An Architecture for Autonomic Web Service Process Planning

Colm Moore and Ming Xue Wang and Claus Pahl

Dublin City University, School of Computing, Dublin 9, Ireland
`christopher.moore4@mail.dcu.ie, [mwang|cpahl]@computing.dcu.ie`

Abstract. Web service composition is a technology that has received considerable attention in the last number of years. Languages and tools to aid in the process of creating composite web services have been received specific attention. Web service composition is the process of linking single web services together in order to accomplish more complex tasks. One area of web service composition that has not received as much attention is the area of dynamic error handling and re-planning, enabling autonomic composition. Given a repository of service descriptions and a task to complete, it is possible for AI planners to automatically create a plan that will achieve this goal. If however a service in the plan is unavailable or erroneous the plan will fail. Motivated by this problem, this paper suggests autonomous re-planning as a means to overcome dynamic problems. Our solution involves automatically recovering from faults and creating a context-dependent alternate plan.

1 Introduction

The Semantic Web is an emerging technology that creates some opportunities in the field of Web services. Automatic composition of semantically described services is an example. Sequencing services together to accomplish more complex tasks can create difficulties when automated at runtime. AI planners can provide a solution in the form of a plan (often a sequence of Web services required to solve the problem at hand). Once these plans have been made, composite web services can be generated and invoked. However, what will happen if a service becomes unavailable or is not functioning properly? As a solution to this problem, we suggest an execution, monitoring and re-planning architecture.

A second component is the service process generation, which creates an executable process from an abstract plan. The component must convert the plan into an executable process. Using this information, a composite Web service is constructed that can communicate with the services in the plan and execute them in order. This service process needs to be deployed on a Web server and then invoked by the execution component.

If an expected result is returned it means that a Web service has executed without problems. If, however, the fault handling mechanisms indicate an error has occurred, other action must be taken. The third and central component is a monitoring and analysis that detects execution problems and analyses possible remedies. Re-planning results in a new plan that contains alternate Web services that can also accomplish the same task. It is necessary for our program to get an alternate plan from the planner and start the execution process again.

A number of papers discuss the automation of service composition. McIlraith and Son [6] use the AI planner Golog [5]. Golog is a logic programming language based on the Situation Calculus, build on top of Prolog. Other planners, like hierarchical task network planners such as SHOP2, are based on the situation calculus. When composing Web services, high level generic planning templates (subplans) and complex goal can be represented by Golog. While these approaches can provide acceptable plans, this technology needs to be adapted to a dynamic environment. We have already identified two components of an architecture – process conversion and process monitoring and analysis – that can accomplish this integration.

An outline of the entire autonomic planning process follows in the Section 2 and introduce service composition, planners and process execution. Section 3 details the autonomous process planning. In Section 4, we introduce the overall system architecture, which is subsequently discussed in detail in terms of plan execution (Section 5) and monitoring and replanning (Section 6). We end with a discussion and some conclusions.

2 Dynamic Composition and Planning

In order to derive from Web service description an executable composite Web service automatically at runtime, a number of steps and transitions are required. Two central activities are plan generation based on abstract goals and service descriptions and plan conversion for execution through an execution engine.

2.1 Service Composition and Process Planning

A crucial step is to create a plan from service descriptions. AI planners are tools that are used to determine a plan, which is composed of a series of actions, an initial state, a goal state and a set of possible actions. SHOP2 is a Hierarchical Task Network (HTN) planner. In HTN planners, each state of the world is represented as a set of atoms with actions corresponding to deterministic state changes [7]. The planning domain is represented by operators (tasks) and methods. The methods decompose a set of complex tasks into subtasks. The plan is a sequence of these tasks. In the case of Web service composition, services are represented as operations. Inputs and outputs of services are represented as preconditions and postconditions joined with other semantic information. The plan is a sequence to execute the services in order to achieve the predefined goal.

The first requirement is to define a goal or overall task that is required. The goal is the desired outcome from the system once it has finished executing. This goal will usually require a series of Web services executions and, most likely, a number of message transactions. For a simple example, the purchase of a book would require first the lookup of stock to make sure the book is available and then the credit transaction. The Web service information gathered will be automatically translated into an AI planner interpretable language from a knowledge-based language such as OWL-S or WSMO [1]. The converted file then contains service information (input/output, pre-/postconditions of operations). The plan is initially not in an executable format. WS-BPEL is a language that allows for the composition and invocation of Web services. WS-BPEL engines are composite service executors. WS-BPEL connects to WSDL directly and provides error handing mechanisms.

