Automated Web Services Composition with the Event Calculus*

Onur Aydın[1] Nihan Kesim Cicekli[2] Ilyas Cicekli[3]

[1]Microsoft Corporation, Seattle, U.S.A.
[2]Department of Computer Engineering, METU, Ankara , Turkey
[3]Department of Computer Engineering, Bilkent University, Ankara, Turkey
onura@microsoft.com, nihan@ceng.metu.edu.tr, ilyas@cs.bilkent.edu.tr

Abstract. As the web services proliferate and complicate it is becoming an overwhelming job to manually prepare the web service compositions which describe the communication and integration between web services. This paper analyzes the usage of the Event Calculus, which is one of the logical action-effect definition languages, for the automated preparation and execution of web service compositions. In this context, abductive planning capabilities of the Event Calculus are utilized. It is shown that composite process definitions in OWL-S can be translated into Event Calculus axioms so that planning with generic process definitions is possible within this framework.
Keywords: Event Calculus, Web Service Composition, Planning

1 Introduction

Web services can be described as a set of related functionalities that can be programmatically accessed through the web protocols [2]. The distribution of the functions of the business through web services helped a lot to integrate services of different companies. However as the web applications flourished and the number of web services increase another difficulty appeared in the horizon. Application integrators now are concerned with finding the correct web service that meets the demands of the customer while building the applications. In such a dynamic domain, automatic integration or composition of web services would be helpful since the unknowns of the demands are too much or too diverse. This brings us to the problem of automatic web services composition.

Given a repository of service descriptions and a service request, the *web service composition problem* involves finding multiple web services that can be put together in correct order of execution to obtain the desired service. Finding a web service that can fulfill the request alone is referred to as *web service discovery problem*. When it is impossible for one web service to fully satisfy the request, one the other hand, one has to compose multiple web services, in sequential or parallel, preferably in an automated fashion.

However, automated composition of services is a hard problem and it is not entirely clear which techniques serve the problem best. One family of techniques that

* This work is supported by the Scientific and Technical Research Council of Turkey, METU-ISTEC Project No: EEEAG 105E068

has been proposed for this task is AI planning. The general assumption of such kind of methods is that each Web service can be specified by its preconditions and effects in the planning context. The preconditions and effects are the input and the output parameters of the service respectively. In general, if the user can specify the preconditions and effects required by the composite service, a plan or process is generated automatically by logical theorem prover or AI planners. The automation of web services composition may mean two things: either the method can generate the process model automatically, or the method can locate the correct services if an abstract process model is given [18]. In this paper we are concerned with defining an abstract process model.

Recently, a considerable amount of work has investigated the potentials and boundaries of applying AI planning techniques to derive web service processes that achieve the desired goals [7,10,12,15,17,18,24]. As mentioned in [17], the event calculus [6] is one of the convenient techniques for the automated composition of web services. In this paper we aim to demonstrate how the event calculus can be used in the solution of this problem. Our goal is to show that the event calculus can be used to define an abstract composite process model and produce a user specific composition (plan). Abductive planning of the event calculus [21] is used to show that when atomic services are available, the composition of services that would yield the desired effect is possible. The problem of matching the input/output parameters to find a web service in a given repository is out of the scope of this work. We assume that these matching tasks are pre-processed and selected.

The idea of using the event calculus in the context of web services and interactions in multiagent systems is not new [4,19,23,25]. In [4], an event calculus based planner is used in an architecture for automatic workflow generation on the Web/Grid. This work is closely related to our work, however since the details of the formalism is not given it is not possible to compare it with ours. In [19] the event calculus has been used in verifying composed web services, which are coordinated by a composition process expressed in WSBPEL. Here the aim is to verify a composed service, not generating the composition itself. The work in [23] attempts to establish a link between agent societies and semantic web-services, but uses another version of the event calculus which avoids abduction and stick to normal logic programs. In [25] an approach for formally representing and reasoning about commitments in the event calculus is developed. This approach is applied and evaluated in the context of protocols, which represent the interactions allowed among communicating agents.

In this paper our aim is to contribute the research along this direction by presenting a formal framework that shows how generic composition procedures are described in the event calculus to produce specific plans for the requested goals. Our main contribution is the translation of OWL-S to event calculus and demonstrating how planning with complex actions is done within this framework. We present the event calculus framework as an alternative approach for building agent technology, based on the notion of generic procedures and customizing user constraints.

The rest of the paper is organized as follows. Section 2 gives insight information about current technologies, the web service composition problem and techniques used to solve the problem. In Section 3, the event calculus as a logical formalism and its abductive implementation are explained. In Section 4, we present the use of abductive event calculus in the solution of automated web services composition problem. Sec-

tion 5 presents a translation of OWL-S service descriptions to the event calculus and how the abductive event calculus can be used to define abstract process model needed for composition. Finally, Section 6 presents conclusions and possible future work.

2 Related Work

Building composite Web services with an automated or semi-automated tool is a critical and hard task. This problem has received a lot of attention recently [18]. In the literature, AI planning algorithms have been widely used to automatically compose web services [9,10,17,24]. Most apparent reason behind this preference is the great similarities between these two fields.

