Fluent Calculus-based Semantic Web Service Composition and Verification using WSSL

George Baryannis and Dimitris Plexousakis

Department of Computer Science, University of Crete, Heraklion, Greece
Institute of Computer Science, FORTH, Heraklion, Greece
{gmparg,dp}@csd.uoc.gr

Abstract. We propose a composition and verification framework for Semantic Web Services specified using WSSL, a novel service specification language based on the fluent calculus, that addresses issues related to the frame, ramification and qualification problems. These deal with the succinct and flexible representation of non-effects, indirect effects and preconditions, respectively. The framework exploits the unique features of WSSL, allowing, among others, for: compositions that take into account ramifications of services; determining the feasibility of a composition a priori; and considering exogenous qualifications during the verification process. The framework is implemented using FLUX-based planning, supporting compositions with fundamental control constructs, including nondeterministic ones such as conditionals and loops. Performance is evaluated with regard to termination and execution time for increasingly complex synthetic compositions.

Keywords: service composition, service verification, service specification, frame problem, ramification problem, qualification problem

1 Introduction

Semantic Web Services technologies aim to enable automatic and dynamic interaction between software systems by combining the machine-interpretable features of the Semantic Web and the Internet-accessible interfaces of Web services [16]. The main discerning characteristic of Semantic Web Services involves describing what a Service-Based Application (SBA) actually does (and possibly how) in a way that is machine-interpretable, employing concepts that are modeled using formal and semantically rich representations, such as ontologies.

The incorporation of Semantic Web features in the service world benefits all phases of an SBA lifecycle, including service composition and verification. Service composition encompasses all methods for creating SBAs by employing the engineering principles of reusability and composability, with the aim of creating value-added services that achieve functionality otherwise unattainable by atomic services. Service verification, then, focuses on checking whether a service, atomic or composite, meets some properties or conforms to a given specification.

Both service composition and service verification benefit greatly when services are described using a formal, well-defined semantic specification language,

detailing service behavior in the form of inputs, outputs, preconditions and effects (collectively known as IOPEs), using concepts defined in ontologies. Such a specification language can assist in automatically deducing service composability according to given composition patterns, by detecting inconsistencies among service specifications; it is also indispensable in verification processes, since one cannot determine whether a service satisfies a property if no detailed specification of the service exists.

Service specifications usually include conditions that should hold before and after service execution. This makes them prone to a family of problems, known in the AI literature as the frame, ramification and qualification problems. The frame problem stems from the need to express in a (service) specification not only what is changed, but also what remains unchanged, as a safeguard against inconsistencies and erroneous formal proofs. The ramification problem is directly related, as it concerns itself with the ability to adequately represent and infer information about the knock-on and indirect effects that might accompany the direct effects of a service. Finally, the qualification problem deals with the inability to take into account every circumstance and condition that must be met prior to a service execution, especially in the case of qualifications that are outside the scope of our knowledge and result in observed behavior that is inconsistent with the specification.

In previous work [4], we defined the Web Service Specification Language (WSSL), designed with the explicit purpose of addressing these problems by exploiting existing solutions proposed for the fluent calculus formalism [20]. In this work, we propose a composition and verification framework for services, based on an extended version of WSSL that supports control and data flow specification.

The fundamental innovative feature of the proposed framework is that it relies on WSSL, the only service description language that supports solutions to the aforementioned problems. Two major novel aspects rise from that fact: the composition process takes advantage of complete behavioral specifications of services, which includes taking ramifications into account, while verification considers qualifications when attempting to provide explanations for unexpected observed behavior.

The second but equally important innovative feature is the fact that the framework supports semantics, while at the same time satisfying a series of desirable requirements: the framework offers an *automated* way of producing *dynamic* service compositions that support a multitude of control constructs including *non-deterministic* ones, are *QoS-aware*, even under *incomplete knowledge* of the initial state, while achieving *scalability*. As analyzed in Section 6, to the best of our knowledge, no Semantic Web Service composition and verification framework simultaneously satisfies all aforementioned requirements. The proposed framework is implemented as a FLUX [19] planner that attempts to achieve composition goals based on heuristic encodings of the planning problem, while also answering verification queries, given a planning problem solution. The framework is evaluated in order to investigate effectiveness and scalability issues.

The rest of this paper is organized as follows. Section 2 offers a motivating scenario illustrating the need for a composition framework that exploits the unique features of WSSL, while Sect. 3 provides an overview of the language. Section 4 extends WSSL to support control and data flow specification, followed by a detailed presentation of the proposed framework, which is then evaluated in Sect. 5. Section 6 offers a concise description of the most prominent related work and Sect. 7 concludes and points out topics for future work.

2 Motivating Scenario

In this section, we present an indicative scenario that illustrates the motivation behind employing an expressive specification language, such as WSSL, to facilitate service composition and verification. The scenario is inspired by HelpMeOut, a process for vehicle drivers to get assistance in case of an emergency, presented in [1]. In our scenario, a call center for road assistance can be reached by vehicle drivers in need of assistance in two ways, via call or via SMS. In order to effectively assist the driver, necessary information has to be collected, such as the driver's location and details about the problem encountered. Based on this information, a search for the most suitable repair center is conducted and they are dispatched to the driver's location. After resolving the issue, the payment process follows. Finally, a report is sent to the driver either electronically or through traditional mail, depending on the driver's choice.

