jisuapi.com https://www.juhe.cn

Table 1. Details of our test case

Ontology	Domain	SKB Num	Service Num
D1	Common	1	0
D2	City traffic	2	17
D3	Train	2	7
D4	Flight	2	8
D5	Restaurant	1	12
D6	Tourist spot	1	7
D7	Express	1	8
D8	Communication	1	14
D9	Hotel	2	7
D10	Inn	2	7
D11	Payment	1	13
D12	Weather	1	3
D13	Vehicle Violation	1	7

To eliminate semantic conflicts among these ontologies, 668 bridge rules are generated including 476 rules among concepts(intoc 32, ontoc 32,equalc 412) and 192 rules among predicates(intor 49, ontor 49, equalr 94). In all bridge rules, 46 rules are created manually.

Meanwhile, we create 30 different service-based processes by means of *GPlanWithStab*, and are called original processes in the following. These processes can basically cover services from various domains in Table 1. We also design 5 running situations with failures: $rs_1, \cdots, rs_4, rs_5$. In each running situation, different numbers of failures(*UnPre*, *UnExe*, or *UnEff*) at service level and different service running histories(invocation frequency and failure frequency) are set in advance. In $rs_n (1 \leq n \leq 5)$, the number of failures is $n * 10$. Specially, we use rs_0 to represent a running situation without failures. That is, In rs_0, all original processes can success to run and can't encounter any runtime failure.

8.2 Experiment Environment

We simulate the implementation of all services in the test case under JavaEE platform. Here, all services are RESTful, and are deployed on the application server *Tomcat8.0*. Original processes and new processes during the following experiments are described by standard *BPMN2.0*. In them, each service is invoked by custom *serviceTask* in *BPMN2.0*. The execution environment of processes is workflow engine *Activiti 5.22*. In addition, service DB, bridge rule DB and workflow engine database use *MySQL5.1*. And the prototype system is installed on *ThinkPad X1* (*1.80GHz,1.99GHz, 16GRAM,*

Win10). In the following experiments, if not otherwise specified, all 642 bridge rules are put in *Bridge DB*, and can't be changed during the experiments.

8.3 Effectiveness Evaluation

8.3.1 General Effectiveness

To evaluate the general effectiveness of SDKEM, we do the first experiment. In this experiment, for each running situation rs, all original processes run two times. Firstly, we make all these processes run once in rs, and record the number of original processes succeeding to run (*SuccessNum*) and the number of original processes without failure services (*NoFailNumBefore*) during the run. After the run, the situation rs is updated to rs', because the service running histories are changed. Then, in rs', we make these processes run once again and record the number of original processes without failure services (*NoFailNumAfter*) during this run. Lastly, we compare these numbers, shown in Fig. 9.

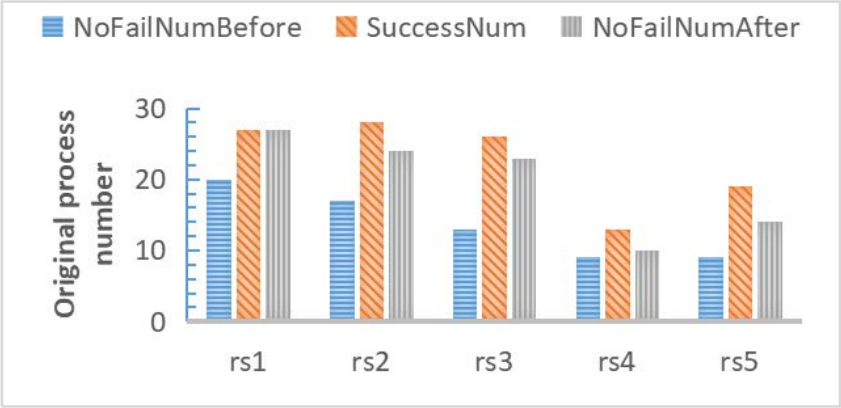


Figure 9. Comparison among *NoFailNumBefore*, *SuccessNum* and *NoFailNumAfter* in each running situation.

It can be seen that, in all running situations, SDKEM improves the response success rate of original processes and also decreases their failure frequency. Here is an example. In $rs2$, *NoFailNumBefore* is 17. It means only 17 original processes can succeed in running if the evolution mechanism don't work. However, under the evolution mechanism, 28 original processes run successfully, and 24 in the 28 processes have no failure service. That is, all failure services in 7 original processes are replaced after the first run. These replacements obviously decrease the runtime failure frequency of the second run. After each run, we also record the numbers of various failures caught, shown in Table 2. From the table, we found that, in each situation, the total number of failures caught (the last row) is less in the second run than in the first run. The maximum reduction is 57.1% in rs_1, and the minimum reduction is 6.4% in rs_4.

Table 2. Details of failures caught in each running situation

Failure	rs_1	rs_1'	rs_2	rs_2'	rs_3	rs_3'	rs_4	rs_4'	rs_5	rs_5'
UnPre	0	0	0	0	0	0	1	1	2	0
UnExe	1	1	13	5	11	3	14	13	21	10
UnEff	9	2	1	1	12	7	11	10	16	11
LocalAdaptFail	4	3	3	2	13	6	21	20	12	14
Sum	14	6	17	8	36	16	47	44	51	35

8.3.2 Effectiveness of Evolution Mechanism

The evolution mechanism plays a key role in improving the response success rate and decreasing runtime failures. To prove this, we do the second experiment. Firstly, we design 5 different evolution situations: es_1 (no evolution), es_2 (only local adaptation), es_3 (only local and global adaptation), es_4(only local evolution), es_5(local and global evolution). Secondly, in each group of $es_i(1 \leq i \leq 5)$ and $rs_j(1 \leq j \leq 5)$, we make all original processes run once. After the run, we record the number of original processes succeeding to run (*Suc*) and the number of original processes without failure services (*NoFail*), shown in Table 3. It is noticed that adaptation operations improve the response success rate of these processes, and that update operations make more and more processes not include any failure service (in es_4 and es_5).

Table 3. Influence of different evolution situations on processes

ES	rs_1		rs_2		rs_3		rs_4		rs_5	
	Suc	NoFail	Suc	NoFail	Suc	NoFail	Suc	NoFail	Suc	NoFail
es_1	20	20	17	17	13	13	9	9	9	9
es_2	26	20	27	17	19	13	12	9	19	9
es_3	27	20	28	17	26	13	13	9	19	9
es_4	26	26	27	24	19	16	12	9	19	14
es_5	27	27	28	24	26	23	13	10	19	14

In rs_3, the advantage of this evolution mechanism is clearly presented. In the first 3 evolution situations, *NoFail* always is 13, because there is no update operation. However, as local and global adaptation measures are added, *Suc* gradually increases. *Suc* in es_2 is 6 more than in es_1, and *Suc* in es_3 is 7 more than in es_2. In es_4, local evolution measure is set, and it includes local adaptation and local update operation(*LReplace*). Under the local evolution, 4 *LReplace*s are carried out, and 3 processes are updated. Based on es_4, in es_5, global evolution measure is added, and it includes global

adaptation and global update operation(*GReplace*). Under the local and global evolution, 4 *LReplace*s and 7 *GReplace*s are invoked, and finally, 10 original processes are updated.

