A Service Execution Control Framework for Policy Enforcement

Masahiro Tanaka, Yohei Murakami, and Donghui Lin

Language Grid Project, National Institute of Information and Communications Technology (NICT),
3-5 Hikaridai, Seika-cho, Kyoto, Japan
{mtnk,yohei,lindh}@nict.go.jp

Abstract. Service-oriented collective intelligence, which creates new value by combining various programs and data as services, requires many participants. Therefore it is crucial for an infrastructure for service-oriented collective intelligence to satisfy various policies of service providers. Some previous works have proposed methods for service selection and adaptation which are required to satisfy service providers' policies. However, they do not show how to check if the selected services and adaptation processes certainly satisfy service providers' policies. In this paper, we propose an execution control framework which realizes service selection and adaptation in order to satisfy service providers' policies. On the framework, the behaviors of composite services are verified against service providers' policies based on model checking. We also formally defined the effect of the proposed execution control APIs. This enabled us to update models for verification at runtime and reduce the search space for verification.

1 Introduction

Services computing technologies had initially aimed to realize flexible development and management of information system of enterprises. However, services computing are now applied to service-oriented collective intelligence, which creates new values by combining a wide variety of programs and contents provided by various providers.

For example, the Language Grid[1] is one of the infrastructures for service-oriented collective intelligence and has achieved interoperability of language resources such as machine translators and dictionaries by wrapping them as Web services with standardized interfaces. More than 120 organizations have joined the Language Grid and 90 language services are available on the infrastructure.

It is crucial to have many service providers join in order to realize for service-oriented collective intelligence. Service providers usually have their own policies about use of their service including limitation of transferred data, constraints on combinations of services and so on. Therefore an infrastructure for service-oriented collective intelligence must be capable of satisfying their policies. From this aspect, service-oriented collective intelligence is quite different with traditional collective knowledge which relies on single license e.g. Wikipedia. As for

P.P. Maglio et al. (Eds.): ICSOC 2010, LNCS 6470, pp. 108–121, 2010.

the Language Grid, it allows service providers to set permission to use their service for each user and limit amount of transferred data.

One of the major advantages of services computing is a flexibility of development and management. This is the reason why a composite service is usually defined as a workflow. Users can select services which are assigned to tasks in the workflow at runtime according to his/her requirements. On the other hand, in the context of service-oriented collective intelligence, the combination of selected services must satisfy service providers' policy. Otherwise, some adaptation process should be applied in order to change the behavior of the services and make them follow the given policy.

Many previous works have proposed methods for service selection for composite services[2,3]. Moreover, to adapt behaviors of services without changing models of the composite services, various methods using Aspect-oriented Programming (AOP)[4] or proxy for message exchange[5] have been proposed. In [6], a comprehensive process for service selection and adaptation is also proposed.

However, service selection and adaptation during execution make it difficult to verify that service providers' policies are satisfied. Applying an adaptation process to satisfy a policy may lead to violation of another policy which was satisfied before the adaptation. Although model checking for composite services[7] can verify behaviors, such method has the following problems.

Dynamic change of model. Runtime adaptation changes the behavior of the target composite service during execution. It is unrealistic to manually change the model for verification.

Verification cost. Verification is performed by exhaustively searching execution states. Runtime verification may decline the performance of the composite service.

To solve the problems, we propose an execution control framework whose behaviors can be verified by extending the methods proposed in [8] and [6]. We also show a reduction of search space for runtime verification according to service selection and adaptation during execution.

The rest of the paper is organized as follows. In Section 2, we show a scenario which represents the problems to be solved. In Section 3, we overview the system architecture which solves the problems. Then we detail the solution, which consists of an execution control framework for composite service and application of model checking in Section 4 and Section 5 respectively. After introducing related works in Section 6, we conclude this paper in Section 7.

2 Scenario

In this section, we show a typical scenario and the problems to be solved by taking a composite service for translation on the Language Grid[1] as an example. Suppose a composite service combines a morphological analyzer, a machine translator, and technical term dictionaries. This service improves translation quality of technical documents by translating technical terms in the given sentences using the technical term dictionaries, not the machine translator.

Figure 1 shows the overview of the composite service. A square which contains a circle represents a service invocation. First the given sentences are divided into morphemes by the morphological analyzer. Next dictionaries find technical terms which consist of the morphemes and return the translation of the technical terms. Finally the translator translates the whole sentences.

