Appl Intell (2012) 36:1–28
DOI 10.1007/s10489-010-0238-z

Dynamic planning approach to automated web service composition

Mehmet Kuzu · Nihan Kesim Cicekli

Published online: 8 June 2010

Abstract In this paper, novel ideas are presented for solving the automated web service composition problem. Some of the possible real world problems such as partial observability of the environment, nondeterministic effects of web services and service execution failures are solved through a dynamic planning approach. The proposed approach is based on a novel AI planner that is designed for working in highly dynamic environments under time constraints, namely Simplanner. World altering service calls are done according to the WS-Coordination and WS-Business Activity web service transaction specifications in order to physically recover from failure situations and prevent the undesired side effects of the aborted web service composition efforts.

Keywords Semantic web services · Automated web service composition · Automated web service invocation · AI planning · Simplanner

1 Introduction

The web is growing very fast and a significant number of web services have become available to be used. Although a large number of web services exist, individually they are not always sufficient to satisfy the user's needs. The absence of a web service that responds to the user's request does not mean that the request cannot be handled. Web services can collaborate to satisfy user requests, that is, a new functionality can be produced by composing the existing functionalities to handle user requests. Since the number of web services and possible collaboration scenarios is huge, analysing them manually for the purpose of achieving user goals is very difficult and beyond human ability. This web service composition process could be done automatically by software agents.

The main goal of this paper is to propose a framework which provides the ability to automate the web service composition task in an effective manner in terms of time and adaptability to the real world environment. The aim of the proposed framework is to find several web services and execute them according to the discovered execution order for handling the user's request, given a repository of services and the user's goal. There are two commonly accepted approaches for solving the web service composition problem: workflow composition and AI planning [1]. In this paper the AI planning approach is adapted for the construction of a web service composer agent.

The automated web service composition problem is a hot research topic. There are excellent surveys [1–5] about the challenges of web service composition (WSC). Although many important studies have been conducted in order to solve this challenging problem, there still exist many open issues. Several open issues about the proposed WSC solutions are summarized in the literature [3, 6]. This paper aims to solve some of the stated problems in [6]. The paper provides solutions to the following problems:

- If unexpected events that cause failures occur, replanning and compensation mechanisms are required.

In this paper, this problem is solved in two levels. In the first level, which is the abstract logical level, a novel AI planner, namely Simplanner [7] which is specially designed for

M. Kuzu
Department of Computer Science, University of Texas at Dallas, Richardson, TX 75080, USA
e-mail: mxk093120@utdallas.edu

N.K. Cicekli (✉)
Department of Computer Engineering, Middle East Technical University, Ankara 06531, Turkey
e-mail: nihan@ceng.metu.edu.tr

highly dynamic, nondeterministic environments is used. The proposed solution interleaves planning and execution. The planner keeps track of the current state by using the initially provided semantic state descriptions and the semantic effects of the executed actions. If something unexpected occurs, such as service unavailability, the web service composer informs the planner about the situation and the planner then changes the current state according to that information and activates dynamic replanning for handling the unexpected situation. In order to cope with the unexpected situations, just replanning is not sufficient. During the real execution of a plan, service failures may occur in later steps and previously executed services may have world altering effects which must be undone or compensated. The second level is the physical level, where failure recovery is provided by using the WS-Business Activity framework [8]. Ours is the first work that combines transactional execution and replanning in the same framework for the automated WSC.

- The environment that is available during planning may not be the same as the environment during execution, which is problematic. Interleaving planning and execution is desirable.

In this paper, the proposed web service composer interleaves planning and execution. Simplanner is an anytime planner. In the anytime planning approach, the planner constructs a very quick initial solution which may include wrong decisions and improves the initial solution if time permits [7]. Simplanner concentrates not on the total plan but on the best initial action for the solution. The service composer requests the best initial action from the planner and does the real execution after doing the other required steps. During real service execution, the planner continues trying to find a new best action. Generally the service execution time is sufficient for the planner to produce the new best action. As a result, interleaving planning and execution is done in a timely manner by using the features of Simplanner.

- Web services may have nondeterministic effects which should be tackled.

Actions may have different effects. Which effects will become true can only be found out during execution. (For instance, before the execution of a reservation service, it is not known if there is an available place or not. The behaviour of the service execution depends on the factors that can be decided at run time). If services have nondeterministic effects, they can be observed by the service composer after the real execution and a state change request can be made to the planner. The planner changes the state according to the effect and continues to plan taking that effect into consideration.

- Service descriptions are provided once and they are not updated frequently. Availability checks should be done from time to time.

The proposed solution initially assumes that, all service descriptions are valid and that the preconditions of action definitions are updated with a logical statement that shows the availability of that particular action. After a real service call, if there is a problem with the service, the service composer informs the planner and makes the logical counterpart of the problematic service unavailable. As a result, the planner does not consider that problematic service for further decisions and tries to solve the problem using alternative services.

- It is possible that web services may produce new objects at run time. It is very difficult to model dynamic object generation with the current AI planners. Therefore a new mechanism should be provided to handle such cases.

The planner can only propose logical actions to execute if the preconditions of those actions are satisfied and the required logical objects for grounding the logical actions are available. It is possible that such logical objects are not available in the initial state but can be produced by other services. In such a case, the service composer constructs a logical object for each defined type and adds a logical statement to the initial sate in order to assert that users can provide the details of logical objects at run time. If the value of the service parameters cannot be provided by the user, the planner tries to find a web service that supplies the values for those objects and changes the plan accordingly.

- Semantic web service descriptions should be connected to real web services.

Semantic web services are web services whose properties and capabilities are described in machine understandable format. Semantic web service descriptions are high level definitions that are used by the AI planner in this paper. For real web service interaction, syntactic counterparts of these semantic service descriptions are needed. The bridge between the syntactic and semantic definitions is provided by using the definitions stated in the grounding section of OWL-S [9] descriptions. The actual values of the syntactic counterparts are obtained either from the user or from other web services. By substituting the actual values for the syntactic interfaces through the reflection mechanism [10], the service composer performs the real interaction with web services.

- User interaction is required during WSC.

During WSC, service execution is needed in order to validate the found plan and act accordingly if something goes wrong. During service execution, inputs to the services should be provided by the users. If they do not know the required input values, some other paths should be found which does not require that particular input.

This paper tries to solve the open issues about automated web service composition and invocation problems mentioned above. To the best of our knowledge this is the first work that proposes complete solutions to these problems in one framework.

The organization of the paper is as follows: In Sect. 2, related works that have been conducted for the solution of the WSC problem are described and their strengths and weaknesses with respect to this work are discussed. In Sect. 3, an adapted AI planner, namely Simplanner, is introduced. In Sect. 4, the general system architecture and the application of Simplanner to the WSC domain is described. In Sect. 5, issues related to automated web service invocation are discussed and an action caching mechanism is proposed. In Sect. 6, transactional issues are discussed and the integration of WS-Transaction frameworks to the system architecture is presented. In Sect. 7, a case study is conducted to clarify the functionality of the proposed system. Finally, in Sect. 8 the paper is concluded and future work is discussed.

2 Related work

There exist several previous studies in automated web service composition with AI planning. In this part of the paper, the most relevant of these studies are briefly described and compared to this work.

The Hierarchical Task Network (HTN) planning approach is adapted for web service composition domain in [11]. The SHOP2 [12] HTN planner is used for finding plans to achieve the user goals. Although HTN planners have very important performance advantages, they need a reasonable amount of domain knowledge, which is a strict requirement. In [11], the execution and planning is interleaved as in our work, but with limitations both in DAML-S service descriptions and in the procedure. They execute only information providing services during plan generation so that the world state is not altered during plan generation. Executing world altering effects is not acceptable before the total plan generation, since it may cause unintentional results if the whole plan cannot be achieved. In our work we have solved this problem by using WS-Coordination [13] and WS-Business Activity [8] frameworks which guarantee the atomicity of composed services. In our system there is no need to have any assumptions for the DAML-S descriptions; instead our system considers only those services that implement the WS-Transaction specification if it is a world altering one.

In [14], a WSC model is proposed based on XPlan and OWLS2PDDL conversion. XPlan is a hybrid planner that is built on an action based Fast Forward planner [15] with an HTN component. OWLS-XPLAN [14] finds the whole plan before execution so it does not work well on partially observable domains and it is not a time efficient solution. It also does not handle nondeterminism that is naturally available in web services. If a found plan is interrupted due to a reason such as a network failure, a new plan needs to be generated from scratch. More importantly, if some actions with world altering effects are executed before the interruption of the plan, inconvenient and undesired states may be constructed. Our work also includes OWSL2PDDL mapping, but with few extensions that are necessary in nondeterministic environments. In contrast to [14], this paper interleaves planning and execution. In addition, the transactional properties are preserved to prevent undesired states. However, [14] is more scalable than our solution since XPlan contains HTN components.

In [16], OWL-S XPLAN is extended with a replanning component. Replanning occurs in the cases of a new operator, a lost operator or a new goal. All of these situations are handled by Simplanner in our system. In [16] replanning cases are defined in general, not connected with the web service composition problem. The execution process of web services is not described. Without real interaction it is hard to know about the lost operators. One of the most frequent reasons of replanning is unknown information (e.g. users may not know the required inputs of web services, so replanning may be needed), and this situation is not mentioned in [16].

In [17] and [18], PDDL is also used as the input language to the planner. Instead of using OWL-S, they constructed a new semantic description language, called SESMA [19] which is designed for WSC purposes. It has the same power as OWL-S with regards to its representational aspects but it is simpler. After mapping SESMA to PDDL, a different planner is selected depending on the application problem. In [17] it is mentioned that if resource optimization is required then the Metric-FF [15] planner is more suitable; if planning with typed variables and lifted actions is required then VHPOP [20] is more suitable; and if durative actions are required then the LPG [21] planner is better. Because of these facts, they propose to plug in any planner instead of using a particular planner. Although the idea seems good initially, it is not well defined. In order to use the differences between the planners, one needs to understand which one to use for a particular problem by inspecting the properties of the problem. This is a very challenging task for machines and it is not examined in [17]. The work discusses nondeterminism in WSC but the proposed solution is a naïve one. Handling nondeterministic cases is ascribed to the application logic, which is totally contradictory to the service oriented architecture. They have also mentioned complex goal definitions that include selection and iteration constructs. However, the proposed solution is not a desirable one; they propose to embed these constructs into the application logic with hard coding. This paper has advantages over the work in [17] in terms of adaptability to dynamic and uncertain environments. However, complex goal definitions are not allowed in our framework.

The approach in [22] is different from the other existing work and this paper. The main focus is to construct new services by using the existing ones by means of BPEL [23] constructs. In order to construct a new service, first its functional (input, output, precondition, effect) and non-functional requirements (quality of service, time constraints, etc.) are described. Then a two-phase process is applied for service composition: logical composition and physical composition. Logical composition provides the functional requirements of the new service that are described through OWL-S. In the physical composition phase, concrete service matching is done. In the logical composition type matching is conducted. There may be more than one concrete service composition that provides such a type matching. The choice between multiple possible services is done in the physical composition phase according to the non-functional attributes that are stated in the new service request. The most attractive part of this work is filtering, where the irrelevant services are eliminated before planning. They mention an important experiment: On a problem with a 7-step plan, the planner can return a solution in 4 seconds if the filter is enabled when 100,000 irrelevant service types/actions are present. However it takes hours without a filter. Filtering determines the irrelevant actions with the specified goal, but the underlying algorithm about this filter and its computational complexity are not presented.

In [4], available approaches for service composition and execution are divided into four categories: interleaved, monolithic, staged and template based. According to [4], the interleaved approach lacks composition control and handling composition failures. However, our approach is an interleaved execution and composition which solves the composition failure problem completely. Composition control is the ability of the user to change the workflow. Our approach has a great potential to give flexibility to the user for controlling the composition. During the composition, users can see the outputs of services and according to those outputs they may want different services to handle their requests (services can easily be made unavailable and new services can be found by using the state change and replanning properties of system).

In [24], the importance of user interaction is discussed. They propose using domain knowledge to estimate if the user can provide the required information. This approach is domain dependent and does not present a recovery mechanism if the estimation is wrong. On the other hand, our approach initially assumes that all the required information can be provided by the user, and asks the user to provide data when required. If the user does not know the answer, dynamic adaptability and recovery properties of our system enables easy handling of such situations. As a result, our system involves the user during composition in a domain independent manner and resolves the problems automatically if initial the assumptions go wrong.

An important open issue in automated web service composition problem is to represent complex requirements. A formal language for representing the control flow and data flow requirements is presented in [25]. Our system does not allow defining complex requirements (i.e. conditional, iterative etc.). In [25], the service composition plan is found according to the control and data flow requirements and it is executed afterwards. It is almost impossible to represent all possible situations (dynamic state changes) that may arise during real execution a priori, so a system that decides on actions during interaction is more suitable for real world applications.

In [26], the general framework for WSC presented in [1] is extended with a workflow repository. Manually constructed workflows are considered as simple web services and can be used during composition. Our approach is comparable to that framework, since we also have a repository similar to the proposed workflow repository, but our repository contains previously found successful service composition flows instead of manually constructed workflows.

In [27], methodologies for recovery mechanisms for semantic web services are discussed. In case of a failure, they propose finding alternative services at run time instead of using the static information about the alternatives of particular services, which increases the chance of a successful recovery and process completion. Our approach proposes a real application of dynamic recovery, not only for service failures but also for the situations where the user cannot provide input for services. If a failure occurs before a compensation decision is made, our system tries to find alternative paths for solving the problem instead of firing compensation mechanisms immediately.

In [28] a different approach is presented for service composition. Entities and services are represented as resources and resource related actions; the interactions between the resources are handled automatically by the composer. The proposed solution requires a manual description of resources and resource interaction instead of considering the semantic service descriptions. Since users define only limited resources which they are interested in, the search space for the composition is small so the application is very practical to use in daily life. Our approach tries to solve the general service composition problem where resources are dynamically constructed by using the semantic web service descriptions and the relations between them are discovered automatically.

3 Simplanner

Simplanner [7] is a very suitable planner for the web service composition domain, since it provides the ability to handle partially observable, nondeterministic environments in a time efficient manner. Some features of Simplanner, such as its ability to deal with unexpected situations and the reactivity that it provides, are very valuable for the WSC domain. In this paper, these features of Simplanner are used and a service composition agent is constructed that is competitive with the ones that are available in the literature.

Simplanner is a kind of domain independent AI planner that is designed to operate on highly dynamic, partially observable environments with time limitations. Simplanner is an anytime planner, that is, it finds an initial solution to the presented problem very quickly and tries to refine the initial solution as time permits. Simplanner is also an online planner which makes it highly resistant to unexpected situations: the plan execution can start without a total plan being generated. Also, it allows for the modification of the current state to another desired state, which enables it to deal with incomplete information and unexpected situations. As a result, the required flexibility for the real world problems can be obtained.

