SERVICE COMPOSITION AND EXECUTION PLAN GENERATION OF COMPOSITE SEMANTIC WEB SERVICES USING ABDUCTIVE EVENT CALCULUS

D.PAULRAJ[1], S.SWAMYNATHAN[2], DANIEL CHANDRAN[3], K.BALASUBADRA[1], M.VIGILSON PREM[1]

[1] *Department of Information Technology, R.M.D Engineering College, Chennai, India.*

[2] *College of Engineering Guindy, Anna University, Chennai, India.*

[3] *Faculty of Engineering and Information Technology, University of Technology Sydney.*

Abstract

Web Service composition is indispensable as a single Web Service cannot satisfy the complex functional requirement of a user. The two key challenges of Semantic Web Service composition are the discovery of most relevant atomic services from the Composite Semantic Web Services and by no means we can assure the execution of the composed atomic services in a proper order. In this work, these two challenges are addressed and also a novel architecture is proposed for atomic service discovery, composition and automatic plan generation for the proper execution of its candidate services. The proposed architecture takes the advantage of abductive event calculus that uses abductive theorem prover to generate a plan for the proper order of execution of the atomic services. The research has found that the plan generated by the proposed architecture is sound and complete.

Keywords: Ontology, Semantic Web Services, Composition, Abductive Event Calculus.

1. INTRODUCTION

In Service Oriented Architecture (SOA), service composition is used to integrate several services together to meet complex business needs. Web Services are loosely coupled platform-neutral and language-neutral entities that are described by WSDL (Web Service Description Language). Generally Web Services do not have homogeneous structure and hence heterogeneity arises from different ways to name parameters and describe their processing. The lack of any machine interpretable semantics requires human intervention for service discovery and composition. This drawback prevents the use of Web Services on complex business contexts, where the automation of these business processes is necessary. Semantic Web Services (SWS) had been proposed to overcome the issues such as interface heterogeneity and keyword-based syntactic search. Various semantic descriptions of Web Services have already been proposed such as Business Process Execution Language for Web Services (BPEL4WS) (Andrews et al., 2003), Web Service Modeling Language (WSML) (Bruijn et al., 2005), Web Service Modeling Ontology (WSMO) (Roman et al., 2005), Web Service Description Language Semantic (WSDL-S) (Akkiraju et al.,2005) and Semantic Annotations for Web Service Description Language (SAWSDL) (Kopecky et al., 2007). In this direction, OWL-S coalition has promoted Ontology Web Languages for Services (OWL-S) (Martin et al., 2004), which enrich WSDL and BPEL4WS with rich semantic annotations to facilitate flexible dynamic Web Services discovery, invocation and composition. The OWL-S coalition proposed the semantic description of Web Services based on its four *ontologies*: *Service, Service profile, Process model (Service Model)* and *Service grounding*. The *Service* ontology serves as an upper ontology of the other three containing references to them. OWL-S defines three types of processes: *atomic*, *simple* and *composite*. Composite Semantic Web Services (CSWS) can be specified by using OWL-S control constructs such as *sequence*, *split*, *split+join*, *choice*, *any-order*, *if-then-else*, *repeat-while*, *repeat-until* and *iterate*. It is to be noted that in OWL-S, the service profile ontology is meant to be mainly used only for the purpose of service discovery and once the service has been discovered the clients will use the process model ontology for the service composition (Martin et al., 2004) but not the service profile ontology. The inequalities and the comparative study about the characteristics and the importance of the process model ontology and service profile ontology are shown in Table 1. The information contained in the service profile ontology of OWL-S would be sufficient for the discovery of atomic services, but it would not be sufficient for the discovery of CSWS (Brogi, Corfini and Popescu, 2008). In CSWS, each atomic service can have arbitrary number of Input, Output, Precondition and Effects (IOPEs). Indeed, the service profile ontology and process model ontology are two different representations of the same service. Therefore, naturally the IOPEs of both service profile ontology and process model ontology should coincide. However, OWL-S framework does not explicitly dictate any such constraint. In case of inconsistency between the service profile ontology and the process model ontology, the interaction with the service will break at some point. Hence, it is essential that the definition of the IOPEs in the service profile ontology should be made carefully, to ensure consistency between service profile ontology and process model ontology. This would help to avoid inconsistency from one side and at the same time the service profile is not overwhelmed or short of IOPEs.

Table 1. Comparative study of the process model and service profile ontologies of OWL-S

No.	Process Model	Service Profile
1.	Describes how the service works.	Describes what the service does.
2.	Is a functional module of the service	Non-functional list of parameters of the service (Advertisement)
3.	IOPEs are the integral part of the process	IOPEs are constructed from the service
4.	Schema Exists in the Process Ontology to describe IOPEs.	Does not provide any schema to describe the IOPEs
5.	Inputs and Outputs are annotated with ontology concepts.	Usually Inputs and outputs are not annotated with ontology concepts.
6.	Provide enough information for service discovery, invocation and composition	Provides information for service discovery only.
7.	Description is Concrete	Description is Concise.
8.	Describe the internal behavior of the service.	Does not describe the internal behavior of the service.
9.	Upper Ontology for specifies the cardinality construct, that is, a service can be *describedBy* at most one service model.	Upper Ontology deliberately does not specify any minimum or maximum cardinality for the properties *describedBy*.

The paper is organized as follows. Related works, limitations of the existing works and the motivation for this research are described in section 2 followed by Conversion of OWL-S control constructs into Event Calculus formulas in section 3.Web Service discovery using process model ontology with experimental results is explained in section 4. Service composition and execution plan generation is elaborated with examples in section 5.The paper is concluded in section 6.

2. RELATED WORKS

There is a considerable amount of research has been carried out in the field of Web Service composition. Some of the most significant contributions related to this work are cited in this section.

Automatically aggregating composite semantic Web Services is a challenging and critical task. In this direction, one of the techniques that has been proposed for this task is the AI planning. In AI planning method, generally the users specify the necessary inputs and required outputs. A plan is generated automatically by an AI planner (Petrie, 2009; Hoffmann et al., 2009; Seog-Chan, Lee, and Kumara, 2005; Seog-Chan, Lee, and Kumara, 2007). Salomie et al.(2008), have used the fluent calculus for AI planning for the composition of Web Services. This approach is a domain specific scenario that uses the domain ontologies and fluent calculus formalisms together for the planning and composition of Web Services. Like AI planning, the

EC is also a very suitable formalism for the problem of Web Service composition. The use of Event Calculus (EC) has been proposed by Aydin et al., 2007, as a solution for the Web Service composition problem. It is shown that when a goal situation is given, the EC can find suitable plans as Web Service composition by using an abductive planning technique. If more than one plan is generated, the execution engine selects the best plan by compiling it in to a graph. A monolithic approach for Web Service composition and execution is proposed by Okutan and Cicekli (2010). In this work also several plans are generated by the planner from which the best solution is selected according to some ranking. Ozorhan, Kuban and Cicekli (2010), have used EC for the solution of automated Web Service composition problem using the interleaved and the template-based approaches. It is shown that, when the initial state and a goal state are given as user specified ontological inputs and outputs, the abductive event calculus planner generates a specific plan.

In order to ensure the correct and reliable composite service execution, an event driven approach has been proposed (Gaaloul et al., 2007). This approach analyzes and checks the Web Service's transactional behavior consistency at design time and report recovery mechanisms deviations after runtime. Chifu et al. (2008) have described how the planning capabilities of the fluent calculus can be used to automatically generate the Web Service composition. The concept between OWL-S process ontology and the fluent calculus are identified and an algorithm is used to translate OWL-S service descriptions into an equivalent fluent calculus service specification. Lecue, Leger and Delteil (2008) have explained how a conditional composition of Web Services can be achieved effectively by integrating causal links through DL reasoning, and laws through situation calculus. A formal approach called SpiG4WSC calculus to carry out to implement secure orchestration and choreography has been proposed by Xu, Qi, Hou and Wan (2008). Spi calculus has been used to identify secure properties for the orchestration. Yolum and Munindar (2004), have proposed an approach for formally representing and reasoning about the commitments in the event calculus. They have formalized commitment operations and domain independent reasoning rules as axioms to capture the evaluation of commitments. An event based approach for modeling and testing of functional behavior of Web Services in SOA by event sequence graph (ESG) has been proposed by Bansal and Vidal (2003).

An OWL-S service profile ontology based framework is used, for the retrieval of Web Services based on subsumption relation and structural case-based reasoning which performs domain-dependant discovery (Meditskos and Bassiliades, 2010). Brogi (2010) have presented an algorithm called Service Aggregation Matchmaking (SAM), for composition-oriented discovery of Web Services. Hypergraph has been used to capture effectively the dependencies among processes on the matched services. It performs a fine-grained matching at the level of atomic process of services. SAM might be the first algorithm that used process model to discover the atomic process in the composite semantic Web Services. The algorithm used in the discovery phase of this paper partially inherits the algorithm used by Brogi (2010). A context-based approach to the problem of matching and ranking semantic Web Services for composition has been proposed by Segev and Toch (2009). BPEL based semantics for a new specification language based on the Π-calculus is used for interacting Web Services (Abouzaid and Mullins, 2008). Rouached and Godart (2006), have addressed the authorization issue within a Web Service composition. For the formal verification of Web Services conformance to choreographies in Abductive Logic Programming, a framework called ALoWS has been proposed (Alberti, 2006). The framework has been proposed for managing authorization policies for Web Service composition. A language has been developed using Event Calculus and abductive reasoning to support specification and analysis of authorization policies for Web Service composition.

2.1 Limitations

Some of the limitations found in the AI based planning approaches are its lack of scalability, extendibility, concurrency and its tendency to generate more plans. Most of the planning approaches hamper the scalability to large scale problems. In Web Service composition, the problem domain tends to be very large. The ability to support an extended goal is imperative in automatic Web Service composition, because requirements (the desire goal) set by the users differ from time to time. The nature of generating more similar or dissimilar plans (Aydin, Kesim Cicekli and Cicekli, 2007; Okutan and Cicekli, 2010) eventually leads to the manual intervention for the selection of the best plan. This is one of the major drawbacks calling for an attention. Almost all of the existing works in the area of composition of the semantic Web Services have concentrated only on atomic services. However, the problems of composing atomic services from the CSWS have received a lot of attention. EC is used to formally specify and check the transactional behavior and consistency of Web Service composition.

EC is a logical mechanism that infers “what’s true when”, given “what happens when” and “what actions do”. The “*what happens when*” part is a *narrative* of events, and the “*what actions do*” part describes the *effects* of actions. In an abductive task, “*what actions do*” and “*what’s true when*” are supplied, and “*what happens when*” is expected (Shanahan, 1999). Traditionally, abductive means inference to a best explanation and is a pattern of reasoning. It proceeds from an outcome to a hypothesis that explains for the outcome. For instance, O ← E produces E as an explanation for the outcome O**.** Abduction is logically inverse of deduction and is used over the event calculus axioms to obtain partially ordered sets of events. Abduction is handled by a second order abductive theorem prover. Shanahan (Shanahan, 2000) explained that, when *EC* formulae are submitted to a suitably tailored resolution based abductive theorem prover, the result is purely logical planning system whose computations mirror closely those of a hand-coded planning algorithm.

2.2 Motivation

The above limitations of the profile based service discovery methods have motivated the authors to propose a complementary service discovery method, which uses the process model ontology of the OWL-S. Regarding service execution, the plan is generated by using second order abductive theorem prover of abductive event calculus. The candidate services in the composition are translated into the axioms of the event calculus and the abductive theorem prover can be used to generate the actual execution plan, which can be executed later on. Literature survey brings out the fact that Abductive Event Calculus is a suitable formalism for the generation of the plan.

The objective of this paper is to prove that the process model ontology based atomic service discovery is more purposeful and better than the service profile ontology based discovery in the CSWS; and the use of abductive event calculus is a better choice to generate a sound and complete plan for better execution of the services in a specific order.

3. TRANSLATION OF OWL-S CONTROL CONSTRUCTS INTO EC FORMULAS.

The Event Calculus (*EC*) was introduced by Kowalski and Sergot (Kowalski and Sergot, 1986) as a logic programming formalism for representing events and their effects. In *EC* it is possible to infer what is true when, given a set of events at certain time points and their effects. The ontology of the *EC* comprises of *fluents*(F), *events*(*E*) (or actions), and *timepoints*(*T*). Fluents are properties that may have different values at different time points and the events manipulate fluents. Fluents are initiated and terminated by events, and that a fluent may be true at the beginning of time. For a given event narrative (a set of events), the *EC* theorem and domain-specific axioms together define which fluents hold at each time. The event calculus used in this paper is a subset of Shanahan's circumscriptive event calculus (Shanahan, 1995). It is based on many-sorted first-order predicate calculus with predicates to reason about events. The predicates are:

Initiates(E,F,T) expresses that fluent *F* holds after timepoint *T* if event *E* happens at T.

Terminates(E,F,T) expresses that fluent *F* does not hold after timepoint *T* if event *E* happens at T.

Releases(E,F,T) expresses that fluent F is not subject to the common sense law of inertia after event E at time T.

InitiallyTrue(F) define if *F* holds at timepoint 0.

InitiallyFalse(F) define if *F* not holds at timepoint 0.

Happens(E,T) is true iff event *E* happens at *T.*

HoldsAt(F,T) is true iff fluent F holds at T.