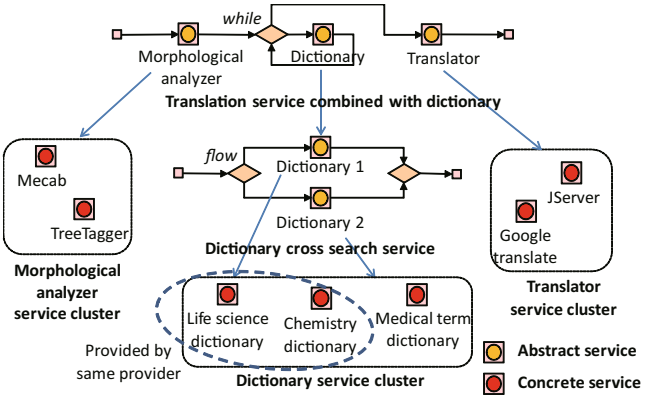

Fig. 1. A composite service for translation

As shown in Fig. 1, we assume that a composite service is defined in a workflow description language such as WS-BPEL[9]. In a composite service, the constituent services define only the interface and are not bound to any endpoint. We refer to such a service as an *abstract service*. For example, on the Language Grid, an abstract service is defined for each service type such as translators and dictionaries. Endpoints for the services are determined when the composite service is invoked or during execution. A service to which an endpoint is bound is called a *concrete service*.

A set of concrete services which can be bound to an abstract service is called a service cluster. In Fig. 1, there are more than one concrete services in each service cluster. In our example, we bind one concrete service to the morphological analyzer and the machine translator. For the dictionary, we first bind a composite service for cross search and then bind two concrete services to abstract services in the cross search composite service.

When we use the composite service, we need to select concrete services which satisfy service providers' policies. Assume that both the life science dictionary and the chemistry dictionary are provided by the same provider and that the provider prohibits concurrent access to the two services to prevent a user from giving too much load. On the other hand, the composite service allows two dictionary services to be concurrently executed because the designer of the composite service does not know which services are bound. In the case that the two dictionaries are bound, the execution of the composite service violates the provider's policy. Since the providers' policies must be certainly satisfied, some verification method such as model checking should be applied.

However, the uncertainty of service execution often causes the following problems. Assume the dictionary service replaces the technical terms found in the input sentences with words in the target languages and that the translator does not translate words in the target language. In this way, we can obtain a translation result whose technical terms are translated by the dictionary, not by the translator. But this may cause a failure of execution of the translator because the length of input sentences changes by the word replacement and may exceeds the limit of input length of the translator.

One of the solutions to this failure is adding an adaptation process, which divides the input string into sentences, translates each of them and merge the translation results. But applying adaptation processes may cause other violation of service providers' policies because the business logic is changed. For example, the service provider for a dictionary service and translator service may give discount for a plan of the same count of invocation of the services. Therefore we need to modify the model for verification according to the changes of business logic and perform verification again.

The first problem of the above approach is that it is unrealistic for the user or the human operator to modify models for verification during execution. Another problem is that verification during execution may decline the performance of the composite service.

3 System Architecture

In this section, we describe the system architecture which contains an execution control framework for runtime service selection and adaptation and a verification framework for dynamic changes of the business change.

Figure 2 shows the overview of the system proposed in this paper. The system consists of the composite service execution engine, service selector and behavior verifier.

The composite service execution engine interprets and executes the deployed composite services. It is extended to provide APIs for adaptation which allows changing business logics of composite services and get/set execution state of running instance of a composite service. Using the APIs, we can define an adaptation service, which is a composite service and implements adaptation process for other composite services. The interaction between the adaptation service and the target composite service is configured by a supervision service, which is also a composite service and uses the APIs.

The service selector finds a combination of services which satisfy service providers' policies and binds them to abstract services. If there is no combination which satisfies the given policies, it searches for adaptation processes which can change properties of the services and satisfy given policies referring to profiles of available services and adaptation processes.

The verifier checks that a composite services satisfies given policy based on the model of the composite service, the model of the adaptation service and changes of business logic defined in the supervision service. It uses SPIN model checker.