8.3.3 Effectiveness of Bridge Rule Evolution

To reflect the role of bridge rule evolution, we do the third experiment. Firstly, $five$ bridge rule evolution situations are created, shown in Table 4. In each situation, three respects are set: the number of existed bridge rules in *Bridge DB* (*BRLib*), ontologies added during the running (*NewOnto*), and the number of manual bridge rules added during the running (*NewMRule*). Then, we invoke algorithm 1 to generate all rules, where we assume vocabularies with the same local name are synonyms. Next, in each group of $brs_i(1 \le i \le 5)$ and $rs_j(1 \le j \le 5)$, we make all original processes run once. After the run, We record the number of newly generated bridge rules (*NewBR*), and the response success rate of these processes, shown in Table 5.

Table 4. Bridge rule evolution situations

BRS	**BRLib**	**NewOnto**	**NewMRule**
brs_1	0	0	0
brs_2	0	$D1, D2$	46
brs_3	0	$D1, D2, D3, D4$	46
brs_4	0	$D1, D2, D3, D4, D9, D10$	46
brs_5	416	0	46

It is noticed that, during the run, those new ontologies and manual bridge rules are perceived, and new bridge rules are generated. Meanwhile, as the number of available bridge rules increases, the response success rate is improved. Except in rs_4, this improvement is obvious in other running situations. For example, in rs_3, the success rate continuously increases from 46.7% in brs_1 to 86.7% in brs_5. In brs_5, after 252 bridge rules are generated, 668 bridge rules are available, the response success rate of original processes in rs_4 also is improved.

Table 5. Bridge rule evolution in various running situations

BRS	**NewBR**	rs_1	rs_2	rs_3	rs_4	rs_5
brs_1	0	66.7%	56.7%	46.7%	30.0%	30.0%
brs_2	136	66.7%	56.7%	50.0%	30.0%	36.6%
brs_3	166	73.3%	73.3%	60.0%	30.0%	46.7%
brs_4	258	86.7%	83.3%	70.0%	30.0%	46.7%
brs_5	252	**90.0**%	**93.3**%	**86.7**%	**43.3**%	**63.3**%

8.4 Response Efficiency Evaluation

To evaluate the influence of SDKEM on the response efficiency of processes, we do the fourth experiment. Firstly, we make all original processes run in rs_0, and record the running time of each original process. For a process, the time is called its *NormalTime*. Then, we redo the two runs in the first experiment and record the running time of each original process in each run. For a process, its running time in the first run is called its *OnEvTime*, and the time in the second run is called its *AfterEvTime*. Lastly, we pick all original processes that are updated and succeed to run in the first run. Meanwhile, we compute two increase rates in the running time of these processes. They reflect the impact of evolution operation on the running time of processes, and respectively are called *OnEvInc* and *AfterEvInc*. For a process, its *OnEvInc* is computed by $(OnEvTime - NormalTime)/NormalTime$, and its *AfterEvInc* is computed by $(AfterEvTime - NormalTime)/NormalTime$. Table 6 shows the statistical data onto the two increase rates for all picked processes in each running situation, including the minimal and maximal *OnEvInc*(*OnEvIncMin* and *OnEvIncMax*), the minimal and maximal *AfterEvInc*(*AfterEvIncMin* and *AfterEvIncMax*), and the number of picked processes.

Table 6. Influence of SDKEM on running time of original processes

Increase	rs_1	rs_2	rs_3	rs_4	rs_5
OnEvIncMin	48.7%	51.8%	34.6%	157.3%	38%
OnEvIncMax	557.4%	121.1%	11387%	288.6%	254.7%
AfterEvIncMin	−0.3%	−1.9%	−20.8%	3.1%	−30.2%
AfterEvIncMax	15.6%	108%	119%	51.6%	52.7%
UpdatedNum	7	$7(1failure)$	10	2	6

It is noticed that SDKEM greatly improves the response efficiency of processes with failure services. The running time of a process increases greatly when an evolution operation is invoked. However, after this evolution, its running time is closer to its *NormalTime* than its *OnEvTime*. Here is an example. In rs_3, there is a process which *OnEvInc* is 11387%. This means, to make the process run successfully, the evolution operations spend a lot of time repairing various runtime failures in the process. After these evolution operations are carried out, all failure services are replaced with those successful services. We found the *AfterEvInc* of this process only is −20.8%, this is also the minimal *AfterEvInc* in rs_3. That is, after evolution, the running time of this process is less than its *NormalTime*.

Specially, in rs_2, there is a special picked process. Although it was updated and succeeded in running, it still encountered a failure in the second run. The failure only was repaired and did not trigger corresponding update operation in the first run, because the stability of its candidate service is lower.

This directly increases the *AfterEvTime* of this process, and its *AfterEvInc* is 108%. However, the highest *AfterEvInc* of other 6 picked processes is only 5.6%. This means that the running time of most of the updated processes is close to the normal time.

8.5 Stability Evaluation

To illustrate the role of the stability evaluation model, we do the fifth experiment. In each running situation rs, we firstly make all original processes run once and use rs' to represent the running situation after the run. Then, we pick all original processes that are updated, and compute their stabilities. For a picked process p, there are two versions: the old version before updated, and the new version after updated. Based on the service running histories in rs', we respectively compute the stabilities of the two version of p, and are expressed as *oldstab* and *newstab*. Lastly, we compute the stability increase value of p by $newstab - oldstab$. Table 7 shows the statistical data onto stability increase values of all picked processes in each running situation, including the minimal and maximal increase values (*SMin* and *SMax*). The last row shows the number of picked processes(*UpdatedNum*) in each running situations.

Table 7. Stability changes of original processes

Increase	rs_1	rs_2	rs_3	rs_4	rs_5
SMin/SMax	$0.31/0.54$	$0.10/0.32$	$0.11/0.61$	$0.14/0.39$	$0/0.47$
UpdatedNum	7	8	10	2	9

It can be seen that, for most processes, the stabilities of their new versions are higher than their old versions. This can guarantee the new version of a process has fewer runtime failures and higher response efficiency. In $rs3$, for all picked processes, the stabilities of their new versions increase at least 0.11, and at most 0.61. For the process with the most *SMax*, the running times of its old version and new version respectively are $25862ms$ and $1721ms$. Obviously, the response efficiency of the process is greatly improved.

Especially, in rs_5,*oldstab*s of 3 processes are equal to the *newstab*s. At the moment when they are updated, their *newstab*s are higher than the *oldstab*s. As the subsequent processes run, current service running histories are changed. This makes the stabilities of services in the new versions become lower. Therefore, it is reasonable that the stability of one process is not changed. Also, two perishing services appear in rs_5, because their stabilities are lower than the given threshold value 0.2. These phenomenons also reflect that the stabilities of services are constantly changing.

In addition, we also compute the stability of each SKB in various running situations. Due to space limitations, Table 8 only shows the stabilities of 9 knowledge bases. In the remaining 9 bases, one has no service, and its stability always is 1, and the stabilities of others are 1 in no less than 2 running situations. It is noticed that, in most cases, the stability of a knowledge base is different in a different running situation. For example, the stabilities of knowledge base $skb1$ in 5 running situations are all different. The highest is 0.94 in rs_4, and the lowest is 0.88 in rs_1. This is because the running histories of services in $skb1$ are different in these running situations. Therefore, the stabilities of services also affect the stabilities of related SKBs.