Given a service repository containing services that implement the separate tasks described above, we would like to automatically create a composite process that realizes the complete HelpMeOut road assistance scenario, such as the one shown in Fig. 1. The following features are also required:

- take into account ramifications of services when attempting to create a composition schema, e.g. the payment process has the knock-on effect of credit card invalidation if a daily spending limit has been reached.
- verify the composability and correctness of a candidate composite process that realizes the HelpMeOut scenario
- determine the changes in the state of affairs that are brought upon by a successful execution of a HelpMeOut composite service
- determine the results of executing a HelpMeOut composition, even under incomplete information, e.g. without knowing in advance if the user will request electronic or mail delivery for the report.
- determine what went wrong when unexpected results occur, e.g. no report is delivered to the driver, when all preceding tasks were successful.

Supporting all these features requires that services participating in the composition are described using semantically rich specifications that take into account the frame, ramification and qualification problems. These problems have been largely ignored by every Semantic Web Service description language that has been proposed in recent years (SAWSDL [9], OWL-S [12] and WSML [22]) and were the main motivation behind the creation of the Web Service Specification Language (WSSL) [4], described in the following section.

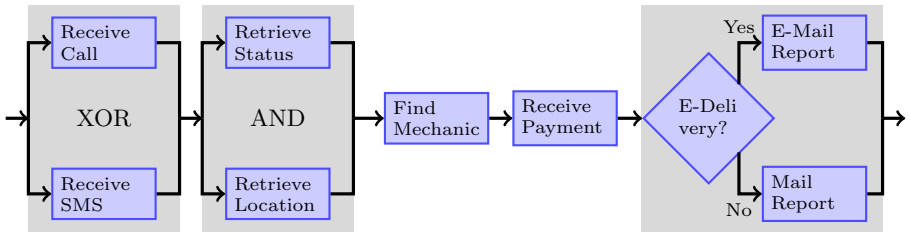

Fig. 1: Composite process of the motivating scenario

3 Web Service Specification Language

In this section, we offer a brief overview of WSSL and its fluent calculus foundations. A detailed definition and analysis of the language syntax and semantics, as well as the problems it addresses, can be found in [4].

3.1 Fluent Calculus basics

The fundamental entity of the fluent calculus is the *fluent*, a single atomic property of the physical world which may change in the course of time. A *state* is a snapshot of the environment at a certain moment. A fluent is equivalent to a state where only this particular fluent holds. An *action* represents high-level actions. Finally, a *situation* is a history of action performances. Predefined function *Do* maps an action to the situation after performing it, while *State* maps situations to equivalent states. A fluent f is said to hold in a state z, if z can be decomposed into two states, one of which is f: $Holds(f, z)$.

While the fluent calculus was conceived for the field of autonomous robotics, its modeling of dynamic environments is perfectly suited for service specifications. Services (atomic or composite) can be represented by actions, while fluents can model the state of the world before and after executing a service operation. The execution of a service and its effects can be described using the fluent calculus definitions that follow. A ***state formula*** $\Delta(z)$ is a first-order formula where z is a free state variable, states occur exclusively in expressions of the form $Holds(f, z)$ and no actions or situations are allowed. A ***situation formula*** is defined accordingly. Given an action $A(x)$[1], a state variable z and a state formula $\Pi_A(z)$ (where z is free by definition and x may also be free), an ***action precondition axiom*** is a formula $Poss(A(x), z) \equiv \Pi_A(z)$, with the semantics that action A is possible at state z, if and only if Π_A is true.

A ***state update axiom*** is a formula $Poss(A(x), s) \rightarrow (\exists y)(\Delta(s) \wedge$ $\wedge State(Do(A(x), s)) = State(s) + \theta^+ - \theta^-)$ stating that, if action A is possible at situation s, executing it results in a successor state derived from $State(s)$ if we add fluents that have been made true (*positive effects* θ^+) and subtract falsified ones (*negative effects* θ^-), under additional conditions $\Delta(s)$ (a situation

[1] Note that actions may also have a vector variable as an argument ($A(\vec{x})$)

formula containing variables x and y). Note that disjunction can be used to express multiple state updates with different $\Delta(s)$ formulas. As analyzed in [18], state update axioms are a provably correct solution to the frame problem.

3.2 Defining WSSL Specifications

Service preconditions and postconditions can be directly represented using action precondition axioms and state update axioms respectively. To represent service inputs and outputs, we introduced two unary predicate symbols corresponding to fluents, namely $HasInput$ and $HasOutput$. $HasInput$ denotes that the associated argument is available to the service as an input while $HasOutput$ denotes that the associated argument is produced as a service output.

Definition 1. An **input formula** in z is a state formula $I(z)$ with free state variable z, which is composed exclusively of $HasInput$ fluents. An **output formula** $O(z)$ is defined accordingly.