Both the planning problem and composition problem seek a (possibly partially) ordered set of operations that would lead to the goal starting from an initial state (or situation). Operations of the planning domain are actions (or events) and operations of the composition domain are the web services. Like actions, Web services have parameters, preconditions, results and effects hence they are very attractive to be used in conventional planning algorithms.

Viewing the composition problem as an AI planning problem, different planners are employed for the solution. An excellent survey of modern planning algorithms and their application to web service composition problem can be found in [17]. Here we highlight some of the existing work that is most relevant to our approach.

Estimated-regression is a planning technique in which the situation space is searched with the guide of a heuristic that makes use of backward chaining in a relaxed problem space [10]. In this approach, the composition problem is seen as a PDDL planning problem and efforts are condensed to solve the problem in PDDL domain referring to the common difficulties of Web Services domain. In fact, a translator [11] has been written which converts DAML-S (former version of OWL-S) and PDDL into each other. This shows that the composition problem can be (informally) reduced to a planning problem and in that sense working in PDDL domain is not much different indeed.

In [12], web service composition problem is assumed to be the execution of generic compositions with customizable user constraints. GOLOG [8], which is a situation calculus implementation with complex actions, is used to write the generic processs model (complex action). It is said to be generic since it is not executable without user constraints. After the user specifies the constraints, it is executed and the solver tries to generate the plan according to the runtime behavior of the services. The output of this method is a running application which satisfies the user requests.

Hierarchical Task Network (HTN) planning has been applied to the composition problem to develop software to automatically manipulate DAML-S process definitions and find a collection of atomic processes that achieve the task [24]. SHOP2, an HTN planner, is used to generate plans in the order of its execution. This work has recently been extended into another planning algorithm called Enquirer, which provides information gathering facilities during planning [7].

In [15], a taxonomy is presented for the classification of the web service composition problem. This taxonomy is used to help select the right solution for the composi-

tion problem at hand. According to this classification, the Event Calculus based approach falls in the category of AI planning methods that best suits to the solution of small scale and simple operator based automated web service compositions.

3 Event Calculus

Event calculus [6] is a general logic programming treatment of time and change. The formulation of the event calculus is defined in first order predicate logic like the situation calculus. Likewise, there are actions and effected fluents. Fluents are changing their valuations according to effect axioms defined in the theory of the problem domain. However there are also big differences between both formalisms. The most important one is that in the event calculus, narratives and fluent valuations are relative to time points instead of successive situations. The most appearing advantage of this approach is the inherent support for concurrent events. Events occurring in overlapping time intervals can be deduced. Inertia is an assumption, which accounts a solution to the frame problem together with other techniques and it is saying that a fluent preserves its valuation unless an event specified to affect (directly or indirectly) the fluent occurs.

Each event calculus theory is composed of axioms†. A fluent that holds since the time of the initial state can be described by the following axioms [20]:

$holdsAt(F, T) \leftarrow initially(F) \wedge \neg clipped(t_0, F, T)$
$holdsAt(\neg F, T) \leftarrow initially(\neg F) \wedge \neg declipped(t_0, F, T)$

Axioms below are used to deduce whether a fluent holds or not at a specific time.

$holdsAt(F, T) \leftarrow$
$\quad happens(E, T_1, T_2) \wedge initiates(E, F, T_1) \wedge \neg clipped(T_1, F, T) \wedge T_2 < T$
$holdsAt(\neg F, T) \leftarrow$
$\quad happens(E, T_1, T_2) \wedge terminates(E, F, T_1) \wedge \neg declipped(T_1, F, T) \wedge T_2 < T$

The predicate *clipped* defines a time frame for a fluent that is overlapping with the time frame of an event which terminates this fluent. Similarly *declipped* defines a time frame for a fluent which overlaps with the time frame of an event that initiates this fluent. The formula *initiates*(E, F, T) means that fluent F holds after event E at time T. The formula *terminates*(E, F, T) denotes that fluent F does not hold after event E at time T. The formula *happens*(E, T_1, T_2) indicates that event E starts at time T_1 and end at time T_2. The instantaneous events are described as *happens*(E, T_1, T_1).

$clipped(T_1, F, T_4) \leftrightarrow (\exists E, T_2, T_3)\ [\ happens(E, T_2, T_3) \wedge$
$\quad terminates(E, F, T_2) \wedge T_1 < T_3 \wedge T_2 < T_4\ ]$
$declipped(T_1, F, T_4) \leftrightarrow (\exists E, T_2, T_3)\ [\ happens(F, T_2, T_3) \wedge$
$\quad initiates(E, F, T2)\ \wedge\ T1 < T3 \wedge T2 < T4\ ]$

† Variables begin with upper-case letters, while function and predicate symbols begin with lower-case letters. All variables are universally quantified with maximum possible scope unless otherwise indicated.

3.1 Abductive Event Calculus

Abduction is logically the inverse of deduction. It is used over the event calculus axioms to obtain partially ordered sets of events. Abduction is handled by a second order abductive theorem prover (ATP) in [21]. ATP tries to solve the goal list proving the elements one by one. During the resolution, abducible predicates, i.e. < (temporal ordering) and *happens*, are stored in a residue to keep the record of the narrative. The narrative is a sequence of time-stamped events, and the residue keeping a record of the narrative is the plan.