Table 8. Comparison of stabilities of SKBs

RS	**skb1**	**skb2**	**skb3**	**skb4**	**skb5**	**skb6**	**skb7**	**skb8**	**skb9**
rs_1	0.875	1.0	0.864	1.0	1.0	1.0	0.962	0.989	0.938
rs_2	0.929	0.924	0.912	0.867	0.948	0.952	0.912	0.986	0.962
rs_3	0.935	0.88	0.933	0.882	0.905	0.917	0.870	0.963	0.976
rs_4	0.942	0.85	0.942	0.889	0.868	0.870	0.858	0.956	0.894
rs_5	0.914	0.963	0.742	0.778	0.896	0.844	0.844	0.891	0.882

8.6 Prototype System

A prototype system for SDKEM can assist developers in managing their Web services, SKBs, and knowledge(ontologies and bridge rules); automatically generate a business process for a given service request, and provide a running environment for automatic evolution of the business process. Fig 10 shows the user interface for business process generation and running.

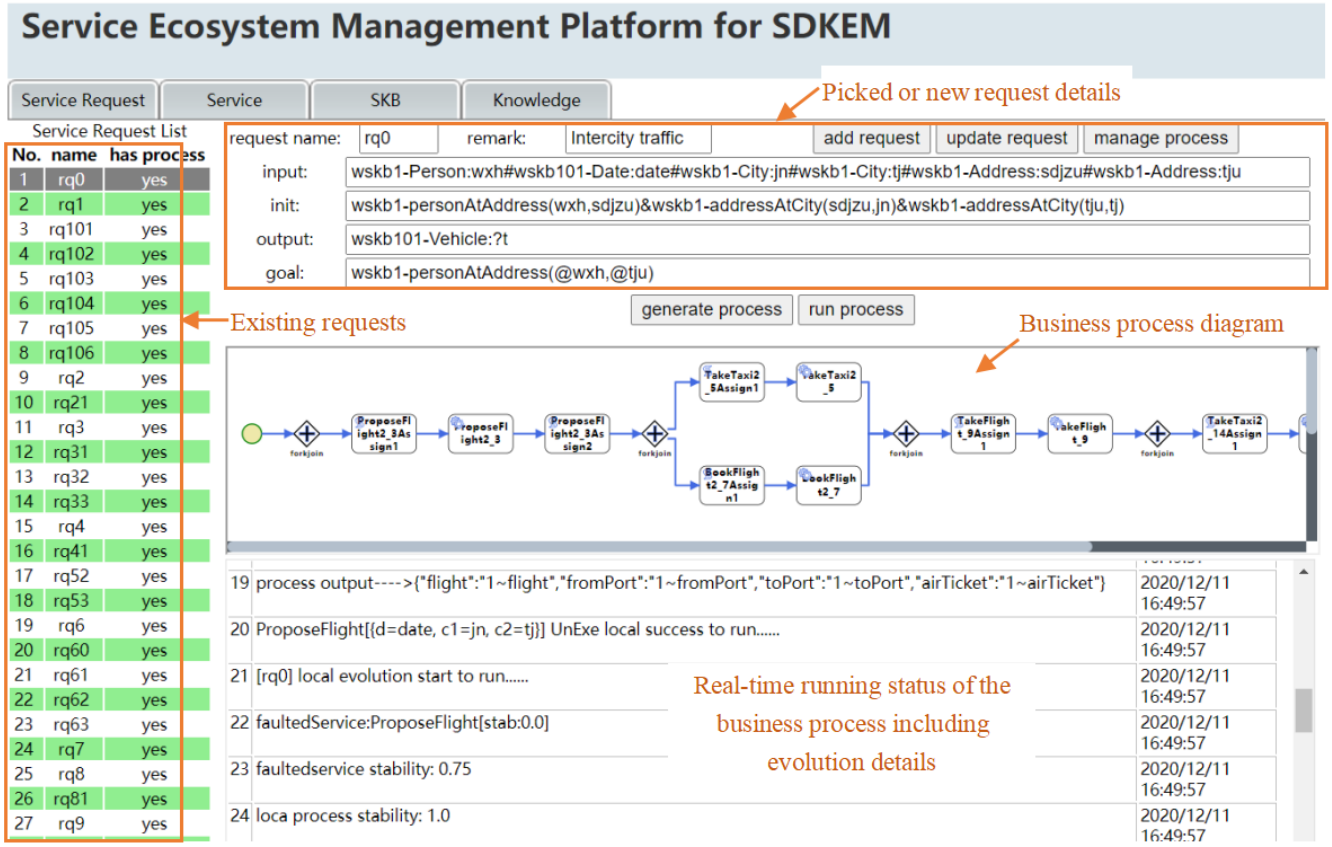


Figure 10. User interface for business process generation and running

In this interface, a developer can define his own service request, generate a corresponding business process, running the process, and look over running details of this process. And evolution operations will be automatically invoked when some running failures occur.

In general, SDKEM can effectively and efficiently evolve various elements in a service ecosystem during a service-based process running, including services, current service-based process, and SKBs. Stabilities of these elements play an core role in "survival of the fittest" of them. Compared with previous approaches, SDKEM has outstanding advantages in real-time, evolved elements, and supporting distributed knowledge, shown in Table 9. Specially, SDKEM is an extended version of DKEM, and more implementation details are presented in this paper.

Table 9. Comparison with existing approaches

No.	Approach	Real-time	Evolved elements	Distributed knowledge
1	Interface Document based [14-16]	Yes	Service	No
2	Running Log based [17]	No	Service	No
3	Complex Network based[18-20]	No	Service-based process	No

4	Run-time Self-adaptation based [21-28]	Yes	Service-based process (only one run)	Partial
5	DKEM[29]	Yes	Service&Service-based process&SKB	Yes
6	SDKEM	Yes	Service&Service-based process&SKB&Bridge rule	Yes

9 Conclusion and future work

In this paper, we propose an automatic service ecosystem evolution model SDKEM. It can capture various evolution opportunities and automatically trigger the evolution of service-based processes. Using bridge rules and self-adaptation technology, SDKEM automatically promotes competition and cooperation among services with distributed knowledge. During evolution, service-based processes with high stability are picked out to replace faulted ones. Service-based processes' stability is evaluated according to a stability evaluation model, which considers the effect of related services and related providers on stability. Especially, local and global evolution patterns are designed to evolve service-based processes, and a bridge rule evolution algorithm is presented to generate bridge rules when ontologies change automatically. Also, perishing services and SKBs are evolved by developers with the help of our prototype system. Ultimately, SDKEM can make a service ecosystem continuously and healthily evolve. Experiment results show that SDKEM can effectively and efficiently achieve holistic and continuous evolution of a service ecosystem, and guarantee more stable response time and a lower failure rate of business processes under dynamic and distributed running environments.

In SDKEM, evolution operations about perishing services are manually completed by developers, including semantic annotation, updating interface description, etc. In the future, we will improve SDKEM to carry out these operations automatically. Also, we will consider more QoS features in the stability evaluation model, such as throughput capacity, cost, the reputation of providers, etc.

Acknowledgment

This work is supported by the key project of the National Natural Science Foundation of China (61832014, 62032016), Natural Science Foundation of Shandong Province (ZR2018MF012, ZR2020MF084), Project of Shandong Province Higher Educational Science and Technology Program (J18KA364), and Doctoral Fund of Shandong Jianzhu University(X19045Z).

References:

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi,R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," Present and Ulterior Software Engineering, pp. 195–216, 2017.

[2] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casal- 'las, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in Computing Colombian Conference, 2015.

[3] K. Huang, J. Yao, J. Zhang, and Z. Feng, "Human-as-a-service: Growth in human service ecosystem," in IEEE International Conference on Services Computing, 2016, pp. 90–97.

[4] L. J. R. Stroppi, O. Chiotti, and P. D. Villarreal, "Extending bpmn 2.0:Method and tool support," in International Workshop, 2011, pp. 59–73.