$Clipped(T_1,F,T_2)$ expresses if fluent *F* was terminated during time interval $[T_1,T_2]$.

$Declipped(T_1,F,T_2)$ expresses if fluent *F* was initiated during time interval $[T_1,T_2]$.

These predicates can be used to translate the OWL-S control constructs into equivalent EC formalisms (Paulraj and Swamynathan, 2010).

Therefore, all OWL-S control constructs must be translated into equivalent EC formulas in order to formally specify services composition and are explained below.

3.1. Sequence.

In a sequence control construct, a list of services S_1, S_2, … ,S_n has to be executed in a sequential order. That is, Web Service S_{i+1} follows Web Service S_i sequentially. This control construct can be translated into the Event Calculus (EC) rules as follows:

$$\begin{aligned}\mathit{Happens}(S,T) \leftarrow\ & \mathit{HoldsAt}(P,T) \wedge \\ & \mathit{Happens}(S_1,T_1) \wedge \\ & \mathit{Happens}(S_2,T_2) \\ & \wedge . \ . \ . \wedge \\ & \mathit{Happens}(S_n,T_n) \wedge \\ & T < T_1 < . \ . \ . < T_n\end{aligned}$$

where, P is the precondition of the Web Service S and T, T_1, … , T_n are the time points.

3.2. Split

In split, a list of services S_1, S_2, … ,S_n has to be executed concurrently. That is, the split completes as soon as all its services are scheduled for execution. In EC split control construct can be represented by a sequence of rules as follows:

$$\begin{aligned}\mathit{Happens}(S,T) \leftarrow\ & \mathit{HoldsAt}(P,T) \wedge \\ & \mathit{Happens}(S_1,T_1) \wedge \\ & \mathit{Happens}(S_2,T_1) \wedge . \ . \ . \wedge \\ & \mathit{Happens}(S_n,T_1) \wedge \\ & T < T_1\end{aligned}$$

3.3. Split + join

A list of services S_1, S_2, … ,S_n has to be executed concurrently with barrier synchronization. That is, *split+join* completes when all of its constituent services have been completed. The following is the EC rule equivalent to the *split+join* control construct.

$$\begin{aligned}\mathit{Happens}(S,T) \leftarrow\ & \mathit{HoldsAt}(P,T) \wedge \\ & \mathit{Happens}(S_1,T_1) \wedge \\ & \mathit{Happens}(S_2,T_2) \wedge . \ . \ . \wedge \\ & \mathit{Happens}(S_n,T_n) \\ & T = \mathrm{Maximum}(T_1,T_2,. \ . \ . \ , T_n)\end{aligned}$$

3.4. Choice

In a list of services, S_1,S_2,…,S_n the choice control construct executes a service if any one of the conditions {*cond₁, cond₂, ..., condₙ*}is true. The choice control construct can be represented in EC as follows:

$$\begin{aligned}\mathrm{Happens}(S,T) \leftarrow\ & \mathrm{HoldsAt}(P,T) \wedge \\ & \mathrm{HoldsAt}(\mathrm{cond}_1,T_1) \wedge [\neg\mathrm{HoldsAt}(\mathrm{cond}_2,T_1) \vee \\ & \neg\mathrm{HoldsAt}(\mathrm{cond}_3,T_1) \vee . \ . \ . \vee \neg\mathrm{HoldsAt}(\mathrm{cond}_n,T_1)] \wedge \\ & \mathrm{Happens}(S_1,T_2) \wedge \\ & T < T_1 < T_2\end{aligned}$$

$$\begin{aligned}\mathrm{Happens}(S,T) \leftarrow\ & \mathrm{HoldsAt}(P,T) \wedge \mathrm{HoldsAt}(\mathrm{cond}_2,T_1) \wedge [\neg\mathrm{HoldsAt}(\mathrm{cond}_1,T_1) \vee \\ & \neg\mathrm{HoldsAt}(\mathrm{cond}_3,T_1) \vee . \ . \ . \vee \neg\mathrm{HoldsAt}(\mathrm{cond}_n,T_1)] \wedge\end{aligned}$$

$$\begin{aligned} & \mathtt{Happens}(S_2,T_2)\ \wedge \\ & T < T_1 < T_2 \\ & \quad . \\ & \quad . \\ & \quad . \\ \mathtt{Happens}(S,T) \leftarrow\ & \mathtt{HoldsAt}(P,T)\ \wedge\ \mathtt{HoldsAt}(cond_n,T_1)\ \wedge\ [\neg\mathtt{HoldsAt}(cond_1,T_1) \vee \\ & \neg\mathtt{HoldsAt}(cond_2,T_1)\ \vee . \ . \ . \vee \neg\mathtt{HoldsAt}(cond_{n-1},T_1)]\ \wedge \\ & \mathtt{Happens}(S_n,T_2)\ \wedge \\ & T < T_1 < T_2 \end{aligned}$$

3.5. Any-Order

A list of services S_1, S_2,…,S_n has to be executed in some unspecified order but not concurrently, and all services would be executed. *Any-order* is expanded into a combination of *choice* and *sequence* control constructs among all of its child nodes. *Any-order* control construct for two services with the permutation of *choice* and *sequence* control constructs is shown in Figure 1. The two services S_1, S_2 are executed either in the order of S_1 followed by S_2 or S_2 followed by S_1, based on any one of the two conditions *cond₁, cond₂* present in the *choice* control construct is true.

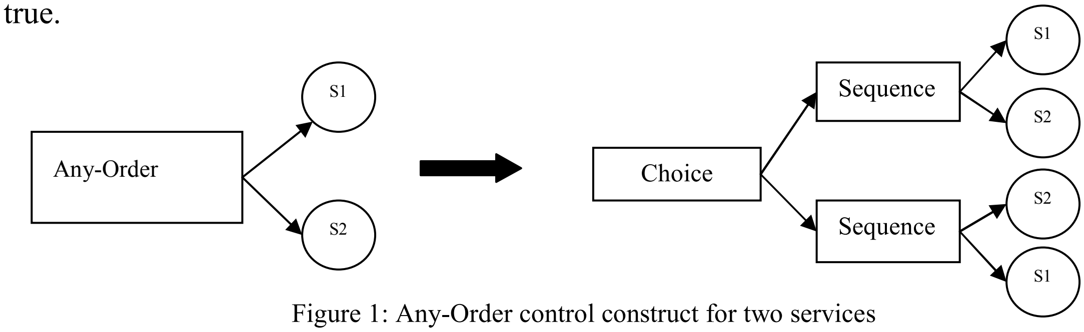


Figure 1: Any-Order control construct for two services

$$\begin{aligned} \mathtt{Happens}(S,T) \leftarrow\quad & \mathtt{HoldsAt}(P,T)\ \wedge \\ & \mathtt{HoldsAt}(cond_1,T_1)\ \wedge \\ & [\neg\mathtt{HoldsAt}(cond_2,T_1)]\ \wedge \\ & \mathtt{Happens}(S_1,T_2)\ \wedge \\ & \mathtt{Happens}(S_2,T_3)\ \wedge \\ & \neg[\mathtt{Happens}(S_2,T_2) \wedge \mathtt{Happens}(S_1,T_3)]\ \wedge \\ & T < T_1 < T_2 < T_3 \end{aligned}$$

$$\begin{aligned} \mathtt{Happens}(S,T) \leftarrow\quad & \mathtt{HoldsAt}(P,T)\ \wedge \\ & \mathtt{HoldsAt}(cond_2,T_1)\ \wedge \\ & [\neg\mathtt{HoldsAt}(cond_1,T_1)\ ]\ \wedge \\ & \mathtt{Happens}(S_2,T_2)\ \wedge \\ & \mathtt{Happens}(S_1,T_3)\ \wedge \\ & \neg[\mathtt{Happens}(S_1,T_2) \wedge \mathtt{Happens}(S_2,T_3)]\ \wedge \\ & T < T_1 < T_2 < T_3 \end{aligned}$$

To further explain the functioning of *any-order* control construct intricately, list of three services with the combination of *choice* and *sequence* control constructs is shown in Figure 2. Three services S_1, S_2 and S_3 are executed either in the order of $\{S_1,S_2,S_3\}$ or $\{S_1,S_3,S_2\}$ or $\{S_2,S_1,S_3\}$ or $\{S_2,S_3,S_1\}$ or $\{S_3,S_1,S_2\}$ or $\{S_3,S_2,S_1\}$ based on any one of the six conditions $\{cond_1, cond_2, cond_3, cond_4, cond_5, cond_6\}$ present in the choice control construct is true.

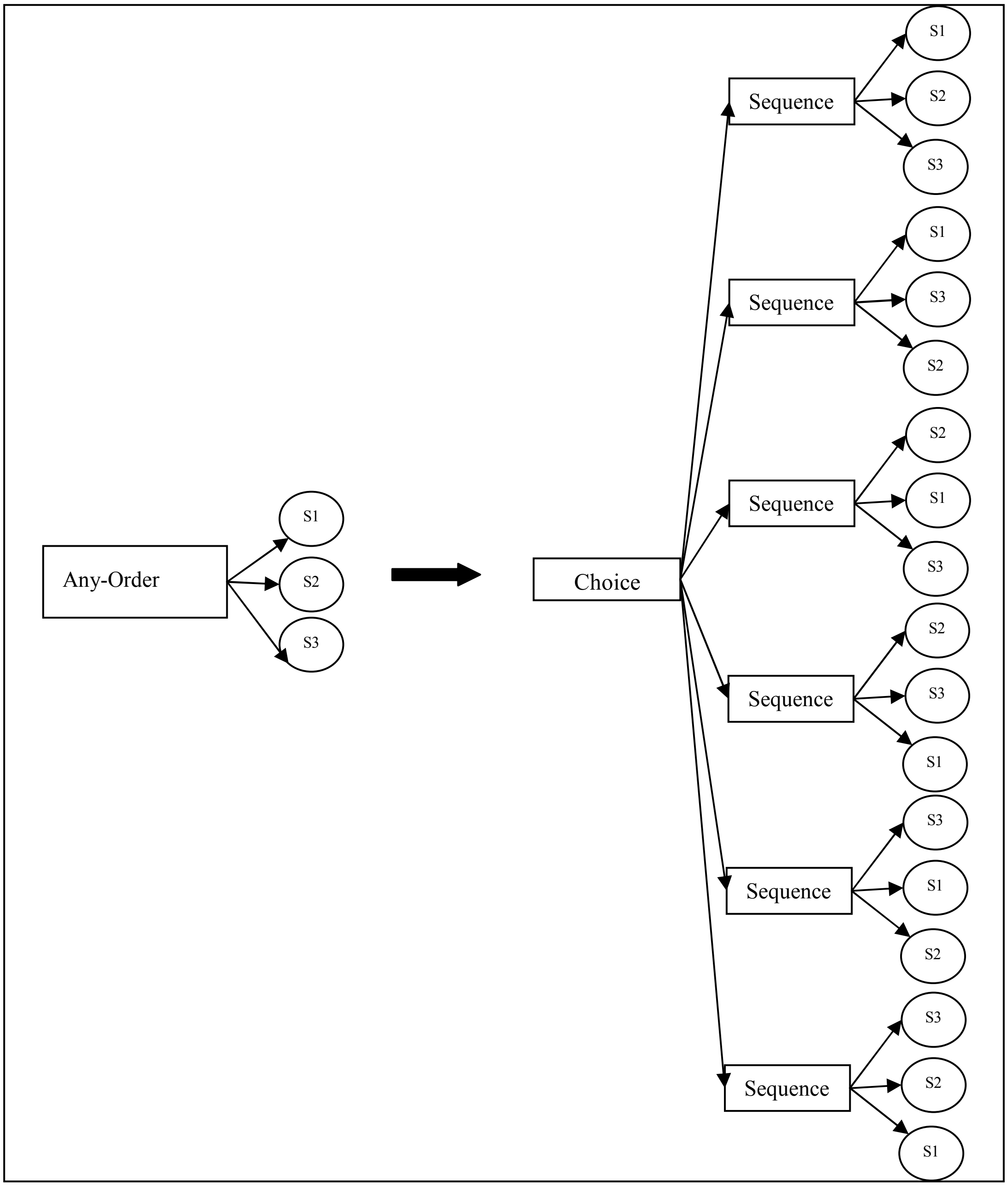


Figure 2: Any-Order control construct for three services

The EC rules for any-order control construct for three services are expressed below.