In this paper, the predicate *ab* is used to denote the theorem prover. It takes a list of goal clauses and tries to find out a residue that contains the narrative. For each specific object level axiom of the event calculus, a meta-level *ab* solver rule is written. For example an object level axiom in the form:

$$A_H \leftarrow AB_1 \wedge AB_2 \wedge \ldots \wedge AB_N$$

is represented with the predicate *axiom* in the ATP theory and it is translated to:

$$axiom(AH, [AB_1, AB_2, \ldots, AB_N])$$

During the resolution process axiom bodies are resolved by the *ab* which populates the *abducibles* inside the residue. A simplified version of *ab* solver is as follows.

$ab([],RL,RL,NL).$
$ab([A|GL],CurrRL,RL,NL) \leftarrow abducible(A),\ NewRL = [A|CurrRL],\ consistent(NL,NewRL),\ ab(GL,NewRL,RL,NL).$
$ab([A|GL],CurrRL,RL,NL) \leftarrow axiom(A,AL),\ append(AL,GL,NewGL),\ ab(NewGL,CurrRL,RL,NL).$
$ab([not(A)|GL],CurrRL,RL,NL) \leftarrow irresolvable(A,CurrRL),\ ab(GL,CurrRL,RL,[A|NL]).$

In this definition *GL* denotes the goal list, *RL* represents the residue list, *NL* represents the residue of negated literals, *A* is the axiom head and *AL* is the axiom body. Intuitively, the predicate *abducible* checks if the axiom is abducible. If it is so, it is added to the residue. If it is an axiom then its body is inserted into the goal list to be resolved with other axioms. Negated literals are proven by negation as failure (NAF). However as the residue grows during the resolution, the negative literals, which were previously proven, might not be proven anymore. This situation may occur when negative literals were proven due to the absence of contradicting evidence; however the newly added literals might now allow the proof of the positive of literals, invalidating the previous negative conclusions. For that reason, whenever the residue is modified, previously proven negated literals should be rechecked. The predicate *irresolvable* checks whether the negated literal is resolvable with the current residue or not. The negative literal in question might also mention a non-abducible predicate. In this case it needs to be resolved with the axioms not the residue. This possibility is studied in [21]. The predicate *consistent* checks that none of the negated literals is resolvable with the current narrative residue using the predicate *irresolvable* for each negated literal.

4 Web Services Composition with Abductive Planning

The event calculus can be used for planning as it is theoretically explained in [21]. The planning problem in the event calculus is formulated in simple terms as follows: Given the domain knowledge (i.e. a conjunction of *initiates, terminates*), the Event Calculus axioms (i.e. *holdsAt, clipped, declipped*) and a goal state (e.g. *holdsAt*(*f,t*)), the abductive theorem prover generates the plan which is a conjunction of *happens* i.e. the narrative of events, and temporal ordering predicates, giving the partial ordering of events.

4.1. Web Services

In the event calculus framework, the web services are modeled as events with input and output parameters. For instance, if a web service returns the availability of a flight between two locations, its corresponding event is given in Fig.1.

```
-- web service description
<message name='GetFlight_Request'>
<part name='Origin' type='xs:string'>
<part name='Destination' type='xs:string'>
<part name='Date' type='xs:date'>
</message>
<message name='GetFlight_Response'>
<part name='FlightNum' type='xs:string'>
</message>

-- event
getFlight(Origin, Destination, Date, FlightNum)
```

Fig. 1. Web Service to Event Translation

The web service operation *GetFlight* is translated to the event *getFlight*. The inputs and outputs of the web service are translated as parameters of the event. The invocation of the web service is represented with the predicate *happens*:

```
happens(getFlight(Origin,Destination,Date,FlightNum), T1, T1) ←
        ex_getFlight(Origin, Destination, Date, FlightNum).
```

The parameters of the event are populated with help of the predicate *ex_getFlight* which is a call to the actual web service. This predicate is used as a precondition for the event and it is invoked anytime it is added to the plan. In order to resolve literals which are non-axiomatic assertions such as conditions or external calls *ab* is extended to contain the following rule:

$$ab([L|GL],\ CL,\ RL,\ NL) \leftarrow \neg axiom(L) \wedge L \wedge ab(GL,\ CL,\ RL,\ NL)$$

In this rule *L*, *GL*, *RL* and *NL* denote, respectively, the non-axiomatic literal, the goal list, the narrative residue and the negation residue. If a non-axiomatic literal is encountered then *ab* directly tries to prove the literal and if it is successful it continues with rest of the goal list.

In ATP implementation, the external call bindings like the predicate *ex_getFlight* are loaded from an external module that is written in C++ programming language. After invoking the associated service, flight number is unified with *FlightNum*, the last parameter of *getFlight* event.