[5] L. Cesari, R. Pugliese, and F. Tiezzi, "A tool for rapid development of ws-bpel applications," Acm Sigapp Applied Computing Review, vol. 11,no. 1, pp. 27–40, 2010.

[6] A. L. Lemos, F. Daniel, and B. Benatallah, "Web service composition:a survey of techniques and tools," Acm Computing Surveys, vol. 48, no. 3,pp. 1–41, 2015.

[7] M. Garriga, C. Mateos, A. Flores, A. Cechich, and A. Zunino, "Restful service composition at a glance: A survey," Journal of Network &Computer Applications, vol. 60, no. C, pp. 32–53, 2016.

[8] G. Castelli, M. Mamei, A. Rosi, and F. Zambonelli, "Engineering pervasive service ecosystems:the sapere approach," Acm Transactions on Autonomous & Adaptive Systems, vol. 10, no. 1, pp. 1–27, 2015.

[9] X. Wang, Z. Feng, K. Huang, and W. Tan, "An automatic self-adaptation framework for service-based process based on exception handling," Concurrency & Computation Practice & Experience, vol. 29, no. 5,2017.

[10] X. Wang, Z. Feng, and K. Huang, "D3l-based service runtime selfadaptation using replanning," IEEE Access, vol. PP, no. 99, pp. 1–1,2018.

[11] H. Wang, Z. Tu, Y. Fu, Z. Wang, and X. Xu, "Time-aware user profiling from personal service ecosystem," Neural Computing and Applications, pp. 1–23, 2020.

[12] O. Adeleye, J. Yu, S. Yongchareon, and Y. Han, "Constructing and evaluating an evolving web-api network for service discovery," in International Conference on Service-Oriented Computing, 2018.

[13] B. Rahul C, "On the evolution of service ecosystems: A study of the emerging api economy," Handbook of Service Science, vol. 2, pp. 479–495, 2018.

[14] M. J. Hadley, "Web application description language (wadl) specification," 2009.

[15] M. Fokaefs and E. Stroulia, "Wsdarwin: Studying the evolution of web service systems," 2014.

[16] H. T. Tran, H. Baraki, R. Kuppili, A. Taherkordi, and K. Geihs, "A notification management architecture for service co-evolution in the internet of things," in Maintenance and Evolution of Service-Oriented and Cloud-Based Environments, 2016, pp. 9–15.

[17] A. R. Sampaio, H. Kadiyala, H. Bo, J. Steinbacher, T. Erwin, N. Rosa,I. Beschastnikh, and J. Rubin, "Supporting microservice evolution," in IEEE International Conference on Software Maintenance and Evolution, 2017, pp. 539–543.

[18] Y. Liu, Y. Fan, and K. Huang, "Service ecosystem evolution and controlling: A research framework for the effects of dynamic services," in International Conference on Service Sciences, 2013, pp. 28–33.

[19] B. Xia, Y. Fan, and K. Huang, "Prediction method of perishing services in web service ecosystem," Computer Integrated Manufacturing Systems, vol. 20, no. 8, pp. 2060–2070, 2014.

[20] Y. Liu, Y. Fan, K. Huang, and W. Tan, "Failure analysis and tolerance strategies in web service ecosystems," Concurrency & Computation Practice & Experience, vol. 27, no. 5, pp. 1355–1374, 2015.

[21] M. Chinosi and A. Trombetta, "Bpmn: An introduction to the standard," Computer Standards Interfaces, vol. 34, no. 1, pp. 124–134, 2012.

[22] P. W. Wang, Z. J. Ding, C. J. Jiang, M. C. Zhou, and Y. W. Zheng, "Automatic web service composition based on uncertainty execution effects," IEEE Transactions on Services Computing, vol. 9, no. 4, pp.551–565, 2016.

[23] Z. Wei, F. Bastani, I. L. Yen, J. Fu, and Y. Zhang, "Automated holistic service composition: Modeling and composition reasoning techniques," in IEEE International Conference on Web Services, 2017.

[24] M. Polese, G. Tretola, and E. Zimeo, "Self-adaptive management of web processes," in IEEE International Symposium on Web Systems Evolution, 2010.

[25] G. H. Alfe'rez, V. Pelechano, R. Mazo, C. Salinesi, and D. Diaz, "Dynamic adaptation of service compositions with variability models," Journal of Systems & Software, vol. 91, no. 5, pp. 24–47, 2014.

[26] A. Murguzur, S. Trujillo, H. L. Truong, S. Dustdar, s. Ortiz, and G. Sagardui, "Runtime variability for context-aware smart workflows." IEEE Software, vol. 32, no. 3, pp. 52–60, 2015.

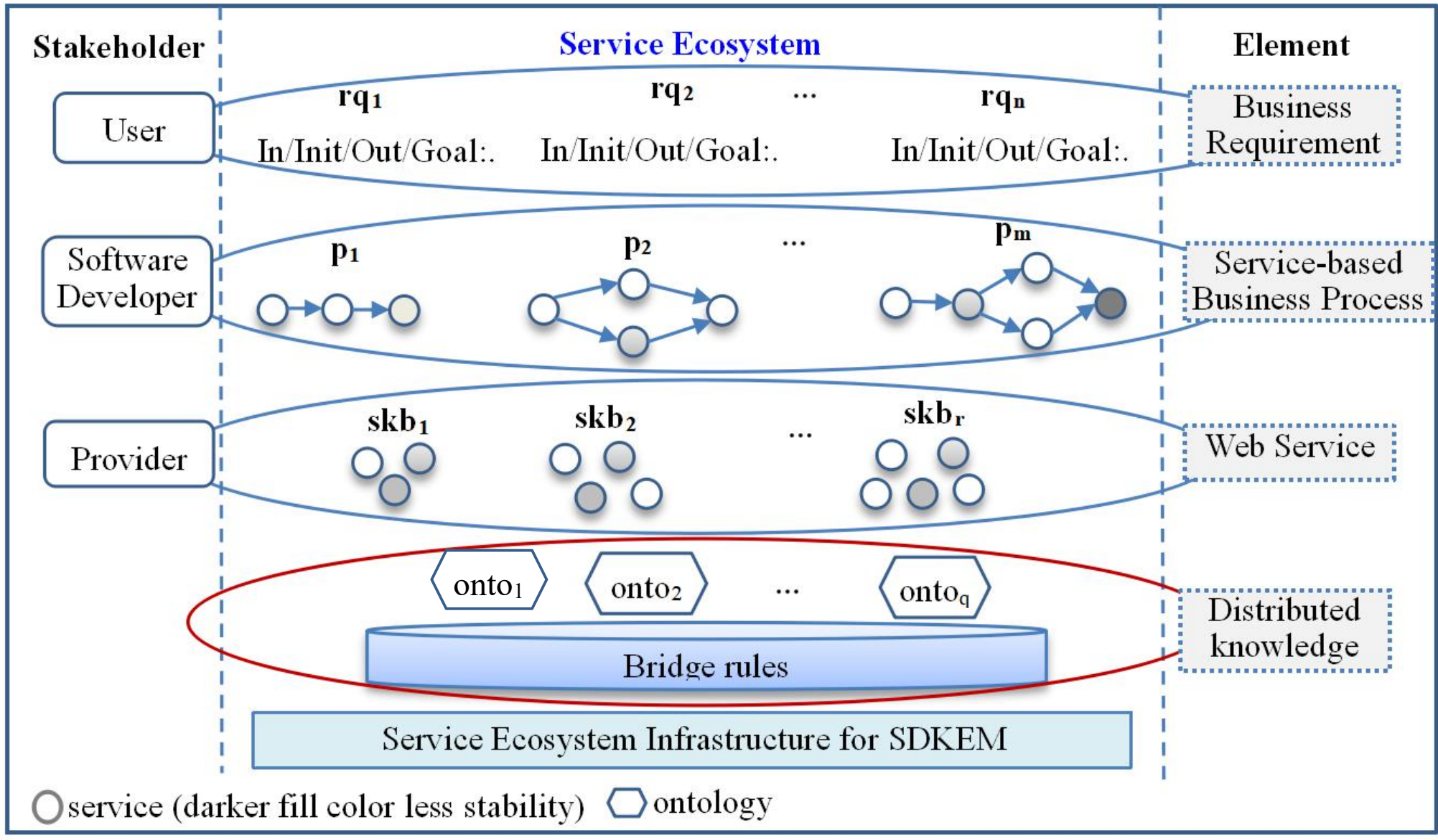
[27] A. Bucchiarone, M. D. Sanctis, A. Marconi, M. Pistore, and P. Traverso, "Incremental composition for adaptive by-design service based systems," in IEEE International Conference on Web Services, 2016.