$$
\begin{aligned}
\mathrm{Happens}(S,T) \leftarrow\quad & \mathrm{HoldsAt}(P,T) \wedge \mathrm{HoldsAt}(cond_1,T_1) \wedge [\neg \mathrm{HoldsAt}(cond_2,T_1) \vee \\
& \neg \mathrm{HoldsAt}(cond_3,T_1) \vee \neg \mathrm{HoldsAt}(cond_4,T_1) \vee \neg \mathrm{HoldsAt}(cond_5,T_1) \vee \\
& \neg \mathrm{HoldsAt}(cond_6,T_1)] \wedge \\
& [\mathrm{Happens}\ (S_1,T_2) \wedge \mathrm{Happens}(S_2,T_3) \wedge \mathrm{Happens}(S_3,T_4)\ ] \wedge \\
& \neg[\mathrm{Happens}(S_1,T_3) \wedge \mathrm{Happens}(S_2,T_4) \wedge \mathrm{Happens}(S_3,T_2) \wedge \\
& \mathrm{Happens}(S_1,T_4) \wedge \mathrm{Happens}(S_2,T_2) \wedge \mathrm{Happens}(S_3,T_3)] \wedge \\
& T < T_1 < T_2 < T_3
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Happens}(S,T) \leftarrow\quad & \mathrm{HoldsAt}(P,T) \wedge \\
& \mathrm{HoldsAt}(cond_2,T_1) \wedge [\neg \mathrm{HoldsAt}(cond_1,T_1) \vee \neg \mathrm{HoldsAt}(cond_3,T_1) \vee \\
& \neg \mathrm{HoldsAt}(cond_4,T_1) \vee \neg \mathrm{HoldsAt}(cond_5,T_1) \vee \neg \mathrm{HoldsAt}(cond_6,T_1)] \wedge \\
& [\mathrm{Happens}\ (S_1,T_2) \wedge \mathrm{Happens}(S_3,T_3) \wedge \mathrm{Happens}(S_2,T_4)\ ] \wedge \\
& \neg[\mathrm{Happens}(S_1,T_3) \wedge \mathrm{Happens}(S_3,T_4) \wedge \mathrm{Happens}(S_2,T_2) \wedge \\
& \mathrm{Happens}(S_1,T_4) \wedge \mathrm{Happens}(S_3,T_2) \wedge \mathrm{Happens}(S_2,T_3)] \wedge \\
& T < T_1 < T_2 < T_3 < T_4
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Happens}(S,T) \leftarrow\quad & \mathrm{HoldsAt}(P,T) \wedge \\
& \mathrm{HoldsAt}(cond_3,T_1) \wedge [\neg \mathrm{HoldsAt}(cond_1,T_1) \vee \neg \mathrm{HoldsAt}(cond_2,T_1) \vee \\
& \neg \mathrm{HoldsAt}(cond_4,T_1) \vee \neg \mathrm{HoldsAt}(cond_5,T_1) \vee \neg \mathrm{HoldsAt}(cond_6,T_1)] \wedge \\
& [\mathrm{Happens}\ (S_2,T_2) \wedge \mathrm{Happens}(S_1,T_3) \wedge \mathrm{Happens}(S_3,T_4)\ ] \wedge \\
& \neg[\mathrm{Happens}(S_2,T_3) \wedge \mathrm{Happens}(S_1,T_4) \wedge \mathrm{Happens}(S_3,T_3) \wedge \\
& \mathrm{Happens}(S_2,T_4) \wedge \mathrm{Happens}(S_1,T_2) \wedge \mathrm{Happens}(S_3,T_2)] \wedge \\
& T < T_1 < T_2 < T_3 < T_4
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Happens}(S,T) \leftarrow\quad & \mathrm{HoldsAt}(P,T) \wedge \\
& \mathrm{HoldsAt}(cond_4,T_1) \wedge [\neg \mathrm{HoldsAt}(cond_1,T_1) \vee \neg \mathrm{HoldsAt}(cond_2,T_1) \vee \\
& \neg \mathrm{HoldsAt}(cond_3,T_1) \vee \neg \mathrm{HoldsAt}(cond_5,T_1) \vee \neg \mathrm{HoldsAt}(cond_6,T_1)] \wedge \\
& [\mathrm{Happens}\ (S_2,T_2) \wedge \mathrm{Happens}(S_3,T_3) \wedge \mathrm{Happens}(S_1,T_4)\ ] \wedge \\
& \neg[\mathrm{Happens}(S_2,T_3) \wedge \mathrm{Happens}(S_3,T_4) \wedge \mathrm{Happens}(S_1,T_2) \wedge \\
& \mathrm{Happens}(S_2,T_4) \wedge \mathrm{Happens}(S_3,T_2) \wedge \mathrm{Happens}(S_1,T_3)] \wedge \\
& T < T_1 < T_2 < T_3 < T_4
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Happens}(S,T) \leftarrow\quad & \mathrm{HoldsAt}(P,T) \wedge \\
& \mathrm{HoldsAt}(cond_5,T_1) \wedge [\neg \mathrm{HoldsAt}(cond_1,T_1) \vee \neg \mathrm{HoldsAt}(cond_2,T_1) \vee \\
& \neg \mathrm{HoldsAt}(cond_3,T_1) \vee \neg \mathrm{HoldsAt}(cond_4,T_1) \vee \neg \mathrm{HoldsAt}(cond_6,T_1)] \wedge \\
& [\mathrm{Happens}\ (S_3,T_2) \wedge \mathrm{Happens}(S_1,T_3) \wedge \mathrm{Happens}(S_2,T_4)\ ] \wedge \\
& \neg[\mathrm{Happens}(S_3,T_3) \wedge \mathrm{Happens}(S_1,T_4) \wedge \mathrm{Happens}(S_2,T_2) \wedge \\
& \mathrm{Happens}(S_3,T_4) \wedge \mathrm{Happens}(S_1,T_2) \wedge \mathrm{Happens}(S_2,T_3)] \wedge \\
& T < T_1 < T_2 < T_3 < T_4
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Happens}(S,T) \leftarrow\quad & \mathrm{HoldsAt}(P,T) \wedge \\
& \mathrm{HoldsAt}(cond_6,T_1) \wedge [\neg \mathrm{HoldsAt}(cond_1,T_1) \vee \neg \mathrm{HoldsAt}(cond_2,T_1) \vee \\
& \neg \mathrm{HoldsAt}(cond_3,T_1) \vee \neg \mathrm{HoldsAt}(cond_4,T_1) \vee \neg \mathrm{HoldsAt}(cond_5,T_1)] \wedge \\
& [\mathrm{Happens}\ (S_3,T_2) \wedge \mathrm{Happens}(S_2,T_3) \wedge \mathrm{Happens}(S_1,T_4)\ ] \wedge \\
& \neg[\mathrm{Happens}(S_3,T_3) \wedge \mathrm{Happens}(S_2,T_4) \wedge \mathrm{Happens}(S_1,T_2) \wedge \\
& \mathrm{Happens}(S_3,T_4) \wedge \mathrm{Happens}(S_2,T_2) \wedge \mathrm{Happens}(S_1,T_3)] \wedge \\
& T < T_1 < T_2 < T_3 < T_4
\end{aligned}
$$

Thus, the EC rules for the *any-order* control construct for N services can be constituted with the combination of choice and sequence control constructs, and are expressed below. N services $\{S_1, S_2 \ldots, S_n\}$ are executed either in the order of $\{S_1, S_2, \ldots, S_n\}$ or $\{S_1, S_3, S_2, \ldots S_n\}$ or … or $\{S_n, S_{n-1}, \ldots, S_1\}$ based on any one of the N conditions $\{cond_1, cond_2, \ldots cond_n\}$ present in the *choice* control construct is true.

$$
\begin{aligned}
\mathrm{Happens}(S,T) \leftarrow\ & \mathrm{HoldsAt}(P,T) \wedge \mathrm{HoldsAt}(cond_1,T_1) \wedge \\
& [\neg\mathrm{HoldsAt}(cond_2,T_1) \vee \neg\mathrm{HoldsAt}(cond_3,T_1) \vee \ldots \vee \\
& \neg\mathrm{HoldsAt}(cond_n,T_1)] \wedge \\
& [\mathrm{Happens}(S_1,T_2) \wedge \mathrm{Happens}(S_2,T_3) \wedge \ldots \wedge \mathrm{Happens}(S_n,T_{n+1})] \wedge \\
& \neg[\mathrm{Happens}(S_1,T_3) \wedge \mathrm{Happens}(S_1,T_4) \wedge \ldots \wedge \mathrm{Happens}(S_1,T_{n+1}) \\
& \wedge \mathrm{Happens}(S_2,T_2) \wedge \mathrm{Happens}(S_2,T_4) \wedge \ldots \wedge \mathrm{Happens}(S_2,T_{n+1}) \\
& \wedge \ldots \wedge \mathrm{Happens}(S_n,T_2) \wedge \mathrm{Happens}(S_1,T_3) \wedge \ldots \wedge \mathrm{Happens}(S_1,T_n)] \wedge \\
& T < T_1 < T_2 < \ldots T_n < T_{n+1}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Happens}(S,T) \leftarrow\ & \mathrm{HoldsAt}(P,T) \wedge \mathrm{HoldsAt}(cond_2,T_1) \wedge \\
& [\neg\mathrm{HoldsAt}(cond_1,T_1) \vee \neg\mathrm{HoldsAt}(cond_3,T_1) \vee \ldots \vee \\
& \neg\mathrm{HoldsAt}(cond_n,T_1)] \wedge \\
& [\mathrm{Happens}(S_2,T_2) \wedge \mathrm{Happens}(S_1,T_3) \wedge \ldots \wedge \mathrm{Happens}(S_n,T_{n+1})] \wedge \\
& \neg[\mathrm{Happens}(S_1,T_2) \wedge \mathrm{Happens}(S_1,T_4) \wedge \ldots \wedge \mathrm{Happens}(S_1,T_{n+1}) \\
& \wedge \mathrm{Happens}(S_2,T_3) \wedge \mathrm{Happens}(S_2,T_4) \wedge \ldots \wedge \mathrm{Happens}(S_2,T_{n+1}) \\
& \wedge \ldots \wedge \mathrm{Happens}(S_n,T_2) \wedge \mathrm{Happens}(S_1,T_3) \wedge \ldots \wedge \mathrm{Happens}(S_1,T_n)] \wedge \\
& T < T_1 < T_2 < \ldots T_n < T_{n+1}
\end{aligned}
$$

.
.
.

$$
\begin{aligned}
\mathrm{Happens}(S,T) \leftarrow\ & \mathrm{HoldsAt}(P,T) \wedge \mathrm{HoldsAt}(cond_n,T_1) \wedge \\
& [\neg\mathrm{HoldsAt}(cond_1,T_1) \vee \neg\mathrm{HoldsAt}(cond_2,T_1) \vee \ldots \vee \\
& \neg\mathrm{HoldsAt}(cond_{n-1},T_1)] \wedge [\mathrm{Happens}(S_n,T_2) \wedge \\
& \mathrm{Happens}(S_{n-1},T_3) \wedge \ldots \wedge \mathrm{Happens}(S_1,T_n)] \wedge \\
& \neg[\mathrm{Happens}(S_n,T_3) \wedge \mathrm{Happens}(S_n,T_4) \wedge \ldots \wedge \mathrm{Happens}(S_n,T_n) \\
& \wedge \mathrm{Happens}(S_{n-1},T_2) \wedge \mathrm{Happens}(S_{n-1},T_4) \wedge \ldots \wedge \mathrm{Happens}(S_{n-1},T_n) \wedge \ldots \wedge \\
& \mathrm{Happens}(S_1,T_2) \wedge \mathrm{Happens}(S_1,T_3) \wedge \ldots \wedge \mathrm{Happens}(S_1,T_{n-1})] \wedge \\
& T < T_1 < T_2 < \ldots T_n < T_{n+1}
\end{aligned}
$$

3.6. If-Then-Else

One service out of two services is chosen for execution if the condition "*cond*" is true. The EC equivalent rule for this control construct is expressed below.

$$
\begin{aligned}
\mathrm{Happens}(S,T) \leftarrow\ & \mathrm{HoldsAt}(P,T) \wedge \\
& \mathrm{HoldsAt}(cond,T_1) \wedge \\
& \mathrm{Happens}(S_1,T_2) \wedge \neg\mathrm{Happens}(S_2,T_2) \wedge \\
& T < T_1 < T_2 \\
\mathrm{Happens}(S,T) \leftarrow\ & \mathrm{HoldsAt}(P,T) \wedge \\
& \neg\mathrm{HoldsAt}(cond,T_1) \wedge \\
& \mathrm{Happens}(S_2,T_2) \wedge \neg\mathrm{Happens}(S_1,T_1) \wedge \\
& T < T_1 < T_2
\end{aligned}
$$

Thus, these EC rules can be efficaciously used to represent the CSWS and by using these rules a plan can be generated for a possible execution of the composed service to meet the user's requirement.

4. SERVICE DISCOVERY USING PROCESS MODEL ONTOLOGY.

Paulraj et al, (2012) have experimentally proved that the process model ontology based service discovery is better than the service profile based one. The data for the experiments were taken from the service retrieval test collection OWLS-TC (version 4.0), which contains 1083 semantic Web Services which are semantically annotated by using 48 different ontologies (Klusch *et al*. 2009). Two discovery algorithms, one using process model ontology and another using service profile ontology has been employed for service discovery of 42 different test queries. Number of matched services returned by process model and service profile ontology algorithms varies considerably and is shown in Table 2.

Table 2. Number of matched services returned by process model and service profile ontology algorithms.