Let us assume that we have the following specific axioms for a very simple travel domain.

```
axiom( happens(getFlight(Origin,Dest,Date,FlightNum), T, T),
     [ ex_getFlight(Origin,Dest,Date,FlightNum)] ).
axiom(initiates(getFlight(Origin,Dest,Date,FlightNum),
       at_location(Dest),T),
     [ holdsAt(at_location(Origin),T), Origin \== Dest ] ).
axiom(terminates(getFlight(Origin,Dest,Date,FlightNum),
       at_location(Origin),T),
     [ holds_at(at_location(Origin),T), Origin \== Dest ] ).
axiom( initially(at_location(ankara)), []).
```

There are two non-axiomatic literals, namely \== and *ex_getFlight* in the bodies of the axioms. The predicate \== checks whether two bound variables are different or not. The predicate *ex_getFlight* represents an external web service operation, and it returns the flight number for the given origin and destination cities. Thus, the parameters of the *getFlight* event are populated. The *initiates* and *terminates* axioms describe how the fluent *at_location* is affected by *getFlight* event. The *initially* axioms says that our initial location is the city *ankara*. Let us assume that, we only have the following three flights in our travel domain, and the predicate *ex_getFlight* returns these flights one by one.

getFlight(*ankara, izmir, tk101*).
getFlight(*ankara, istanbul, tk102*).
getFlight(*istanbul, izmir, tk103*).

In order to find the possible plans for the goal of being in *izmir*, the abductive theorem prover is invoked with the goal *ab*([*holdsAt*(*at_location*(*izmir*), *t*)], [], *RL*, []). The theorem can find the following two plans one by one.

plan1: [*happens*(*getFlight*(*ankara, izmir, tk101*), *t1, t1*), $t1 < t$]
plan2: [*happens*(*getFlight*(*ankara,istanbul,tk102*),*t2,t2*),
happens(*getFlight*(*istanbul,izmir,tk103*), *t1, t1*), $t2 < t1$, $t1 < t$]

Here each plan contains the time stamped *happens* predicates and temporal ordering between these time stamps. The time constants in the plan (*t1* and *t2*) are generated by the abductive reasoner. The abductive planner binds the given time parameter to a unique constant if the time parameter is an unbound variable.

4.2 Plan Generation

ATP returns a valid sequence of time stamped events that leads to the goal. If there are several solutions they are obtained with the help of backtracking of Prolog. Multiple solutions could be thought as different branches of a more general plan. For instance, assume that the following event sequence is generated after a successful resolution process.

happens(E_1, T_1 , T_1). *happens*(E_2, T_2, T_2). *happens*(E_3, T_3 , T_3). $T_1 < T_2 < T_3$

It can be concluded that when executed consecutively, the ordered set $\{E_1, E_2, E_3\}$ generates the desired effect to reach the goal. In addition to this plan, alternative solutions could be examined. In order to do such a maneuver, the executer should have a tree like plan where proceeding with alternative paths is possible. Assume that the following totally ordered sequences of events also reach the same goal.

$\{E_1, E_5, E_4\}$, $\{E_1, E_2, E_4\}$, $\{E_6, E_7\}$

When these separate plans are combined, a graph which describes several compositions of web services, is formed (see Fig. 2). In this graph the nodes represent the events (web services) and *CS* is the start of composition (i.e. the initial state). The nodes with the label *Exclusive-Or-Split* (*XOr*) represent alternative branches among which only one could be chosen. Also several alternative paths are joined in the nodes with the label *XOr-Join. XOr-Joins* mandate that only one of the branches is active at the joining side. This graph contains all plans (i.e. composite services) generated by the planner. This graph is used to evaluate the composed services according to the non-functionality values such as cost, quality and response time, and the best plan can be chosen afterwards, which will be executed by the execution engine.

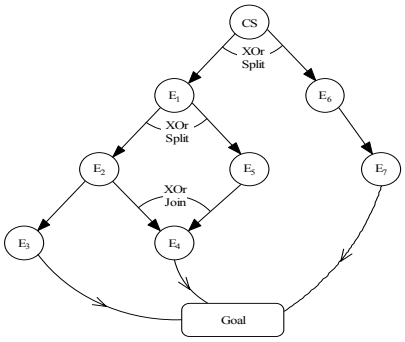

Fig. 2. All generated compositions

4.3 Concurrency of Events

The narratives generated by the ATP are partially ordered set of events. Due to the partial ordering, events, for which a relative ordering is not specified, can be thought to be concurrent. For instance, assume ATP has generated the following narrative:

happens(E_1, T_1 ,T_1). *happens*(E_2, T_2 , T_2). *happens*(E_3, T_3, T_3).

happens(E_4, T_4 ,T_4). $T_1 < T_2 < T_4$, $T_1 < T_3 < T_4$

Since there is no relative ordering between E_2 and E_3 they are assumed to be concurrent. If this is the only narrative generated by the ATP then the plan can be shown as in Fig. 3.

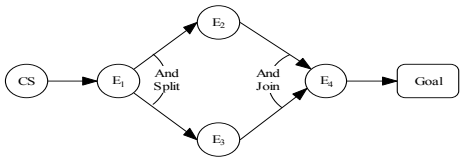

Fig. 3. Concurrent Composition

In this graph, concurrent events are depicted as *And-Split* since both of the branches should be taken after the event E_1. Before the event E_4 *And-Join* is required since both of E_2 and E_3 should be executed.

5 Web Services Composition with Generic Process Definition

In the literature, one of the most promising leaps on automating the Web Service Composition is taken with the OWL-S language [16]. In OWL-S, Web Services are abstracted, composed and bound to concrete service providers. Web Services are composed by a series of operations, which atomically provide certain functions. Service interactions can be as simple as a single operation invocation (e.g. http://www.random.org returns random numbers with a single Web Service operation). They can be as complicated as a multi-department electronic commerce site for shopping, where catalog browsing, item selection, shipment arrangements and payment selection are accomplished by invoking a series of operations. (e.g. Amazon Web Service http://www.amazon.com).