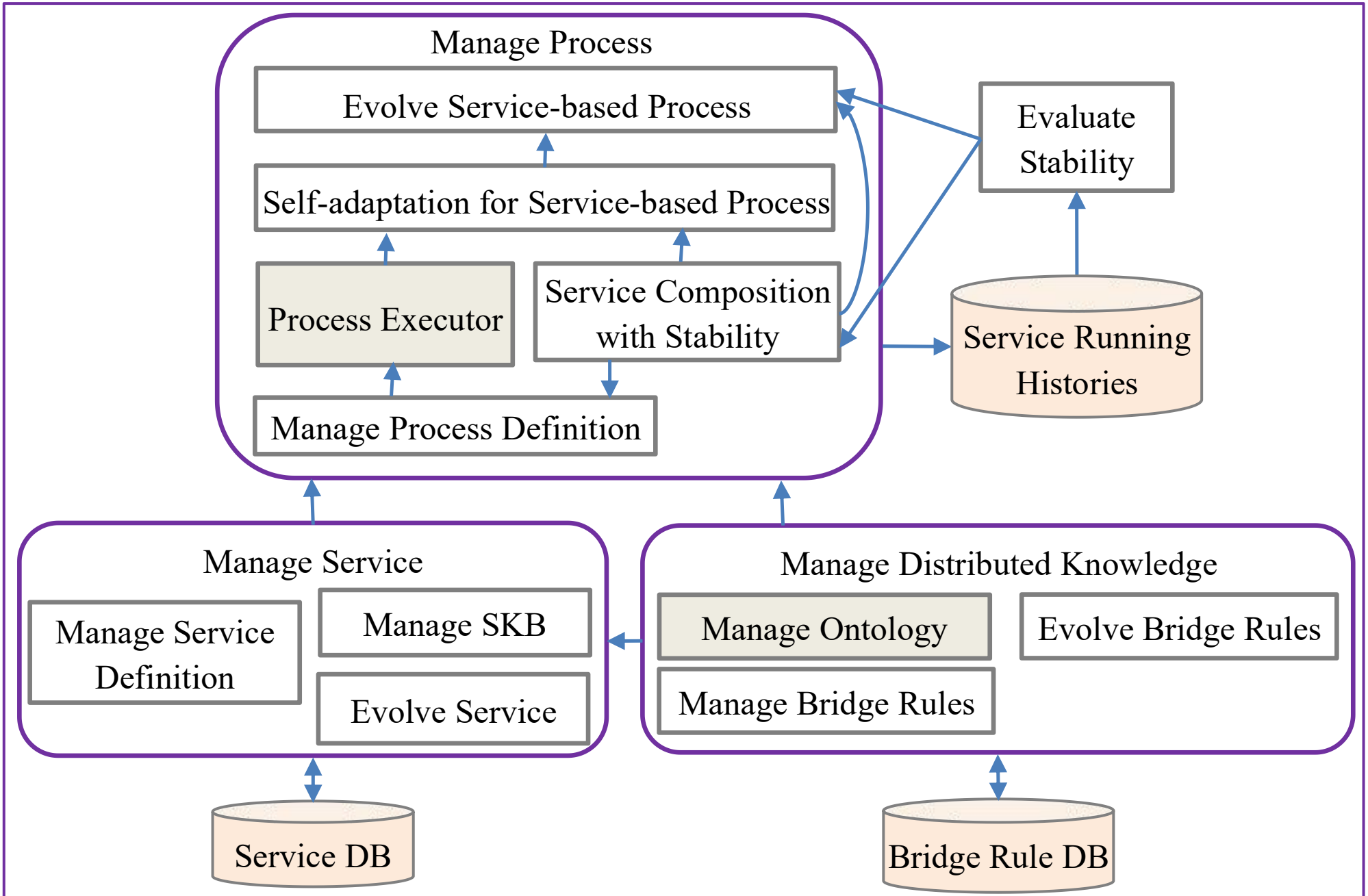
[28] N. van Beest, E. Kaldeli, P. Bulanov, J. Wortmann, and A. Lazovik, "Automated runtime repair of business processes," Information Systems,vol. 39, pp. 45–79, 2014.

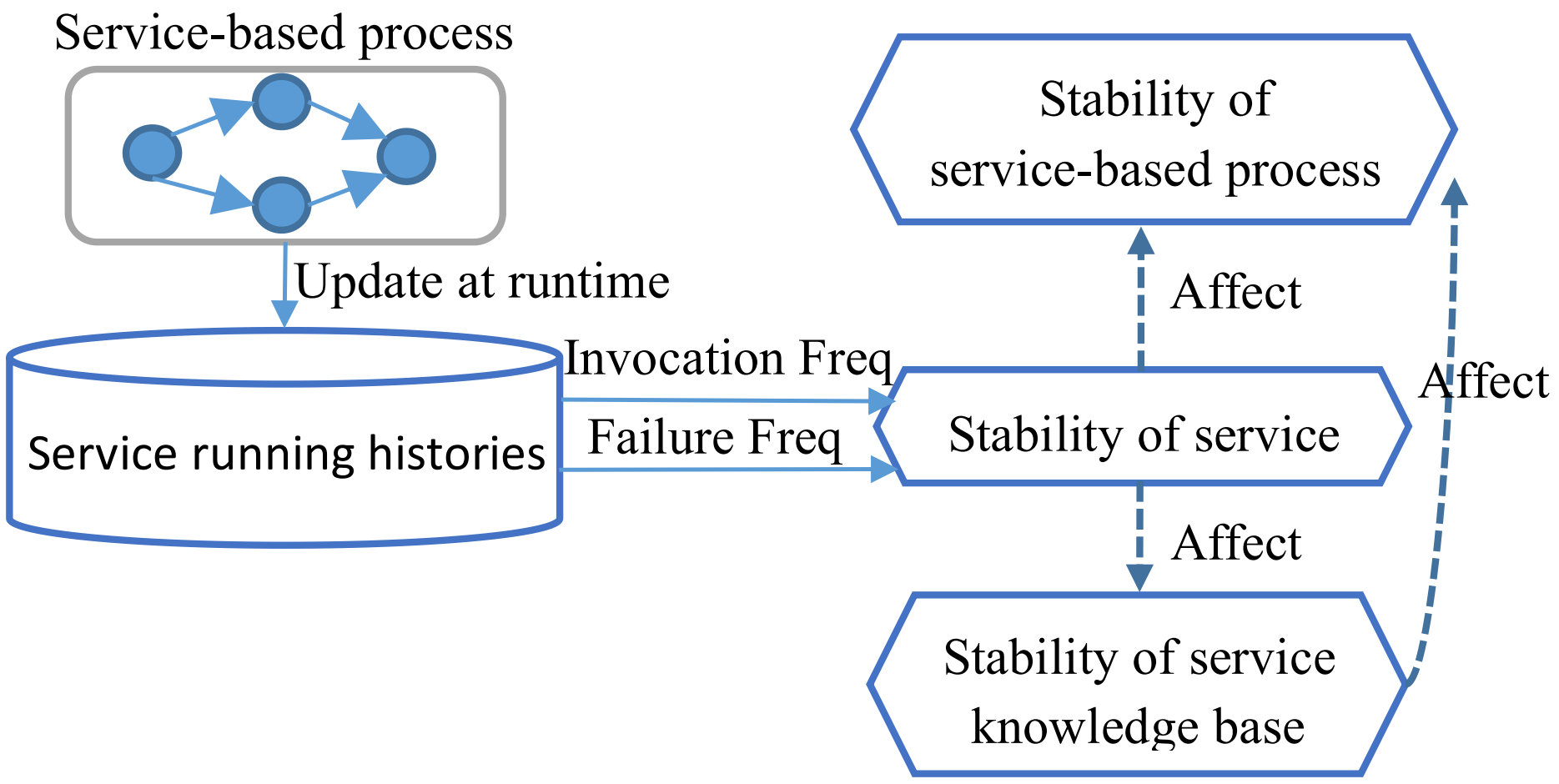
[29] X. Wang, Z. Feng, S. Chen, and K. Huang, "Dkem: A distributed knowledge based evolution model for service ecosystem," in 2018 International Conference on Web Services, 2018.