Test Data Set	**Query**		**No. of Matched Services**	
	Input	**Output**	**Process Model**	**Service Profile**
1	BOOK	PRICE	20	15
2	CAR, 1PERSONBICYCLE	PRICE	21	12
3	PERSON,BOOK, CREDITCARDACCOUNT	BOOK	86	94
4	PERSON, BOOK, CREDITCARDACCOUNT	PRICE	23	18
5	CAR	PRICE	18	9
6	CITY, COUNTRY	HOTEL	6	6
7	COUNTRY	SKILLEDOCCUPATION	10	8
8	MP3PLAYER, DVDPLAYER	PRICE	16	11
9	EBOOKREQUEST, USERACCOUNT	EBOOK	6	6
10	MESSEMODUL	SLIDER, CUP, ULTRASOUNDSENSOR, PILL, SLIDERZONE, CUPZONE	1	1
11	USERID, LATITUDE, LONGITUDE	ALTITUDE	3	2
12	COUNTRY1, COUNTRY1, STATE1, STATE2, CITY1, CITY2	DISTANCE	1	1
13	CITY, STATE	ZIPCODE, REACODE, LATITUDE, ONGITUDE	1	1
14	CITY,STATE	LATITUDE, ONGITUDE	2	2
15	ZIPCODE	LATITUDE, ONGITUDE	1	1
16	ADDRESS, CITY, STATE, ZIPCODE	MAP	1	1
17	LICENSEKEY, LATITUDE, LONGITUDE, DATE	SUNRISE, SUNSET	1	1
18	LICENSEKEY, CITY, STATE	ZIPCODE	3	2
19	ADDRESS, CITY, STATE, ZIPCODE	LATITUDE, LONGITUDE	6	3
20	GEOGRAPHICAL-REGION1, GEOGRAPHICAL-	MAP	7	6

Test Data Set	Query		No. of Matched Services	
	Input	Output	Process Model	Service Profile
	REGION2			
21	GEOPOLITICAL-ENTITY	WEATHERPROCESS	2	2
22	DEGREE, GOVERNMENT	SCHOLARSHIP	8	6
23	MISSILE, GOVERNMENT	FUNDING	11	8
24	GROCERYSTORE	FOOD	4	3
25	HOSPITAL	INVESTIGATING	4	3
26	DOOR	LOCK	65	52
27	MAXPRICE	COLA	5	4
28	MILES	KILOMETERS	1	1
-29	NOVEL	AUTHOR	14	11
30	DOOR	OPEN	65	47
31	PREPAREDFOOD	PRICE	14	10
32	PUBLICATION-NUMBER	PUBLICATION	4	4
33	RECOMMENDEDPRICE	COFFEE, WHISKEY	5	3
34	RESEARCHER-IN-ACADEMIA	ADDRESS	4	3
35	SHOPPINGMALL	CAMERA, PRICE	3	2
36	SURFING	DESTINATION	4	3
37	HIKING, SURFING	DESTINATION	9	8
38	ORGANIZATION, SURFING	DESTINATION	9	8
39	TITLE	COMEDYFILM	10	6
40	TITLE	VIDEOMEDIA	12	7
41	UNIVERSITY	LECTURER-IN-ACADEMIA	6	6
42	SCIENCE-FICTION-NOVEL, USER	PRICE	18	18

4.1. Precision and Recall.

Precision is the ability to retrieve the most precise services. Higher the precision means better the relevance and more precise results, but may imply fewer results returned. Recall means the ability to retrieve as many services as possible that matches or related to a query.

The precision of the first *n* results of a query *q* is computed as following:

$$\frac{\text{Number of relevant services retrieved}}{\text{Number of relevant services retrieved} + \text{Number of irrelevant services retrieved}} \times 100$$

Table 3. Percentage of Precision and recall of the services.

S. No.	Test Data Set	No. of Matched Services		Relevant Retrieved		Relevant not Retrieved		Irrelevant Retrieved		Precision (%)		Recall (%)	
		Process Model	Service Profile	Process Model	Service Profile	Process Model	Service Profile	Process Model	Service Profile	Process Model	Service Profile	Process Model	Service Profile
1	1	20	15	12	7	8	9	8	8	60.00	46.67	60.00	43.75
2	2	21	12	15	8	5	4	6	4	71.43	66.67	75.00	66.67
3	3	86	94	79	71	13	14	15	15	84.04	82.56	85.87	83.53
4	4	23	18	17	12	5	7	6	6	73.91	66.67	77.27	63.16

S. No.	Test Data Set	No. of Matched Services		Relevant Retrieved		Relevant not Retrieved		Irrelevant Retrieved		Precision (%)		Recall (%)	
		Process Model	Service Profile	Process Model	Service Profile	Process Model	Service Profile	Process Model	Service Profile	Process Model	Service Profile	Process Model	Service Profile
5	5	18	9	10	4	28	26	8	5	55.56	44.44	26.32	13.33
6	6	6	6	5	4	3	4	1	2	83.33	66.67	62.50	50.00
7	7	10	8	9	7	4	6	1	1	90.00	87.50	69.23	53.85
8	8	16	11	12	8	5	4	4	3	75.00	72.73	70.59	66.67
9	18	3	2	3	2	1	1	0	0	100.00	100.00	75.00	66.67
10	19	6	3	6	3	2	4	0	0	100.00	100.00	75.00	42.86
11	20	7	6	2	2	4	4	5	4	28.57	33.33	33.33	33.33
12	22	8	6	4	4	4	4	4	2	50.00	66.67	50.00	50.00
13	23	11	8	3	1	13	15	8	7	27.27	12.50	18.75	6.25
14	24	4	3	2	1	10	8	2	2	50.00	33.33	16.67	11.11
15	25	4	3	4	2	2	3	0	1	100.00	66.67	66.67	40.00
16	26	65	52	58	42	5	6	7	10	89.23	80.77	92.06	87.50
17	27	5	4	4	3	9	9	1	1	80.00	75.00	30.77	25.00
18	29	14	11	11	8	6	7	3	3	78.57	72.73	64.71	53.33
19	30	65	47	54	39	10	9	11	8	83.08	82.98	84.38	81.25
20	31	14	10	11	7	4	4	3	3	78.57	70.00	73.33	63.64
21	33	5	3	5	2	4	3	0	1	100.00	66.67	55.56	40.00
22	34	4	3	4	3	1	3	0	0	100.00	100.00	80.00	50.00
23	35	3	2	3	2	1	1	0	0	100.00	100.00	75.00	66.67
24	36	4	3	4	3	0	0	0	0	100.00	100.00	100.00	100.00
25	37	9	8	4	3	5	5	5	5	44.44	37.50	44.44	37.50
26	38	9	8	7	6	2	3	2	2	77.78	75.00	77.78	66.67
27	39	10	6	8	6	3	3	2	2	80.00	75.00	72.73	66.67
28	40	12	7	9	5	3	2	3	2	75.00	71.43	75.00	71.43
29	41	6	6	5	4	1	1	1	2	83.33	66.67	83.33	80.00
30	42	18	18	13	11	3	4	5	7	72.22	61.11	81.25	73.33

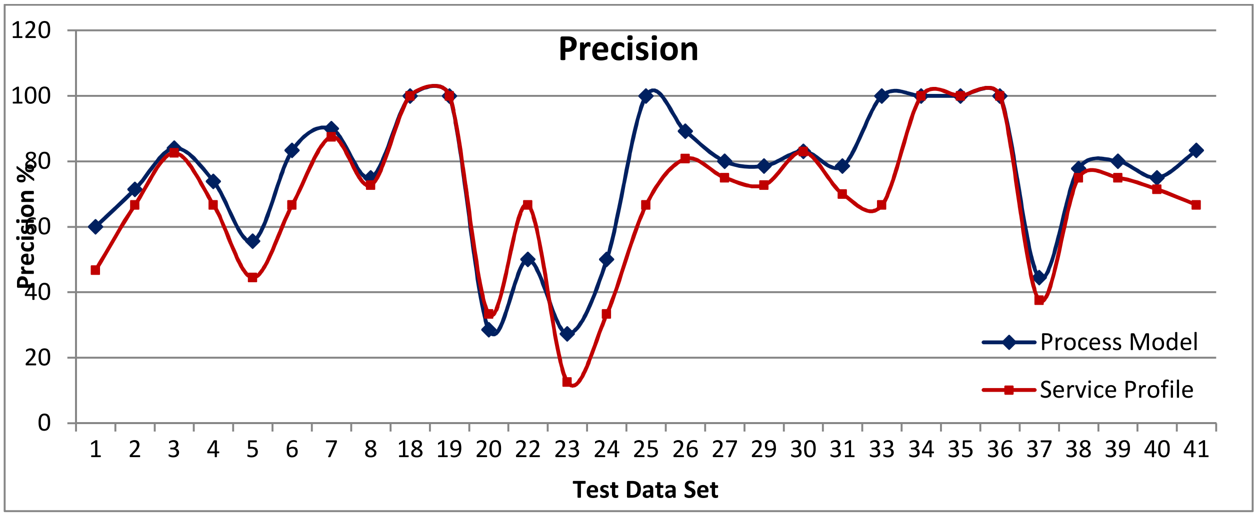


Figure 3. Performance comparison in terms of Precision.

Thus, precision is the ratio of the total number of relevant services retrieved to the total number of irrelevant and relevant services retrieved.

As well as the recall of the first *n* results of a query *q* is computed as follows:

$$\frac{\text{Number of relevant services retrieved}}{\text{Number of relevant services retrieved} + \text{Number of relevant services not retrieved}} \times 100$$

Recall is the ratio of the number of relevant services retrieved to the total number of relevant services in the repository. The precision and recall of service profile ontology and process model ontology based algorithms for selected test queries are tabulated in table 3.

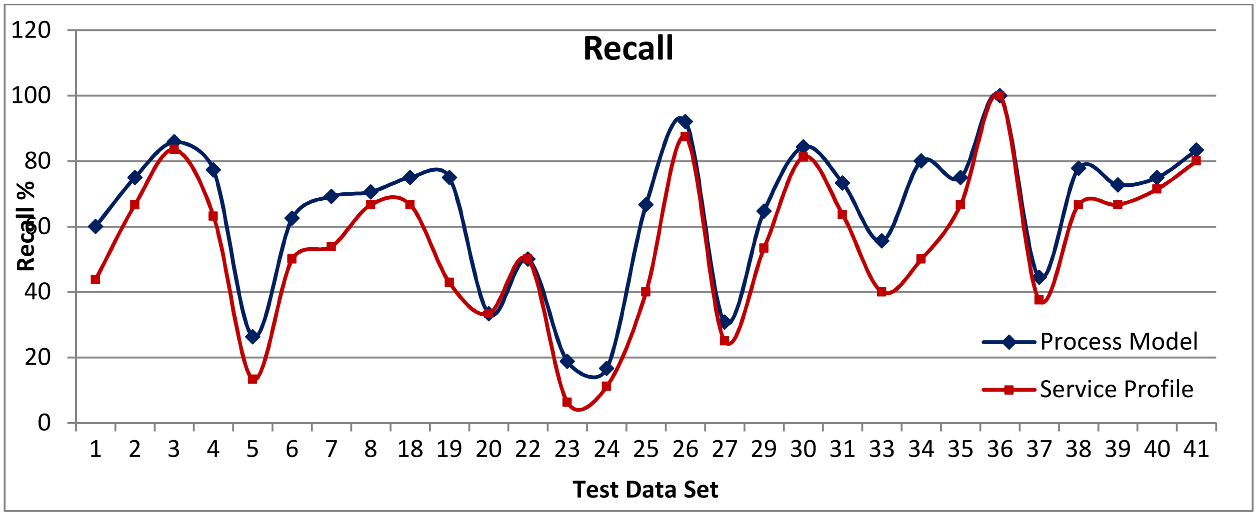


Figure 4. Performance comparison in terms of Recall.

It is evident that, process model based results have higher precision and recall than that of the service profile based results and are shown in Figure 3 and Figure 4 respectively.

4.2. Processing Time

In order to compare the performance of both the algorithms, the processing time also has been calculated in addition to the precision and recall. Figure 5 shows that the processing time of the two algorithms in terms of their performance. It is evident that for each query, the time taken by the service profile based algorithm is relatively more than that of process model based one. The reason is, usually the service profile is not annotated with ontology concepts. Therefore, to access the ontology concepts for the given input, the service profile based algorithm has to invoke the sub tag `<process:parameterType> . . . </process:parameterType>` of `<process:hasInput . . ."/>` tag of process model ontology. Whereas the process model based algorithm will directly invoke the ontology concept, so naturally its processing time is less when compared to the service profile ontology based algorithm.

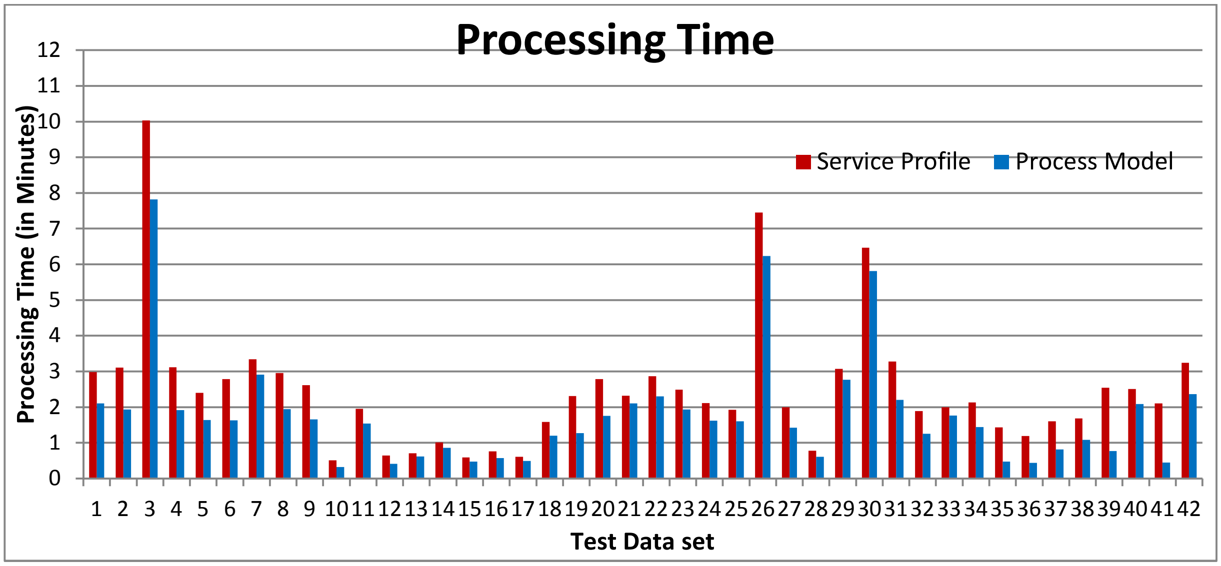


Figure 5. Performance in terms of query processing time.