Problems can occur during the execution of these processes. Web services are often not reliable, which affects both the composition and execution activities. Web service can become unavailable for many reasons. If this happens between discovery and invocation, the goal becomes unachievable. Using error handling and re-planning it is possible to recover from problems.

2.2 Web Service Composition

Web services are platform-independent Internet-accessible software components [10]. WSDL files describe the service and how to connect and interact with it. Web service composition is the linking of Web services to perform some new complex task. WS-BPEL (Business Process Execution Language) is an orchestration language used to define business processes based on Web services. It controls message passing and execution of the process. The message handling in WS-BPEL refers to WSDL to define how the incoming and outgoing messages are handled. WS-BPEL defines how the services can be scheduled and organized into an executable process that provides an integrated service [8]. WSDL files are defined as ”partnerLinks” where their role in relation to the WS-BPEL file is determined.

3 Autonomous Service Process Planning

The purpose of this investigation is to address dynamic re-planning in Web service composition. This involves using the outlined technologies to actually build a system dynamically and automatically. This system must be capable of creating plans, converting them to a usable language and then executing them. In addition, the system must detect and handle errors from faulty Web services and then automatically create a new alternate plan. The context of the system determines the quality and consequence of errors.

3.1 Service Description

We use a book search feature as our running example. Four OWL-S files describing four basic services define the service repository used here. There is service to find information on a book given a title, two alternative services that find the price of the book from an ISBN number and a service that converts the price from one currency to another. The goal of the problem is to get a price for a

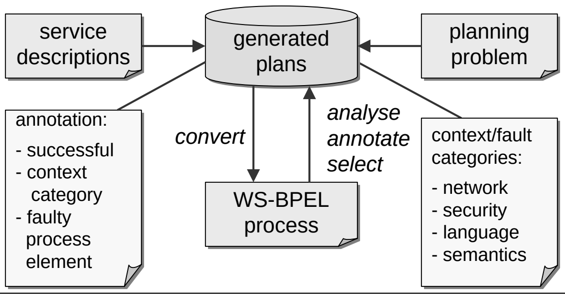

Fig. 1. Information Architecture

book in a given currency from the title of the book. We assume the four services as the result of a discovery activity.

```
<rdf:RDF xml:base="BookFinder.owl">
  <owl:Ontology rdf:about=""> ... </owl:Ontology>
  <!-- Service, Profile, and Process descriptions -->
   <process:AtomicProcess rdf:ID="BookFinderProcess">
    <service:describes rdf:resource="#BookFinder"/>
    <process:hasInput rdf:resource="#BookName"/>
    <process:hasOutput rdf:resource="#BookInfo"/>
  </process:AtomicProcess>
  <process:Input rdf:ID="BookName">
    <process:parameterType rdf:datatype="..">string</process:parameterType>
    <rdfs:label>Book Name</rdfs:label>
  </process:Input>
  <process:Output rdf:ID="BookInfo">
    <process:parameterType rdf:datatype="..">Book</process:parameterType>
    <rdfs:label>Book Info</rdfs:label>
  </process:Output>
  <!-- Grounding description -->
</rdf:RDF>
```

3.2 Planning

A planner generates an execution plan based on a given planning domain and planning problem, see Fig. 1. In SHOP2, the planning domain is established by a set of operators and methods. The input and output of the services are represented as preconditions and postconditions, respectively. For example, the book lookup service requires a book title to function; for the operator this would have a precondition that requires a BookName element to be accessible.

SHOP2 operator definitions consist of different parts: preconditions, which guards the operator execution. A delete list for negative postconditions and a add list for positive postconditions.

```
   (:operator (!BookFinderService)
    ( (BookName ?bookName) )    ; preconditions
    ()                          ; negative postconditions
    ( (BookInfo bookInfo) ) )   ; positive postconditions
```

This SHOP2 interpretable code shows the BookFinder operator. The precondition is that there is a book name available. There is nothing in its delete list and its add list contains BookInfo (information about the book). Once the operation is executed, the process will have the variable BookInfo available.