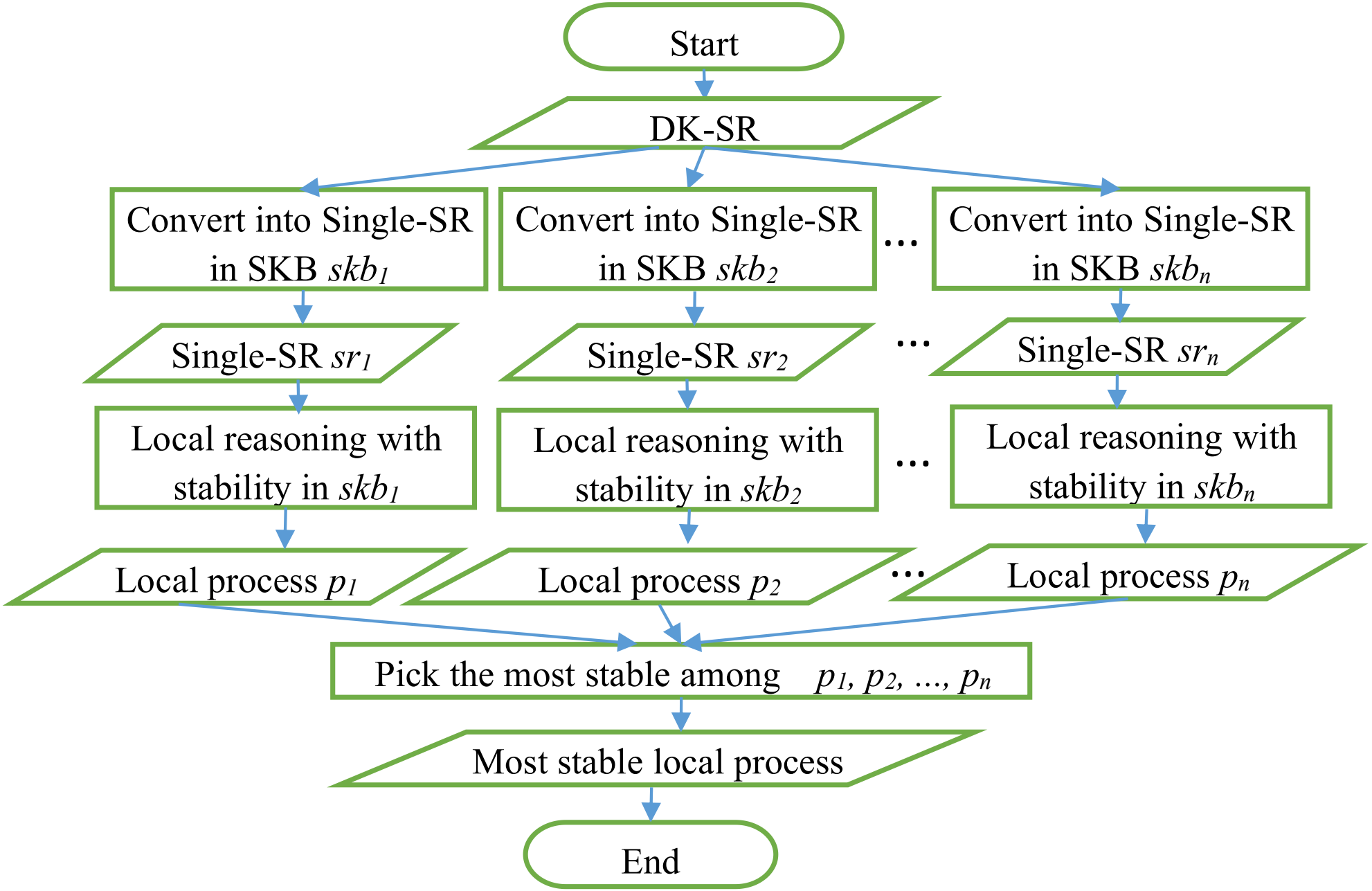
[30] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. Mcdermott, S. Mcilraith, S. Narayanan, M. Paolucci, B. Parsia, and T. R. Payne, "Owl-s:Semantic markup for web services," In Proceedings of the International Semantic Web Working Symposium (SWWS), 2004.