5. SERVICE COMPOSITION AND EXECUTION PLAN GENERATION.

Service composition is the process of composing autonomous services to attain new functionality and is another challenging task of CSWS. Service composition has the potential to reduce the development time and effort for new applications. Since the web has several independent service providers, there is an inherent need for composing complementary services to achieve the end user's needs. Preliminary experiments and performance studies discussed in the previous sections shows that the process model ontology can effectively be used for service discovery. Indeed a novel architecture is proposed in this work for service discovery, composition and execution plan generation is shown in Figure 6. This architecture consists of two distinct phases, namely, the service discovery phase and the service composition phase. The main function of the service discovery phase is to find the relevant atomic services from the CSWS. The service composition phase focuses mainly on the composition of the atomic services discovered from the discovery phase. The output of the first phase of the architecture is a list of atomic services that are to be composed in the composition phase. The composition phase utilizes the advantage of the abductive event calculus to generate all possible plans for the composition of the atomic services.

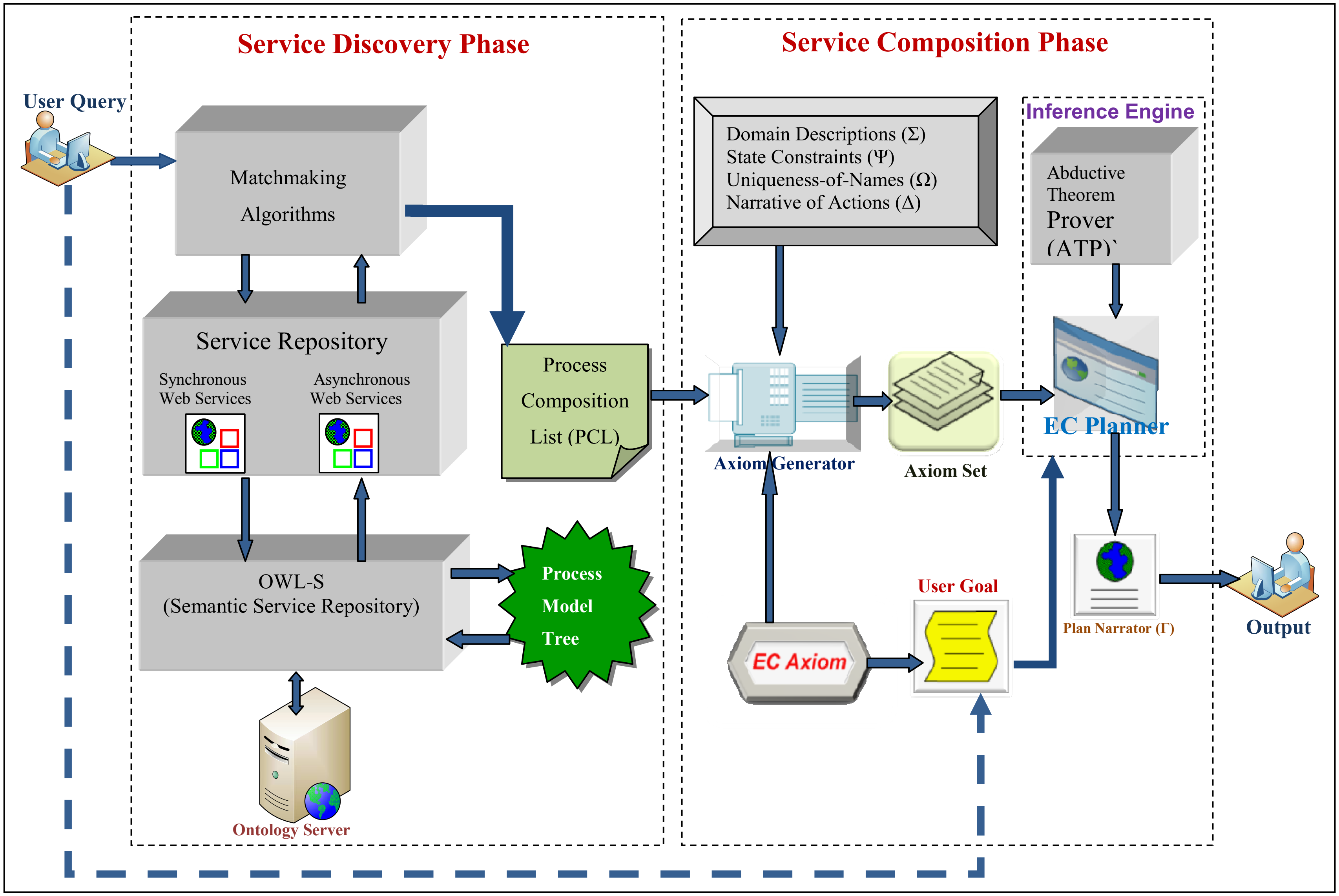


Figure 6. Architecture for service discovery, composition and execution plan generation with abductive reasoner

The roles and functionalities of the modules of the architecture are explained with a simple example in the following sub section.

5.1. Example Scenario 1: Find the price of a book.

A user wants to know the price of a book published by a specified publisher. Assuming the user has given the query as:

```
Input: Book,Title,Publisher
Output: Price
```

In this query the user enters the required inputs as “Book, Title, Publisher” and these parameters are used to match the atomic services of the CSWS. At the same time the composition of the atomic services must produce the output “Price” as the user expects.

According to Martin *et al*. (2004), any CSWS can be considered as a tree whose nonterminal nodes are labeled with control constructs. The terminal nodes of the tree are atomic services, which are directly invoked by a client or an agent. Based on this concept, the Process Model Tree (PMT) representation of the BookService composite service is shown in Figure 7.

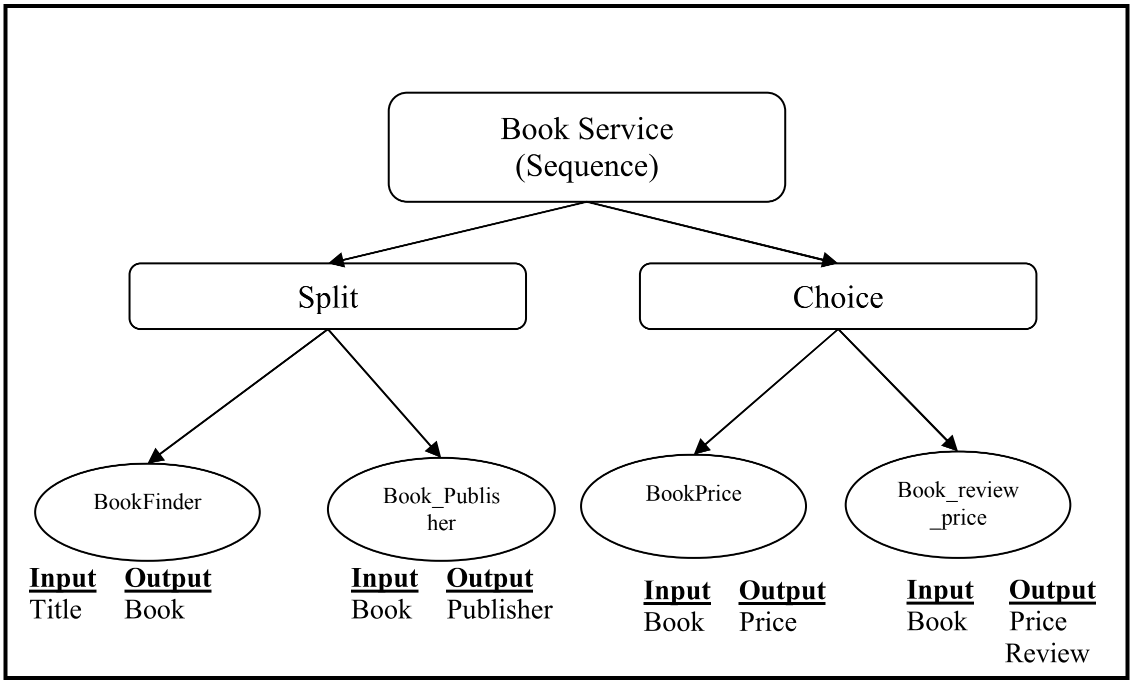


Figure 7 Process Model Tree of the BookFinder CSWS

During the matching process, the algorithm first invokes the root of the tree and descends towards the leaf node. Once it reaches the leaf node, the user input query is matched with the inputs of the process model ontology. If no match is found, the algorithm will further move to the next node and the process will continue until all the nodes are visited. If a match is found, the Process Composition List (PCL) is updated with the atomic services along with its inputs and outputs and also the control construct present in the root node of that atomic service. The Pseudo code of the atomic service discovery algorithm is shown in Listing 1.

PCL is the input to the axiom generator which will convert the OWL-S description of the discovered set of atomic services into equivalent EC axioms. For this conversion, the axiom generator utilizes the EC literals, such as Domain Description (Σ), State Constraints (Ψ), Uniqueness-of-Names (Ω) and Narrative of actions (Δ). The axiom generator interprets the details of the atomic services present in the PCL one by one, and converts them into EC axioms and produces an axiom set which is the output of the axiom generator. This axiom set is the actual axiomatic representation of all the discovered services and the axiom set generated by the axiom generator for the BookFinder composite service is shown in Figure 8. The second order abductive theorem prover present in the inference engine uses this axiom set to generate all possible plans for the service composition.

```
axiom(initiates(book_finder(A,B),f_bookfinder,T),[bookfinder(A,B)]).
axiom(initiates(book_publisher(A,B),f_bookpublisher,T),
[bookpublisher(A,B),holds_at(f_bookfinder,T)]).
axiom(initiates(book_price(A,B),f_bookprice,T),
[bookprice(A,B,C,D,E,F),holds_at(f_bookpublisher,T)]).

/* state constraints */
axiom(holds_at(price(title, book, publisher, price),T)).

/* contextual conditions */
axiom(bookfinder(title, book),[]).
axiom(bookpublisher(book, publisher),[]).
axiom(bookprice(book, price),[]).

/* actions(services) used to construct the plan */
executable(book_finder(A,B)).
executable(book_publisher(A,B)).
executable(book_price(A,B,C,D,E,F)).
```

Figure 8. The axiom set for the BookFinder composite service

The Inference engine encompasses the second order Abductive Theorem Prover (ATP) and an EC planner (Shanahan 2000). The inference engine generates a sequence of actions (a.k.a., a plan or a composition) that achieves the goals specified in the query. The query mentioned in this example, that is, the user expected output “Price” is first converted into an EC axiom and fed into the inference engine. A goal is a finite conjunction of the formulae of the form (¬) HoldsAt(β,τ) where β is a ground fluent term and τ is a ground time.

The abductive theorem prover present in the inference engine, coded as a Prolog meta-interpreter, will populate the plan narrator with the plans for the given goal. The plans generated by the inference engine consists of a set of *happens* and *before* literals. The plan narrator contains the set of events that are to be executed in some order. Planning can be thought of as the inverse operation to temporal projection which in the event calculus is naturally cast as a deductive task. A narrative is a finite conjunction of the formulae of the form, Happens (α,τ), $\tau 1 < \tau 2$ where α is a ground action and τ1 and τ2 are time points (Shanahan 1995). Thus, the plan generated for the given goal "Price" for the Book Service example is shown below.

[[happens(book_finder(title,book), t3,t3),
happens(book_publisher(book, publisher),t2,t2),
happens(book_price(book,price), t1, t1)],
[before(t3, t1),
before(t2, t1),
before(t1, t)]];

Here, **happens**(book_finder(title, book), t3, t3) means that the book_finder atomic service with input "title" and output "book" is executed within the time interval [t3, t3]. The temporal ordering of the time intervals are indicated by the "before" literal. The execution plan generated by the inference engine for the BookFinder example is shown in Figure 9.