The Simplanner algorithm is based on goal decomposition and searching with heuristics obtained from the relaxed planning graph (RPG) [29]. RPG is a rich source of information and it is used by most AI planners, and especially by the domain independent ones. The algorithm of the Simplanner contains three important steps: Relaxed Plan Graph generation, Subplan Construction, and Subplan Ordering. The details of the algorithm can be found in [7].

One of the most important features of Simplanner is its responsiveness in real time. Planning problems are generally very complex to solve and their computational complexity is generally exponential so it is very difficult to give a solution in a short period of time. There are different approaches to produce solutions in a timely manner, such as precompiled solutions to the problems and any time planning approaches. In most real world cases as in the WSC, using precompiled solutions is out of the question since there are millions of possible problems and solutions, so what is needed is to use an anytime approach like Simplanner. By using an anytime approach, Simplanner gives an initial solution in polynomial time which is reasonable and it continues to plan up to the execution point. The execution can start any time before the total plan generation. After each execution step, Simplanner considers the current state and produces a plan for the current situation [7]. Producing the initial plan in polynomial time does not mean that the planning problem can be solved in polynomial time. Anytime planning a provides quick reaction decision which is important for interactive applications that perform interleaved planning and execution and interact with users continuously.

Achieving any time planning is a very difficult task and generally requires an important amount of domain knowledge as in the HTN case that uses domain information for task decomposition. Using HTN based solutions is limited in most real world problems since there does not exist sufficient domain knowledge in most of the cases. What we need is a domain independent anytime planner for time critical operations. The novelty of Simplanner comes from the fact that it achieves anytime planning without the need of domain specific information. Simplanner continuously interacts with the custom executor logic. It provides high level logical actions to the executor and gets information about unexpected events from the executor. If an unexpected event occurs, the planner rejects the current plan and tries to find a new plan that is suitable for the current state. If everything goes well, the planner does not try to find a new plan from scratch but to improve the current plan by searching the state space as time permits [7].

Like other AI planners, the aim of Simplanner is to find a complete solution to the presented problem, but Simplanner also has another goal which is very important and provides its main distinction from other planners. Simplanner concentrates on the initial action but not on the whole plan because of its anytime principles. The algorithm of the Simplanner is based on depth limited heuristic search that can be interrupted at any time. If an interruption occurs, the planner returns the most promising action that will be used to reach the goal state. Otherwise it continues to provide better solutions [7].

Generally speaking planning problems require searching the whole state space for completeness which is in fact very time consuming and impractical. As many other effective planners, Simplanner uses some heuristics which work very well in practice. These heuristics are very useful for plan detection but if the plan does not exist for a particular problem, Simplanner starts to search the whole state space, which is not practical. When the state space search is started, an action may be offered by the planner several times. If the same action with the same logical parameters is offered again our system decides to abort the operation.

Without a complete state space search, we cannot say that a solution does not exist but the time complexity of searching the whole state space is exponential and cannot be used in real applications. Simplanner does not inform the application program when it starts to search the complete state space, so our system tries to infer this situation by examining the actions offered by the planner. As a result, the system may decide to abort when the planner starts a complete state space search in some necessary situations too (e.g. iterative service calls). However, such situations are infrequent so it is not a big problem for the time being. Our concern is not to make the solution complete but to use the heuristics when available.

We have examined many off-the-shelf planners for our purposes and finally decided that Simplanner is the planner that we need because of the mentioned properties. Although it is one of the most important components in the framework, it constitutes only a small part of the system. Simplanner mainly provides actions in the abstract level as it performs the planning.

The specific contributions of this paper related to the integration of Simplanner to the framework are:

- converting the abstract level actions to real interactions,
- representing the abstract state that handles information unavailability (user may not know the required inputs of services) and service unavailability (service may not respond),
- deciding about the state changes after real interactions.

It was not possible to use Simplanner directly in our framework, because Simplanner was written in Delphi and the proposed framework involves many components related to real service execution (dynamic code generation, service calls, transactional execution, etc.) many of which are handled with the available libraries written in the Java programming language. In order to connect the Delphi implementation of Simplanner with other components, the JNI interface of Java is not sufficient since Simplanner is a multithreaded application and JNI does not allow to access distinct threads at the same time. Therefore we re-implemented Simplanner in Java programming language from scratch in order to integrate it with other components of the framework.

4 Simplanner application to WSC domain

A high level view of interactions between the proposed automated web service composition core and external components is presented in Fig. 1. Simplanner works with the PDDL data format [30], as do most of the other off-the-shelf planners. Therefore, before using Simplanner, semantic web service descriptions and user requests are converted from OWL-S to the PDDL data format.

The service invocation component is integrated with Simplanner through communication interfaces. The actual service execution component requests high level logical actions from the planner. The execution component performs some processing on the provided actions, and informs the unexpected event handler about unexpected events, if any. An interface is provided between the planner and the unexpected event handler component. The handler requests state changes from the planner that are suitable to the current situation.

The proposed architecture provides a highly dynamic, interactive, flexible and time efficient service composition and invocation framework. The general web service composition process is composed of five phases which are *preprocessing*, *planning*, *action handling*, *executing* and *unexpected event handling*. A formal algorithmic description of the proposed system is presented in Table 1. The algorithmic details of the phases are presented in the forthcoming sections. (Third party software and noncritical implementation details are represented by function calls or skipped completely. Only important parts are included in the algorithmic description, function calls with bold font represent sub-components of the system).

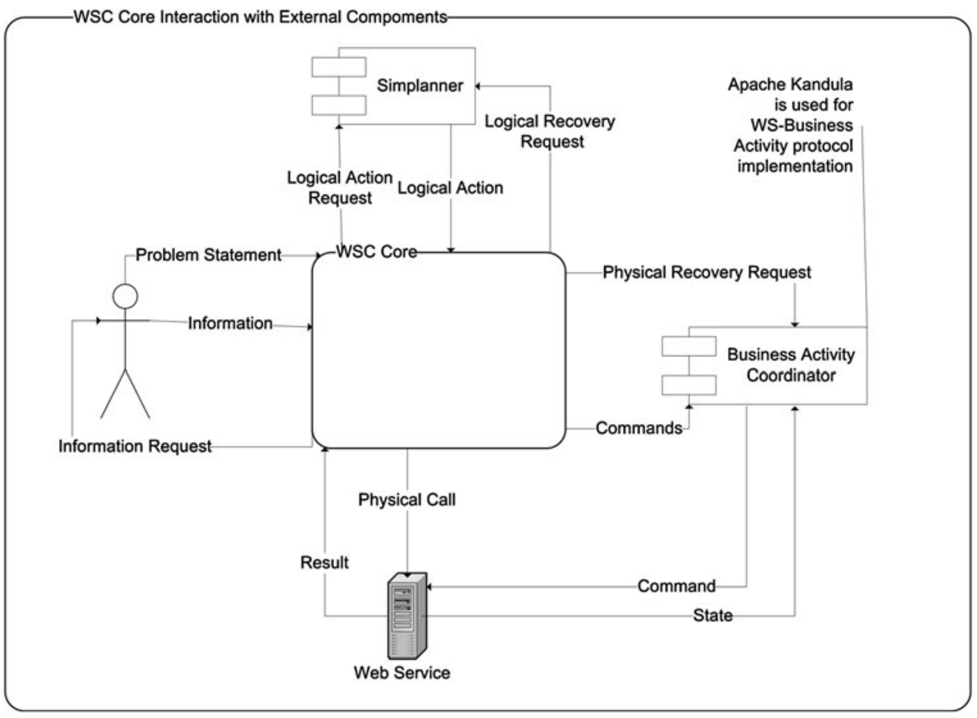


Fig. 1 Interaction with external components

Table 1 WSC algorithm

```
WS_Composer (repository)

OBTAIN initialState described with OWL constructs // end user is asked to provide the initialState
OBTAIN goalState described with OWL constructs // end user is asked to provide the goalState

[domain_PDDL , problem_PDDL, physicalMap, logicalObjects] ← PREPROCESS (repository, initialState, goalState)

// physicalMap contains the mapping between semantic action definitions that are extracted from OWL-S files and machine codes that
// implement the corresponding web service client. physicalMap also contains the syntactic type definitions corresponding to the semantic
// types. logicalObjects are the semantic objects that are obtained from the init section of problem_PDDL

LogicalPhysicalMap ← constructLogicalPhysicalMap(logicalObjects, physicalMap)
// logical/physical map contains the details of logical objects, such as their syntactic definitions and current value of their syntactic counters

currentState ← initialState;    //currentState contains the predicates that describes the current state, it is initialized by the initial state description
executedActions ← ∅             //executedActions keep track of the logical action list executed in this session

cachedActions ← loadCachedActions()    //cachedActions are the composite action definitions that are formed after successful WSC processes
businessActivityCoordinator ← construct BusinessActivityCoordinator() //coordinator for current WSC session, provides transactional execution
                                                                      //Apache Kandula project is used for WS-Business Activity protocol impl.
termination ← SUCCESSFUL

// variables constructed above are global variables and they are used by submodules

PLAN (plan_SIGNAL) //Simplanner is fired to work on the problem

WHILE (currentState != goalState) OR (termination = UNSUCCESSFUL)

    prev_state ← current_state //planner will change the current state during action proposal, if action cannot be executed physically
                               //state should be rolled back to its previous state, previous state is saved for possible uses later

    [currentState, action] ← PLAN (proposeAction_SIGNAL)

    IF executedActions CONTAIN action        // heuristic for termination decision, precise determination of plan inexistency requires
        termination ← UNSUCCESSFUL           // full space search which is too costly, iterative service calls are rare
    ELSE
        handleResult ← LOGICAL_ACTION_HANDLER(action) //construct physical counter of the logical action
        IF handleResult == SUCCESSFUL // physical counter is constructed, values for action parameters are collected successfully
            execSuccess ← EXECUTE (action, Execute_SIGNAL)
            IF execSuccess == SUCCESSFUL //service is successfully executed
                executedActions ← executedActions ∪ {action}     //update executedAction list with new executed act.
            ELSE    //problem occurred during service execution
              UNEXPECTED_EVENT_HANDLER(action, prev_state, UnavailableService_SIGNAL)
            ENDIF
        ELSE    //required parameters for calling web service cannot be obtained, service is not invoked
            UNEXPECTED_EVENT_HANDLER(action, prev_state, UnavailableInformation_SIGNAL)
        ENDIF
    ENDIF
END WHILE
IF termination == UNSUCCESSFUL //plan is not found
    UNEXPECTED_EVENT_HANDLER ( NoPlan_SIGNAL)
ELSE    //WSC session is successfully completed
    businessActivityCoordinator.closeAll()  //commit the transaction
    newLogicalAction ← constructNewAction (initalState, goalState, executedActions)  //save the steps of the found plan as a new action
    cachedActions ← cachedActions ∪ {newLogicalAction}                              //definition to use for future problems
ENDIF

PLAN(terminate_SIGNAL)  //stop the Simplanner thread
```

4.1 Pre-processing phase

The preprocessing phase prepares all the required objects for the upcoming phases. In this phase, the system uses semantic and syntactic service descriptions that are located in the service repository, the user provided domain ontology and the initial state and goal state ontologies. The graphical representation of this phase is given in Fig. 2 and a formal algorithmic description is presented in Table 2.

The framework produces the domain knowledgebase in PDDL by using the OWL-S semantic service descriptions that are provided by the service providers in the service repository, and the selected domain ontologies by the service requester. The user request consists of the representation of the initial state and the goal state in OWL [31]. The request can require the invocation of both world altering and information gathering services. The action definitions that are given in the problem statement (OWL constructs that presents the desired properties between OWL individuals) are considered high level, logical actions, since both actions and their parameters do not exist in the real physical world. Therefore, they cannot be used directly.

One of the important operations of this phase is the generation of physical counterparts of logical actions that are represented in the domain knowledgebase. In order to achieve this, first the OWLS-WSDL mapping is extracted from the grounding part of the OWL-S files in order to find the syntactic requirements of the domain actions and their parameters. In the physical state, the logical actions and their parameters are represented by machine interpretable codes and types respectively. By using WSDL [32] descriptions of services, client stubs are automatically generated which contain both service client implementations and complex type implementations that are defined in WSDL files by the service providers. As a last step PDDL action-Physical action mapping information is constructed using the automatically generated codes and the OWLS-WSDL mapping. This information is then used by the executor in the service execu-