In addition to operators, planning methods define how composite tasks are decomposed. A simple method includes a precondition and the subtasks that need to be accomplished in order to accomplish the composite task.

```
(:method (GetBookPrice)
 ( (BookInfo ?bookInfo) (Currency ?currency) )
 ( (!BookFinderService) (!AmazonPriceService) (!CurrencyConverterService) )
```

If preconditions are satisfied, the method decomposes GetBookPrice into subtasks, composed of BookFinderService, AmazonPriceService and CurrencyConverterService. A second GetBookPrice method has a different ShopPriceService.

3.3 Goals and Plan Creation

In addition to the operator and method input files, a planning problem file is created that represents the goal of the plan. When the Java version of SHOP2 executes, it takes the two files and converts them to Java, which can subsequently be executed to attain the plan. As there are alternate services available that can implement identical functionality as defined in the methods, there can be multiple plans. In the example GetBookPrice, when book name and a desired currency format is available in the initial state, SHOP2 returns two separate and both equally valid plans for the book price conversion goal:

```
Plan 1: BookFinderService; AmazonPriceService; CurrencyConverterService;
Plan 2: BookFinderService; BNPriceService; CurrencyConverterService;
```

SHOP2 can create an indexed list of plans. Multiple plan generation is a central feature since it allows different alternative plans to be executed in case of failure without the need to re-start the planning itself. Plan 2 above is such an alternative. Differences between plans can be noted and future selection can be based on this. We create an index to a plan repository to enable efficient access.

4 An Execution, Monitoring and Planning Architecture

A monitoring system with two components is the backbone, see Fig. 2:

– The first component is an autonomous plan execution component. Its aims are: conversion of abstract plans into executable service processes, pre-execution preparation of the execution environment including service description and deployment files, but also context fault-handling determination in addition to plan conversion, execution of the process using a service process engine.

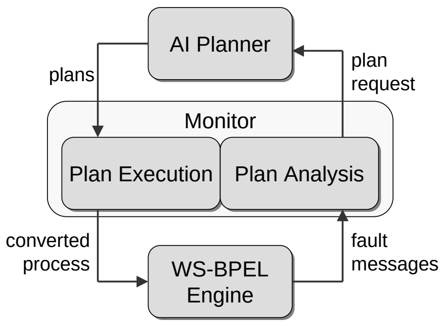

Fig. 2. System Architecture

– The second component is the context-dependent replanning component. Its objectives are: monitoring of process execution and fault capture, analysis of faults that have occurred during execution and determination of remedies (includes use of alternate existing plans or restart of planning process).

The necessary infrastructure consists of an execution engine at the core. The WS-BPEL execution engine that is used in this project is ActiveBPEL. It is an open source project written in Java. In terms of choice, the two most popular open source engines are Apaches ODE and ActiveBPEL. In terms of performance, the Apache engine has the advantage. ActiveBPEL however, provides excellent support for its engine, including many online guides and an actively monitored forum. In terms of the infrastructure, additionally Ant scripts add files to ActiveBPEL deployment folder. To simplify the integration of the planner into the architecture, the use of the Java version of SHOP2 called JSHOP2 is used instead of the Lisp version. The planner creates Java files to represent the problem/goal and the service description data.

5 Autonomous Plan Execution

5.1 Plan Conversion

Plan conversion involves two activities: conversion of the SHOP2 generated plan to a WS-BPEL representation and provision of input WSDL services and WS-BPEL deployment files for the BPEL engine. As part of the actual conversion of the plan into an executable process in WS-BPEL, a number of files need to be created. These are the WSDL files of the Web services that the plan requests to be invoked, the WSDL file of the generated WS-BPEL process and a number of deployment files, which are created by the WS-BPEL deployment tool.

5.2 Plan Execution

Plan execution – the second subcomponent – involves two activities: execution of WS-BPEL code and input OWL-S to XML parsing, which is done on the fly. As the sample data originates from a number of OWL-S files, it is necessary to search through these to determine the location of the WSDL files which are needed for the WS-BPEL process, as WS-BPEL does not refer to OWL-S directly. This is done through XML parsing. Once the location is found, the WSDL file is analyzed and relevant information is selected. Information such as the how to connect, what message formats are needed, the names of services and others details are vital for the correct invocation of a service by the WS-BPEL process.