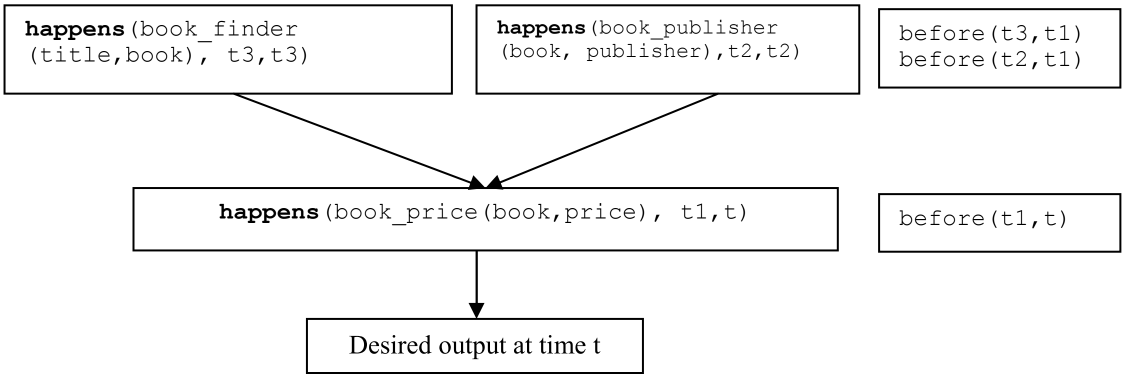


Figure 9. Execution plan generated by EC planner for the BookFinder example.

5.2. Example scenario 2: Micro Finance Institutions and Self Help Groups in the Social Network.

The term Micro Finance Institution (MFI) refers to the provision of financial services for low-income households and micro entrepreneurs (both urban and rural) for productive purposes. MFI is the supply chain of loans, savings and other basic financial services to the poor. Financial services needed by the poor include working capital loans, consumer credit, and savings, pensions, insurance and money transfer services. The majority of the poor people in rural areas around the world lack access to financial services. In India alone, over 200 million people (36% of the rural population) do not have access to a bank. Although India has more MFIs than any other countries, these programs only reach a small percentage of needy households. Most Micro Finance Institutions in India are built upon the grassroots infrastructure of a self-help group (SHG). These are small village-based groups within which micro finance transactions are conducted. Due to programs such as NABARD's (National Bank for Agricultural and Rural Development) SHG-bank linkage programme, these groups are increasingly being seen as a viable market for financial services in rural areas in India. NABARD is the arm of the central bank which encourages Indian banks to invest in microfinance. Most SHGs operate in far-flung rural areas, making travel cumbersome and face high transaction cost while dealing with banks due to distances, small value of financial transactions etc. To combat this problem, some MFIs are using new information and communication technologies (ICTs) such as smart cards, handhelds, and modified ATMs. In this paper a new approach based on Web Services is proposed to overcome the difficulties faced by the SHGs and minimizes the transaction costs and help to extend the availability of microfinance.

This work has taken up two composite semantic Web Services namely, INSURANCE_PAYMENT service and a MICRO_FINANCE_INSTITUTION service, having several atomic services as example. Insurance premium is paid using the INSURANCE_PAYMENT service. MICRO_FINANCE_INSTITUTION service is used to make financial transactions like fund transfer, loan repayments. It also provides facility to create virtual credit cards. In the event of a SHG member from the rural area who has to pay the insurance premium as well as make two more financial transactions such as fund transfer from savings bank (SB) account to recurring deposit (RD) and repayment of loan will specify the necessary input and required output from the service as a query as illustrated below.

Inputs: *username, password, policy_num, bank_info, acc_num, rd_acc_num, loan_acc_num, loan_type, amount*
Outputs: *premium_paid_receipt, money_transfer_receipt, loan_paid_receipt*

According to this query, neither of these two services alone can fulfill the request by itself. For example *cr_card_type*, *cr_card_num* and *pin_num* are the three inputs required by *Pay_Premium* atomic service of the INSURANCE_PAYMENT service. But these three inputs are not provided in the query. So, INSURANCE_PAYMENT service requires another service to generate these three parameters. However, these three parameters can be obtained from the MICRO_FINANCE_INSTITUTION service, as one of its atomic services namely *Virtual_Credit_Card* generates the required outputs *cr_card_num, cr_card_type* and *pin_num*.

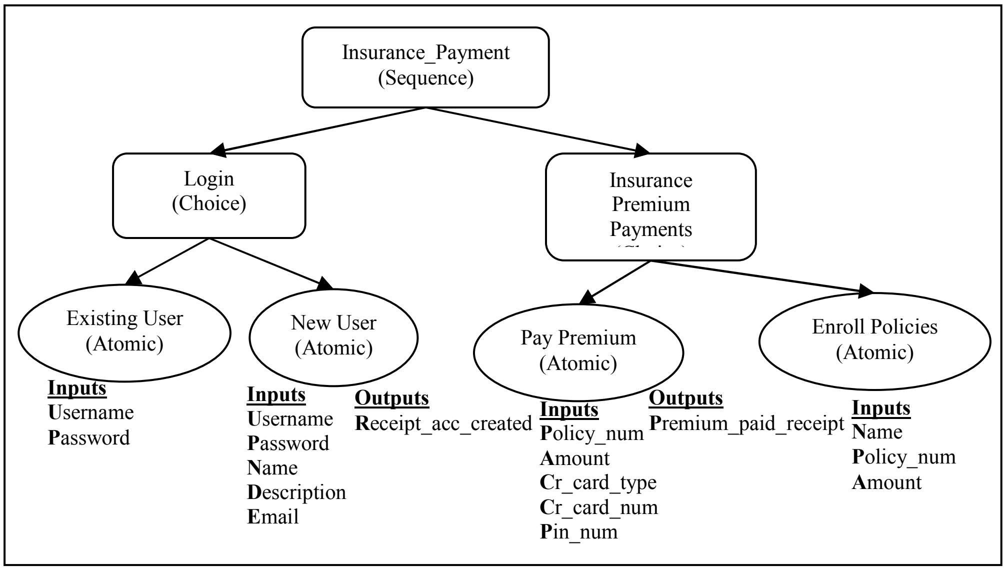


Figure 10. Process Model Tree (PMT) of insurance_payment service.

The PMT of INSURANCE_PAYMENT and MICRO_FINANCE_INSTITUTION services are shown in Figure 10 and Figure 11 respectively.

The function *MatchAtomicProcess* present in the pseudo code in Listing 1 first invokes the root node of the PMT shown in Figure 10 of the INSURANCE_PAYMENT Composite Semantic Web Service. The function further descends and reaches out to the leaf node and invokes the atomic service namely the *Existing_User* and checks if the PCL already contains this atomic service, if not, the *MatchAtomicProcess* calls the *AddAtomicProcess* to add this atomic service into the PCL. The *AddAtomicProcess* checks whether all the inputs of the *Existing_User* are given in the query Q_{input}. Since all the inputs of the *Existing_User* are matching with Q_{input}, the *Existing_User* atomic service is identified as a required service for composition. Therefore, the *Existing_User* entity in the PCL is updated with all its inputs and outputs. The algorithm proceeds in this way and at one point it invokes the *Pay_Premium* atomic service of INSURANCE_PAYMENT Web Service. The *AddAtomicProcess* matches all the inputs, - *policy_num, amount, cr_card_type, cr_card_num, pin_num* - with Q_{input}. Here the user has not provided the inputs, such as, *cr_card_type*, *cr_card_num* and *pin_num* in the query Q_{input} which are required to match the *Pay_Premium* atomic service.

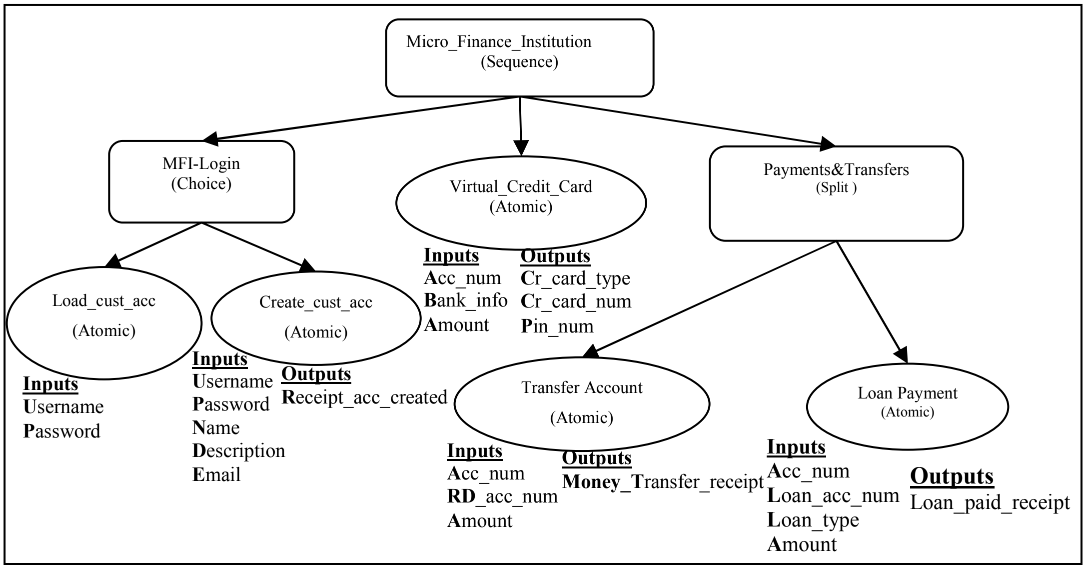


Figure 11. Process Model Tree (PMT) of micro_finance_institution service.

Listing 1. Pseudocode for the service discovery

```
Inputs
Q – The given query consists of $Q_{input}$ and $Q_{output}$ where,
        $Q_{input}$ – A set of user's input parameters
        $Q_{output}$- A set of user expected outputs.
Output
        PCL – Process Composition List
Local Resources
$P_{Ai}$ – Process node of atomic node $A_i$
SR – Service Registry
$A_{input}$ – Input of Atomic Process A
$A_{output}$ – Output of Atomic Process A
$E_{SQ}$ = ({$P_1$},{$P_2$} ) – be two sets of processes, which are to be executed sequentially. An
    event of type $E_{SQ}$ is a sequence event.
$E_{XE}$ = ({p},P ) – be a set P of processes which are in mutual exclusion relationship with
    the process node {p}. An event of type $E_{XC}$ is an excluding event.
$E_{CE}$ = ({$P_1$},{$P_2$} ) – be two sets of processes, which are to be executed concurrently. An
    event of type $E_{CE}$ is a concurrent event.
$SEQUENCE_A$ – be the set of atomic processes which can be executed sequentially.
$CHOICE_A$ – be the set of atomic processes out of which only A can be chosen for
    execution.
$SPLIT_A$ – be two sets of atomic processes which all can be executed concurrently.
GeneratePCL(ServiceRegistry SR, Query Q, Process-Composition-List PCL)
 repeat
```

```
    forall services $S_i$ in $\{S_1,S_2,\ldots,S_n\} \in$ SR do
        MatchAtomicProcess(Root($S_i$),PCL)
 until no more process node is added into PCL
MatchAtomicProcess(ProcessNode P, Process-Composition-List PCL)
forall $P_i$ in $\{P_1,P_2,\ldots,P_n\}$ do
        forall Atomic Node $A_i$ in $P_i$ do
if ($A_i \notin$ PCL) then
                AddAtomicProcess($A_i$,PCL);
AddAtomicProcess(AtomicNode $A_i$, Process-Composition-List PCL)
findmatch = true;
add the atomic process $A_i$ into PCL;
if ($\forall A_{i\text{-input}} \in Q_{input}$) then
        forall input ip $\in A_{i\text{-input}}$ do
        add ip to the input parameter list of $A_i \in$ PCL;
        forall output op $\in A_{i\text{-output}}$ do
        add op to the output parameter list of $A_i \in$ PCL;
else    // $A_i$ fails to match with $Q_{input}$, so search other services in the service registry SR,
        except the present service $S_i$, to find another atomic process.
     repeat
    forall services $S_j$ in $\{S_1,S_2,\ldots,S_n\} \setminus \{S_i\}$ do
        findmatch =MatchAnotherAtomicProcess(Root($S_j$) ,PCL);
 until no more process node is added into PCL
     if (findmatch) then
SetControlConstruct($P_{Ai}$)
    else
    remove $A_i$ from PCL;      // No match found and the atomic process $A_i$ is not
                                required process, so remove from the composition list
boolean MatchAnotherAtomicProcess (ProcessNode P, Process-Composition-List PCL)
addNewAtomicProcess = true;
forall $P_j$ in $\{P_1,P_2,\ldots,P_n\}$ do
 forall AtomicNode $A_j$ in $P_j$ do
 if ($A_j \notin$ PCL) then
       addNewAtomicProcess = AddSupportingAtomicProcess($A_j$,PCL);
     else
                    return false;
boolean AddSupportingAtomicProcess(AtomicProcess $A_j$, Process-Composition-List PCL)
foundNewAtomicProcess = true;
add the atomic process $A_j$ to PCL;
if ($\forall A_{j\text{-output}} \in Q_{input}$ ) then
  if ($\forall A_{j\text{-input}} \in Q_{input}$ ) then
      forall output op $\in A_{j\text{-output}}$ do
     add op to the output parameter list of $A_j \in$ PCL;
           forall input ip $\in A_{j\text{-input}}$ do
       add ip to the input parameter list of $A_j \in$ PCL;
```

```
            return true;
    else
            remove $A_j$ from PCL;    // No match found and the atomic process $A_j$ is
                                         not a candidate process, so remove it from PCL

            return false;

void SetControlConstruct(ProcessNod $P_{Ai}$)
  forall choice node P in $CHOICE_A$ do
        add  $A_i$ into PCL such that $(\{C_N\}, A_i) \in E_{XE}$
  forall sets of process P in $SPLIT_A$ do
        add  $A_i$ to PCL such that $(\{P\}, A_i) \in E_{CE}$
  forall sets of process P in $SEQUENCE_A$ do
        add  $A_i$ to PCL such that $(\{P\}, A_i) \in E_{SQ}$
```