Apart from the frame problem, the fluent calculus offers a solution to the problem of representing ramifications as well, using causal relationships that link a ramification to the direct effect (or ramification) that brings it about (see Chap. 9 in [20]). A ***causal relationship*** is formally defined as a formula $(\forall)(\Gamma \rightarrow Causes(z, p, n, z', p', n', s)$ with the semantics that in situation s, under conditions expressed in Γ, the positive and negative effects p and n that have occurred cause an automatic update from state z to z', with effects p' and n'. A ***state update axiom with ramifications*** is a formula $Poss(A(x), s) \rightarrow (\exists y)(\Delta(s) \wedge Ramify(z, \theta^+, \theta^-, z', Do(A(x), s))$, where $Ramify$ stands for applying any matching causal relationships in state z, leading to z'.

Finally, the fluent calculus offers a way to represent qualifications outside the scope of our knowledge, solving the exogenous qualification problem (see Chap. 10 in [20]) by modeling any unforeseen situation that obstructs an action as an *accident*, represented by new predicate $Acc(c, s)$, with the semantics that accident c, which is a variable of new sort $ACCIDENT$, happened in situation s. In order to assume away accidents, default logic rules are employed, such as $\frac{:\neg Acc(c,s)}{\neg Acc(c,s)}$, which is essentially a universal default on the non-occurrence of all accidents. To express the case where no accident has taken place, the conjunct $(\forall c)\neg Acc(c, s)$ is included in the right-hand side of a state update axiom. Any other accident case is expressed as a separate state update using disjunction. Action precondition axioms are rewritten in the form $Poss(A(x), s) \equiv [(\forall c)\neg Acc(c, s) \rightarrow \Pi_A(x, s)]$ meaning that A is possible at s provided that no accidents have happened and the preconditions are true.

Using all of the above, we are able to formally define a WSSL specification:

Definition 2. A **WSSL specification** is a 6-tuple
$\mathcal{S} = \langle \textbf{service}, \textbf{input}, \textbf{output}, \textbf{pre}, \textbf{post}, \textbf{causal}, \textbf{default} \rangle$ where:

- **service**: identifiers offering general information about the service (e.g. service or operation name, invocation information), or the symbol *nil*,

- **input**: the required input of the service, expressed as *input formulas*
- **output**: the expected output of the service, in the form of *output formulas*,
- **pre**: service preconditions, expressed as *action precondition axioms*,
- **post** service postconditions, in the form of *state update axioms*,
- **causal**: causal relationships linking effects and ramifications,
- **default**: default qualifications formalized as default rules.

Table 1 offers a WSSL specification of indicative services for the tasks in the motivating scenario (omitting *Poss* and no-accident clauses). We also defined an XML syntax for WSSL, named WSSL/XML, in order to provide machine readability for WSSL documents and facilitate standard parsing processes. The XML Schema can be found online at www.csd.uoc.gr/˜gmparg/research.html.

Table 1: Example WSSL specifications

Service	**Inputs**
ReceiveSMS/Call	-
RetrieveLocation	$Holds(HasInput(request), ?z\_in)$
RetrieveDiag	$Holds(HasInput(request), ?z\_in)$
FindMech	$Holds(HasInput(status), ?z\_in) \wedge Holds(HasInput(loc), ?z\_in)$
ReceivePay	$Holds(HasInput(payform), ?z\_in) \wedge Holds(HasInput(loc), ?z\_in)$
EReport	$Holds(HasInput(invoice), ?z\_in)$
MReport	$Holds(HasInput(invoice), ?z\_in)$
Service	**Preconditions**
ReceiveSMS/Call	$Holds(CallCenterUp)$
RetrieveLocation	$Holds(GPSActive(user), ?z\_in)$
RetrieveDiag	$Holds(SystemActive(vehicle), ?z\_in)$
FindMech	$Holds(Rcvd(loc, ?user), ?z\_in) \wedge$ $Holds(Rcvd(status, vehicle), ?z\_in) \wedge \neg Holds(Solved(status, loc), ?z\_in)$
ReceivePay	$Holds(HasInput(credCard), ?z\_in) \wedge Holds(Solved(status, loc), ?z\_in)$
EReport	$Holds(PayCompleted(payform), ?z\_in) \wedge$ $Holds(Generated(mechlog), ?z\_in) \wedge \neg Holds(Emailed(report), ?z\_in)$
MReport	$Holds(PayCompleted(payform), ?z\_in) \wedge$ $Holds(Generated(mechlog), ?z\_in) \wedge \neg Holds(Delivered(report), ?z\_in)$
Service	**Outputs and Postconditions**
ReceiveSMS	$?z\_out = ?z\_in + HasOutput(request) + Rcvd(request, sms)$
ReceiveCall	$?z\_out = ?z\_in + HasOutput(request) + Rcvd(request, call)$
RetrieveLocation	$?z\_out = ?z\_in + HasOutput(loc) + Rcvd(loc, user) - HasInput(request)$
RetrieveDiag	$Ramify(?z\_in, HasOutput(status) + Rcvd(status, vehicle),$ $HasInput(request), ?z\_out)$
FindMech	$?z\_out = ?z\_in + HasOutput(payform) + HasOutput(credCard) +$ $Solved(status, loc) - HasInput(status) - HasInput(loc)$
ReceivePay	$?z\_out = ?z\_in + HasOutput(invoice) + PayCompleted(payform)$ $-HasInput(payform)$
EReport	$?z\_out = ?z\_in + HasOutput(report) + Emailed(report) - HasInp(invoice)$
MReport	$(\forall ?c) \neg Acc(?c, ?s)(?z\_out = ?z\_in + HasOutput(report) + Delivered(report)$ $-HasInput(invoice)) \vee (\exists deliv)(Acc(Failure(deliv, s) \wedge ?z\_out = ?z\_in)$
Causal Relationships	
$?p = HasOutput(status) + Rcvd(status, vehicle) \wedge ?n = HasInput(request) \Rightarrow$ $Causes(?z, ?p, ?n, ?z + Generated(mechlog), ?p + Generated(mechlog), ?n, ?s)$	
$DailyLimitReached(payform) \wedge ?p = HasOutput(invoice) + PayCompleted(payform) \Rightarrow$ $Causes(?z, ?p, ?n, ?z + Invalid(credCard), ?p + Invalid(credCard), ?n, ?s)$	