Several atomic processes constitute a *Composite Process w*hen connected with the flow control constructs of OWL-S. If an automated system requires the provided service it should execute the composite processes as they are defined in the OWL-S, supplying the intermediate inputs to the atomic services nested under them.

The Event Calculus framework can be used to define composite processes (i.e. complex goals) and ATP can be used to generate a plan which corresponds to the user specific composition of the web service. Composite processes will correspond to compound events in the Event Calculus [3]. Like the composite processes, compound events provide the grouping of sub-events. In the following sections, first, an OWL-S to event calculus translation scheme is presented to show that OWL-S composition constructs can be expressed as event calculus axioms[‡]. Then an example application will be presented to illustrate the use of generic process definition and its use in the abductive event calculus planner.

5.1 OWL-S to Event Calculus Translation

Composite processes are composed of control constructs which closely resemble to standard workflow constructs. Since further composite processes can be used inside a

[‡] For readability purposes, we will omit *axiom* predicate in the rest of the paper and present object level axioms only. However, note that they are converted into the *axiom* predicate in the implementation.

composite process, the translation is recursively applied until all composite processes are replaced with the corresponding axioms that contain atomic processes. Here we present an OWL-S to event calculus translation scheme. The automatic mapping is possible, but we have not implemented it yet.

5.1.1 Atomic Processes

Atomic processes are translated into simple events of the Event Calculus. An abstract representation of an atomic process of OWL-S is given in Fig. 4.

```
Atomic Process<A, V, P, E, O, E^C, O^C>
  A : Atomic Process Functor
  V : Set of Inputs: {V_1, V_2, ..., V_N}
  P : Preconditions: Conjunction of Literals (P_1 ∧ P_2 ∧ ... ∧ P_M)
  E : Effects: Conjunction of Literals (E_1 ∧ E_2 ∧ ... ∧ E_K)
  O : Outputs: Set of Outputs {O_1, O_2, ..., O_L}
  E^C : Conditional Effects: Set of literals {E^C_1, E^C_2, ..., E^C_R}
        where each E^C_i has a condition such as
        E^C_i ← BE^C_i : BE^C_i are conjunction of literals
  O^C : Conditional outputs
```

Fig. 4. Atomic Process Definition of OWL-S

This definition is translated to the Event Calculus as an event with the same name as the atomic process *A* and the effect axioms are defined according to the preconditions and effects. The translation is given in Fig. 5.

```
initiates(A(V, O), E_i, T) ← holdsAllAt(P, T) ∧ invoke(A, V, O, T)
        where E_i ∈ E^+ (positive literals of E)
terminates(A(V, O), E_i, T) ← holdsAllAt(P, T) ∧ invoke(A, V, O, T)
        where E_i ∈ E^- (negative literals of E)
initiates(A(V, O), E^C_i, T) ← holdsAllAt(P, T) ∧
                holdsAllAt(BE^C_i, T) ∧ invoke(A, V, O, T)
        where E^C_i ∈ E^C+
terminates(A(V, O), E^C_i, T) ← holdsAllAt(P, T) ∧
                holdsAllAt(BE^C_i, T) ∧ invoke(A, V, O, T)
        where E^C_i ∈ E^C-
holdsAllAt({F_1, F_2, ..., F_Z}, T) ↔ holdsAt(F_1, T) ∧
                holdsAt(F_2, T) ∧ ... ∧ holdsAt(F_Z, T)
```

Fig. 5. Atomic Process Translation

The meta predicate *holdsAllAt* has an equivalent effect of conjunction of *holdsAt* for each fluent that *holdsAllAt* covers. The predicate *invoke* is used in the body of effect axioms to generate the desired outputs (it corresponds to the invocation of external calls through the happens clause as illustrated in the example in Section 4.1). It takes the name of the atomic process, input parameters and unifies the outputs with the results of the corresponding Web Service operation invocation.

5.1.2 Composite Process Translation

Composite processes combine a set of processes (either atomic or composite) with different control constructs. An example composition which is composed of nested structures is given in Fig. 6. Split, Join and Repeat-While control constructs are used

in this composite process. It is necessary to be able to express such control constructs in the event calculus framework. This problem has been studied earlier in different contexts [3,5]. For the purpose of the web services composition problem, OWL-S constructs should be translated into compound events in the event calculus framework.

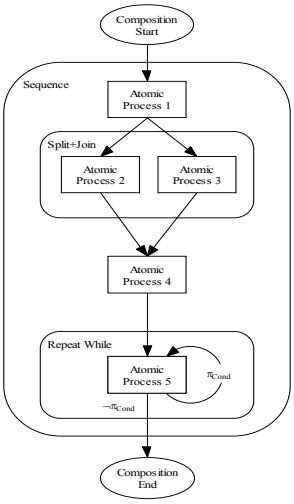

Fig. 6. Example of a Composite Process

The translation of some of the flow control constructs into the Event Calculus axioms is summarized in the following. Others can be found in [1]. Constructs are originally defined in XML (to be more precise in RDF) document structure however since they are space consuming only their abstract equivalents will be given.