MatchAnotherAtomicProcess function is called to check the next service in the service registry to find an atomic service, whose output can produce the inputs *cr_card_type, cr_card_num* and *pin_num* required by the atomic service *Pay_Premium*. However the algorithm will invoke the second service MICRO_FINANCE_INSTITUTION and when the function invokes the *Virtual_Credit_Card* atomic service, the *AddSupportingAtomicProcess* matches all the outputs of *Virtual_Credit_Card* with the input query Q_{input} and finds a match. In order to compose this service into the composition, the inputs of *Virtual_Credit_Card* are also matched with Q_{input} and eventually a match is found. So the atomic service *Virtual_Credit_Card* is added into the PCL, because it produces the necessary outputs required by the *Pay_Premium* atomic service. Once the algorithm exits, PCL will have all the atomic services that are needed to be composed and executed in the specific order to produce the user expected output.

```
Existing_user(username, password),
Load_Cust_Acc(username, password),
Virtual_Credit_Card(acc_num,bank_info, amount, cr_card_num,
cr_card_type,pin_num),
Pay_Premium(policy_num,amount,cr_card_num,cr_card_type,pin_num,premi
um_paid_receipt),
Transfer_Account(acc_num,rd_acc_num,amount,money_transfer_receipt),
Loan_Payment(acc_num, loan_acc_num, loan_type, amount,
loan_paid_receipt)].
```

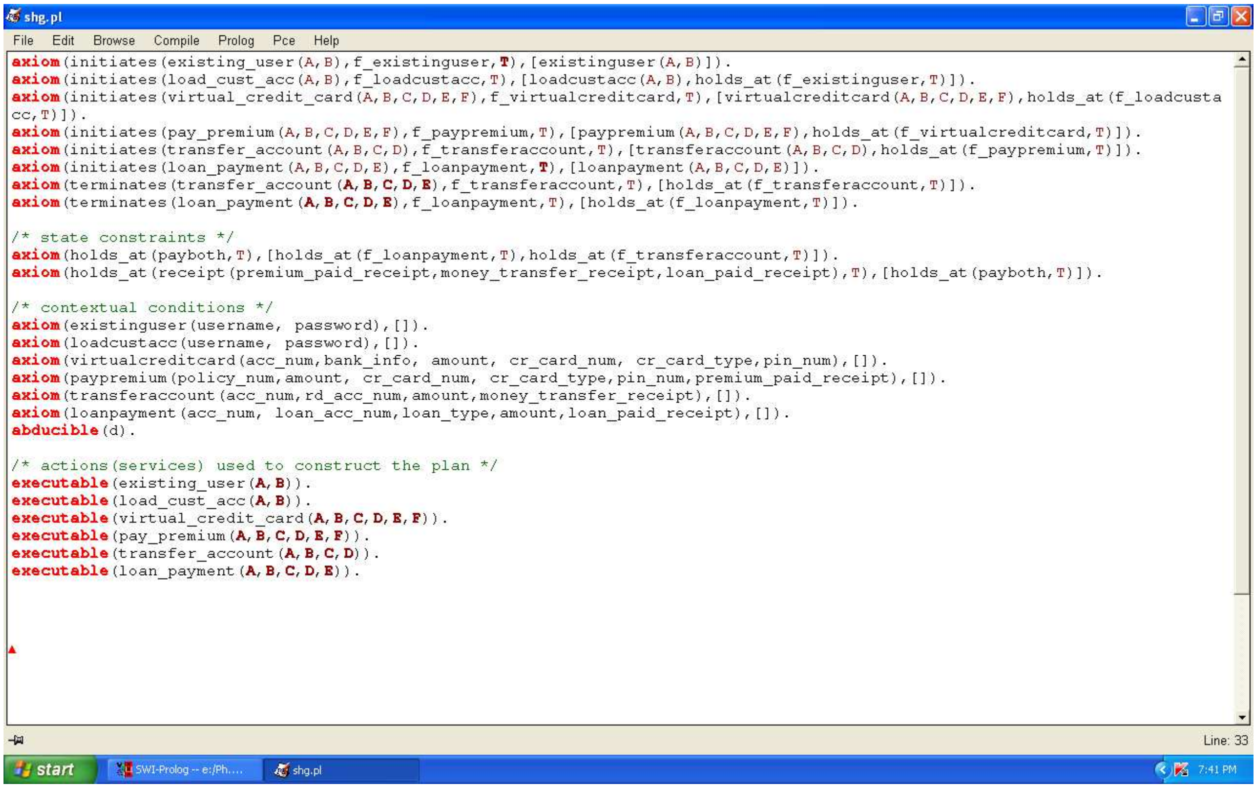
Figure 12. Process Composition List (PCL).

Listing 2. Axiom set generated by the axiom generator.

```
axiom(initiates(existing_user(A,B),f_existinguser,T),[existinguser(A,B)]).
axiom(initiates(load_cust_acc(A,B),f_loadcustacc,T),[loadcustacc(A,B),holds_at(f_existing
user,T)]).
axiom(initiates(virtual_credit_card(A,B,C,D,E,F),f_virtualcreditcard,T),[virtualcreditcar
d(A,B,C,D,E,F),holds_at(f_loadcustacc,T)]).
axiom(initiates(pay_premium(A,B,C,D,E,F),f_paypremium,T),[paypremium(A,B,C,D,E,F),holds_a
t(f_virtualcreditcard,T)]).
axiom(initiates(transfer_account(A,B,C,D),f_transferaccount,T),[transferaccount(A,B,C,D),
holds_at(f_paypremium,T)]).
axiom(initiates(loan_payment(A,B,C,D,E),f_loanpayment,T),[loanpayment(A,B,C,D,E)]).
axiom(terminates(transfer_account(A,B,C,D,E),f_transferaccount,T),[holds_at(f_transferacc
ount,T)]).
axiom(terminates(loan_payment(A,B,C,D,E),f_loanpayment,T),[holds_at(f_loanpayment,T)]).

/* state constraints */
axiom(holds_at(payboth,T),[holds_at(f_loanpayment,T),holds_at(f_transferaccount,T)]).
axiom(holds_at(receipt(premium_paid_receipt,money_transfer_receipt,loan_paid_receipt),T),
[holds_at(payboth,T)]).

/* contextual conditions */
axiom(existinguser(username, password),[]).
axiom(loadcustacc(username, password),[]).
axiom(virtualcreditcard(acc_num,bank_info, amount, cr_card_num,
cr_card_type,pin_num),[]).
axiom(paypremium(policy_num,amount, cr_card_num,
cr_card_type,pin_num,premium_paid_receipt),[]).
axiom(transferaccount(acc_num,rd_acc_num,amount,money_transfer_receipt),[]).
axiom(loanpayment(acc_num,loan_acc_num,loan_type,amount,loan_paid_receipt),[]).

/* actions(services) used to construct the plan */
executable(existing_user(A,B)).
executable(load_cust_acc(A,B)).
executable(virtual_credit_card(A,B,C,D,E,F)).
executable(pay_premium(A,B,C,D,E,F)).
executable(transfer_account(A,B,C,D)).
executable(loan_payment(A,B,C,D,E)).
```

After each successful match, the *SetControlConstruct* module is called to set the control construct present in the root node corresponding to the matched atomic node, and the PCL is updated accordingly. The output of the first phase of the proposed architecture is a list of atomic services that are required for the composition and execution and that the PCL list is shown in Figure 12. Also, the EC axiom set generated by the axiom generator for the candidate services present in the PCL is shown in Listing 2 and the corresponding Prolog editor screen is shown in Figure 13. This axiom set along with a list of *holds_at* predicates that represents the goal (user required output) is presented to the EC planner as input. The user goal of the given example is fed into the inference engine is as follows:

```
abdemo([holds_at(receipt(premium_paid_receipt,money_transfer_rec
        eipt,loan_paid_receipt),t)],R).
```



```
axiom(initiates(existing_user(A,B),f_existinguser,T),[existinguser(A,B)]).
axiom(initiates(load_cust_acc(A,B),f_loadcustacc,T),[loadcustacc(A,B),holds_at(f_existinguser,T)]).
axiom(initiates(virtual_credit_card(A,B,C,D,E,F),f_virtualcreditcard,T),[virtualcreditcard(A,B,C,D,E,F),holds_at(f_loadcusta
cc,T)]).
axiom(initiates(pay_premium(A,B,C,D,E,F),f_paypremium,T),[paypremium(A,B,C,D,E,F),holds_at(f_virtualcreditcard,T)]).
axiom(initiates(transfer_account(A,B,C,D),f_transferaccount,T),[transferaccount(A,B,C,D),holds_at(f_paypremium,T)]).
axiom(initiates(loan_payment(A,B,C,D,E),f_loanpayment,T),[loanpayment(A,B,C,D,E)]).
axiom(terminates(transfer_account(A,B,C,D,E),f_transferaccount,T),[holds_at(f_transferaccount,T)]).
axiom(terminates(loan_payment(A,B,C,D,E),f_loanpayment,T),[holds_at(f_loanpayment,T)]).

/* state constraints */
axiom(holds_at(payboth,T),[holds_at(f_loanpayment,T),holds_at(f_transferaccount,T)]).
axiom(holds_at(receipt(premium_paid_receipt,money_transfer_receipt,loan_paid_receipt),T),[holds_at(payboth,T)]).

/* contextual conditions */
axiom(existinguser(username, password),[]).
axiom(loadcustacc(username, password),[]).
axiom(virtualcreditcard(acc_num,bank_info, amount, cr_card_num, cr_card_type,pin_num),[]).
axiom(paypremium(policy_num,amount, cr_card_num, cr_card_type,pin_num,premium_paid_receipt),[]).
axiom(transferaccount(acc_num,rd_acc_num,amount,money_transfer_receipt),[]).
axiom(loanpayment(acc_num, loan_acc_num,loan_type,amount,loan_paid_receipt),[]).
abducible(d).

/* actions(services) used to construct the plan */
executable(existing_user(A,B)).
executable(load_cust_acc(A,B)).
executable(virtual_credit_card(A,B,C,D,E,F)).
executable(pay_premium(A,B,C,D,E,F)).
executable(transfer_account(A,B,C,D)).
executable(loan_payment(A,B,C,D,E)).
```

Figure 13. Prolog editor screenshot for the axiom set.

The abductive theorem prover present in the inference engine will populate the plan narrative into the residue R given in the goal. The residue R populated by the inference engine consists of a set of *happens* and *before* literals as shown in Figure 14. Services are the events in the domain and they are to be defined in the axiom definition through the *executable* clause. The **`executable`**`(existing_user(A,B)` clause in Listing 2 means that the `existing_user` event is used in constructing a plan.

```
R = [[happens(existing_user(username, password), t6, t6),
happens(load_cust_acc(username, password), t5, t5),
happens(virtual_credit_card(acc_num, bank_info, amount, cr_card_num,
cr_card_type, pin_num), t4,t4),
happens(pay_premium(policy_num, amount, cr_card_num, cr_card_type,
pin_num, premium_paid_receipt), t3, t3),
happens(transfer_account(acc_num, rd_acc_num, amount,
money_transfer_receipt),t2,t2),
happens(loan_payment(acc_num, loan_acc_num, loan_type, amount,
loan_paid_receipt), t1, t1)],
[before(t6, t5),
before(t5, t4),
before(t4, t3),
before(t3, t2),
before(t2, t),
before(t1, t)]] ;
```

Figure 14. The plan generated by Prolog inference engine.

The EC predicates such as *initiates* and *terminates* are used to specify how the execution can evolve. In addition to defining the fluent they initiate or terminate, the required preconditions for activating these predicates must be specified. In the initiates axiom, the first argument is the *initiates* clause, and the second argument is a set of preconditions necessary for the *initiates* clause to become applicable. This fact is illustrated in the following *initiates* axiom:
`axiom(initiates``(load_cust_acc(A,B),f_loadcustacc,T),[loadcustacc (A,B),holds_at(f_existinguser,T)]).`
Evidently, the event `load_cust_acc` initiates the fluent `f_loadcustacc` with the precondition `holds_at(f_existinguser,T)`. That is, the requirement to execute the event `load_cust_acc` is that, the fluent `f_existinguser` of `existing_user` event must hold. The argument `loadcustacc(A,B)` is known to be the *contextual condition*. Contextual condition can also be represented by using the *holds_at* predicates. But then there is no distinction between the precondition and the contextual condition, in this formalism. So the *domain constraints* or the *state constraints* (Ψ) are used only to represent the preconditions, which give rise to actions with indirect effects. This expresses a logical relationship held between fluents at all times. In EC, state constraints are *holds_at* formulae with a universally quantified time argument as the one given below:

```
axiom(holds_at(payboth,T),[holds_at(f_loanpayment,T),holds_at(f_
     transferaccount,T)]).
```

5.1 Concurrent Execution of events in the plan

The most significant advantage of the EC is the inherent support for concurrency. The events h*appens(e_1,t_1), happens(e_2,t_2), happens(e_3,t_3), happens(e_4,t_4)*, $t_1 < t_2 < t_4$, $t_1 < t_3 < t_4$ are examined. Since there is no relative ordering between e_2 and e_3 they are assumed to be concurrent as shown in Figure 15. It is to be observed that the service model tree of the MICRO_FINANCE_INSTITUTION service having two atomic services namely,

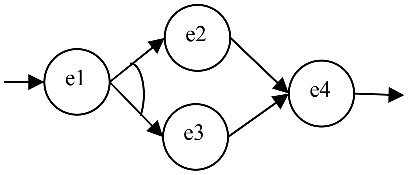


Figure 15. Concurrency of events.