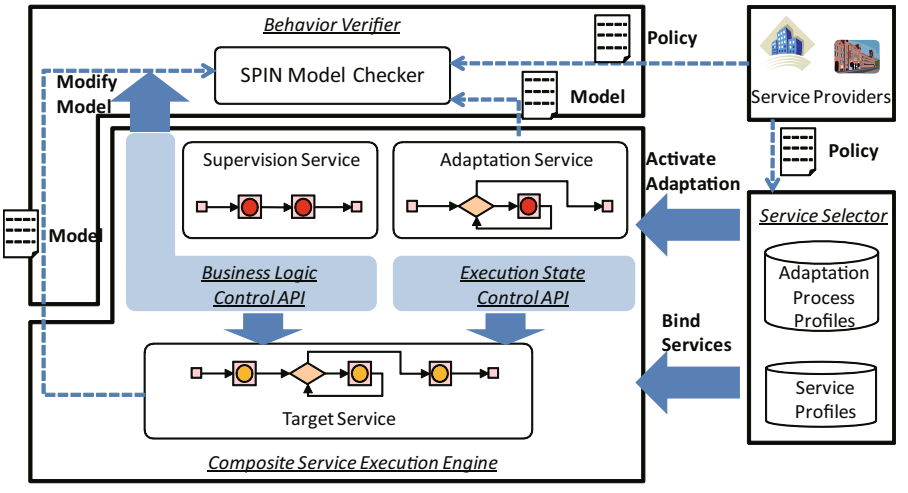

Fig. 2. An architecture for policy enforcement

We describe the steps of service execution on the architecture below. First the composite service execution engine is invoked by a request from a user. Since abstract composite services are deployed on the engine, user can specify bindings for constituent services of the composite services. For abstract services to which the user does not bind a concrete service, the service selector selects concrete services to be bound. Adaptation processes to be applied are also activated if needed.

Through execution of the composite service, the following two types of check are performed.

Execution state check. Check if the combination of services and variables defined in a composite service satisfy the given policies. This check is performed before each service invocation.

Behavioral check. Check if the execution state and protocols of service invocation. This check is performed before an adaptation process is executed.

Both checks are also performed when execution of a composite service starts. If the service selector cannot find a combination which passes execution state check, it applies adaptation processes to services. For example, when the system finds that the input string exceeds the limit of length, the adaptation process which divides an input string into sentences can be applied to pass the check. The selected adaptation process must pass behavioral check using SPIN model checker[10]. When it cannot finally find an appropriate combination of services or adaptation processes, the execution of the composite service fails. The method for finding a combination of services and adaptation processes is proposed in [6].

4 Adaptation Using Execution Control

The architecture shown in the previous section contains adaptation mechanism. The adaptation requires flexible changes of behaviors of composite services, e.g.

adding a parallel process and skipping defined a service invocation. We defined two types of execution control APIs to realize the adaptation. In this section, we describe the details of the APIs and a framework for adaptation using the APIs.

4.1 Execution State Control

The composite service defined in WS-BPEL, the standard language for describing composite services, consists of *activities*. An activity corresponds to an atomic process in a composite service such as service invocation and assignment of variables. Most of WS-BPEL execution engines define state for activities. We assume that an activity is in one of the following states: Ready, Running, Finished, Suspended.

A composite service defined in WS-BPEL also has variables and partner links which are information about service to be invoked. A partner link has an endpoint address of the invoked service. Therefore we can change invoked services at runtime by changing the endpoint address set to the partner link, without changing the model of the composite service.

In this paper, we assume the state of a running instance of a composite service is defined by the combination of states of activities, values of variables and endpoint addresses set to partner links. Most of constraints which come from service providers' policies are defined by the information. For example, the violation of the limit of input length can be found by checking the value of the variable input to the translator before invocation.

Table 1. Execution control APIs

API	Effect
`getVariable(pid, varname)`	Get the value of the specified variable. This API takes a process ID and a variable name as parameters.
`setVariable(pid, varname, value)`	Set a value to the specified variable. This API takes a process ID, variable name and values to be set as parameters.
`getPartnerLink(pid, plname)`	Get the specified partner link. This API takes a process ID and a partner link name as parameters.
`setPartnerLink(pid, plname, address)`	Set a partner link. This API takes a process ID, a partner link and an endpoint address to be set to the partner link as parameters.
`getActivityState(pid)`	Get states of all activities in the specified process. This takes a process ID as parameters.
`suspend(pid, activity)`	Suspend the specified running activity. This API takes a process ID and activity as parameters.
`resume(pid, activity)`	Resume the specified suspended activity. This API takes a process ID and activity as parameters.
`getProcessIds(sid)`	Get process IDs of running instances of the specified composite service. This API takes a service ID as a parameter.