Sequence

The *Sequence* construct contains the set of all component processes to be executed in order. The abstract OWL-S definition of the composite process containing a *Sequence* control construct and its translation into an Event Calculus axiom is given in Fig 7. The translation is accomplished through the use of compound events in the Event Calculus which contains sub-events. The sequence of events are triggered from the body of the compound event and the ordering between them is ensured with the predicate < (precedes).

Sequence Composite Process<C, V, P, S> *C : Composite Process Functor* *V : Set of Inputs* $\{V_1, V_2, \ldots, V_N\}$ *P : Preconditions* $(P_1 \wedge P_2 \wedge \ldots \wedge P_M)$ *S : Sequence of Sub-Processes* *Ordered set of* $\{S_1, S_2, \ldots, S_K\}$	$happens(C, T_1, T_N) \leftarrow$ $holdsAllAt(P, T_1) \wedge$ $happens(S_1, T_2, T_3) \wedge$ $happens(S_2, T_4, T_5) \wedge \ldots \wedge$ $happens(S_K, T_{2K}, T_{2K+1}) \wedge$ $T_1 < T_2 \wedge T_3 < T_4 \wedge \ldots \wedge$ $T_{2K-1} < T_{2K} \wedge T_{2K+1} < T_N$

Fig. 7. Sequence Composite Process

If-Then-Else

If-Then-Else construct contains two component processes and a condition. Its structure and translation are given in Fig. 8. Two *happens* axioms are written for both cases. With the help of *notholdsAllAt* which is logically the negation of *holdsAllAt*, the second axiom is executed when the *else-case* holds.

If-Then-Else Composite Process *<C, V, P, π, $S_π$, $S_{¬π}$>* *C : Composite Process Functor* *V : Set of Inputs {V_1, V_2, ..., V_N}* *P : Preconditions (P_1 ∧ P_2 ∧ ... ∧ P_M)* *π : If condition ($π_1$ ∧ $π_2$ ∧ ... ∧ $π_K$)* *$S_π$: If condition Sub-Process* *$S_{¬π}$: Else condition Sub-Process*	*happens(C, T_1, T_N) ←* *holdsAllAt(P, T_1) ∧* *holdsAllAt(π, T_1) ∧* *happens($S_π$, T_2, T_3) ∧* *T_1 < T_2 ∧ T_3 < T_N* *happens(C, T_1, T_N) ←* *holdsAllAt(P, T_1) ∧* *notholdsAllAt(π, T_1) ∧* *happens($S_{¬π}$, T_2, T_3) ∧* *T_1 < T_2 ∧ T_3 < T_N* *notholdsAllAt({F_1,F_2,...,F_F},T)* *↔ holdsAt(¬F_1, T) ∨* *holdsAt(¬F_2, T) ∨ ... ∨* *holdsAt(¬F_N, T)*

Fig. 8. If-Then-Else Composite Process

Repeat-While and Repeat-Until

Repeat-While and *Repeat-Until* constructs contain one component process and a loop controlling condition. The loop iterates as long as the condition holds for *Repeat-While* and does not hold for *Repeat-Until*. They have a common structure and it is given in Fig. 9. The figure presents the translation of *Repeat-While* only, since the translation of the other is similar. Two *happens_loop* axioms are written for both states of the loop condition. The composite event is triggered when preconditions hold. The body of the loop is recursively triggered as long as the loop condition permits. The preconditions and the loop condition are checked at time T_1. If they hold, the component process is invoked at a later time T_2.

Repeat-While/Unless Composite Process <C, V, P, π, $S_π$> *C : Composite Process Functor* *V : Set of Inputs {V_1, V_2, ..., V_N}* *P : Preconditions (P_1 ∧ P_2 ∧...∧ P_M)* *π : Loop condition ($π_1$ ∧ $π_2$ ∧...∧ $π_K$)* *$S_π$: Loop Sub-Process*	*happens(C, T_1, T_N) ←* *holdsAllAt(P, T_1) ∧* *happens_loop(C, π, T_1, T_N).* *happens_loop(C, π, T_1, T_N) ←* *holdsAllAt(π, T_1) ∧* *happens($S_π$, T_2, T_3) ∧* *happens_loop(C, π, T_4, T_5) ∧* *T_1 < T_2 ∧ T_3 < T_4 ∧ T_5 < T_N* *happens_loop(C, π, T_1, T_1) ←* *notholdsAllAt(π, T1)*

Fig. 9. Repeat-While/Unless Composite Process

In the given abstract translations, it may seem that the set of inputs are not used, but the actual translations spread out the contents of the set of inputs as the appropri-

ate parameters of the component processes. This is illustrated in the example given in Section 5.2.

5.2 Example of a Composition

In this section we illustrate the use of the abductive event calculus in generating compositions from a given composite procedure. The example illustrates how one can describe a complex goal and find a plan to achieve that goal.

The implementation of the traveling problem given in [12] is formulated in the Event Calculus. In [12], a generic composition is presented for the traveling arrangement task. In this procedure, the transportation and hotel booking are arranged and then mail is sent to the customer. Finally an online expense claim is updated. The transportation via air is selected with the constraint that it should be below the customer's specified maximum price. If the destination is close enough to drive by car then instead of air transportation, car rental is preferred. The customer specifies a maximum drive time for this purpose. If the air transportation is selected then a car is arranged for local transportation. Also a hotel is booked for residence at the destination.