Transfer_account and *Loan_payment* that are connected under a *split* control construct thus, these two atomic services are to be executed concurrently. The axiom generator in the proposed architecture is designed in such a way that, it generates the axiom sets that are proved by the abductive theorem prover and the inference engine generates the plans with simultaneous occurrence of two events. Following is a couple of events generated by the inference engine for the two atomic services:

```
happens(transfer_account(acc_num, rd_acc_num, amount,
     money_transfer_receipt),t2,t2),
happens(loan_payment(acc_num, loan_acc_num, loan_type, amount,
     loan_paid_receipt), t1, t1)],
. . .
before(t2, t),
before(t1, t)
```

The literal `before(t2,t)` means that, t2 < t. Here, the events *transfer_account* and *loan_payment* are to be executed at time t2 and t1 respectively. But both should be executed just before t and since there is no relative ordering between t2 and t1, it is assumed that these two events are to be executed concurrently, as shown in the execution order of the plan in Figure 16.

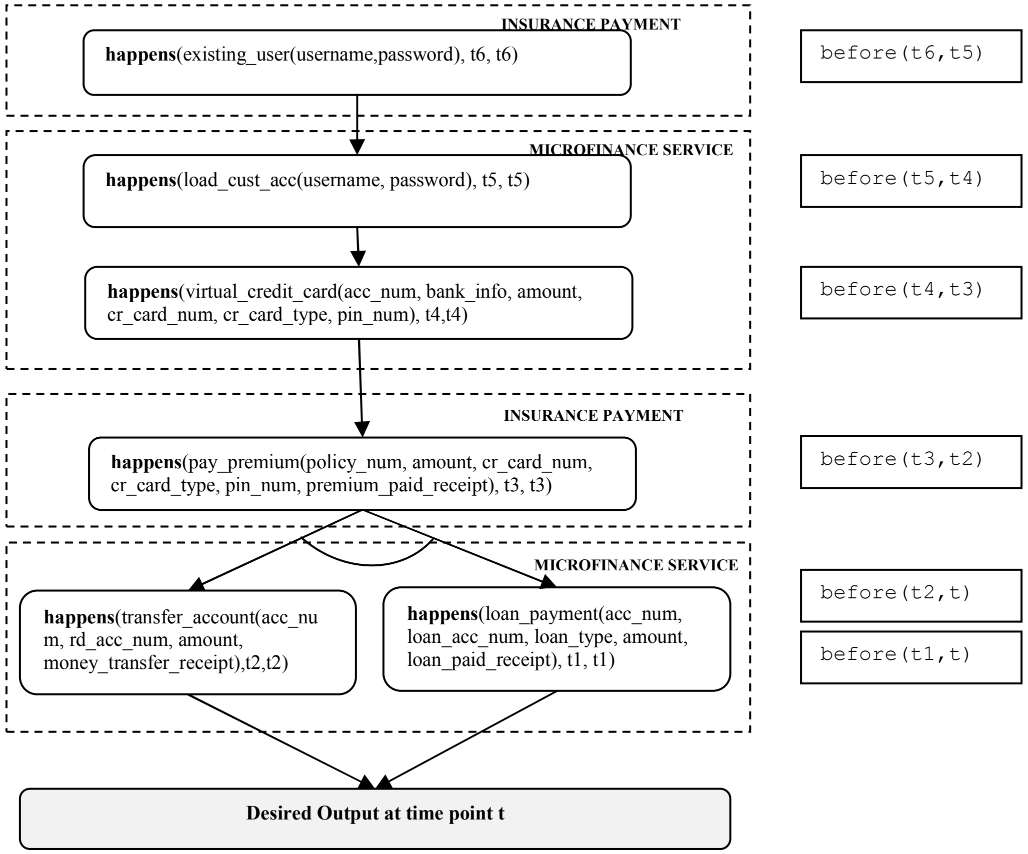


Figure.16 Execution path generated by the EC planner.

5.2. Soundness and Completeness

A plan is *sound* if and only if it does not allow deducing invalid conclusions and *complete* if and only if it allows deducing all the possible and valid conclusions by applying the axioms permitted. The inference engine in the proposed architecture in this work has generated the plans that are proper and sensible. Hence, the emphasis is that the plan generated by the proposed architecture is sound and complete.

6. CONCLUSION.

The experimental result of atomic service discovery and composition from Composite Semantic Web Service is promising. The first phase of the architecture takes the strength, advantage and features of service model ontology for atomic service discovery. The second phase takes the advantage of the abductive event calculus. The inference engine in the planner uses the second order abductive theorem prover as its main inference method. All the time the planner generates domain independent correct plans and never generates an invalid plan. The plans are scalable and extendible to meet the desired goal set by the user. Since the inference engine generates all possible plans and never generates invalid plans, it is emphasized that the plans are sound and complete. Other works in this area have proposed solutions for the composition of *atomic services* only, but this work has proposed a solution for the composition of *composite semantic web services*.

REFERENCES

Abouzaid, F. and Mullins, J. (2008). A Calculus for Generation, Verification and Refinement of BPEL Specifications. *Electronic Notes in Theoretical Computer Science*., **200**(3), 43–65.

Akkiraju, R., Farrell,J., Miller,J., Nagarajan,M., Schmidt,M., Sheth,A. and Verma, K.(2005). Web Service Semantics - WSDL-S. *Version1.0 W3C Member Submission, http://www.w3.org/Submission/2005/SUBM-WSDL-S-20051107/*

Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P. and Montali, M. (2006). A-Priori Verification of Web Services with Abduction. *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming – 2006*, Venice, Italy, July 10-12, (pp. 39-50), ACM, New York, USA.

Andrews,T. et al. (2003). Business Process Execution Language for Web Services Version 1.1.Retrieved January 15, 2010, from *http://dev2dev.bea.com/technologies/webservices/BPEL4WS.jsp*

Aydin, O., Kesim Cicekli, N. and Cicekli, I. (2007). Automated Web Service Composition with the event calculus. *Engineering Societies in the Agents World, 8th International Workshop 2007*, (pp. 142–157). Springer-Verlag Berlin Heidelberg.

Bansal, S. and Vidal, M. (2003). Matchmaking of Web Services Based on the DAMLS Service Model. *Proceedings of the second international joint conference on Autonomous agents and multiagent systems-2003*, Melbourne, Australia, July 14-18, (pp. 926-927), ACM Press.

Brogi, A., Corfini, S. and Popescu, R. (2008). Semantic-Based Composition-Oriented Discovery of Web Services. *ACM Transactions on Internet Technology*., **8**(4), 19:1-19:33.

Brogi, A. (2010). On the potential advantages of exploiting behavioural information for contract based service discovery and composition. *The Journal of Logic and Algebraic Programming*, **80** (1), 3-12.

Bruijn, J., Lausen, H., Polleres, A. and Fensel, D. (2005). The Web Service Modeling Language WSML: An Overview. *DERI- Digital Enterprise Research Institute*, Retrieved December 20,2009, from *http://www.wsmo.org/wsml/wsml-resources/deri-tr-2005-06-16.pdf*

Chifu, V., Salomie, I. and Chifu, E. (2008). Fluent Calculus-Based Web Service Composition - From OWL-S to Fluent Calculus. *Intelligent Computer Communication and Processing*, Cluj-Napoca, Romania,August 28-30, (pp. 161-168).

Evren, S., Bijan, P.,Bernardo, C., Aditya, K. and Yarden, K., (2007). Pellet: A Practical OWL DL Reasoner, *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, **5** (2), 51-53.

Gaaloul, W., Rouached, M., Godart, C. and Hauswirth,M. (2007). Verifying Composite Service Transactional Behavior Using Event Calculus. *Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg*., 4803, (pp. 353–370).

Hoffmann,J., Bertoli, P., Helmert, M. and Pistore, M. (2009). Message-Based Web Service Composition, Integrity Constraints, and Planning under Uncertainty: A New Connection. *Journal of Artificial Intelligence Research*., **35**(1), 49-117.

Klusch, M., Kapahnke, P., Fries, B., Khalid, M.A. and Vasileski, M. "OWLS-TC - OWL-S Service Retrieval Test Collection Version 4.0 revision 1", http://projects.semwebcentral.org/projects/owlstc/, 2009.

Kopecky,J., Vitvar, T., Bournez,C., Farrell, J. (2007). SAWSDL: Semantic Annotations for WSDL and XML Schema *IEEE Internet Computing.*, **11**(6), 60-67.

Kowalski, R.A and Sergot, M.J. (1986). A Logic–based calculus of events. *New Generation Computing*., **4**(1), 67-95.

Lecue, F., Leger, A. and Delteil, A. (2008). DL Reasoning and AI Planning for Web Service Composition. *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, Sydney, Australia, 9-12 December, (pp. 445-453.

Martin, D. et al. (2004). OWL-S Semantic Markup for Web Services 2004. Retrieved October 10, 2009, from *http://www.w3.org/submission/owl-s*

Meditskos, G. and Bassiliades, N. (2010). Structural and Role-Oriented Web Service Discovery with Taxonomies in OWL-S. *IEEE Transactions On Knowledge And Data Engineering*., **22**(2), 278-290.

MINDSWAP: Maryland Information and Network Dynamics Lab Semantic Web Agents Project, OWL-S API, *http://www.mindswap.org/2004/owl-s/api/*

Okutan, C. and Cicekli, N. (2010). A monolithic approach to automated composition of semantic web services with event calculus. *International Journal of Knowledge-Based Systems.*, **23**(5)**,** 440-454.

Ozorhan, E., Kuban, E.' and Cicekli, N. (2010). Automated composition of web services with the abductive event calculus. *International Journal of Information Sciences*., **180**(19), 3589-3613.

Paulraj, D. and Swamynathan, S. (2010). Dynamic Discovery and Composition of Semantic Web Services Using Abductive Event Calculus. *Proceedings of ITC 2010- IEEE International Conference on Recent Trends in Information, Telecommunication, and Computing*., Kochi, India, 12-13 March, (pp. 70-74).

Paulraj, D, Swamynathan, S and Madhaiyan, M (2012), Process Model-Based Atomic Service Discovery and Composition of Composite Semantic Web Services using Web Ontology Language for Service (OWL-S), *International Journal of Enterprise Information Systems*. **6(4)**, 445-471

Petrie, C. (2009). Planning Process Instances with Web Services. *Proceedings of the International Conference on Enterprise Information Systems AT4WS 2009*, Milan, Italy, May 6-7, (pp. 31-35).

Roman,D., Keller,U., Lausen,H., Bruijn,J., Lara,R., Stollberg,M., Polleres,A., Fensel,D. and Bussler, C. (2005). Web Service Modeling Ontology. *Applied Ontology Journal*.,**1**(1),77-106.

Rouached, M. and Godart, C. (2006). Securing Web Service Compositions: Formalizing Authorization Policies Using Event Calculus. *ICSOC 2006, Lecture Notes in Computer Science.*, 4294, (pp. 440–446).

Salomie,I., Viotica, R., Harsa, I. and Gherga, M. (2008).Towards Automated Web Service Composition with Fluent Calculus and Domain Ontologies. *Proceedings of the iiWAS2008*, Linz, Austria, 24-26 November 2008, (pp. 201-207).

Segev, A. and Toch, E.(2009). Context-Based Matching and Ranking of Web Services for Composition. *IEEE Transactions On Services Computing*., **2**(3), 210-222.

Seog-Chan, O., Lee, D and Kumara, S. (2005). A Comparative Illustration of AI Planning-based Web Services Composition. *ACM SIGecom Exchanges.*, **5**(5), 1-10.

Seog-Chan, O., Lee, D. and Kumara, S.(2007). Web Service Planner (WSPR): An Effective and Scalable Web Service Composition Algorithm. *International Journal of Web Services Research.*, **4**(1)**,** 1-22.

Shanahan, M. (1995). A Circumscriptive Calculus of Events. *International Journal of Artificial Intelligence*., **77**(2), 249-284.

Shanahan, M. (1999). The Event Calculus Explained. *Springer Lecture Notes in Artificial Intelligence*., 1600, (pp. 409-430).

Shanahan, M. (2000). An Abductive Event Calculus Planner. *Journal of Logic Programming*., **44**(1-3), 207-240.

Xu, D., Qi, Y., Hou, D. and Wan, G. (2008). An Improved Calculus for Secure Dynamic Services Composition. *Proceedings of the 2008-32nd Annual IEEE International Computer Software and Applications Conference*, Turku, Finland, July 28 - August 01, (pp**.** 686-691)**.**

Yolum, P. and Munindar, P. (2004). Reasoning About Commitments in the Event Calculus: An Approach for Specifying and Executing Protocols. *Annals of Mathematics and Artificial Intelligence*., **42**(1-3)**,** 227–253.