All fundamental entities of WSSL, from fluents to accidents, can be expressed using concepts defined in service ontologies. It is envisioned that existing OWL-S and WSMO descriptions can be ported to WSSL and then annotated to fill up information related to causal relationships and accident modeling, resulting in complete semantic service specifications that take into account the frame, ramification and qualification problems.

4 Composition and Verification of WSSL services

In this section, we present a composition and verification framework for services specified using WSSL. First, we extend WSSL to support control and data flow of compositions, as well as planning and then analyze the composition and verification capabilities separately.

4.1 WSSL for Composition

The definition of WSSL in Sect. 3 allows for black-box specifications of services where only IOPEs are considered, disregarding any knowledge about its control and data flow. In order to be able to employ WSSL for composition, we need to extend it to include the definition of fundamental control constructs.

Definition 3. A tuple $S \cup \langle \epsilon, ;, If, \cdot, +, \oplus, Loop \rangle$ is an extended WSSL signature for composition if S is a WSSL signature and: $\epsilon : ACTION$ (empty action), *If:* $FLUENT \times ACTION \times ACTION \rightarrow ACTION$ (conditional execution), *Loop:* $FLUENT \times ACTION \rightarrow ACTION$ (iterative execution) and $;, \cdot, +, \oplus : ACTION \times ACTION \rightarrow ACTION$ (sequence, AND-Split/AND-Join, OR-Split/OR-Join and XOR-Split/XOR-Join, respectively).

It follows that the foundational axioms that govern the fluent calculus and WSSL, as expressed in [4], need to be extended in order to account for the newly introduced function symbols. The extension is based on the definition and analysis of control constructs conducted in previous work [2].

Definition 4. The foundational axioms for **preconditions** consist of:

1. $Poss(\epsilon, s) \equiv T$
2. $Poss(a_1; a_2, s) \equiv Poss(a_1, s) \wedge Poss(a_2, Do(a_1, s))$
3. $Poss(If(f, a_1, a_2), s) \equiv [Holds(f, s) \wedge Poss(a_1, s)] \vee [\neg Holds(f, s) \wedge Poss(a_2, s)]$
4. $Poss(a_1 \cdot a_2, s) \equiv Poss(a_1 + a_2, s) \equiv Poss(a_1 \oplus a_2, s) \equiv Poss(a_1, s) \wedge Poss(a_2, s)$
5. $Poss(Loop(f, a_1), s) \equiv [Holds(f, s) \Rightarrow Poss(a_1, s)] \wedge$
   $[Holds(f, Do(a_1, s) \Rightarrow Poss(a_1, Do(a_1, s))] \wedge ...$

These foundational axioms allow for calculating preconditions for composite services, based on their composition schema. For instance, the precondition of the *RetrieveStatus/RetrieveLocation* composition can be calculated using axiom 4 of Definition 4 as the conjunction of the preconditions of the two services. Note that this axiom may appear too strong for OR and XOR cases, but it stems directly from the fact that, at design time, we do not know which branch is going to be executed; hence, we cannot disregard either precondition. If such knowledge is available at runtime, then the conditions may be adapted accordingly.