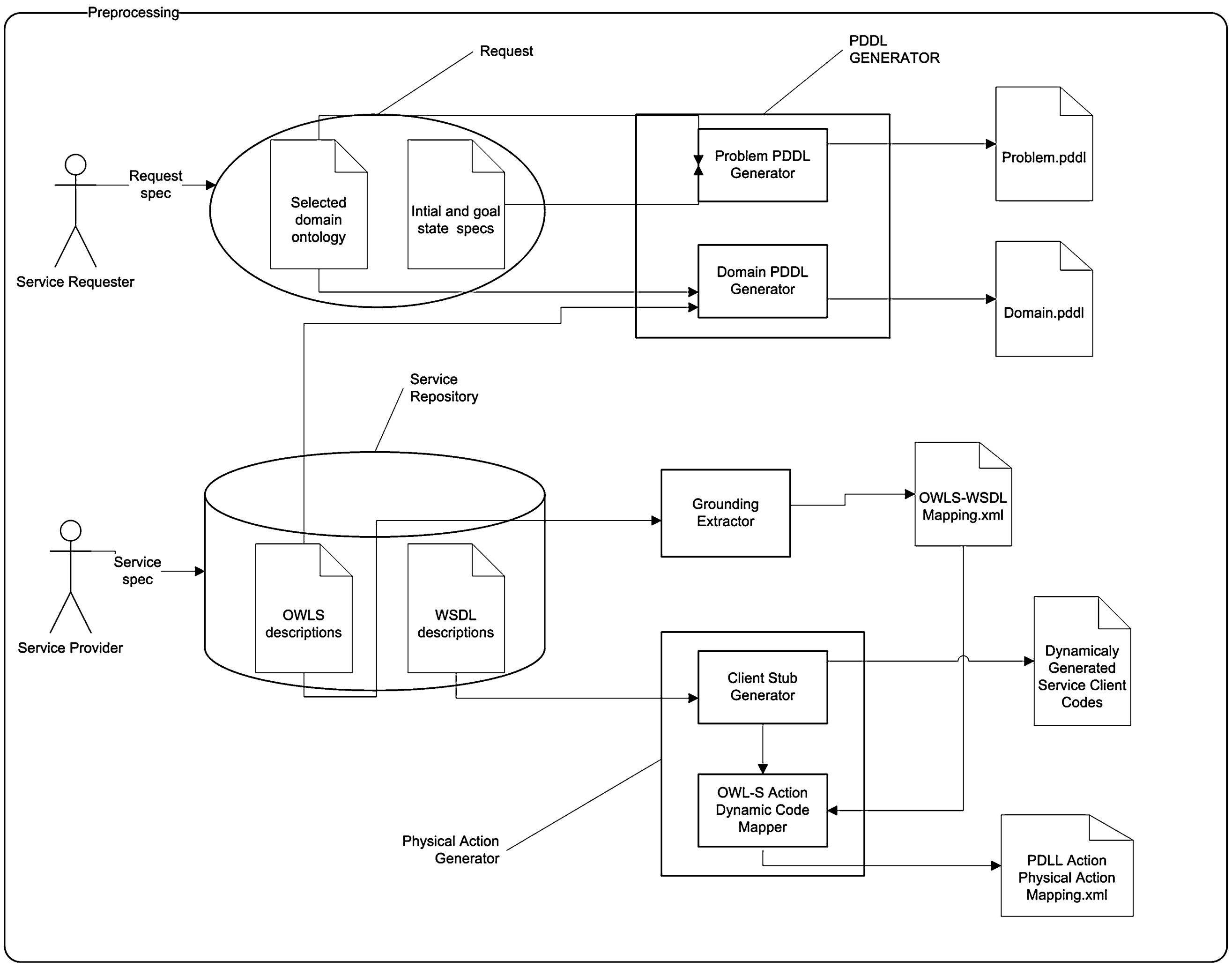


Fig. 2 Pre-processing

Table 2 Pre-processing algorithm

```
PREPROCESS (repository, initial_state, goal_state)

action_list ← ∅          //action,
predicate_list ← ∅       //predicate, and
type_list ← ∅            //type definitions that will be obtained by parsing the OWL descriptions in repository
physical_Map ← ∅         //mapping between semantic definitions and their syntactic counters

FOR EACH service serv_i in repository
    action_i ← owls2PDDL (owl-s definition of serv_i) //standard conversion rules are applied
    action_i.Precondition ← action_i.Precondition ∪ {"validService" + identifier OF serv_i} //each service should be operational
                                                                                   //precondition of action is updated

    FOR EACH input inp_i OF action_i //all input parameters should be known to execute action
        action_i.Precondition ← action_i.Precondition ∪ {"agentHasKnowledgeAbout" + identifier OF input_i }
    FOR EACH output out_i OF action_i //services provide information after successful execution if they have output
        action_i.Effect ← action_i.Effect ∪ {"agentHasKnowledgeAbout" + identifier OF output_i }

// domain information such as types and predicates are obtained by parsing the OWL definitions used in OWL-S service descriptors
    FOR EACH ontology ont_i used in OWL-S service definition OF serv_i
        [types,predicates] ← owl2PDDL(ont_i)
        type_list ← type_list ∪ {types}
        predicate_list ← predicate_list ∪ {predicates}

    action_list ← action_list ∪ {action_i}

init ← owl2PDDL( initial_state) //initial state is obtained from user, OWL is used for representation
goal ← owl2PDDL (goal_state) //goal state is obtained from user, OWL is used for representation

// Initial state contains the objects defined by the user to represent initial state and goal state, some additional objects may be required to execute
services that will be found by the planner later, for each available type construct a logical object*/

FOR EACH type typ_i in type_list
    IF init.Objects NOT CONTAIN any object with type typ_i
        obj_i ← constructObject (typ_i)
        init.Objects ← init.Objects ∪ {obj_i}
    ENDIF

// Logical objects will be used as inputs to the web services; if user has written a goal with "agentHasKnowledgeAbout obj_i", it means
// that the value of it is not known by the user; for other logical objects, initially assume that values can be provided by the user, if not
// appropriate actions will be fired in coming phases*/

FOR EACH object obj_i in init.Objects
    IF goal.Predicates NOT CONTAIN {"agentHasKnowledgeAbout" + obj_i}
        init.Predicates ← init.Predicates ∪ {"agentHasKnowledgeAbout" + obj_i}
    ENDIF

[domainPDLL, problemPDDL] ← constructPDDLDefinitions(action_list,predicate_list,type_list,init, goal)

FOR EACH service serv_i in repository

    mapEntry.machinecode_i ← wsdl2Java (wsdl definition of serv_i) //form service client codes and complex type definitions

    /*mapEntry contains mapping between logical types and physical types for each input and output of serv_i*/
    mapEntry.physicaltype_i ← typeMapping(owls grounding definition of serv_i, wsdl definition of serv_i)
    mapEntry.logicalAct ← identifier OF serv_i

    physical_Map ← physical_Map ∪ mapEntry //save mapping to physical_Map for later use

RETURN [domainPDLL, problemPDDL, physical_Map, init.Objects]
```

tion phase. The details of the execution issues are presented in Sect. 5.

The problem is provided by the user through logical statements using OWL individual definitions and relations between OWL individuals. The statements that are represented in the Web Ontology Language are translated into PDDL objects and PDDL initial and goal states according to the rules described in works [14, 33, 34]. In [14], a new predicate is generated for the PDDL representation of OWL-S action parameters, namely agentHasKnowledgeAbout(X) which is used to show information availability. The construct agentHasKnowledgeAbout(X) is necessary for the web service composition domain, especially for cases where the information is partially observable. In our work, this construct is used to decide whether the user or a web service is the source of information.

The transformation from OWL-S to PDDL is done based on the ideas presented in two other previous works [14, 33] with some modifications that are needed in nondeterministic domains. No particular language is determined for the representation of preconditions and effects by the OWLS specification and therefore it allows custom languages as well. SWRL, KIF and some other custom languages such as PDDXML can be used in order to represent preconditions and effects of actions. We have used PDDXML as in [14] in the proof of concept implementation; therefore the solution is not generic. During the transformation there is no information loss and the conversion is always possible but specific implementations are needed for particular precondition and effect languages. These conversion rules are not new; we have only two modifications to the available conversion rules.

First, we have added the predicate "valid[serviceID]" as a precondition to each service and we have added the predicate "valid[servicedID]" for all services in the init section of the generated PDDL. During the real service interaction, if a service does not respond, the predicate "valid[serviceID]", corresponding to the failed service, is removed from the current state and that service is not considered as a valid action afterwards.

The second modification to the rules is to use the idea presented in [14] in a more realistic way. In [14] the predicate agentHasKnowledgeAbout(X) is proposed in order to represent the information availability. The hasInput and hasOutput components of OWLS are converted as agentHasKnowledgeAbout(X) predicates in the precondition and the effect parts of the PDDL actions. agentHasKnowledgeAbout(X) is given manually in the initial state description of PDDL. This is not realistic, because it is not known initially which services are required, which inputs are required and whether the user can provide the required inputs for the solution of a given problem. In this work we have created one logical object for each PDDL type and we have added agentHasKnowledgeAbout(X) for each PDDL type in the initial state description of PDDL (the same logical type can be used for multiple physical calls). If the user can not provide a particular input (say X) during the real service interactions, the system removes the predicate agentHasKnowledgeAbout(X) from the state and the planner starts searching for alternative ways (i.e. it tries to find either another plan which does not require that service or another service which provides the missing input). In summary, "valid[serviceID]" and the practical usage of "agentHasKnowledgeAbout(X)" are new to this work and the other conversion rules are adapted from previous works.

Both OWL statements and PDDL statements are high level constructs and give the same information. The difference between the two lies in their syntax. Users provide their requests abstractly and the details of the request are extracted by the system in later stages by asking the user for the required information. The following example clarifies the things mentioned here.

Suppose the user wants to make a reservation for the transportation of a person to the hospital. The user defines his request by using the domain ontology constructs and defines the initial state and the desired state. In the initial state, the following statements are provided by the user to show the logical availability of a particular person and a particular transportation

```
<Person rdf:ID = "Patient_0"/>
<VehicleTransport rdf:ID = "TransportToHospital"/>
```

In the goal statement, the user shows the availability of the same logical objects and presents the desired relation between these two logical objects. They are all logical; there is no information about the details of a particular transportation and a particular person. The only useful information is the semantic types of objects which are sufficient in this phase.

```
<Person rdf:ID = "Patient_0"/>
<VehicleTransport rdf:ID = "TransportToHospital"/>
<VehicleTransport rdf:resource = "#TransportToHospital">
<isBookedFor rdf:resource = "#Patient_0"/>
</VehicleTransport>
```

OWL statements represent the fact that there is a desire to book a thing whose semantic meaning is "VehicleTransport" to another thing whose semantic meaning is "Person". The semantic details are going to be obtained by the actions proposed by the planner and the syntactic details are going to be obtained by the service composer by processing the WSDL and asking questions to the user about the required parameters.

Sometimes users do not request a change in the world state but they need only to find information. In such cases, the agentHasKnowledge(X) construct is used. In this case, users provide limited information about their requests. For instance, suppose the user wants to find the number of a particular flight. In such a case s/he initially presents the following statements to the system.

```
Init: <Flight rdf:ID = “Flight”/>

Goal:
<Flight rdf:ID = “Flight”/>
<agentHasKnowledgeAbout rdf:resource = “#Flight”>
```

The details of the flight such as its source and destination locations and arrival time are all unknown at this stage. The system only knows that the user wants to retrieve the flight number of a particular flight. The details are determined according to other services offered by the planner and WSDL description of services and the required values are requested from the user. It is possible to collect more details about the problem at the initial stage, but this is an ineffective way. The user cannot know all of the information that will be needed in later stages. In our work, we propose to request only the necessary information from the user after understanding their problem at a higher level.

4.2 Planning phase

In this phase, all work is handled by Simplanner. Initially, Simplanner does grounding by using the available PDDL objects and PDDL action definitions and produces the possible logical action instances. After grounding, it constructs an initial plan and continues to plan in the lifetime of the session. The interface with the Simplanner is presented in Table 3.

Simplanner produces a quick logical action and continues with planning as time permits. The planner continuously collaborates with the action handle/execution module and the unexpected event handler module. The planner sends logical actions to the action handle/execution module one at a time. During the actual service execution that is conducted by the executor, Simplanner continues to operate in order to refine the current plan. During service execution some problems may appear, such as information unavailability or service execution failures. In such cases the unexpected event handler examines the state and informs the planner about the unexpected situations. Simplanner then produces a new plan according to the current state.

After the high level problem and domain information are presented to the planner, it tries to find out a logical solution to the current problem. Two important assumptions are made at the beginning of planning. First, all the actions that are available in the domain knowledge base are executable. Second, all the necessary information that will be required in later stages can be provided by the user. These assumptions are the initial assumptions. The validity of these assumptions is determined after some interactions with the user and web services. If the assumptions are wrong, they are handled easily by the features of Simplanner.

The initial assumptions are asserted to the planner through the use of two predicates: “validService(X)” and “agentHasKnowledgeAbout(X)”. Each logical action defin-

Table 3 Simplanner interface

```
PLAN (intrerruptSIGNAL)

SIMPLANNER ← constructSimplannerThread()
terminate ← FALSE

WHILE terminate == FALSE
        LISTEN INTERRUPTS

        CASE interruptSIGNAL OF    /*functions corresponding to given signals are provided by the Simplanner*/

                plan:              SIMPLANNER .start(domain.pddl, problem.pddl)

                proposeAction:     SIMPLANNER .pause()
                                   [action, currentState] ← SIMPLANNER.proposeAction() //action will be sent to the module that
                                                                                       // causes interrupt
                                   SIMPLANNER.continue()

                changeState:       SIMPLANNER.pause()
                                   currentstate ← SIMPLANNER.changeState(currentState) //currentState is global variable
                                   SIMPLANNER.continue()                               // keep track of the current logical state

                                   terminate: SIMPLANNER.stop()
                                   terminate ← TRUE
END WHILE
```

ition that corresponds to a physical web service operation includes statements with these predicates in their precondition part. In order to execute any service, it should be physically available and the parameters that are required for executing that service should be known a priori. Consider the following example:

```
(:action BookFlightService
: parameters (?Customer-Person ?AccountData-Account
?Flight-Flight)
: precondition (and
(validBookFlightService)
(agentHasKnowledgeAbout ?Customer) (agentHasKnowledgeAbout
?AccountData) (agentHasKnowledgeAbout ?Flight))
```

"BookFlightService" is the logical counterpart of the service that does the booking operation. The service requires three parameters, the semantic types of which are "Person", "Account" and "Flight" respectively. The precondition statements represent that the parameters should be known and the service should be available.

At the initial problem statement, the user defines some logical objects and represents the relationship between them, but some other objects with other types may also be necessary to solve the problem. Suppose the user wants to do a booking and represents his problem as described earlier. Initially the user does not know that "BookFlightService" is going to be used for the solution of the problem, so he does not know the required parameters of the service either. As a result, some of the required logical objects that are necessary for the solution of the problem are not provided by the user. For the "BookFlightService" example, the logical service instantiation needs an "Account" typed object. If it is not defined by the user in the problem statement, the service cannot be used even if it is required for the solution.

The proposed solution to this problem is to construct one logical object corresponding to each PDDL type if it is not constructed by the user explicitly. The instantiation of any action becomes possible after the logical object construction with each PDDL type, but the instantiation is not sufficient for the physical execution. Before the physical execution, the system asks for the values of the logical objects and acts according to the answer. Since the PDDL objects are logical objects, the same objects can be used for the instantiation of different actions. The actual values of these logical objects are obtained at run time so the use of the same logical object by different services does not mean that distinct services are called with the same arguments. Suppose "service1" uses "obj1" as a parameter and "service2" also uses "obj1" as a parameter. During "service1" call, the actual value of "obj1" is requested from the user or from another service the details of which are described later. During "service2" call, the same procedure is applied from scratch. Therefore, different arguments are used for the same logical objects. In conclusion, one logical object for each PDDL type is sufficient, and they do not cause confusion during the actual service call since physical values are obtained in later stages.

If the user's goal includes an information gathering request, the system does not include the particular objects on the list of known objects. For instance, if the user presents an information gathering request like "agentHasKnowledgeAbout Flight"; the system does not add "agentHasKnowledgeAbout Flight" statement to the init section of the problem PDDL. Thus, the planner tries to find a solution. When the execution module requests an action, the planner returns the most promising logical action (that is grounded action with the logical objects) to the executor. While the executor is doing its own job (actual service execution, information collection from user, etc....), the planner continues to search for a better solution and to repair any problems that may occur in the later steps of the proposed initial solution. The planning procedure continues until another interruption by the executor.

4.3 Action handling phase

In this phase, the logical action that is provided by the planner is handled in order to satisfy user needs. The graphical representation of this phase is given in Fig. 3 and a formal algorithmic description is presented in Table 4. The action handler prepares the logical actions for execution in two ways. One is for single logical actions and the other is for complex (cached) actions.