On the basis of the above assumption, we realized the APIs for execution control shown in Table 1. These APIs get/set variables, partner links and states of activities. We also provide an API for getting information of a target process.

WS-BPEL provides functionalities for getting/setting variables and partner links in the same running instance. On the other hand, we provide APIs for access information in other instances of composite services. This is because we need to consider all processes which use a service in order to satisfy policies of the provider of the service.

Moreover, even in the same instance, WS-BPEL does not provide functionalities for getting/setting states of activities. Therefore we usually need to modify the model of the composite services and add an interface which allows access to the execution state of the composite services. However, the designer of the composite service does not know what interface is required because a combination of services are decided at runtime.

Generally speaking, human operator monitors and manages execution state of composite services. They usually have a process or a workflow for the monitoring and management. Therefore we allow them to implement a composite service which realizes the same process as they perform by providing the APIs as Web service. The composite service can be defined also in WS-BPEL. Therefore we apply existing methods or tools for business process modeling.

4.2 Business Logic Control

After defining an adaptaion process, we need to integrate it into the target composite service. For example, using the APIs shown in Table 1, we can implement a composite service which gets an input string to translator, divides it into sentences, and merges after translating each of them. We need to define the protocol between the target composite service and the composite service for adaptation.

Most of previous works on adaptation for composite services have proposed methods for adding processes to existing composite services based on AOP[4,5]. But more flexible adaptations such as adding a parallel process or skipping activity are often required for adaptation in order to satisfy service providers' policies. Moreover, we need to apply adaptation at runtime because some violations of policies are found during execution.

According to these requirements, we provide the APIs shown in Table 2 for changing business logic. We also make these APIs accessible as Web service. Therefore we can define a composite services for changing business logic in order to integrate a certain adaptation process.

4.3 Adaptation Example

We can realize various adaptation processes by implementing composite services which use the APIs shown in this Section. We refer to a composite service which uses APIs for getting/setting execution state as an adaptation service, and a composite service which uses APIs for changing business logic as a supervision service.

Table 2. Business logic control APIs

API	Effect
`skip(pid, start, end)`	Skip activities in the specified range. This API takes a process ID and activity locations which specify the range to be skipped.
`fork_join(pid, start, end, invocation)`	Add a parallel process to the specified instance. This takes a process ID, activity locations where the parallel process starts/finishes and invocation information (endpoint address, operation and input data) as parameters.
`fork(pid, start, invocation)`	Add a parallel process to the specified instance. This takes a process ID, an activity location where the parallel process starts and invocation information as parameters.
`insert(pid, start, pname, param)`	Invoke a service at the specified location. This takes a process ID, an activity location where the invoked process starts and invocation information as parameters.

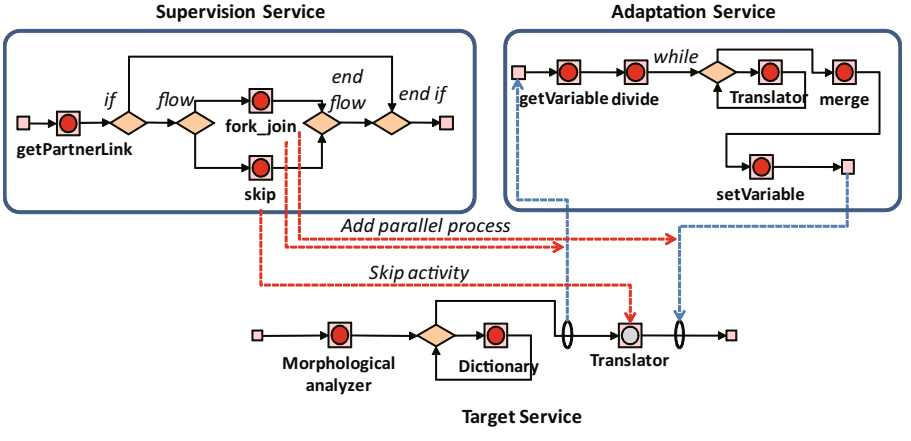

Fig. 3. Adaptation for division and merging

Figure 3 shows an example of adaptation for the composite service shown in Fig. 1. The adaptation divides input string to translator and merges them after translating each of them.