Creating the WS-BPEL file and its "partnerlink" WSDL file is done automatically. WS-BPEL files contain a number of sections which each have a particular role, sections such as partnerLinks, variables, faultHandlers and flow. These sections are made up individually and added to the file as they are required. Each section containing a template for standard layout in a section with relevant information simple is inserted as required. Information about Web service invocations is taken from the relevant WSDL file.

Here is a brief structural outline of the WS-BPEL specification:

```
<process>
   <partnerLinks>
      <partnerLink   name="BookFinder"
                     partnerLinkType="print:FinderLink"
                     partnerRole="BookFinderProcess"/>      ...
   </partnerLinks>
   <variables>
      <variable      name="BookName" ...  />      ...
   </variables>
   <flow>
      <invoke> partnerLink="BookFinder"
               operation="find" inputVariable="BookName" </invoke>
      <invoke> partnerLink="BookPriceCalc" ... </invoke>
      <invoke> partnerLink="PriceConvert" ...  </invoke>
   </flow>
<process>
```

Once WS-BPEL is created, it is deployed. Using the ActiveBPEL execution engine, deployment involves using Apache's Ant. This causes the files to be added to the ActiveBPEL's deployment folder and then deployed once it is noticed.

The deployed WS-BPEL service can be invoked from a manager component. Values are passed to the service; in our example the values would be the name of the book and information about the currencies needed. Once this invocation is made, the WS-BPEL process begins to execute its Web service references.

6 Monitoring and Context-dependent Replanning

6.1 Fault Handling

A vital element of WS-BPEL is fault handling. This is important due to the possibility if failure, but essential to our context to achieve autonomy. Fault handlers can be defined in WS-BPEL to handle the exceptions thrown when a process is executing. Adding handlers to the invocations of Web services allows us to catch a fault when it arises. When a fault occurs and fault handlers have been defined, we use handlers to determine remedies in order to achieve the overall execution goal. Technically, a reply message indicating the fault is send by the handler (part of the execution engine) to the monitor (a separate component).

6.2 Context

In order to structure the failure handling aspect, possible failures are organised into context categories. The context notion refers here to execution environment factors that might impact the execution (and result in failure).

Context constraint violations need to be analysed and solutions determined. We distinguish for this implementation a number of (not necessarily exhaustive) context constraint violation categories:

- non-responsiveness of services: the service invoked does not respond
- security: a desired level of security cannot be achieved
- performance: the requested service cannot deliver efficiently enough

6.3 Analysis

The analysis component determines the actions to be taken from a failure in order to achieve the overall goal. It carries out the following steps:

- analysis of context constraint violation: an initial configuration can indicate whether violations of constraints are acceptable,
- a short planning cycle is necessary if violations are not acceptable: the analysis component detects previously generated plans (using the plan index) that can be tried as a remedy.
- a full planning cycle is necessary if violations are not acceptable and previously generated plans are not suitable (or not available): an invocation of the planner with the original goal is necessary.

Clearly crucial here is the decision whether a a time-consuming replanning (and possible service discovery) is necessary or whether an existing alternative plan can be used. This decision is context- and state-dependent. We annotate the indexed plan repository as follows: successful plan completion rate (probability of successful execution), fault type and associated context category, fault-generating plan/process elements. The plan annotation actually allows sets of fault type / process elements as a plan execution can cause different faults.

By distinguishing short- and full-cycle replanning, we achieve an improvement of planning performance; repeated generation of unsuccessful plans is avoided. The plan repository is updated (annotated unsuccessful ones) In the future, we aim to improve the annotation and analysis of unsuccessful plans. We plan to implement a learning technique that reliably allows to determine plans with a high degree of success from a plan repository. Clustering of faults/context categories and fault-causing elements is at the core of this endeavour. Our observation so far is that the success probability depends on the context category.

6.4 Implementation

Our WS-BPEL process has a number of fault handlers defined – corresponding to the context categories under consideration. In the case of an inaccessible service for instance, an error will occur. At this point the fault handlers take over. An automatic reply is sent to the monitor. If this message is a fault message and it indicates a non-responsive service the analysis component is called. It knows the plans that have already been produced and which of those have been (unsuccessfully) executed. It takes the next plan from the AI planner.