Definition 5. The foundational axioms for **postconditions** consist of:

1. $State(Do(\epsilon, s)) = State(s)$
2. $Poss(a_1; a_2, s) \Rightarrow State(Do(a_1; a_2, s)) = State(Do(a_2, Do(a_1, s)))$

3. $Poss(If(f, a_1, a_2), s) \Rightarrow [Holds(f, s) \wedge State(Do(If(f, a_1, a_2), s)) = State(Do(a_1, s))] \vee \neg[Holds(f, s) \wedge State(Do(If(f, a_1, a_2), s)) = State(Do(a_2, s))]$
4. $Poss(a_1 \cdot a_2, s) \Rightarrow State(Do(a_1 \cdot a_2, s)) = State(Do(a_2, s)) + \theta_1^+ - \theta_1^- = State(s) + \theta_2^+ - \theta_2^- + \theta_1^+ - \theta_1^-$
5. $Poss(a_1 + a_2, s) \Rightarrow [State(Do(a_1 + a_2, s)) = State(s) + \theta_1^+ - \theta_1^-] \vee [State(Do(a_1 + a_2, s)) = State(s) + \theta_2^+ - \theta_2^-]$
6. $Poss(a_1 \oplus a_2, s) \Rightarrow [State(Do(a_1 \oplus a_2, s)) = State(s) + \theta_1^+ - \theta_1^-] \oplus [State(Do(a_1 \oplus a_2, s)) = State(s) + \theta_2^+ - \theta_2^-]$
7. $Poss(Loop(f, a_1), s) \Rightarrow [\neg Holds(f, s) \Rightarrow (State(Do(Loop(f, a_1))), s) = State(s)] \wedge [Holds(f, s) \wedge \neg Holds(f, Do(a_1, s)) \Rightarrow (State(Do(Loop(f, a_1), s))) = State(s) + \theta_1^+ - \theta_1^-] \wedge ...$

These axioms complement the ones in Definition 4, resulting in a whole view of a composite service execution. For instance, by combining the third axioms in Definitions 4 and 5, we can express the fact that the conditional execution of *EReport* and *MReport* requires only one of the two services' preconditions to be true, depending on the truth value of the condition fluent, while a successful execution leads to a state change as a result of *EReport* or *MReport*.

Definitions 4 and 5 can be extended in a straightforward way for compositions of more than two services. As it will be discussed later on, the nature of loops leads to an infinite expression for the associated foundational axioms, which can only be made finite if the number of iterations is known or limited beforehand.

Apart from defining control flow for service composition, WSSL needs to account for data flow as well. The following axiom models the simplest case of routing between outputs and inputs of services:

Definition 6. The foundational axiom for **data flow** expresses the fact that any produced output can potentially be consumed as an input from that state onward and is written as $Holds(HasOutput(f), z) \Rightarrow Holds(HasInput(f), z)$.

QoS-Awareness By definition, any WSSL term can be associated with concepts defined in a knowledge representation model, using IRI [8] sequences. For instance, expressions used in WSSL preconditions can refer to concepts of any origin, including ontology-based QoS models. We are investigating the integration of such a model, OWL-Q [11], with WSSL. OWL-Q is an OWL-S [12] extension that provides a semantic, rich and extensible model for describing QoS aspects, which can be used by service providers to model QoS attributes. Such models can be referenced in either WSSL specifications or queries, realizing in that way QoS-aware service description and composition, respectively.

4.2 Service Composition Planning

Based on planning in the fluent calculus with FLUX (see Chap. 6 in [20]), we define planning for service composition using WSSL.

Definition 7. A WSSL planning problem is defined as the problem of reaching a goal state defined by a state formula $\Gamma(z)$, starting from an initial state defined by a state formula $\Phi(z)$. A WSSL plan is a sequence $\alpha_1, ..., \alpha_n$ of service executions, with $n \geq 0$. The plan is a solution to the problem iff the following holds: $Poss([\alpha_1, ..., \alpha_n], \Phi(z)) \wedge \Gamma\{z/State(Do([\alpha_1, ..., \alpha_n], \Phi(z)))\}$

A planning problem is encoded in FLUX in a sound and complete way using the following two clauses: $P(z,p,z) \Leftarrow Goal(z), p = []$ and $P(z,[a|p],z_n \Leftarrow Poss(a,z), StateUpdate(z,a,z_1,[]), P(z_1,p,z_n)$, stating that if we are at the goal state, the solution is either the empty plan, or a sequence of actions constructed recursively until the goal state is reached. For the motivating scenario, a planning problem encoding is: $AssistPlan(z,p,z) \Leftarrow Holds(Solved(status,location),z)$, $Holds(PayCompleted(payform),z), Holds(HasOutput(report),z), p = []$ and $AssistPlan(z,[a|p],z_n) \Leftarrow Poss(a,z), StateUpdate(z,a,z_1,[]), P(z_1,p,z_n)$.

Encodings based on Definition 7 have two major drawbacks: they do not take into account the issues of termination and computational complexity and can produce only sequential compositions. The first step towards handling both issues is introducing heuristics, further specifying the planning encoding.

Definition 8. A heuristic encoding of a WSSL planning problem is defined as a FLUX program P_{plan} defining a predicate $P(z,p,z_n)$ that describes the problem of reaching a goal state $\Gamma(z)$, starting from an initial state $\Phi(z)$. The encoding is sound iff the following holds: for every computed answer θ to the FLUX query $\Leftarrow \Phi(z) \wedge P(z,p,z_n)$, $p\theta$ is a solution to the planning problem and $Poss(p\theta, \Phi(z)) \wedge \Gamma\{z/State(Do(p\theta,\Phi(z)))\}$.