The Supervision Service shown in Fig. 3 is invoked after Service Selector decides to use the Adaptation Service. First it checks the endpoint for translator `getPartnerLink`, and then integrates the Adaptation Service into the target composite service (`fork_join`) and skip invocation of translator defined in the target service (`skip`).

The adaptation service first reads the variable which is set as the input to the translator (`getVariable`). Then it divides the input to sentences and translates each of them in loop. The traslation results are merged and set to the variable which is set to the output of the translator (`setVariable`).

5 Verification during Execution

When a supervision process is executed, the business logic of the target service is changed as shown in Fig. 3. Therefore execution of the target service may violate policies which were once satisfied. This is the reason we should perform behavioral check whenever adaptation is applied. In this section, we first show how models for verification are changed based on business logic APIs described in the previous section. Then we propose an idea for reducing computational cost for verification in order to prevent decline of performance.

5.1 Updating Model

Behavioral check described in Section 3 uses SPIN model checker and requires model of composite services described in Promela. We assume the models for SPIN can be generated from WS-BPEL processes. Although it is impossible to generate models which represents complete behaviors of a composite service, in [7], the author shows the method generating a model described in Promela from composite services described in WS-BPEL by introducing some abstraction.

Using SPIN and model described in Promela, we can verify that policies represented as LTL (Liner Temporal Logic) formula are satisfied. For example, the following formula f represents that the life science dictionary service ($S_{life\_science}$) and the chemistry dictionary service ($S_{chemistry}$) are not concurrently executed. $state()$ returns the state of the given service.

$$f = \Box\neg((state(S_{life\_science}) = Running) \wedge (state(S_{chemistry}) = Running))$$

SPIN transforms models described in Promela into an automaton which represents behaviors of the system. The automaton of the whole system including the target composite services and adaptation services can be obtained as an asynchronous product of automatons of target composite services $S_1...S_m$ and those of adaptation services $a_1...a_n$ as shown below.

$$M_A = M_{S_1} \times ...M_{S_m} \times M_{a_1} \times ... \times M_{a_n}$$

The LTL formula f is satisfied if a synchronous product of M_A and $\overline{M_f}$, which is an automaton corresponding to the negation of f, is empty.

M_A needs to be changed when business logic control APIs are executed. Since it is unrealistic to manually modify the model during execution, we defined modification of an automaton for each API in Table 2.

Figure 4 shows the process for `fork_join`. The idea of the modification is to add states for sending requests/receiving response. In Fig. 4, we omit the process ID from the parameter list for simplicity. Channels which represent sending a request to/receiving a response from the adaptation service are defined. States which correspond to activity location specified as `start` or `end` are also defined. The automaton can have more than one state which corresponds to `start` or `end` because the states can have the values of variables and partner links.

```
fork_join(start, end, invocation)

ch!op <- send action on channel which represents invocation
S_start <- set of states which corresponds to start
ch?op <- receive action on channel which represents invocation
S_end <- set of states which corresponds to end

for each s_start in S_start
  S_to_start <- set of states which have transition to s_start
  for each s_to_start in S_to_start
    create node S_new
    replace transition(s_to_start, s_start, act) with transition(s_to_start, s_new, act)
    create transition(s_new, s_start, ch!op)
  end for
end for
for each s_end in S_end
  S_to_end <- set of states which have transition to s_end
  for each s_to_end in S_to_end
    create node S_new
    replace transition(s_to_end, s_end, act) with transition(s_to_end, s_new, act)
    create transition(s_new, s_end, ch?op)
  end for
end for
```

Fig. 4. Updating automaton of verification model (`fork_join`)

In the first loop, the algorithm inserts a new state before the activity location specified as `start`. The transition from/to the new state are also created. The transition from the new state to the state which corresponds to the activity location specified as `start` is performed when the send action on the channel is executed. In the latter part of the algorithm, the same goes for states which correspond to the activity location specified as `end`.

The modification process for other APIs are shown in Fig. 5.

5.2 Reducing Execution State Space

Behavioral check requires exhaustive search on state space and may decline the performance of composite services. Therefore reducing search space contributes to improving the performance.