Compound events are used to express generic compositions in the Event Calculus in a similar way that they have been used in OWL-S translation. The whole operation is decomposed into smaller tasks which are separately captured with other compound events [1]. The Event Calculus translation is given in Fig. 10.

$$
\begin{aligned}
&happens(travel(O, D, D_1, D_2), T_1, T_N) \leftarrow \\
&\quad [[happens(bookFlight(O, D, D_1, D_2), T_2, T_3) \wedge \\
&\quad happens(bookCar(D, D, D_1, D_2), T_4, T_5) \wedge T_3 < T_4] \vee \\
&\quad happens(bookCar(O, D, D_1, D_2), T_2, T_5)] \wedge \\
&\quad happens(bookHotel(D, D_1, D_2), T_6, T_7) \wedge \\
&\quad happens(SendEmail, T_8) \wedge \\
&\quad happens(UpdateExpenseClaim, T_9) \wedge \\
&\quad T_5 < T_6 \wedge T_7 < T_8 \wedge T_8 < T_9 \wedge T_9 < T_N \\
\\
&happens(bookFlight(O, D, D_1, D_2), T_1, T_N) \leftarrow \\
&\quad ex\_GetDriveTime(O, D, Tm) \wedge \\
&\quad Tm > userMaxDriveTime \wedge \\
&\quad ex\_SearchForFlight(O, D, D_1, D_2, Id) \wedge \\
&\quad ex\_GetFlightQuote(Id, Pr) \wedge \\
&\quad Pr < UserMaxPrice \wedge \\
&\quad ex\_BookFlight(Id) \\
\\
&happens(bookCar(O, D, D_1, D_2), T_1, T_N) \leftarrow \\
&\quad [[ex\_GetDriveTime(O, D, Tm) \wedge \\
&\quad Tm < userMaxDriveTime] \vee O = D] \wedge \\
&\quad ex\_BookCar(O, D, D_1, D_2)
\end{aligned}
$$

where O *: Origin,* D *: Destination,* D_1 *: Traveling Start Date,* D_2 *: Traveling End Date*

Fig. 10. Generic Composition in the Event Calculus

In this translation *userMaxDriveTime* and *userMaxPrice* are the user preference values which alter the flow of operations. Based on traveling inputs and user prefer-

ences the traveling arrangement is accomplished with the help of external Web Service calls (in Fig. 10 they are represented with predicates with *ex_* prefix). When this composition is implemented in the ATP, a residue which contains the sequence of events to arrange a travel will be returned as the plan. For instance let us assume that we have the definitions of several external web services for the atomic processes like *GetDriveTime, SearchForFlight, GetFlightQuote etc.*), and we have the following initiates axiom:

initiates(*travel*(*O,D,SDate,EDate*)*, travelPlanned*(*O,D,SDate,EDate*)*,T*).

If we want to find a travel plan from Ankara to Athens between the dates October 22 and October 24, we can invoke the ATP with the following goal:

ab([*holdsAt*(*travelPlanned*(*ankara,athens,october22,october24*)*, t*)]*, R*)

The variable *R* will be bound to a plan, for instance, of the following form:

```
[ hapens(updateExpenseClaim, t7, t7),
  happens(sendEmail, t6, t6),
  happens(bookHotel(athens, october22, october24), t5, t5),
  happens(bookCar(athens, athens, october22, october24), t4, t4),
  happens(bookFlight(ankara, athens, october22), t3, t3),
  happens(travel(ankara, athens, october22, october24), t1, t2)
  t7 < t2,  t6 < t7,  t5  < t6,  t4 < t5,  t3 < t4,  t1 < t3,  t2 < t ].
```

The plan shows which web services must be invoked for the composition and also the temporal ordering among them.

6 Conclusions

In this paper, the use of the event calculus has been proposed for the solution of web service composition problem. It is shown that when a goal situation is given, the event calculus can find proper plans as web service compositions with the use of abduction technique. It is also shown that if more than one plan is generated, the solutions can be compiled into a graph so that the best plan can be chosen by the execution engine.

In [24], SHOP2 is used to translate DAML-S process model into SHOP2 operators. This translation assumes certain constraints for the process model to be converted. The first assumption in SHOP2 translation is that the atomic processes are assumed to be either output generating or effect generating but not both. Atomic processes with conditional effects and outputs are not converted at all. Our translation supports atomic processes with outputs, effects and conditional effects. Another limitation of SHOP2 translation is the support for concurrent processes. Since SHOP2 cannot handle parallelism the composite constructs *Split* and *Split+Join* cannot be translated. On the other hand, our translation supports for these constructs since event calculus is inherently capable of handling concurrency.

Both the event calculus and GOLOG can be used to express composite process models. The most important difference between the Event Calculus and GOLOG is the syntax of the languages. GOLOG provides extra-logical constructs which ease the definition of the problem space as it is given in [12] for the same example above. These constructs can be easily covered with Event Calculus axioms too. Furthermore,

since the event calculus supports time points explicitly, it is easier to model concurrency and temporal ordering between actions in the Event Calculus. Therefore it is more suitable to the nature of web services composition problem with respect to expressiveness and ease of use.