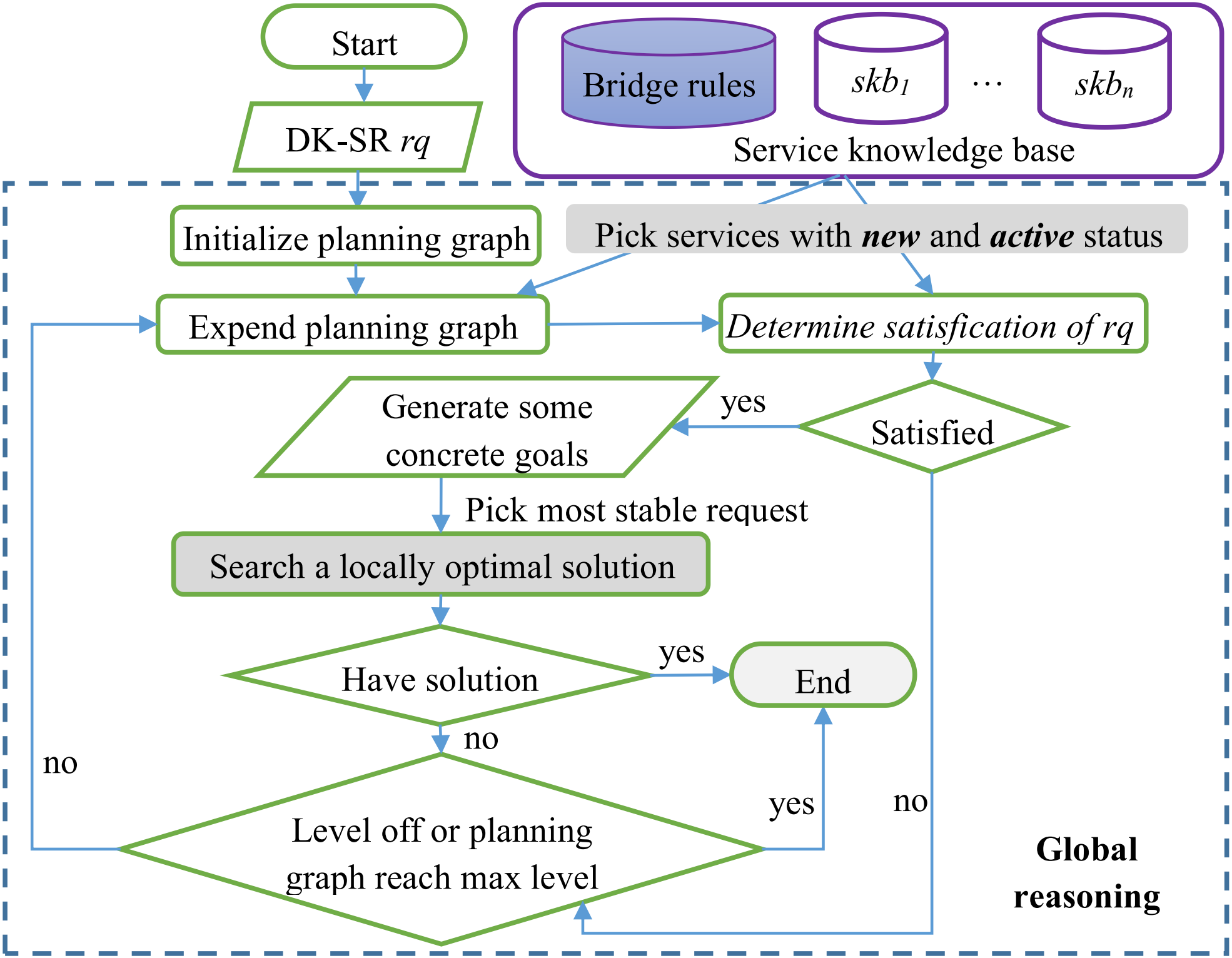
[31] X. Wang and Z. Feng, "Semantic web service composition considering iope matching," Journal of Tianjin University, vol. 50, no. 9, pp. 984–996, 2017.

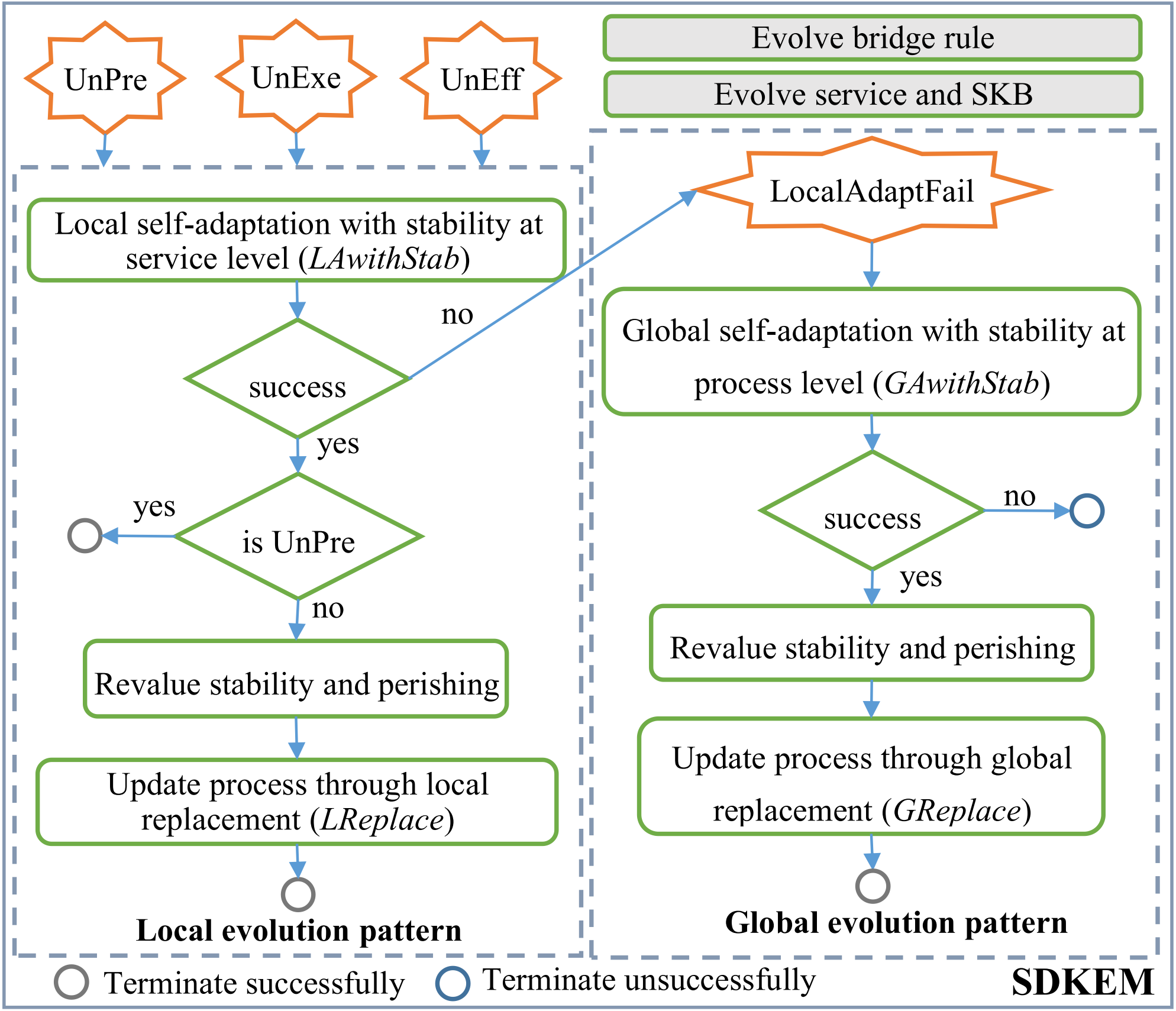

Stakeholder
Service Ecosystem
Element
User
rq1
rq2
...
rqn
In/Init/Out/Goal:.
In/Init/Out/Goal:.
In/Init/Out/Goal:.
Business Requirement
Software Developer
p1
p2
...
pm
Service-based Business Process
Provider
skb1
skb2
...
skbr
Web Service
onto1
onto2
...
ontoq
Bridge rules
Distributed knowledge
Service Ecosystem Infrastructure for SDKEM
service (darker fill color less stability)
ontology


Manage Process
Evolve Service-based Process
Self-adaptation for Service-based Process
Process Executor
Service Composition with Stability
Manage Process Definition
Evaluate Stability
Service Running Histories
Manage Service
Manage Service Definition
Manage SKB
Evolve Service
Manage Distributed Knowledge
Manage Ontology
Evolve Bridge Rules
Manage Bridge Rules
Service DB
Bridge Rule DB