Business logic control APIs does not affect states between the initial state and states which corresponds to the activity location specified as `start`. This is the reason the states do not have to be checked once verification is performed before starting execution of the composite service.

On the other hand, we need to check reachability from the current state to the states which are added by the processes shown in Fig. 4 and Fig. 5.

Based on the above idea, we search only transition to states which are newly added by the processes in Fig. 4 and Fig. 5 if the search reaches the state which has a transition to the newly added states.

```
fork(start, invocation)

ch!op <- send action on channel which represents invocation
S_start <- set of states which corresponds to start

for each s_start in S_start
  S_to_start <- set of states which have transition to s_start
  for each s_to_start in S_to_start
    create node S_new
    replace transition(s_to_start, s_start, act) with transition(s_to_start, s_new, act)
    create transition(s_new, s_start, ch!op)
  end for
end for
```

```
insert(start, invocation)

ch!op <- send action on channel which represents invocation
ch?op <- receive action on channel which represents invocation
S_start <- set of states which corresponds to start

for each s_start in S_start
  S_to_start <- set of states which have transition to s_start
  for each s_to_start in S_to_start
    create node S_new1, S_new2
    replace transition(s_to_start, s_start, act) with transition(s_to_start, s_new1, act)
    create transition(s_new1, s_new2, ch!op)
    create transition(s_new2, s_start, ch?op)
  end for
end for
```

```
skip(start, end)

S_start <- set of states which corresponds to start
S_end <- set of states which corresponds to end

for each s_start in S_start
  S_to_start <- set of states which have transition to s_start
  for each s_to_start in S_to_start
    for each s_end in S_end
      act <- action for transition from s_to_start to s_start
      replace path(s_to_start, s_end) with transition(s_to_start, s_new, act)
    end for
  end for
end for
```

Fig. 5. Updating automaton of verification model (`fork`, `insert`, `skip`)

Figure 6 shows the algorithm. First $Initialize()$ is executed, then search is recursively executed. This is based on depth-first search, which is used by SPIN as default, and `satisfy` checks if the condition are satisfied or not. The condition is given according to f which represents service providers' policies. If any of states does not satisfy the condition, the algorithm immediately exits and returns error.

```
Initialize()
  Add the initial state to State Space
  Push the initial state to stack

search()
  Stop <- top of stack

  if Stop does not satisfy the condition
    exit(error)
  end if

  Snext <- set of states to which can be directly moved from Stop
  for each Snew in Snext
    if Snew is a state added by business logic control APIs
      if Snew is not in State Space
        add Snew to State Space
        push Snew to stack
        search()
      end if
    end if
  end for
  pop from stack
```

Fig. 6. An execution state search algorithm

6 Related Works

This work shows architecture based on runtime service selection and adaptation with verification for an infrastructure where many service providers join. In this section, we introduce some related works on verification of composite services, service selection and adaptation.

For verification, various approaches including process algebra, Petri net, finite state machine or logic have been proposed. Nakajima proposed a method for generating description in Promela, which is used for SPIN model checker, from a WS-BPEL process[7]. In the paper, the author shows an abstraction for verification and refers to features which are characteristic of WS-BPEL such as dead path elimination. Our paper assumes that the initial model of composite services are generated by such a method.

Narayanan et al. modeled composite Web services in OWL-S using Petri nets in order to verify the composite Web services [11]. Their work verifies the reachability to certain states and detects deadlocks.

Ankolekar et al. proposed a method transforming a composite Web service in OWL-S into descriptions in a language for model checking[12]. This makes it possible to verify not only control flow but also dataflow. Fu et al. focused on interactions between services in a composite Web service[13]. They transformed interaction protocols described in BPEL into Guarded Automata(GA), which reacts to certain messages. Then the GA is transformed into descriptions in a language for model checking. Their method can flexibly adopt various combinations of languages for composite Web service and languages for model checking.

Many previous works have proposed methods for service selection. For example, the method proposed in [2] focuses on finding a combination of services which gives the best QoS. The method proposed in [3] selects services considering interfaces of services in addition to QoS. These works assume that vast amount of services are stored in a service cluster. In reality, however, the number of services which have equivalent functions is limited. That is the reason the previous works often cannot find a combination of services. Moreover, to handle the policies of service providers, we need not only finding a static combination of services but also dynamic adaptation and meta-level control of composite services.