As a future work, the results that are theoretically expressed in this paper will be put into action and implemented within a system which works in a real web environment. It would be helpful if a language is developed for the event calculus framework, in order to define common control structures of web service compositions in a more direct way. In fact, there has been already some efforts along this direction, i.e. extending the event calculus with the notions of processes [3,5].

Evaluation and execution of the generated plans are the final phases of automatic web service composition. These phases are left out of the scope of this paper. However, in a realistic implementation, these issues and other aspects like normative ones need to be studied. It would be interesting to formalise the rights, responsibilities, liabilities that are created by composing different web services [9].

As another further work, it is worth trying event calculus planners that employ SAT solvers for efficiency reasons.

References

1. Aydin, O., Automated web service composition with the event calculus, M.S. Thesis, Dept. of Computer Engineering, METU, Ankara, 2005.
2. Berners-Lee T., Hendler J., Lassila O. The Semantic Web. Scientific American Magazine, May 2001
3. Cicekli N. K., Cicekli I. Formalizing the specification and execution of workflows using the event calculus, to appear in Information Sciences.
4. Chen L., Yang X., Applying AI Planning to Semantic Web Services for workflow Generation, Proc. of the 1st Intl. Conf. on Semantics, Knowledge and Grid (SKG 2005).
5. Jacinto J.D., REACTIVE PASCAL and the event calculus: A platform to program reactive, rational agents. Proc. of the Workshop at FAPR'95: Reasoning about Actions and Planning in Complex Environments, 1996.
6. Kowalski R. A., Sergot M. J.A Logic-Based Calculus of Events. New Generation Computing, Vol. 4(1), pp. 67--95, 1986.
7. Kuter U., Sirin E., Nau D., Parsia B., Hendler J., Information gathering during planning for web service composition, ISWC, 2004.
8. Levesque H., Reiter R., Lesperance Y., Lin F., Scherl R. GOLOG: A Logic programming language for dynamic domains, Journal of Logic Programming, Vol. 31(1-3) pp. 59--84, April/June 1997.
9. Marjanovic O., Managing the normative context of composite e-services, ICWS-Europe, 2003, pp. 24-36.
10. McDermott D. Estimated-regression planning for interactions with Web Services. In Sixth International Conference on AI Planning and Scheduling. AAAI Press, 2002.
11. McDermott D. V., Dou D., Qi P. PDDAML, An Automatic Translator Between PDDL and DAML, http://www.cs.yale.edu/homes/dvm/daml/pddl_daml_translator1.html
12. McIlraith S. A., Son T. Adapting Golog for composition of semantic Web services. In Proceedings of Eight International Conference on Principles of Knowledge Representation and Reasoning, pp. 482--493, 2002.

13. McIlraith S. A., Son T., and Zeng H. Semantic Web services. In IEEE Intelligent Systems, March/April 2001.
14. Medjahed B., Bouguettaya A., Elmagarmid A. K. Composing web services on the semantic web. The VLDB Journal, Vol. 12(4), pp. 333--351, November 2003.
15. Oh SG., Lee D., Kumara S.R.T., A comparative Illustration of AI planning-based web services composition, ACM SIGecom Exchanges, vol. 5, Dec. 2005, pp. 1-10.
16. OWL-S: Semantic Markup for Web Services Version 1.1, November 2004. Publish of Semantics Web Services Language (SWSL) Committee. Last Accessed: 17 September 2005. http://www.daml.org/services/owl-s/1.1/overview/
17. Peer J., Web Service Composition as AI Planning- a Survey*, Technical report, Univ. of St. Gallen, Switzerland (2005), http://elektra.mcm.unisg.ch/pbwsc/docs/pfwsc.pdf.
18. Rao J., Su X., A Survey of Automated Web Service Composition Methods. In Proceedings of First International Workshop on Semantic Web Services and Web Process Composition, July 2004.
19. Rouached M., Perrin O., Godart C., Towards formal verification of web service composition, 4th Intl. Conference on Business Process Management, BPM 2006.
20. Shanahan M. P. The Event Calculus Explained. In Artificial Intelligence Today, Springer-Verlag Lecture Notes in Artificial Intelligence no. 1600, Springer-Verlag, pp. 409--430, 1999.
21. Shanahan M.P. An abductive event calculus planner. Journal of Logic Programming, Vol. 44(1-3), pp. 207--240, July 2000.
22. Sirin E., Hendler J., Parsia B. Semi-automatic Composition of Web Services using Semantic Descriptions. Web Services: Modeling, Architecture and Infrastructure workshop in conjunction with ICEIS2003, 2002.
23. Stathis K., Lekeas G., Kloukinas C., Competence checking for the global e-service society using games, In Proceedings of Engineering Societies in the Agents World (ESAW06), G. O'Hare, M. O'Grady, O. Dikinelli, and A Ricci (Eds).
24. Wu D., Sirin E., Parsia B, Hendler J., Nau D. Automatic web services composition using SHOP2. In Proceedings of Planning for Web Services Workshop, in ICAPS 2003, June 2003.
25. Yolum P., Singh M., Reasoning About Commitments in the Event Calculus: An Approach for Specifying and Executing Protocols, Annals of Mathematics and AI, Vol:42(1-3), 2004.