7 Discussion and Related Work

The solution that we implemented through our prototype indicates that an autonomic composition approach is feasible. Some concerns have, however, arisen.

A challenge that we encountered was the correctness of the conversion of a Web service composition plan into an actual working service process. Plans are abstract instructions, whereas WS-BPEL is executable process language with binding and deployment information. Information gathered, interpreted and converted to the correct format. This would include creating the WSDL files (from an OWL file) and extracting the data from these files to define a process that complies with the plan specification.

We have already discussed that performance is crucial and that we have provided a solution that targets plan reuse without replanning whenever possible. Improvements in this respects are, however, still possible. We mentioned an intelligent, context-dependent plan selection feature as a promising direction. We have focussed on communications-specific fault categories in our context definition. A range of other context aspects such as language, semantic context, a full range of quality criteria, etc. can be considered.

Many planning tools have been integrated into autonomic composition architectures. In [6], Golog is used as the planning component. In [7], with SHOP2 the same planner that we used is proposed based on OWL-S semantic Web service descriptions. [9] applied planning using a model checking approach. The plan generation is done by exploring the state space of the semantic model. In a recent hybrid AI planner [3], different planning techniques are combined. The major focus of these activities is discovery and service composition. However, they are lack fault-tolerance, which in distributed service infrastructures is a necessity for reliable implementations.

Many researches are looking into self-healing mechanisms [4] for service composition to achieve dependable systems. The self-healing approach focuses on monitoring and recovery activities for overcoming faulty behaviours of service oriented system. In [2], a self-healing composition strategy is defined, which includes assertion-based monitoring, event-based monitoring, history-based monitoring, recovery through a retry-failure service, recovery through a substitute-failure service, and recovery by restructuring plans. [11] presents an enhanced BPEL engine for self-healing. The engine is extended by planning, monitoring, diagnosis and recovery modules. However, none of these activities provides a complete architecture solution for autonomic service composition.

8 Conclusions

In this paper, the problem of dynamic Web service composition and execution failure and error handling and re-planning has been addressed. The causes of this problem and the effects have been discussed. An architecture for autonomic, i.e. dynamic and automated service composition has been discussed.

One of the crucial characteristics of autonomic composition is a self-healing ability of the dynamically deployed composition system. It needs to deal with execution faults of a very different nature. We have proposed a context-based fault handling strategies that efficiently determines remedies in terms of reuse of plans or AI-based replanning and subsequent plan conversion. As indicated, our aim is to extend the current system by considering more context categories and to make the decision processes more efficient and reliable.

References

1. OWL-S Coalition. *OWL-S 1.1.* http://www.daml.org/services/owl-s/1.1, 2003.
2. S. Guinea. Self-healing web service compositions. *27th International Conference on Software Engineering*, 2005.
3. M. Klusch and A. Gerber. Semantic web service composition planning with owls-xplan. *1st Int. AAAI Fall Symposium on Agents and the Semantic Web*, 2005.
4. P. Koopman. Elements of the self-healing system problem space. *Workshop on Software Architectures for Dependable Systems*, 2003.
5. H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R.B. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–83, 1997.
6. S. McIlraith and T. Son. Adapting golog for composition of semantic web services. *Eighth International Conference on Knowledge Representation and Reasoning (KR2002)*, pages 482–493, 2002.
7. D. Nau, T. C. Au, O. Ilghami, U. Kuter, W. J. Murdock, D. Wu, and F. Yaman. Shop2: An htn planning system. *Journal of Artificial Intelligence Research*, 20:379–404, December 2003.
8. L. Padgham and W. Liu. Internet collaboration and service composition as a loose form of teamwork. *Journal of Network and Computer Applications*, 30(3):1116–1135, 2007.
9. M. Pistore, P. Bertoli, E. Cusenza, A. Marconi, and P. Traverso. Ws-gen: A tool for the automated composition of semantic web services. *3rd International Semantic Web Conference*, 2004.
10. B. Srivastava and J. Koehler. Web service composition - current solutions and open problems. *ICAPS'2003 Workshop on Planning for Web Services*, 2003.
11. S. Subramanian. On the enhancement of bpel engines for self-healing composite web services. *IEEE Symp. on Applications and the Internet*, pages 33–39, 2008.