In order to consider plans more complex than sequences of services, heuristic encodings need to include control construct definitions. Based on Definitions 4, 5 and 6, we extend the FLUX Prolog kernel with clauses that support fundamental control constructs and data flow between inputs and outputs. For instance, plans that contain looped executions are considered based on the following rules:

```
poss_loop(F,K,A,Z) :- K\==0, (holds(F,Z)->poss(A,Z)),update(Z,A,Z_PR),poss_loop(F,K-1,A,Z_PR).
state_update_loop(Z,F,K,A,Z_PR) :- not_holds(F,Z)->Z_PR=Z ; K\==0,(holds(F,Z),update(Z,A,Z_1),
           not_holds(F,Z_1)) -> state_update(Z,A,Z_1), state_update_loop(Z_1,F,K-1,A,Z_PR).
```

The iterative nature of loops is expressed using Prolog rules that refer to themselves. As already mentioned, it is necessary to impose an upper bound K on the number of iterations, to avoid non-terminating executions. The complete extended kernel can be found online at www.csd.uoc.gr/~gmparg/research.html. One possible heuristic encoding for the problem of the motivating scenario is:

```
assist_plan(Z,[A|P],Z_PR) :- A1=receivesms, A2=receivecall, A=xor(A1,A2),
     poss_xor(A1,A2,Z), state_update_xor(Z,A1,A2, Z_1), assist_plan1(Z_1,P,Z_PR).
assist_plan1(Z,[A|P],Z_PR) :- A1=retrievelocation, A2=retrievediagnostics,
     A=and(A1, A2), poss_and(A1,A2,Z),state_update_and(Z,A1,A2,Z_1),assist_plan2(Z_1,P,Z_PR).
assist_plan2(Z,[A|P],Z_PR) :- A=findmech, poss(A,Z),
                              state_update(Z,A,Z_1), assist_plan3(Z_1,P,Z_PR).
assist_plan3(Z,[A|P],Z_PR) :- A=receivepay, poss(A,Z),
                              state_update(Z,A,Z_1), assist_plan4(Z_1,P,Z_PR).
assist_plan4(Z,A,Z_PR) :- F=req_deliv, A1=ereport,A2=mreport, A=if(F,A1,A2),
                              poss_if(F,A1,A2,Z), state_update_if(Z,F,A1,A2,Z_PR).
```

Executing the FLUX query `assist_plan([callcenterup, gpsactive(user1), systemactive(vehicle1), req_deliv], P, Z_PR).` will yield a plan corresponding to the composite process of Fig. 1. FLUX is also able to handle incomplete initial states thanks to embedded constraint handling rules. For instance, we may exclude `req_deliv` (which corresponds to the user wanting the report e-mailed to him) from the definition of the initial state and the planner will still produce plans that assume either case, with or without that fluent.

4.3 Verification

Given a generated plan that solves a planning problem, service verification aims to check that the composite service that corresponds to the plan meets some properties. The verification process in our framework focuses mainly on answering questions about the behavior of the composition. Examples of the properties that can be verified are the following:

- *Composability* of a set of services: given a composition goal, the nature of the composition process yields results about whether this particular set can lead to a valid composition.
- *Liveness* and *safety* properties that check whether the composition plan realizes the desired behavior.
- *Conformance* of an observed composite service behavior to the corresponding plan specification and, in case conformance fails, possible *explanations* for the conflict in order to perform troubleshooting actions.

For example, a liveness property in our motivating scenario would be to verify whether the composition plan leads to the final report being delivered (either by mail or electronically), by proving $Emailed(report, z) \vee Delivered(report, z)$, where z is the final state, while a safety property would be to make sure that payment is performed for the correct payment form ($HasInput(payform, z_{in}) \wedge \neg PayCompleted(payform2, z)$). Verification queries of the third type can be answered due to WSSL's solution to the qualification problem. For instance, after executing the composite process of Fig. 1, we observe its behavior in the form of WSSL state descriptions and pose the query: in the final state z, is the goal `holds(solved(status, location), z), holds(paycompleted(form), z), holds(hasoutput(report), z)` satisfied? If the answer is no and no accidental qualifications have been expressed, then the observed behavior is deemed inconsistent with the composition specification. However, given the specification shown in Table 1, the framework deduces that an accident has occurred, namely `failure(deliv)`. Such explanations are valuable for determining follow-up actions to unexpected situations, such as re-executing services that failed or adapting the composition in order to replace them.

4.4 Complexity, Decidability and Planning Efficiency

As discussed in [4], decidability results for WSSL are directly related to its foundations in the fluent calculus, as well as default logic. Decidability is guaranteed for the fragment that is equivalent to the two-variable situation calculus with counting, and for default theories without prerequisites. Such results also hold for the extended version that supports control and data flow, provided that an upper bound on iterations is imposed for all loops.

In general, the planning problem is considered undecidable. Even in the case of decidable planning problems, the complexity of finding a solution is directly analogous to the number of choice points, since each choice point splits the search

tree into two branches. Heuristic encodings based on problem-specific knowledge can improve efficiency by trimming parts of the search tree that do not conform to the heuristics definition. However, even then scalability is not guaranteed, because increasing the problem size linearly may lead to a non-linear increase in the number of choice points. In Sect. 5, we investigate scalability issues for our framework, while a complexity and decidability analysis for heuristics-based WSSL planning is part of ongoing research we are currently conducting.