When the planner proposes a single action, the action handler tries to collect the actual parameters, which are the physical counterparts of logical action parameters, from the user or from other web services. For instance, suppose the planner provides a logical action like "BookFlightService Person1 Account1 Flight1". "Person1", "Account1" and "Flight1" are the objects that are defined in the problem.pddl. The only information about these logical objects is their semantic types at this stage, which is not usable for the execution of the real service. The Logical/Physical map, which is kept in memory, is the core of the mechanism that associates logical entities with their physical counterparts. At the beginning of each session, a fresh Logical/Physical map is constructed (the details are presented in Sect. 5). The Logical/Physical map contains the syntactic counterparts of the semantic objects and their current values. The syntactic counterparts of the semantic objects are obtained by processing the grounding part of OWL-S service descriptions and WSDL descriptions of services. Initially the actual values of the syntactic counterparts of logical entities are unknown.

During the processing, these values are obtained from the user or from other services.

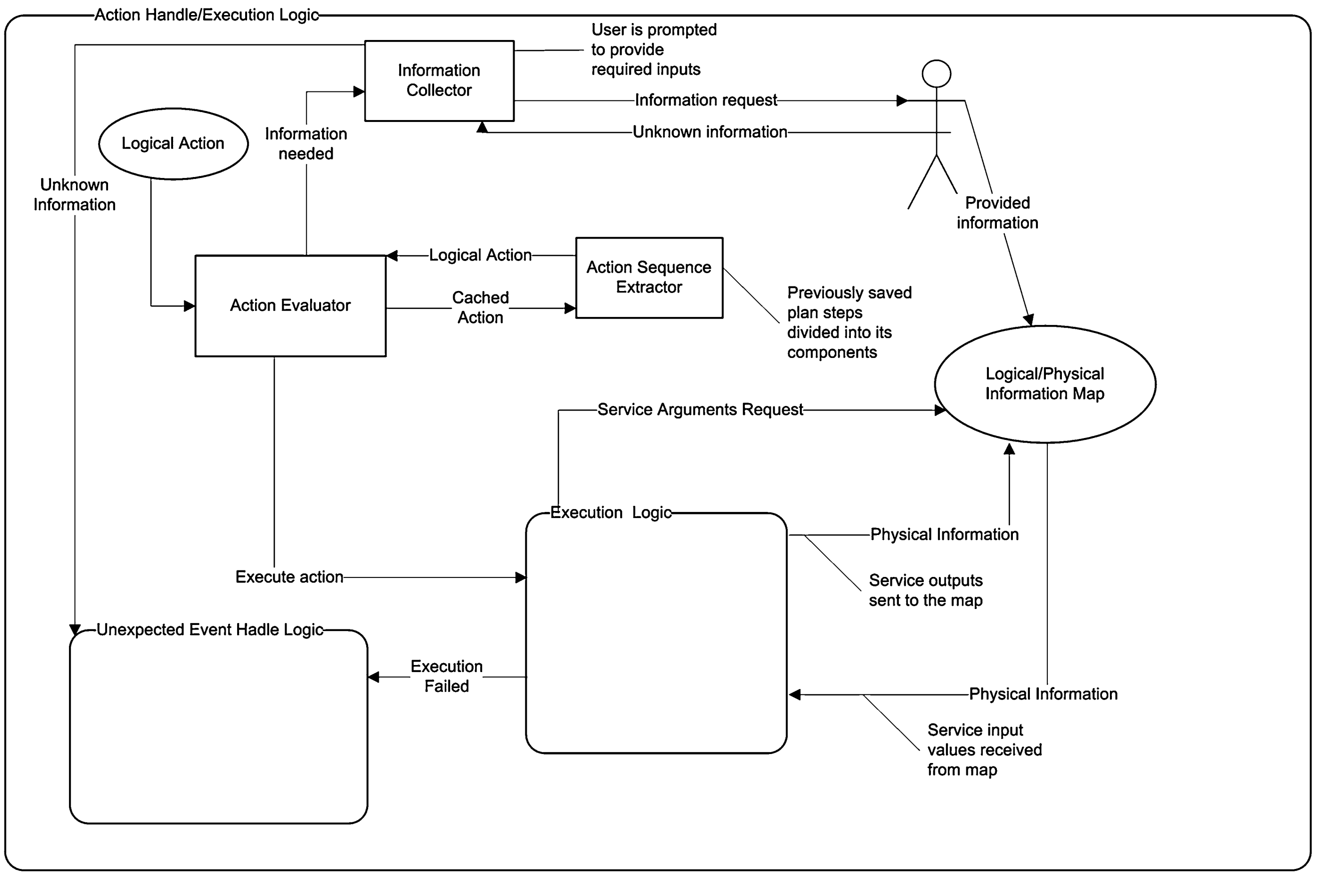

Action Handle/Execution Logic
User is prompted to provide required inputs
Information Collector
Information request
Unknown information
Logical Action
Information needed
Unknown Information
Provided information
Logical Action
Action Sequence Extractor
Action Evaluator
Cached Action
Previously saved plan steps divided into its components
Logical/Physical Information Map
Service Arguments Request
Execution Logic
Physical Information
Service outputs sent to the map
Execute action
Unexpected Event Hadle Logic
Execution Failed
Physical Information
Service input values received from map

Fig. 3 Action handling

When a logical action is presented to the action handler, it extracts the parameters and looks up the Logical/Physical map to learn their actual values. If they can be found in the Logical/Physical map, the action handler directs the action to the real executor. If the answer "unknown" is returned from the Logical/Physical map, the information collector's turn starts. Initially, the information collector gets the syntactic details of logical objects from the Logical/Physical map. For instance, a logical object "Request" whose semantic type is "RequestParameters" is contained in the parameters of the current action definition. The WSDL counterpart of "RequestParameters" that is obtained from the OWL-S grounding section is the "RequestInfo" type. It has a complex type definition in the WSDL. Suppose the "RequestInfo" complex type contains the parts "DestinationLocation-xsd:string SourceLocation-xsd:string ArrivalTime-xsd:dateTime". When the information collector requests the details of the "Request" object, the Logical/Physical map provides all the syntactical details and the information collector asks the user the values of "DestinationLocation", "SourceLocation" and "ArrivalTime". The user will provide "DestinationLocation", "SourceLocation" and "ArrivalTime" for the example above. If the user can not provide the values of the requested parameters, another strategy should be used. As an example, suppose the user wants to book a flight and a flight number is needed by the booking service. The above procedure is executed and the user is asked for the value of the flight number. If the user does not know the value, s/he tells the system that the value is not known. In this case the "unknown information" signal is fired. The details of the unexpected event are sent to the Unexpected Event Handler Module.

The action handler also extracts the details of cached actions. Action caching is a mechanism for making use of the previous executions in order to speed up the system. When a cached action is proposed by the planner, the procedure that is used for non-cached actions cannot be applied. A cached action is not a single action but it is a set of actions (i.e the steps of a previously successful WSC plan). There does not exist any information about the details of cached actions in the Logical/Physical Map and the details of the actions are obtained from other resources (Action caching is described in Sect. 5). Action handler finds the components of the cached action and does the grounding (constructing logical action with logical objects) of sub-actions according to the types of their arguments. After the action handler

Table 4 Action handling algorithm

```
LOGICAL_ACTION _HANDLER (action)

result ← SUCCESSFUL

IF cachedActions CONTAIN action          //cachedActions are loaded when session starts, it contains the composite action definitions
    subActions ← extractSubActions(action) //subactions are parts of the previously found plan for a particular problem

    FOR EACH action act ELEMENT OF subActions
        act ← groundLogicalActions(act)  //in this step logical objects are found and associated with the actions according to
                                         //types of action parameters
        LOGICAL_ACTION _HANDLER (act)  //composite actions are divided into its original actions, Logical_Action_Handler
                                       //can now operate on it
ELSE
    inputList ← extractActionInput(action) //get the input parameters of the proposed action

    FOR EACH input inp ELEMENT OF inputList
        physical_parts ← extractPhysicalDetail(inp) //syntatic counters of semantic object parameters are obtained in this step
        createGUI(physical_parts) //by using the syntactic definitions graphical user interface is generated for collecting inputs
        OBTAIN input from USER //user is prompted to provide required input

    //syntatic counter of semantic objects may contain multiple parts, corresponding WSDL type definition may be complex

        FOR EACH component comp ELEMENT OF physical_parts
            comp.real_value ← user provided value
            inp.comp ← inp. comp ∪ {comp}

            IF inp.comp.real_value == UNKNOWN  //user cannot provide value for some parts of the required input
                result ← UNSUCCESSFUL          //this situation will be handled by Unexpected Event Handler

        updateLogicalPhysicalMap(inp)  //Logical/Physical Map contains the information about syntactic counter of semantic
                                       //parameters and their current values, update the map with new information
ENDIF
RETURN result
```

performs its function, the real executor goes through the remaining steps for calling the web service.

4.4 Execution phase

In this phase, the actual service call is performed. The execution is done in two ways according to the service behaviour. If only an information gathering service is executed, it is done as a usual service call. However, if a world altering service is executed, then the service is called indirectly in conformity with WS-Business Activity and WS-Coordination specifications. The graphical representation of this phase is given in Fig. 4 and a formal algorithmic description is presented in Table 5.

In this phase, dynamically generated service client codes and PDDL action- physical action mappings are used in order to make a real service call. The objects are dynamically constructed at run time by using the above mentioned information resources and by the help of the reflection mechanism [10]. The actual values of the arguments of the service are collected from the Logical/Physical map and the constructed object instances are modified with the collected real data with reflection. If the service only provides information, the provided information is taken to the Logical/Physical information map and the consumed information is deleted from the Logical/Physical map. If the called service has world altering effects, its call is done through the business activity coordinator which is generated at the beginning of each session.

When the actual execution is conducted, some problems may arise because of network problems or other external reasons. In such cases the executor informs the Unexpected Event Handler about the unexpected situation and completes its work for that particular action.

4.5 Unexpected event handling phase

During service execution and information collection, some unexpected situations may occur. The service execution may fail because of network problems or wrongly provided argu-

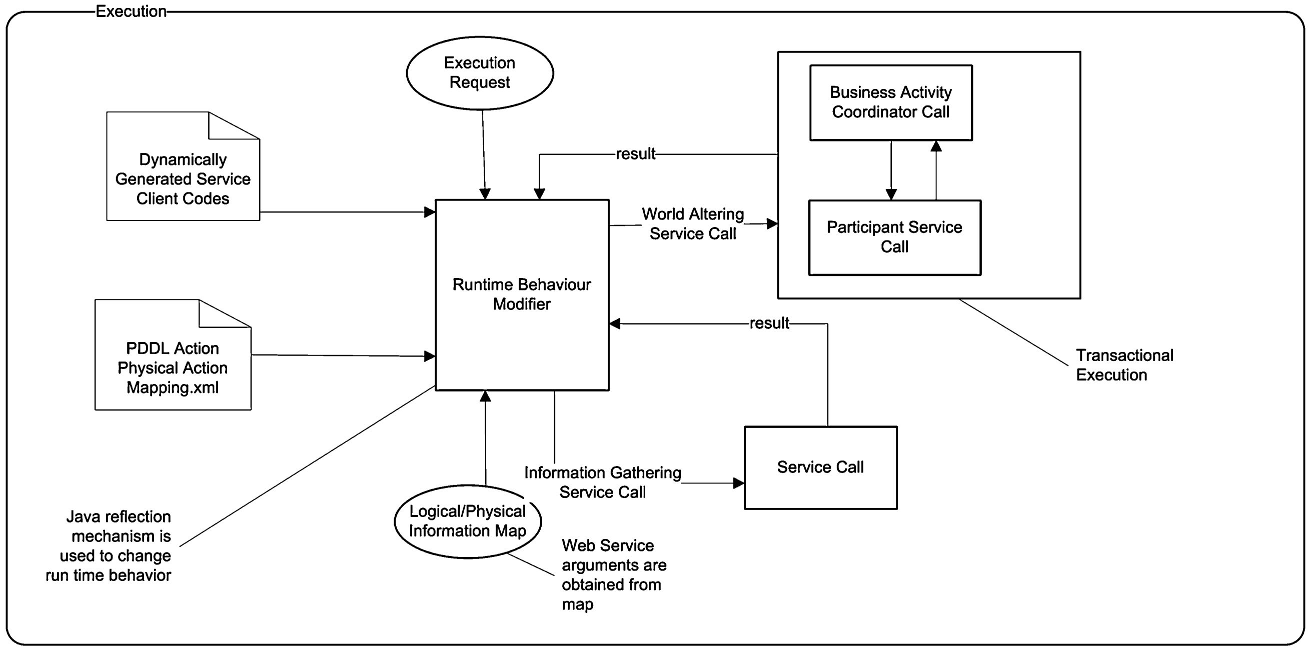

Execution
Execution Request
Dynamically Generated Service Client Codes
PDDL Action Physical Action Mapping.xml
Runtime Behaviour Modifier
result
World Altering Service Call
Business Activity Coordinator Call
Participant Service Call
result
Transactional Execution
Information Gathering Service Call
Service Call
Logical/Physical Information Map
Web Service arguments are obtained from map
Java reflection mechanism is used to change run time behavior

Fig. 4 Execution

Table 5 Execution algorithm

```
EXECUTE ( action)

service_function ← loadService(action)  //by using the outputs of the preprocessing phase and java reflection mechanism
                                        //web service client is constructed
inputList ← extractActionInputs (actions)

FOR EACH input inp ELEMENT OF inputList

        type_implementation ← loadType(inp)  // by using the outputs of the preprocessing phase and java reflection mechanism
                                             //required type definitions of web service arguments are constructed (complex types)

// LogicalPhysicalMap contains real values for input components, service arguments should be set with those values before invocation
        physical_components ← lookupLogicalPhysicalMap(inp)
            FOR EACH component comp ELEMENT OF physical_components
                setType_parts (comp.real_value)  //real values of input components are set through java reflection mechanism

IF action.Effects IS NOT EMPTY //world altering service call, transaction execution is required
        [result, serviceOutput] ← invoke(service_function, businessActivityCoordinator)  //service is invoked through businessActivity
                                                                                         //coordinator with reflection mechanism
ELSE
        [result, serviceOutput] ← invoke(service_function)     //result is the success of the execution and serviceOutput is the output values
                                                               //provided by the service if it is information providing
END IF

out ← extractActionOutput(action)    //extract output parameter of the logical action if it has

physical_components ← lookupLogicalPhysicalMap (out)  //get parts of the logical object corresponds to output from Logical/Physical map
        FOR EACH component comp ELEMENT OF physical_components
                comp.real_value ← extractRealValue(serviceOutput)     //set physical value of the object with service provided value

            updateLogicalPhysicalMap(out)    //update the Logical/Physical map with new information
RETURN result
```