Also in the area of dynamic adaptation, there have been some previous works. Most of the works can be classified into three types: weaving a new process based on AOP (Aspect-oriented Programming), using a proxy to monitor/change messages exchanged between a composite service and invoked services, and transforming the model of a composite service based on definition of additional processes.

AO4BPEL[4] is one of the framework for realizing AOP of composite services. It allows a user to define a pointcut in a WS-BPEL process and weave a process described in WS-BPEL as an advice. This can add processes for adaptation without changing the model of a composite service.

For service-oriented collective intelligence, however, it is required to satisfy various policies of service providers. AOP is suitable for adaptation as described in the previous section, but it is not flexible enough to coordination such as controlling order of service execution.

The work proposed in [5] adopts a framework using a proxy. It checks if messages exchanged among a composite services and the constituent services satisfy the given conditions when the composite service execution engine invokes the constituent services. If any of conditions is not satisfied, it performs some recovering processes, retries invocation, or changes the service to an alternative[14]. But this focuses on adaptation of single service and does not deal with policies of all service provider concerned.

7 Conclusion

In this paper, we proposed a framework of execution control for composite services. The framework realizes service selection and adaptation in order to satisfy service providers' policies. Moreover, adaptation applied at runtime can be verified using model checking.

The contributions of this paper are as follows.

- We defined procedures which update description of model checking for each business logic control operation used for adaptation.
- We showed an algorithm which reduces the search space for verification by focusing on search states updated by business logic control operations.

Although there have been many previous works on adaptation and verification, they have not focused on verifying composite services which are adapted at runtime. This paper is the first work that tries to verify service providers' policies

are satisfied even when adaptation is applied by formally defining effects of execution control for adaptation.

For future works, we are going to propose generation of adaptation process which satisfies service providers' policies based on formal definition of execution control APIs and business logic control APIs.

Acknowledgments

This works was partially supported by Strategic Information and Communications R&D Promotion Programme from Ministry of Internal Affairs and Communications.

References

1. Ishida, T.: Language Grid: An infrastructure for intercultural collaboration. In: IEEE/IPSJ Symposium on Applications and the Internet (SAINT 2006), pp. 96–100 (2006)
2. Zeng, L., Benatallah, B., Ngu, A.H., Dumas, M., Kalagnanam, J., Chang, H.: Qos-aware middleware for web services composition. IEEE Transactions on Software Engineering 30, 311–327 (2004)
3. Hassine, A.B., Matsubara, S., Ishida, T.: A constraint-based approach to horizontal web service composition. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 130–143. Springer, Heidelberg (2006)
4. Charfi, A., Mezini, M.: AO4BPEL: An aspect-oriented extension to bpel. World Wide Web 10(3), 309–344 (2007)
5. Baresi, L., Guinea, S., Plebani, P.: Policies and aspects for the supervision of BPEL processes. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 340–354. Springer, Heidelberg (2007)
6. Tanaka, M., Murakami, Y., Lin, D., Ishida, T.: Service supervision for service-oriented collective intelligence. In: IEEE 7th International Conference on Services Computing (SCC 2010) (to appear 2010)
7. Nakajima, S.: Model-checking behavioral specification of bpel applications. Electronic Notes in Theoretical Computer Science 151, 89–105 (2006)
8. Tanaka, M., Ishida, T., Murakami, Y., Morimoto, S.: Service supervision: Coordinating web services in open environment. In: IEEE International Conference on Web Services (ICWS 2009), pp. 238–245 (2009)
9. Business process execution language for web services (BPEL), version 1.1 (2003), `http://www.ibm.com/developerworks/library/ws-bpel/`
10. Holzmann, G.: The SPIN Model Checker. Addison-Wesley, Reading (2004)
11. Narayanan, S., McIlraith, S.A.: Simulation, verification and automated composition of web services. In: The 11th International Conference on World Wide Web (WWW 2002), pp. 77–88 (2002)
12. Ankolekar, A., Paolucci, M., Sycara, K.: Towards a formal verification of owl-s process models. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 37–51. Springer, Heidelberg (2005)
13. Fu, X., Bultan, T., Su, J.: Analysis of interacting bpel web services. In: The 13th conference on World Wide Web (WWW2004), pp. 621–630 (2004)
14. Mosincat, A., Binder, W.: Transparent runtime adaptability for bpel processes. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 241–255. Springer, Heidelberg (2008)