For a given planning problem, there may be more than one solution. For instance, in the motivating scenario, the goal of delivering the report can be satisfied by two different tasks, *EReport* and *MReport*. The planner can either return the first solution found, or generate all possible solutions and rank them according to a ranking function. FLUX supports ranking based on cost.

4.5 Other Features

The fluent calculus foundations of WSSL imbue our framework with several other interesting features. First of all, the fluent calculus has been extended to support knowledge (or belief) states (see Chap. 5 in [20]), a generalization of expressing incomplete initial states (supported by our framework) in the direction of defining which states are possible at a given situation, based on how complete our knowledge is. Knowledge states, combined with non-determinism and qualifications, allow for comprehensive modeling of partially observable behavior.

An especially desirable feature in the case of services is supporting asynchronous execution, i.e. services that do not wait for a response after being invoked. Asynchronous services can be easily modeled as a pair of distinct WSSL services, similarly to the invoke/receive combination of WS-BPEL [13], thanks to the definition of states in WSSL: the first service has no postconditions, since it simply invokes the operation, while the second has no preconditions, since they have already been checked on invocation. Thus, the state after invoking the service is decoupled from the state after receiving the reply.

Finally, WSSL compositions can be translated to executable processes based on the included definition of control and data flow. For instance, plans comprising of WSSL services that do not contain invocation information can be translated to abstract BPEL processes, which can then be concretized via suitable service discovery mechanisms. If WSSL specifications are linked to specific service endpoints, then they directly lead to executable BPEL processes.

5 Experimental Evaluation

In order to evaluate the proposed composition framework, we run a series of experiments, calculating the time needed for the planner to produce a valid service composition plan, given a set of services, an initial state and a goal state. To investigate scalability, we varied the complexity of the planning problem in two ways: by increasing the size of the service repository and by allowing for more elaborate plans. The service specifications are synthetically generated in FLUX,

each one consisting of one or two sets of IOPEs. The computation time values are an average of 10 runs in the ECL^iPS^e constraint programming system. The evaluation was performed on an Intel® Core™ i7-740QM processor running at 1.73GHz, with 6 GB RAM.

In the first two experiments, we examined the scalability of our approach given simple composition schemas, considering only sequential composition in the first and only parallel composition (AND-Split/Join) in the second. In both cases, we increased linearly the number of services that need to be considered in order to find an executable one at a given state. As shown in Fig. 2, even for a repository of 1000 services, computation time is around 1 second, which is rather efficient considering the fact that choice points increase from 1 in the initial state to 1000 in the penultimate state.

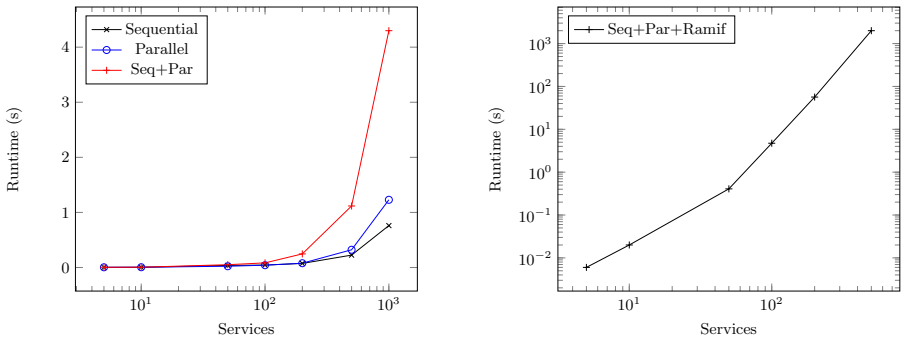

Fig. 2: Scalability results for compositions of varying complexity

For the third experiment, we combine the first two, creating a composition schema of alternating sequential and parallel executions. Fig. 2 shows that there is a reasonable increase to around 4 seconds for the case of 1000 services. Note that other composition schemas such as OR and XOR Split/Join, conditionals and loops are not included in our experiments, since their use is expected to be dictated explicitly in heuristic encodings, as in the motivating scenario; in our test sets, AND Split/Join schemas are always the first to evaluate to true.

In the final experiment, we investigate the effect of including ramifications in service specifications. Each postcondition in our test set is associated with a ramification through the inclusion of a causal rule (a 50% increase in the size of specifications). Once again, we increase linearly the number of causal rules that are considered until a matching one is found. As we can see in Fig. 2 on the right, there is a significant increase in computation time, up to 2000 seconds for 500 services. These results are expected, considering that the planner has to examine around 500 causal rules one by one at the final states of the composition. Note that real-world problems are expected to be much simpler than the ones generated synthetically for our evaluation, since only some postconditions are usually associated with ramifications.