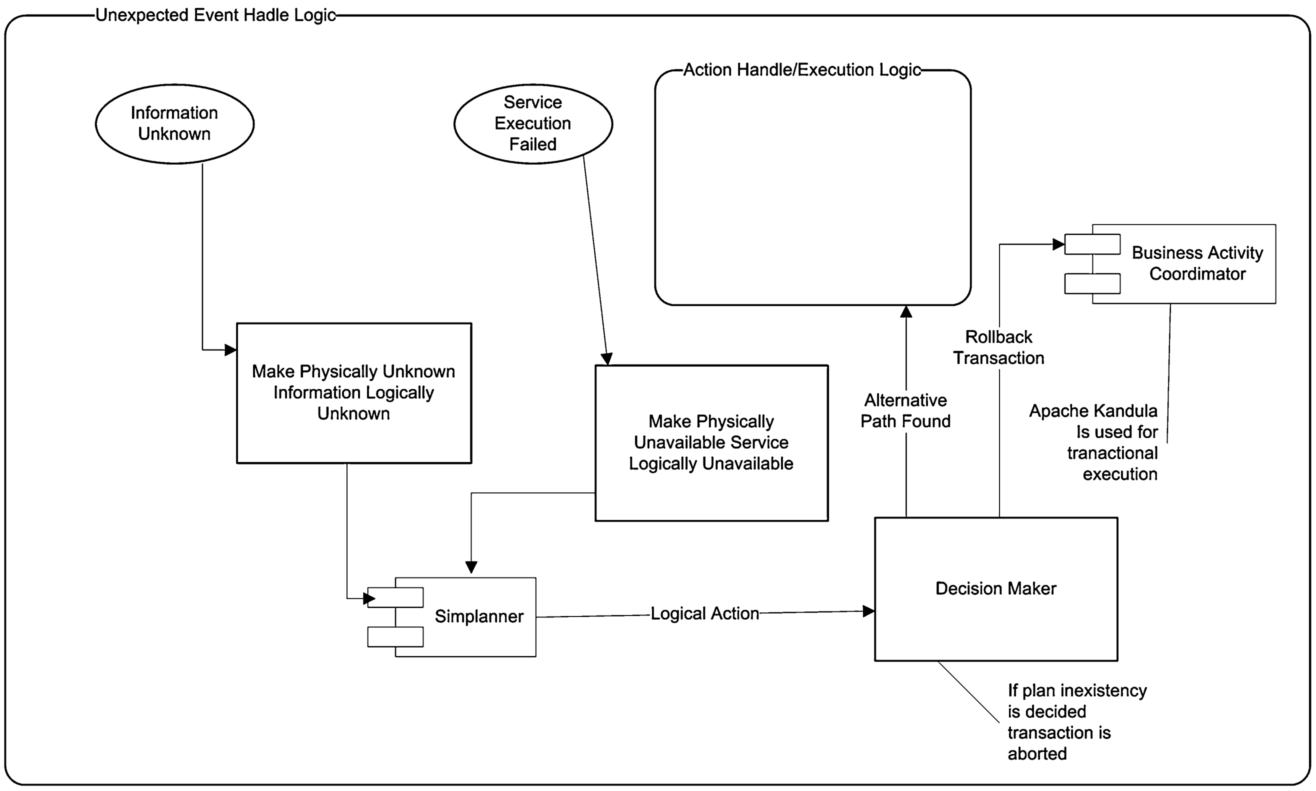

Unexpected Event Hadle Logic
Information Unknown
Service Execution Failed
Action Handle/Execution Logic
Business Activity Coordinator
Make Physically Unknown Information Logically Unknown
Make Physically Unavailable Service Logically Unavailable
Rollback Transaction
Alternative Path Found
Apache Kandula Is used for tranactional execution
Simplanner
Logical Action
Decision Maker
If plan inexistency is decided transaction is aborted

Fig. 5 Unexpected event handler

ments or because the user may not know the information that is required by the service. Such unexpected situations are handled in this phase. The operations of this phase are described in Fig. 5 and a formal algorithmic description is presented in Table 6.

The executor component sends two kinds of unexpected events to the Unexpected Event Handler: service unavailability and unknown information. The Unexpected Event Handler tries to solve these problems in collaboration with the planner. The Unexpected Event Handler keeps the logical state that is valid before the service execution. If any problem occurs, it uses this saved state and the information returned from the executor about the problem, to construct a new state that describes the current situation. After a new logical state is constructed, it is presented to the planner and the planner starts to find a solution according to the current situation. Simplanner allows state changes at any time. After a state change request is received by the planner, the logical actions that it provides to the executor conform to the current situation afterwards.

When the service unavailability message comes from the executor to the unexpected event handler, it requests a state change from the planner. The new state is constructed by removing the "valid[serviceID]" predicate from the last saved state. For instance, if there is a problem during the execution of the booking service "BookFlightService", the unexpected event handler removes the "validBookFlightService" predicate from the newly constructed state and informs the planner. From that point on, "BookFlightService" cannot be used, since its precondition "validBookFlightService" is not satisfied. The planner will not consider this logical action again and it tries to find other ways of solving the problem. There may exist alternative ways to solve the same problem. The planner constructs a new solution with alternative services, if available. Otherwise, the session is terminated unsuccessfully. If the session cannot be terminated successfully, there may be side effects in the environment due to the previously executed world altering services. Such side effects are compensated by the used transaction mechanism. The session termination condition and the used transactional mechanism are described in detail in Sect. 6.

The other unexpected event is information unavailability. The information collector component of the execution module asks the user for the values of the physical counterparts of logical objects as described earlier. If the user can not provide the requested information, the unexpected event handler changes the logical state for the planner. Suppose there is a logical object "Flight1" which has a semantic type of Flight. Initially, the logical state contains the predicate "agentHasKnowledgeAbout Flight1" which assumes that the user can provide the required information about the logical object "Flight1". If the information collector cannot

Table 6 Unexpected event handling algorithm

```
UNEXPECTED_EVENT_HANDLER (action, previousState, signal)

CASE signal OF
    UnavailableService_SIGNAL : //if service does not respond, make it invalid, planner does not consider it any more
        currentState ← removePredicate(previousState, {"validService" + identifer OF action})
        PLAN (changeState_SIGNAL)

    UnavailableInformation_SIGNAL: //if user cannot provide input value, change the state that describes this situation
        inputList ← extractActionInputs(action)
        FOR EACH input inp ELEMENT OF inputList
            physical_components ← lookupLogicalPhysicalMap(inp)
            FOR EACH component comp ELEMENT OF physical_components
                IF comp.real_value == UNKNOWN
                    currentState ← removePredicatecurrentState ({"agentHasKnowledgeAbout" +identifer OF inp})
        PLAN (changeState_SIGNAL)
    NoPlan_SIGNAL:   //if plan inexistency decision is reached, abort the transaction
            businessActivityCordinator.cancelOrCompensateAll()  //abort transaction

ENDCASE

RETURN currentState
```

get the necessary information about "Flight1" from the user, it delivers the situation to the unexpected event handler. The unexpected event handler removes the predicate "agentHasKnowledgeAbout Flight1" from the last saved state and delivers the state change request to the planner. After that state change, the planner has two alternatives: it can either search for other actions that do not require the unknown information or try to find other services that provide the required information. Suppose there is an action whose partial definition is presented below.

```
(:action ProposeFlight
(:parameters
( . . . . . . . ?Flight-Flight)
(:effect (and
(agentHasKnowledgeAbout ?Flight)))
```

This action is grounded with "Flight1" during the Simplanner action grounding procedure and it can provide the required information about the logical object "Flight1". If the planner proposes this action to the executor, the executor does the actual service call. The physical counterpart of the logical object "Flight" is constructed with the reflection mechanism. The actual values of the physical counterparts are then updated in the Logical/Physical map. When an action that requires the information about "Flight1" is again proposed by the planner, the action handler can get the actual values that are required for the service call from the Logical/Physical map without any need to ask the user again. An important issue about the Logical/Physical map is that once the actual values of the physical counterparts of logical objects that are provided by either the user or another web service are used, those real values are cleared from the map even if they are required in later steps for the reasons described in Sect. 5.2.

If replanning cannot produce new ways to achieve the user's goal after the occurrence of unexpected events, the transactional operations are rolled back. The business initiator sends the necessary signal (i.e. compensate) to all participating services through a business activity coordinator. This prevents the occurrence of side effects of unsuccessful attempts.

5 Automated service invocation issues

The automated web service invocation module is one of the core components of the proposed framework. The service invocation mechanism uses off-the-shelf technologies such as the Java reflection mechanism [10], the web service invocation framework (WSIF) [35], the WSDL2Java tool of the Apache Axis [36] framework and integrates them in an efficient manner for achieving an automated service call. During service invocation, the arguments that are needed for the service calls are obtained from the data structure, namely the Logical/Physical map. This data structure provides the communication between the service invocation mechanism and the input values provided by either the user or a service. After the termination of each successful session, the system caches the action sequence that is used for handling the current problem in order to use them directly when a similar request comes again later. The following sections present the details of the mechanism that is used for the automated service invocation.

5.1 Automated service invocation

The provided solution for linking abstract actions with actual service calls is highly generic. There is no manual coding, classes are dynamically generated and objects are constructed dynamically at run time with the user provided inputs automatically. After the planner provides a logical action to the execution module, the action handler prepares the logical action for the actual service invocation with the help of the information collector and the unexpected event handler. The action handler makes sure that the parameters of logical actions have the desired syntactic counterparts in the Logical/Physical map, so the real service executor can get the required service argument values from the Logical/Physical map. If the required arguments can be supplied by neither the user nor another service, the planner tries to find an alternative path. If an alternative path cannot be discovered, the session is terminated before reaching the automatic service invocation phase.

There are two kinds of service executions in the system. One is the information gathering service call and the other one is the world altering service call. The difference between them is that the world altering service call causes side effects. In order not to change the world in an unintended way, one makes sure that such a call is made transactional. The mechanisms that are used to call these two kinds of services are different. The information gathering services are called as usual, but the world altering ones are called by conforming to the WS-Business Activity and WS-Coordination standards [8, 13].

Simplanner uses the semantic knowledge extracted from OWL-S service descriptions, but the actual service executor needs syntactic information in order to operate. The syntactic information is collected from WSDL descriptions of the used web services. WSDL contains all necessary details for service calls such as syntactic types of operation arguments, service end point and the required communication protocol.

For a real service call, the client stubs of web services are generated through the WSDL2Java tool during the preprocessing phase. The connection between the semantic information that is used by the planner and the syntactic information that is required for the actual service call is provided in the grounding part of OWL-S definitions. The software processes the grounding section of the OWL-S file and extracts the relationships between semantic types and syntactic types of service arguments. The syntactic counterpart of the semantic type might have complex type definitions. The system extracts the details by processing the WSDL document of the web services. The output of this phase is the "PDDL Action-Physical Action mapping.xml" file whose structure is shown below.

```
<actions><action name = "LogicalOperation"
class = "ImplementationClass" endpoint = "ServiceEndPoint">
<inputs>
    <input name = "Input" class = "ImplementationClass">
        <subtype name = "SubType"
        class = "ImplementationClass"/>
        ....
    </input>
    ....
</inputs>
<outputs>
    <output name = "Output" class = "ImplementationClass">
    <subtype name = "SubType" class = "ImplementationClass"/>
        ....

    </output>
    ....

</outputs>
</action>
    ....
</actions>
```

The information in the "PDDL Action-Physical Action mapping.xml" file is used for dynamic method construction corresponding to the planner-provided logical action and logical parameters through the Java reflection mechanism.

After a logical action with its logical parameters is provided to the executor by the planner and the action handler does its processing, the real service invoker starts the execution. The service invoker discovers the implementation code of the requested operation and its parameters by using the metadata that are produced before, and it constructs a dynamic method for the real service invocation. For instance, the logical action "BookFlightService Param1 Param2" is produced and "BookFlightService" has an implementation, namely "BookFlightServiceClass", Param1 and Param2 have implementations, "Param1Class" and "Param2Class" respectively, and BookFlightService has an endpoint namely "BookFlightServiceEndpoint". The following java code shows how a service call method is dynamically generated for the "BookFlightService" case.

```
Class c = Class.forName("BookFlightServiceClass");
Constructor construct = c.getConstructor(new Class[] {URL.class,
Service.class})
Object stb = construct.newInstance(new
URL("BookFlightServiceURL", null);
Class partypes[] = new Class[2];
partypes[0] = Class.forName(Param1);
partypes[1] = Class.forName(Param2);
Method meth = c.getMethod("BookFlightOperation", partypes);
```

The information that is needed for dynamically generating a service call method is collected during preprocessing phase and it is written to the metadata files as described before. Actually, it is not sufficient to generate only a method for service invocation. The values of the method arguments are needed as well. The Logical/Physical map is used for this purpose. The action handler makes sure that the map contains the required syntactic values of logical parameters before the real service execution begins. After the method construction, the service invoker gets the actual values of the logical parameters from the Logical/Physical map and instantiate the dynamically constructed methods with the obtained values. The Java code for the instantiation of the example above and the real service call is shown as follows:

```
Object arglist[] = new Object[2];
arglist[0] = “Param1Value”;
arglist[1] = “Param2Value”;
meth.invoke(stb,arglist);
```

Param1Value and Param2Value are the real values of Param1 and Param2 logical objects respectively which are obtained from the Logical/Physical map. If the called service is an information providing service, it returns some values and these values are handled in a similar mechanism that is applied for inputs with the help of the reflection mechanism and Java beans. If the returned type is a complex type, the sub-parts are obtained by using the properties of Java beans. The generated code for complex types is in the form of Java beans which provides a static interface for reaching the subcomponents of the returned values so the handling of them can be done automatically without any human assistance.

5.2 Logical/Physical map

The Logical/Physical map associates the logical objects that are used as action parameters by the planner with their syntactic counterparts and values which are required for the real service calls. The syntactic counterpart of a semantic type may be a complex type and it may contain subcomponents. Therefore more than one physical part may exist for a semantic type as shown in Fig. 6. The values of the physical parts are needed for the real service call.

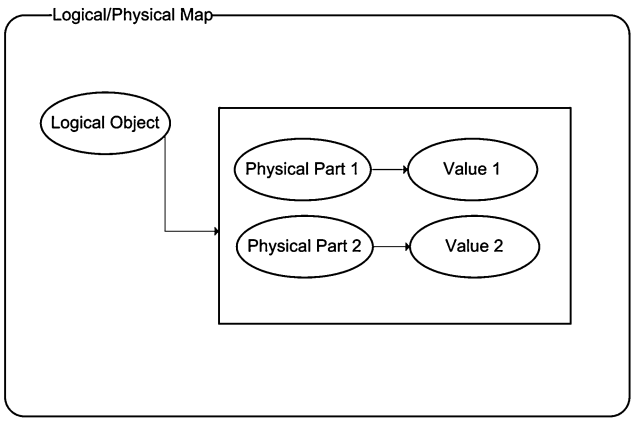


Fig. 6 Logical/physical map

The values are obtained from either the user or other services before each service call. At the beginning of each session, a fresh map is constructed where the values of the physical parts are initialized to the value “unknown”.