Service-based process
Update at runtime
Service running histories
Invocation Freq
Failure Freq
Stability of service-based process
Affect
Stability of service
Affect
Affect
Stability of service knowledge base


Start
DK-SR
Convert into Single-SR in SKB skb_1
Convert into Single-SR in SKB skb_2
...
Convert into Single-SR in SKB skb_n
Single-SR sr_1
Single-SR sr_2
...
Single-SR sr_n
Local reasoning with stability in skb_1
Local reasoning with stability in skb_2
...
Local reasoning with stability in skb_n
Local process p_1
Local process p_2
...
Local process p_n
Pick the most stable among $p_1, p_2, ..., p_n$
Most stable local process
End


Start
DK-SR rq
Bridge rules
skb1
…
skbn
Service knowledge base
Initialize planning graph
Pick services with new and active status
Expend planning graph
Determine satisfication of rq
Generate some concrete goals
yes
Satisfied
Pick most stable request
Search a locally optimal solution
Have solution
yes
End
no
no
yes
no
Level off or planning graph reach max level
Global reasoning


UnPre
UnExe
UnEff
Evolve bridge rule
Evolve service and SKB
Local self-adaptation with stability at service level (LAwithStab)
success
no
yes
yes
is UnPre
no
Revalue stability and perishing
Update process through local replacement (LReplace)
Local evolution pattern
LocalAdaptFail
Global self-adaptation with stability at process level (GAwithStab)
success
no
yes
Revalue stability and perishing
Update process through global replacement (GReplace)
Global evolution pattern
Terminate successfully
Terminate unsuccessfully
SDKEM

Algorithm 1 EvolveBRLib

Inputs: *brlib*: all bridge rules in current rule library; *newmbrs*: new bridge rules added manually; synvoctb: synonym vocabulary table

Outputs: updated *brlib*

1. **FOR** each $<v_1, rtype, v_2>$ in *newmbrs* **DO**
2. **IF** *rtype* is not $equalr_F$ **THEN** $newmbrs \leftarrow newmbrs \cup \{ <v_2, \textbf{\textit{inverse}}(rtype), v_1>\}$
3. **END FOR**
4. $brlib = brlib \cup \textbf{\textit{synequal}}(synvoctb)$
5. **FOR** each $<v_1, rtype, v_2>$ in *newmbrs* **DO**
6. **FOR** each $<v_3, rtype', v_4>$ in *brlib* **DO**
7. **IF** $rtype == rtype'$ and $v_1 == v_4$ **THEN** $brlib = brlib \cup \{<v_3, rtype, v_2>\}$
8. **IF** $rtype == rtype'$ and $v_2 == v_3$ **THEN** $brlib = brlib \cup \{<v_1, rtype, v_4>\}$
9. **IF** *rtype* start with '*equal*' and *rtype'* start with '*into*' or '*onto*' **THEN**
10. **IF** $v_1 == v_3$ **THEN** $brlib = brlib \cup \{<v_2, rtype', v_4>\}$
11. **IF** $v_1 == v_4$ **THEN** $brlib = brlib \cup \{<v_3, rtype', v_2>\}$
12. **IF** $v_2 == v_3$ **THEN** $brlib = brlib \cup \{<v_1, rtype', v_4>\}$
13. **IF** $v_2 == v_4$ **THEN** $brlib = brlib \cup \{<v_3, rtype', v_1>\}$
14. **END IF**
15. **IF** *rtype'* start with '*equal*' and *rtype* start with '*into*' or '*onto*' **THEN**
16. **IF** $v_1 == v_3$ **THEN** $brlib = brlib \cup \{<v_4, rtype, v_2>\}$
17. **IF** $v_1 == v_4$ **THEN** $brlib = brlib \cup \{<v_3, rtype, v_2>\}$
18. **IF** $v_2 == v_3$ **THEN** $brlib = brlib \cup \{<v_1, rtype, v_4>\}$
19. **IF** $v_2 == v_4$ **THEN** $brlib = brlib \cup \{<v_1, rtype, v_3>\}$
20. **END IF**
21. **END FOR**
22. **END FOR**
23. **RETURN** $brlib \cup newmbrs$

Algorithm 2 Evolving perishing service and service knowledge base

Inputs: *s*: a perishing service, *skb*: a service knowledge base containing *s*

Outputs: an updated *skb*

01. *newInvSyn* ← new invocation information of *s* from its official website
02. *oldInvSyn* ← *s.InvSyn*, that is, old invocation information of *s*.
03. **IF** *newInvSyn* doesn't exist or *newInvSyn*==*oldInvSyn* **THEN** remove *s* from *skb.TP*
04. **ELSE**
05. *s.InvSyn* ← *newInvSyn*
06. *s.FunSem* ← new function semantics manually annotated by developers
07. *s.IF* ← 0
08. **END IF**
09. **IF** *skb.TP*==Ø **THEN** remove *skb*
10. **RETURN** *skb*


NoFailNumBefore
SuccessNum
NoFailNumAfter
Original process number
30
20
10
0
rs1
rs2
rs3
rs4
rs5

Service Ecosystem Management Platform for SDKEM

Service Request | Service | SKB | Knowledge

Service Request List

No.	name	has process
1	rq0	yes
2	rq1	yes
3	rq101	yes
4	rq102	yes
5	rq103	yes
6	rq104	yes
7	rq105	yes
8	rq106	yes
9	rq2	yes
10	rq21	yes
11	rq3	yes
12	rq31	yes
13	rq32	yes
14	rq33	yes
15	rq4	yes
16	rq41	yes
17	rq52	yes
18	rq53	yes
19	rq6	yes
20	rq60	yes
21	rq61	yes
22	rq62	yes
23	rq63	yes
24	rq7	yes
25	rq8	yes
26	rq81	yes
27	rq9	yes

Existing requests

Picked or new request details

request name: rq0 remark: Intercity traffic add request | update request | manage process

input: wskb1-Person:wxh#wskb101-Date:date#wskb1-City:jn#wskb1-City:tj#wskb1-Address:sdjzu#wskb1-Address:tju

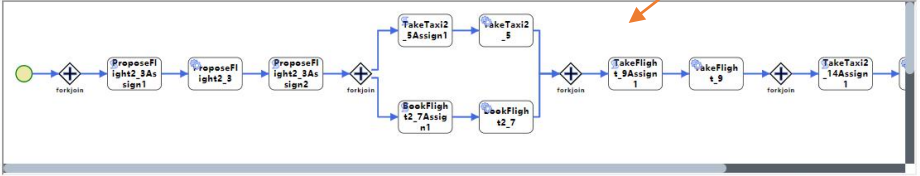
init: wskb1-personAtAddress(wxh,sdjzu)&wskb1-addressAtCity(sdjzu,jn)&wskb1-addressAtCity(tju,tj)

output: wskb101-Vehicle:?t

goal: wskb1-personAtAddress(@wxh,@tju)

generate process | run process

Business process diagram



19	process output---->{"flight":"1~flight","fromPort":"1~fromPort","toPort":"1~toPort","airTicket":"1~airTicket"}	2020/12/11 16:49:57
20	ProposeFlight[{d=date, c1=jn, c2=tj}] UnExe local success to run.....	2020/12/11 16:49:57
21	[rq0] local evolution start to run.....	2020/12/11 16:49:57
22	faultedService:ProposeFlight[stab:0.0]	2020/12/11 16:49:57
23	faultedservice stability: 0.75	2020/12/11 16:49:57
24	loca process stability: 1.0	2020/12/11 16:49:57

Real-time running status of the business process including evolution details