When a logical action with logical object parameters comes to the execution module, the action handler examines the values of logical objects in the map. If they are “unknown”, the information collector asks the user for the values of the physical components. If the user provides the values, the map is updated with these values and the service invocation begins. If the user does not know the values, the unexpected event handler and the planner together try to find a service that can provide the particular information. If such a service is found, the current plan changes and the information providing service is invoked with the same steps applied. The map is updated with the collected values. The action handler then discovers that the required information is available in the map and allows the service invocation to operate.

The construction of the map is dynamic and it is carried out throughout the current session according to the logical action definition. There may be several logical actions that are grounded with logical objects but the syntactic structure of distinct services may be different. For instance consider the services “BookFlightService ?Flight-Flight ?Person-Person” and “BookMedicalFlightService ?Flight-Flight ?Person-Person” and PDDL objects “Flight1-Flight and Person1-Person”. The two services given above are two distinct services, but their semantic parameter types are the same. After the planner performs grounding, two distinct logical actions are generated with the same logical object parameters as follows: “BookFlightService Flight1 Person1”, “BookMedicalFlightService Flight1 Person1”.

The intended semantic meanings of “Flight1” and “Person1” objects are the same in the two grounded services, since “Flight1” and “Person1” are used with the same meaning in mind by the user. However, syntactic differences may occur in the physical definition of the logical objects. For instance, “Flight” type may have a WSDL counterpart such as “xsd: int” for the “BookFlightService”, but a complex type counterpart that contains a string and an integer value “sequence (xsd: int, xsd: string)” for the “BookMedicalService”.

Since one logical object may be used by different services, we need to make sure that the correct syntactic counterpart is available in the map before each service call. For instance, before the “BookFlightService” is called, the map contains (Flight $\rightarrow$ int) information and before the “BookMedicalService” is called the map contains (Flight $\rightarrow$ (int, string)). Thus a part of the map is reconstructed every time an action is called. After each service call, that part of the map is destroyed. By using this reconstruction methodology the syntactic differences can be avoided for the parameters with the same intended semantic meaning.

The same problem can also occur when a service is invoked to obtain values of unknown parameters if the user does not know the values. For instance during a flight reservation, the web service may need the flight number which is not known by the user. Another service can provide the flight number with the source, destination and time data which can be provided by the user. In such a case, the map is updated with values obtained from the other service. The syntactic incompatibility problem mentioned above may arise in this case too, since the other service may provide the required input of the reservation service with a different syntax. This problem can be avoided by using the syntactic checks between the input structures of both services, but it is not implemented in the current version of our system.

5.3 Action caching mechanism

Although Simplanner provides important advantages for providing timely response, precompiled solutions are still very valuable. Different users may have similar requests, so if previously found plans are saved they can be used again later.

In this paper, such a previous experience caching mechanism is proposed. During the processing of each session, the system keeps track of the successfully executed services and the initial and final logical states of the current session. After the successful termination of the session, the system, constructs a new PDDL action with a precondition which represents the initial state of the current session and with an effect which represents the final state of the current session. Parameters of the action are determined according to the requirements of the constructed precondition and effect parts. The new PDDL action is added to the PDDL action definitions in the domain.pddl file. After the addition of the newly constructed action to the domain.pddl, the metadata of the new action is written in an xml file, namely “complexActions.xml”. The metadata contains the successfully executed logical actions in an order which represents the components of the constructed complex action. The structure of the xml file that holds the components of the complex action is as follows:

```
<complexActions>
    <action name = “LogicalActionName”>
        <subAction name = “SubAction1”/>
        ....
    </action>
    ....
</complexActions>
```

Simplanner usually proposes the shortest path to the solution. Because of its any-time principles, Simplanner concentrates on finding the first action instead of the entire plan. And, it needs some deliberation time to propose the new best action after any state change due to an action execution or an unexpected event. If enough deliberation time is provided, it proposes the action that is part of the shortest plan. For instance, suppose two plans are available, such as complexAction $\rightarrow$ B and C $\rightarrow$ D $\rightarrow$ A $\rightarrow$ B, in order to reach a given goal. If enough deliberation time is provided the planner proposes the complexAction.

In our system the deliberation time is determined experimentally to be 5 seconds after each state change. It is the average time to collect inputs from user and execute the web service. During that period planning process can proceed in parallel to refine the plan found so far. However it does not mean that this time is always sufficient to find the optimal solution. The complexAction will be in the optimal plan if the new problem is similar to a previously solved problem. Having a small number of steps in the plan is valid only for logical actions; physically, more steps may be required to reach the goal since the complexAction creates the effect of executing several actions.

6 Transactional issues

One of the most important goals for the automated web service composition that this work focuses on is to deal with the nondeterminism of services and partial observability of the environment. Simplanner that is chosen for the logical action construction of WSC is very effective in dealing with these problems. Service and information unavailability is effectively handled by the planner. In case of such problems, the planner tries to find alternative paths for solving the problem. However, sometimes it is not possible to discover alternative ways when unexpected situations arise, and in these cases the session is terminated unsuccessfully.

The planner handles the unexpected situations at the logical level which is not sufficient in fact. Some physical mechanisms are also needed for handling such cases. The proposed system works step by step and executes the actions physically. If any problem occurs in the later steps of processing and the previously executed actions have world altering effects, some undesired situations may arise. Transactional execution is the solution that we propose to solve this problem. Although the transaction concept is very tightly coupled with databases, it is widely used in distributed systems for the same purposes at an application level.

Specifications, such as WS-Coordination, WS-Atomic Transaction and WS-Business Activity, are proposed to enable transactional features during web service collaboration [8, 13, 37]. The WS-Business Activity framework is more suitable for long running transactions. The web service composition may also be considered as a long running transaction because generally a large amount of user interaction is needed for information collection and the planner needs a deliberation time before the real service execution. Therefore WS-Business Activity framework is used in the proposed system.

The WS-Business Activity framework is used for coordinator-participant communication and the Web Services-Business Activity Initiator Protocol [38] is used for the communication between the service composer agent and the coordinator in the proposed system. The implementations of both protocols are done by the Apache Kandula project [39] and Kandula is adapted to the proposed system. The coordinator implementation is deployed as a web service and the world altering service calls are done through the coordinator. When the service composer decides to make a real service call, it checks to see if the service is a world altering or information gathering service by examining the logical effects of the logical action that corresponds to the service to be executed. If it is a world altering service, the service composer agent invokes the service in a transactional activity.

In this work, world altering services are assumed to implement the WS-Business Activity participant specification. This assumption is not needed in fact, because it is possible to find out if the web services implement that specification or not from their WSDL definitions. However, such an assumption is still made for the sake of simplicity. For instance, a booking flight service with inputs “user account”, “flight number” and “personname” has an input message part as follows in its WSDL definition.

```
<message name = “BookFlightInputMsg”>
 <part name = “transactionalContext”
type = “tns:contextChoiceType” />
 <part name = “useraccount” element = “tns:UserAccount” />
 <part name = “flightnumber” type = “xsd:int” />
 <part name = “personname” type = “xsd:token” />
</message>
```

The type “contextChoiceType” is defined according to the “WS-Business Activity” specification. When a transaction is required, the coordination context that is requested by the service composer agent from the coordinator is transported to the web service. For the example above, the first input argument represents the context and it is sent to the real service by the dynamically constructed method.

The coordinator keeps the status of all world altering web service participants through interfaces provided by the WS-Business Activity. However, all commands that are sent to the participants by the coordinator are determined by the service composer itself. The service composer sends the commands to the coordinator through the Web Services-Business Activity Initiator protocol according to its decisions. The coordinator works as a proxy that conveys commands from the service composer agent to the web services.

When the system decides that the current session can be terminated successfully, it sends the command “closeAllParticipants” to the coordinator, which informs each participant about the situation. The execution component of the agent checks to see if there exists a remaining goal to be achieved before requesting a new action from the planner. If there are no remaining goals, that is if all requests are handled, then the system decides to terminate the session successfully and informs the world altering services about the decision by means of issuing the command “closeAllParticipants”.

The planner tries to find alternative paths when an unexpected situation arises. It is sometimes possible that such paths do not exist. In such cases a solution cannot be found for the requested goal and sometimes, even if the unexpected events do not happen, it is understood that the planner cannot provide a solution to the problem after several steps. In such cases, the system issues the command “cancelorCompensateAllParticipants” to the coordinator for the current activity and the coordinator asks the participating web services to execute their compensation operations.

The problem of proving that a plan does not exist is very difficult not only for Simplanner but for all existing planners. Generally the whole search space needs to be examined, which is impossible for big environments such as the WSC domain since a huge amount of time is needed. Some admissible heuristics are needed for understanding why a plan does not exist. In the WSC case, it is very rare that the same service is called with the same input values in a single session. Although sometimes a need for such calls occurs, such rare cases are not considered for the sake of a timely response to all other cases. In this system the abort decision is made if the same service call is proposed by the planner with the same logical object parameters and with the same physical values (obtained from the Logical/Physical map). Some more restricted admissible heuristics can be applied, such as considering the current logical state. That is, the abort decision is made if the same service call is proposed by the planner with the same logical object parameters and with the same physical values in the same current state. In fact all these session abort heuristics are domain dependent and they cannot be used in a domain independent manner.

In the case of cached actions, a complex logical action is composed of multiple physical actions. The service calls of the physical actions are done by conforming the transactional rules mentioned before, that is, world altering service components are called through the coordinator. The logical effects of each physical service component are made visible after each service call. Therefore, the planner finds alternative paths in case a complex action fails, by considering the logical effects of successfully called physical components of the complexAction. As a result, when the successful termination decision is made by the system, the compensation operations of the physical components of the failed complexAction do not need to be called since the planner considers their logical effects while discovering the new path. If an alternative path cannot be found, the usual compensation mechanism is applied to all the previously executed world altering services.

7 Case study: travel domain

In this section, a simple case study is presented in order to illustrate the implementation and the functionality of the proposed system. The system is highly resistant to unexpected real world situations and it provides a timely response.

Domain independent AI planners cannot be practically used when the number of available actions is more than a few thousands. The current system is applicable in relatively small environments; it is applicable in some prefiltered environments. What we mean by a prefiltered environment is that the relevant web services are discovered and saved in a temporary registry. The planner considers only the services in the local registry. Scalability cannot be provided within the current domain independent planners. Some filtering mechanisms are offered by other works to provide scalability. We are currently working on such a mechanism in order to provide scalability on this work but it is out of the scope of this paper.

The case study is based on a travel domain scenario which is used as a data set in [40]. The following web services are used in this scenario. This case study contains two parts that focus on information and service unavailability, respectively.

RequestMedicalFlight: provides the same functionality as the ProposeFlight service.

CreateMedicalFlightAccount: provide the same functionality as the CreateFlightAccount service

CreateVehicleTransportAccount: This service creates a vehicle transport account for a particular person. The user provides personal information such as name, address, password, etc. and the service creates a transport account that is required for reservation in later stages.

RegisterPersonWithTransport: This service reserves a particular transport to a particular person. It requires that the person has a transport account and it requires transport id information. If the requirements of the service are provided, it books the transport for the person.

RequestTransport: This service provides a transport id corresponding to request parameters such as source and destination locations and arrival time.

CreateFlightAccount: This service creates a flight account for a particular person. The user provides personal information such as name, address, password, etc. and the service creates a flight account that is required for reservation in later stages.

BookFlight: This service reserves a particular flight to a particular person. It requires that a person who has a flight account and it requires flight id information. If the requirements of the service are provided, it books the flight for the person.

ProposeFlight: This service provides a flight id corresponding to request parameters such as source and destination locations and arrival time.

BookMedicalFlight: provides the same functionality as the BookFlight service.

7.1 Case 1: information unavailability

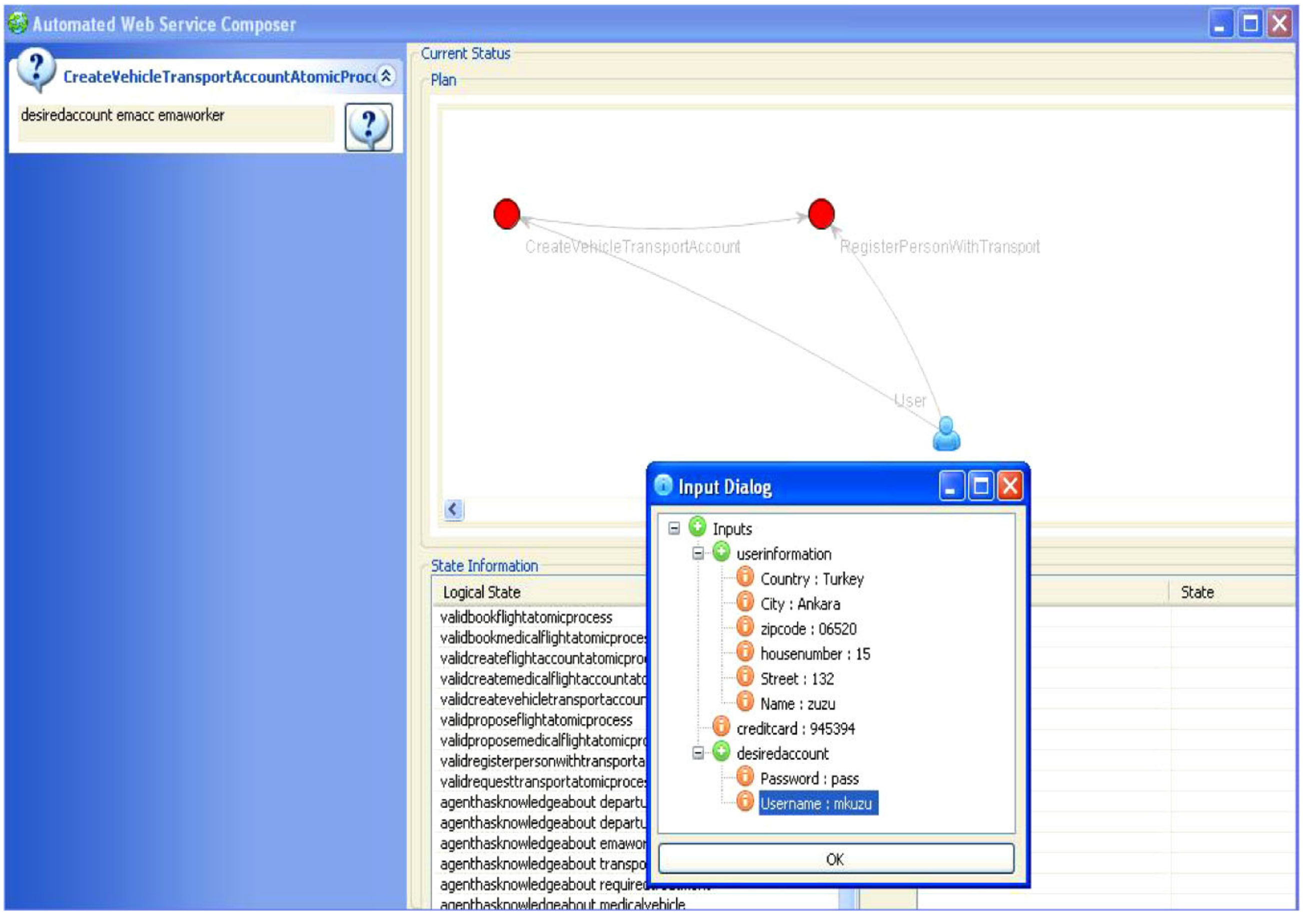
In this scenario, the user requests the reservation of transportation, using the constructs of the available travel ontology.

```
<Patient rdf:ID = “Patient_0”/>
<VehicleTransport rdf:ID = “TransportToHospital”>
  <isBookedFor rdf:resource = “#Patient_0”/>
</VehicleTransport>
```

This request is constructed using OWL individuals and their relationships in the travel ontology by the user explicitly (At this version, requests are manually constructed; an ontology editor will be integrated to the system to construct initial requests visually in later versions). The user asserts the system that s/he wants to make a VehicleTransport reservation for a Patient. The details of the request are asked to the user in later stages by the system (who the patient is, reservation details such as time, departure, etc…) according to the semantic and syntactic details of services that will be used. This request is converted to PDDL and the planner starts to work on the problem. The service “RegisterPersonWithTransport” does the booking operation, but it has a precondition that it requires a valid customer account and some other inputs. The service “CreateVehicleTransportAccount” satisfies the precondition of the service “RegisterPersonWithTransport”. As a result the planner produces the plan given in Fig. 7.

The planner proposes an action with logical parameters such as “desiredaccount”, “emacc” and “emaworker” for this particular case. Before the service composition begins, the Logical/Physical map is constructed and the physical counterparts of all logical objects are written on the map by using the grounding part of OWL-S and the WSDL definitions of the services. The “Input Dialog” is prompted to the user in order to obtain the physical values of logical objects. The user enters the inputs and the map is updated with the given values. By using the provided inputs and previously constructed service client stubs, the system does the actual service call.

Since the service “CreateVehicleTransportAccount” has world altering effects, its call is done through the WS-Business Activity coordinator. After calling “CreateVehicleTransportAccount”, the precondition of “RegisterPersonWithTransport” is satisfied and the system prompts the “Input Dialog” to the user that asks for the physical counterparts of the logical parameters (See Fig. 8). The user does not know the “transportid”, which is the physical counterpart of the logical object “transporttohospital”. Since its real value is left “unknown”, the system removes the logical statement “agentHasKnowledgeAbout transporttohospital”. So the planner searches for an action that provides the required input.


Automated Web Service Composer
Current Status
Plan
CreateVehicleTransportAccount
RegisterPersonWithTransport
User
State

CreateVehicleTransportAccountAtomicProc

desiredaccount emacc emaworker

Input Dialog

- Inputs
  - userinformation
    - Country : Turkey
    - City : Ankara
    - zipcode : 06520
    - housenumber : 15
    - Street : 132
    - Name : zuzu
  - creditcard : 945394
  - desiredaccount
    - Password : pass
    - Username : mkuzu

OK

State Information

```
Logical State
validbookflightatomicprocess
validbookmedicalflightatomicproce
validcreateflightaccountatomicpro
validcreatemedicalflightaccountato
validcreatevehicletransportaccour
validproposeflightatomicprocess
validproposemedicalflightatomicpr
validregisterpersonwithtransporta
validrequesttransportatomicproce
agenthasknowledgeabout departu
agenthasknowledgeabout departu
agenthasknowledgeabout emawor
agenthasknowledgeabout transpo
agenthasknowledgeabout required
agenthasknowledgeabout medicalvehicle
```

Fig. 7 Initial plan generation

The planner discovers another service "RequestTransport" which is able to provide the missing information. The required inputs are requested from the user for firing the service "RequestTransport", and after the required information is collected the actual service is called.

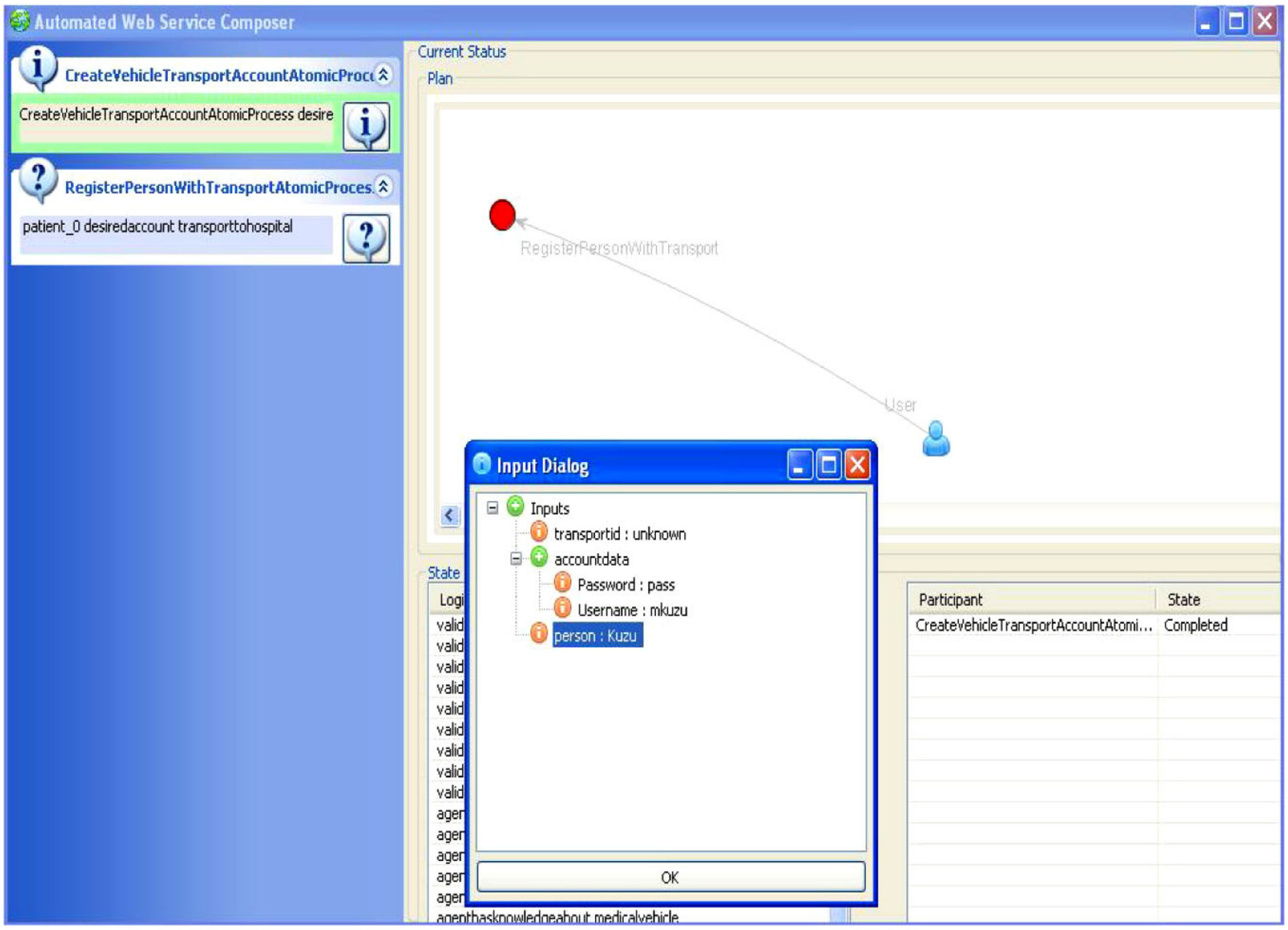
The service "RequestTransport" is called directly because it is an information providing service. Since the "transportid" is provided by another service, the required inputs for "RegisterPersonWithTransport" service become ready and the actual service call is done by the system using these inputs.

After the service "RegisterPersonWithTransport" is executed, the session is terminated successfully since the user's goal has been reached.

7.2 Case 2: service unavailability

In this case, when the user tries to reserve a flight, some unexpected situations occur (i.e. service failures). Only the most important parts of this case are presented in order not to repeat things that were already explained. Initially the system discovers a plan which contains services "CreateFlight", "ProposeFlight" and "BookFlight". The services "CreateFlight" and "ProposeFlight" are successfully executed, but during the execution of the service "BookFlight", the execution fails because of some network problems. The system then removes the logical statement "validBookFlightAtomicProcess" from the current state and tries to find an alternative path in order to respond the user request. Since "validBookFlightAtomicProcess" statement is a precondition for firing the "BookFlight" service and it is not true anymore, that service becomes unavailable to the planner. The planner discovers a new path for achieving the goal by dynamic replanning. The new plan contains "CreateMedicalFlightAccount", "ProposeMedicalFlight" and BookMedicalFlight" respectively (see Fig. 9).

The services "CreateMedicalFlightAccount" and "ProposeMedicalFlight" are executed successfully. During the execution of the "BookMedicalFlight" service, a failure occurs because of service unavailability. The predicate "validBookMedicalFlightAtomicProcess" is removed from the current state so that the service will not be considered as an available action any more for this particular session. This time the planner cannot produce an initial plan but proposes


Automated Web Service Composer
CreateVehicleTransportAccountAtomicProc
CreateVehicleTransportAccountAtomicProcess desire
RegisterPersonWithTransportAtomicProces
patient_0 desiredaccount transporttohospital
Current Status
Plan
RegisterPersonWithTransport
User
State
Input Dialog
OK

Inputs
transportid : unknown
accountdata
Password : pass
Username : mkuzu
person : Kuzu

Participant	State
CreateVehicleTransportAccountAtomi...	Completed

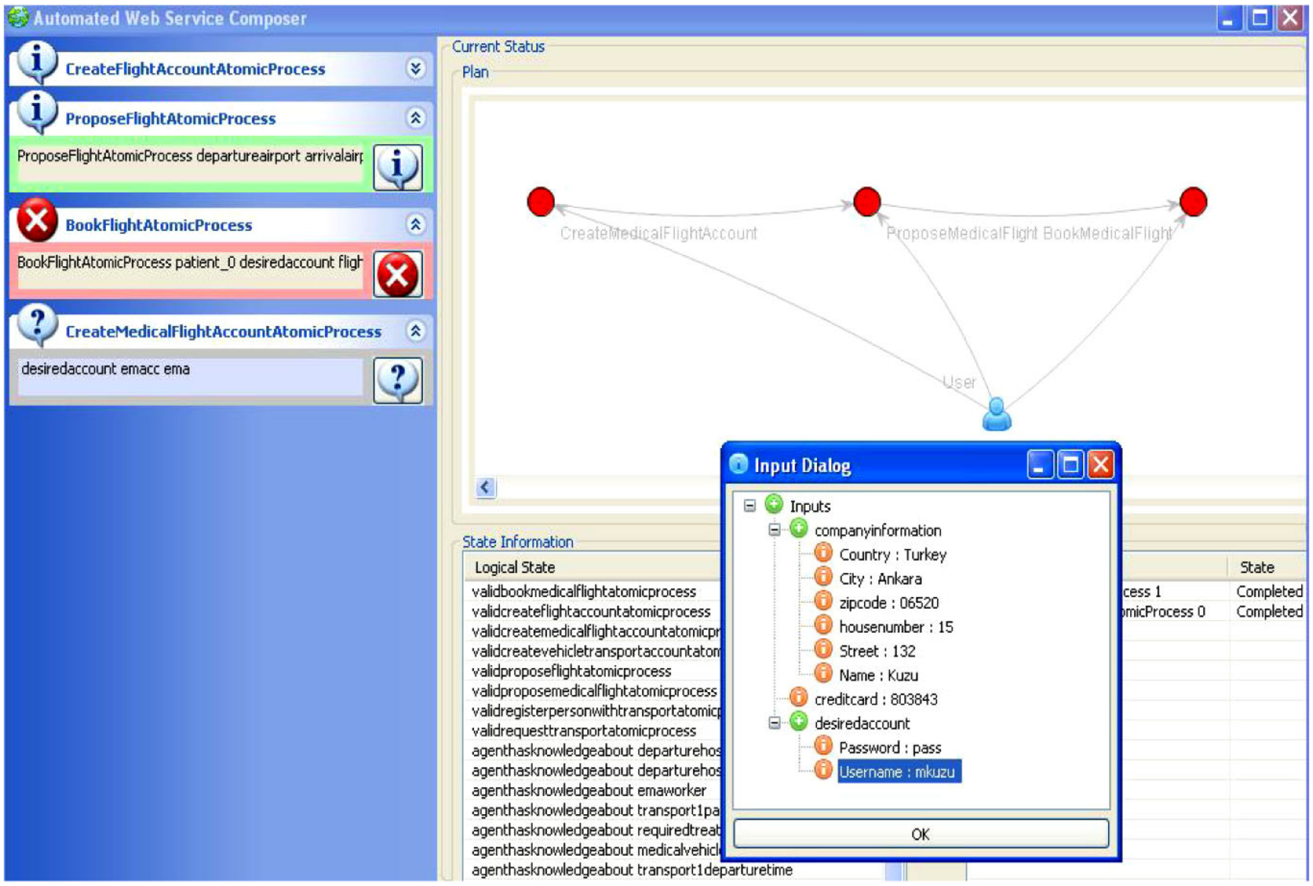
Fig. 8 Unknown information

the best available action which is the "CreateVehicleTransportAccount" service.

After the service "BookMedicalFlight" has failed, the planner cannot find a new plan and starts to search the state space. During the state space search, the same action, with the same logical parameters is proposed by the planner. However this causes the session to abort. All of the services that are executed up to the abort decision have world altering effects in this scenario. Therefore their calls are conducted through the business activity coordinator. After the abort decision is made, the system sends a "compensateAll" signal to the coordinator which then transmits it to the participants. The participant services fire their compensation mechanisms. As a result, undesired side effects of the previously executed actions are prevented.

Besides algorithmic scalability, the interactive web service execution is realistic. The user interfaces are not manually engineered; all of them are generated at run time automatically by using the xml schema definitions provided in the WSDL files. Before any service call, the logical actions and their syntactic counterparts are available in the Logical/Physical map. When a logical action is proposed by the planner, its physical counterparts (WSDL counterparts) are retrieved from the map and the user interface is constructed with that information at run time (the input and output interface is represented as a dom tree and the tree is updated with the information retrieved from the map). When the user provides the required inputs the map is updated with the values and the real service call is done if all inputs are provided. The map contains all syntactic details (if a complex type definition exists in WSDL, it is represented in the map and the user interface is constructed according to it).

It is clear that the proposed system is highly adaptable for the real world environment. A direct comparison with other service composers in terms of algorithmic efficiency is not possible, since the approaches are quite different. To the best of our knowledge this is the first work that combines replanning with transactional execution and combines planning and user interaction for service execution. For instance, to run OWS-XPLAN [40], all information must be given as a config file and there is no user interaction. The system proposes the complete plan without considering the possible service failures. It only gives a sequence of service calls and does not execute the services. The web service


Automated Web Service Composer
Current Status
Plan
CreateMedicalFlightAccount
ProposeMedicalFlight BookMedicalFlight
User
Input Dialog
OK
State Information

CreateFlightAccountAtomicProcess

ProposeFlightAtomicProcess

ProposeFlightAtomicProcess departureairport arrivalair

BookFlightAtomicProcess

BookFlightAtomicProcess patient_0 desiredaccount fligh

CreateMedicalFlightAccountAtomicProcess

desiredaccount emacc ema

Inputs
- companyinformation
  - Country : Turkey
  - City : Ankara
  - zipcode : 06520
  - housenumber : 15
  - Street : 132
  - Name : Kuzu
- creditcard : 803843
- desiredaccount
  - Password : pass
  - Username : mkuzu

Logical State		State
validbookmedicalflightatomicprocess	cess 1	Completed
validcreateflightaccountatomicprocess	micProcess 0	Completed
validcreatemedicalflightaccountatomicpr		
validcreatevehicletransportaccountatom		
validproposeflightatomicprocess		
validproposemedicalflightatomicprocess		
validregisterpersonwithtransportatomic		
validrequesttransportatomicprocess		
agenthasknowledgeabout departurehos		
agenthasknowledgeabout departurehos		
agenthasknowledgeabout emaworker		
agenthasknowledgeabout transport1pa		
agenthasknowledgeabout requiredtreat		
agenthasknowledgeabout medicalvehic		
agenthasknowledgeabout transport1departuretime		

Fig. 9 Service failure

composer software of Mindwap lab [41] allows interaction with the user, but it works with only information providing services. It only considers inputs and outputs of the services but not preconditions and effects. Our framework on the other hand supports world altering web services; allows user interaction; and considers preconditions and effects. It has more functionality as an integrated framework. Therefore the available softwares are not directly comparable with our system.

However the existing approaches may be compared with respect to the planners they use, since the bottleneck of the systems is usually the planning algorithm. If the used planner is effective then the system becomes effective too. Simplanner is compared with similar planners and it is shown that Simplanner is timely effective [7]. As a result our system is effective. If something unexpected occurs during operation, its handling should be done in a timely manner with the help of Simplanner's algorithm.

Experiments that are presented in [7] show that the deliberation time is high at the initial steps when compared to the final steps, since the goal is far away. However, even in the initial states the time required for an action decision can be limited to less than the execution time of a particular service. According to experiments, unexpected situations do not increase the action production time, so dealing with the unexpected situations can be done in a responsive way.

8 Conclusions and future work

This paper proposes a novel solution for performing automated web service composition and invocation. Users present their requests to the proposed framework and the system handles the required analysis. It finds the required web services, the execution order of the found web services and possible data transfers among them.

In the literature, a considerable amount of work on the automated WSC problem has been conducted, but there is no complete solution to the problem yet. The literature lists several open problems. There are solutions proposed to solve some of these problems but the open problems are not independent of each other. For instance proposing a re-planning component without a real interaction is not sufficient alone, dynamic object generation is part of integrating planning and execution. The execution component without transactional properties and without user interaction is unacceptable in real world. Therefore we propose the framework in this paper as a solution to the open problems in one framework. This paper proposes the use of an AI planner, namely Simplanner, and it also proposes integrating the web service transaction frameworks (i.e. WS-Coordination and WS-Business Activity) to its automatic web service invocation mechanism. The solution that is proposed by this paper is highly adaptable, which makes it appropriate for real world applications.

The important features of the proposed "automated web service composition and invocation framework" can be summarized as follows:

- It is highly fault tolerant, which is very important for real world applications.

The proposed system interleaves planning and execution. If any problem occurs during service execution, the unexpected event handler is fired. The unexpected event handler initially tries to resolve the problem at a high level by using Simplanner's dynamic replanning features. If it cannot solve the problem at high level, physical operations are conducted. Since world altering service calls are done by conforming the WS-Business Activity specification, in failure situations compensation mechanisms are fired which prevent undesired side effects.

- It is responsive; users do not to wait for long periods of time to get the results of their requests.

The component that determines the time for responding to the user is the AI planner, since computationally complex operations are handled by it. Simplanner finds the first best action in polynomial time, and during real execution it finds the next action. The deliberation time required by the planner is very short, and sometimes this short deliberation time can be eliminated. The system has an action caching mechanism. If a similar problem has previously been solved by the system, the precompiled solutions are directly used. Shortly, the proposed system is highly responsive.

- Users do not need to provide an excessive amount of information initially; the system asks for only the required information during the composition process.

It is impossible for users to know which services are going to be used in the handling of their requests, so they cannot provide all required inputs initially. In the proposed system, users initially give a very high level description of their requests. The system discovers the required services for the high level definitions and asks the user for the required input. If users cannot provide the input, it discovers new paths that do not require that particular information or, if possible, it discovers other web services that can provide the particular information.

- The dynamic object generation of web services is modeled by using in-memory data structures and with continuous user interactions.

Web services sometimes produce information which is not available initially. The system, constructs an object for each type definition of the worked domain. Their availability and unavailability is represented by using logical statements and determined according to user interactions.

The paper provides these important contributions to the automated web service composition and invocation problem. However, some important future work still exists. The most important future work is providing scalability. Almost all domain independent AI planners fail to work with more than thousands of actions, which is a very small number for real world cases. One of the possible solutions is to do some kind of filtering before transferring the problem to the planner. A filtering operation will eliminate irrelevant web services according to the user's goal and give the planner a problem with a reasonable search space. A filtering procedure may eliminate some of the relevant services as well, but it is acceptable, otherwise the proposed system cannot be used in the real world where there exist millions of web services.

Another direction for future study is to include more syntactic analysis to the system. In the current system, sometimes the inputs of the service cannot be provided by the users but by other services. In this case, semantic types are compared. However semantic type similarity does not mean that syntactic types are equal too. For instance, the output of service A provides the input of service B, that is; the output of A and the input of service B has the same semantic type, but their WSDL counterparts may be different. Such cases are rare, but they are problematic situations which should be solved by a syntactic analysis.

Another future work is to increase the user's involvement in the WSC procedure. The users can see the results of the executed services immediately and according to those results they might be able to direct the system. If a service returns an undesired result, they will be able to invalidate that service for that session through the interfaces. In the current system, service invalidations are conducted automatically for the unexpected situations. This mechanism can easily be used by the users themselves manually when desired.

Acknowledgements The bulk of this work was completed while Mehmet Kuzu was supported by the Scientific and Technical Council of Turkey (TUBITAK). The authors would like to thank anonymous referees for many helpful suggestions which have improved the presentation of the paper. The authors would also like to thank one of the creators of Simplanner Dr. Oscar Sapena for helping them understand the details of Simplanner.

References

1. Rao J, Su X (2004) A survey of automated web service composition methods. In: Proceedings of 1st international workshop on semantic web services and web process composition, pp 43–54

2. Milanovic N, Malek M (2004) Current solutions for web service composition. IEEE Trans Internet Comput 8(6):51–59
3. Srivastava B, Koehler J (2003) Web service composition—current solutions and open problems. In: Proceedings of ICAPS'03 workshop on planning for web services, Trento, Italy, 2003, pp 28–35
4. Agarwal V, et al (2008) Understanding approaches for web service composition and execution. In: Proceedings of the 1st Bangalore annual compute conference, India, 2008, pp 1–8
5. Alamri A, Eid M, Saddik AE (2006) Classification of the state-of-the-art dynamic web services composition techniques. Int J Web Grid Serv 2(2):148–166
6. Polleres A (2004) AI planning for web service composition. Presentation, Ilog, Paris, France, 2004. http://axel.deri.ie/~axepol/presentations/20040907-paris-ilog-AIplanning4WSC.ppt
7. Sapena O, Onaindia E (2007) Planning in highly dynamic environments: an anytime approach for planning under time constraints. J Appl Intell 29(1):90–109
8. OASIS (2006) Web services business activity specification. http://docs.oasis-open.org/ws-tx/wsba/2006/06
9. OWL-S Semantic markup for web services. http://www.w3.org/Submission/OWL-S/
10. Java Reflection. http://java.sun.com/docs/books/tutorial/reflect/index.html
11. Sirin E, Parsia B, Wu D, Hendler J, Nau D (2004) HTN planning for web service composition using SHOP2. J Web Semant 1(4):377–396
12. Nau D, Au TC, Ilghami O, Kuter U, Murdock W, Wu D, Yaman F (2003) SHOP2: an HTN planning system. J Artif Intell Res (JAIR) 20:379–404
13. OASIS (2006) Web services coordination specification. http://docs.oasis-open.org/ws-tx/wscoor/2006/06
14. Klusch M, Gerber A, Schmidt M (2005) Semantic web service composition planning with OWLS-XPlan. In: Proceedings of the AAAI fall symposium on semantic web and agents, Arlington VA, USA. AAAI Press, Menlo Park
15. Hoffmann J (2003) The metric-FF planning system: translating ignoring delete lists to numeric state variables. J Artif Intell Res (JAIR) 20:291–341
16. Klusch M, Renner K-U (2006) Fast dynamic re-planning of composite OWL-S services. In: Proceedings of IEEE/WIC/ACM international conferences on web intelligence and intelligent agent technology—workshops, 2006, pp 134–137
17. Peer J (2004) A PDDL based tool for automatic web service composition. In: Proceedings of the 2nd international workshop on principles and practice of semantic web reasoning, 2004, pp 149–163
18. WSPlan http://sourceforge.net/projects/wsplan/
19. Peer J (2005) Semantic service markup with SESMA. In: Proceedings of the web service semantics workshop (WSS'05) at the 14th international world wide web conference (WWW'05), 2005, Chiba, Japan, pp 100–116
20. Younes HLS, Simmons RG (2003) VHPOP: versatile heuristic partial order planner. J Artif Intell Res (JAIR) 20:405–430
21. Gerevini A, Saetti A, Serina I (2003) Planning through stochastic local search and temporal action graphs. J Artif Intell Res 20(1):239–290
22. Agarwal V, Dasgupta K, Karnik N, Kumar A, Kundu A, Mittal S, Srivastava B (2005) A service creation environment based on end to end composition of web services. In: Proceedings of the 14th international conference on world wide web, Chiba, Japan, 2005, pp 128–137
23. WS-BPEL http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
24. Bartalos SP, Bieliková M (2008) Enhancing semantic web services composition with user interaction. In: Proceedings of the IEEE international conference on services computing (SCC), Honolulu, Hawaii, USA, 2008, pp 503–506
25. Marconi SA, Pistore M, Traverso P (2008) Automated composition of web services: the ASTRO approach. IEEE Data Eng Bull 31(3):23–26
26. Digiampietri LA, Alcázar JJP, Medeiros CB (2008) AI planning in web services composition: a review of current approaches and a new solution. In: Proceedings of the XXVII Brazilian computer society conference (CSBC), July 2008
27. Wiesner K, Vaculín R, Kollingbaum MJ, Sycara KP (2009) Recovery mechanisms for semantic web services. In: Meier R, Terzis S (eds) International conference on distributed applications and interoperable systems (DAIS). LNCS, vol 5053. Springer, Berlin, pp 100–105
28. Kazhamiakin R, Bertoli P, Paolucci M, Pistore M, Wagner M (2009) Having services "YourWay!": towards user-centric composition of mobile services. In: Future Internet FIS 2008. LNCS, vol 5468. Springer, Berlin, pp 94–106
29. Bryce D, Kambhampati S (2007) A tutorial on planning graph-based reachability heuristics. AI Mag 28(1):47–83
30. Ghallab M, Howe A, Knoblock C, McDermott D, Ram A, Veloso M, Weld D, Wilkins D (1998) PDDL: the planning domain definition language, AIPS-98 planning committee
31. Smith MK, Welty C, McGuinness DL OWL web ontology language guide. http://www.w3.org/TR/owl-guide/
32. Christensen E, Curbera F, Meredith G, Weerawarana S Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl
33. Kim H, Kim I (2007) Mapping semantic web service descriptions to planning domain knowledge. Proc IFMBE 14(1):388–391
34. OWLS2PDDL tool http://projects.semwebcentral.org/projects/owls2pddl/
35. WSIF Web services invocation framework. http://ws.apache.org/wsif/
36. Axis Apache web services project. http://ws.apache.org/axis/
37. OASIS (2006) Web services atomic transaction specification. http://docs.oasis-open.org/ws-tx/wsat/2006/06
38. Erven H, Hicker G, Huemer C, Zaptletal M (2007) The web services-business activity-initiator (WS-BA-I) protocol: an extension to the web services-business activity specification. In: IEEE international conference on web services (ICWS), 2007, Salt Lake City, pp 216–224
39. Kandula Apache WS-transaction project. http://ws.apache.org/kandula/, http://ws.apache.org/axis/
40. OWLS-XPLAN http://projects.semwebcentral.org/projects/owls-xplan/
41. Mindswap Web service composer software. http://www.mindswap.org/~evren/composer/
42. Srivastava B (2004) A software framework for building planners. In: Proceedings of knowledge based computer systems (KBCS), 2004, Hyderabad, pp 382–392

Mehmet Kuzu is currently a PhD student in Computer Science department at the University of Texas at Dallas. He received his BS and MS degrees in Computer Engineering at the Middle East Technical University, Ankara, Turkey, in 2007 and 2009, respectively. His research areas include intelligent systems, data mining, security and privacy issues related to the management of data.

Nihan Kesim Cicekli is an Associate Professor in the Department of Computer Engineering at the Middle East Technical University, Ankara, Turkey. She received her BSc degree in Computer Engineering at the Middle East Technical University in 1986. She received the MSc degree in Computer Engineering at Bilkent University in Ankara in 1988; and the PhD degree in Computer Science at Imperial College, London, UK in 1993. She was a visiting associate professor at the University of Central Florida, Orlando, USA, from 2001 till 2003. Her current research interests include multimedia databases, semantic web, web services, workflow management systems, recommender systems and temporal reasoning. She is a member of IEEE. She served on the program committee of several international conferences including VLDB and ICDE. For more information about her, see http://www.ceng.metu.edu.tr/~nihan.