6 Related Work

Employing fluent calculus in order to solve the service composition problem has been attempted in [7] and [6]. In both approaches, Semantic Web Services are translated to fluent calculus theories and forward chaining composition algorithms implemented using FLUX are proposed, with [6] considering QoS thresholds as well. However, neither takes into account the fluent calculus extensions that solve the ramification and qualification problems, resulting in frameworks that ignore their effects and fail to capitalize on the benefits of their solutions, such as expressing ramifications or explaining non-conforming behavior. Moreover, both planners produce at best sequences of parallel executions, disregarding any other control constructs, such as the ones supported by our framework. Finally, their choice to model inputs and outputs using the $KnowsVal$ macro is invalid since, by definition, it was introduced to represent a subset of fluent-related variables that is true in all possible states, and not the semantics of a service input, which is a fluent that holds at the state before service execution.

Extensive literature exists that employs AI planning techniques to realize automated service composition. Klusch and Gerber [10] propose OWLS-XPlan, a framework that combines the benefits of graph-based planning with HTN planning for service composition, while also employing re-planning techniques to adjust outdated plans during execution time. However, the framework does not support non-deterministic control constructs or planning with incomplete state descriptions, while the authors do not consider QoS-awareness or the ramification and qualification problems. The work of Peer [14] translates semantic Web services to PDDL descriptions which are then fed to a VHPOP planner framework that supports re-planning as well as non-determinism in the sense of considering failure in service executions. Looped execution, ramifications, QoS-awareness and incomplete states are again not supported.

Arguably the most prominent realization of service composition using planning techniques is the WS-SYNT mechanism included in the ASTRO framework. As analyzed in [5], WS-BPEL processes are translated into state transition systems (STSs), which are then combined to construct an STS that represents all possible behaviors and afterwards this STS is searched in order to find subsystems of it that satisfy the composition goal. The solutions are then translated back to executable WS-BPEL processes. This work realizes two features, namely support for asynchronous services and translation to executable processes, that are currently not supported by our framework but can be easily integrated, as indicated in Section 4.5. Unlike our framework, however, WS-SYNT does not support Semantic Web services (although one of the earliest works [21] in the authors' research line did), is not QoS-aware and, more importantly, does not consider the frame, ramification and qualification problems and their effects.

[17] and [15] share similar logic foundations with our approach. [15] is based on GOLOG, a logic programming language based on the situation calculus in the same way that FLUX is based on the fluent calculus. The authors integrate user preferences in a GOLOG-based planner, modeling them using a first-order logic language. Employing preferences drastically limits the search space for

the planner, resulting in significantly less computation time. [17] proposes the representation of service I/O schemas and behavioral constraints as Horn clauses and realizes service composition through logical inference as well as structural analysis of Petri nets that model the Horn clause set and the goal. The resulting compositions, however, are restricted to sequences of parallel executions, while behavioral constraints in both approaches ignore knock-on or indirect effects and accidental qualifications, which are supported in our framework.

7 Conclusions and Future Work

In this paper, we proposed a service composition and verification framework using WSSL, a novel semantics-aware service specification language based on the fluent calculus. The framework satisfies significant requirements such as automation, dynamicity, nondeterminism and incomplete state knowledge, in addition to supporting semantics and exploiting WSSL's solutions to the frame, ramification and qualification problems. Experimental evaluation shows that efficiency is achieved even in the presence of ramifications. The framework is an effective demonstration of the benefits of rich semantic behavior specifications in the context of service science and an indicative example of how such benefits can be reaped for the purposes of service verification and composition.

Future work includes further concretizing the link between WSSL and OWL-Q [11], exploring QoS-aware matchmaking and selection mechanisms and identifying ways to improve efficiency by limiting the search space before the planning process, as well as exploring graph-based rule optimization. Additionally, we plan to build upon the discussion in Sect. 4.5 in order to support knowledge states, asynchronous service interactions and derivation of executable composite processes. Moreover, we intend to integrate the ideas of formal behavior specifications in the lifecycle of Cloud-based services deployed on multiple Cloud providers, as initially explored in [3].

Acknowledgments. The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 317715 (PaaSage), as well as the Special Account for Research Grants, University of Crete (Grant No. 3742).

References

1. Ali, S.A., Roop, P.S., Warren, I., Bhatti, Z.E.: Unified Management of Control Flow and Data Mismatches in Web Service Composition. In: Gao, J.Z., Lu, X., Younas, M., Zhu, H. (eds.) SOSE. pp. 93–101. IEEE (2011)
2. Baryannis, G., Carro, M., Plexousakis, D.: Deriving Specifications for Composite Web Services. In: COMPSAC. pp. 432–437 (2012)
3. Baryannis, G., Garefalakis, P., Kritikos, K., Magoutis, K., Papaioannou, A., Plexousakis, D., Zeginis, C.: Lifecycle Management of Service-based Applications on Multi-Clouds: A Research Roadmap. In: Proceedings of the international workshop on Multi-cloud applications and federated clouds. pp. 13–20. ACM (2013)

4. Baryannis, G., Plexousakis, D.: WSSL: A Fluent Calculus-Based Language for Web Service Specifications. In: Salinesi, C., Norrie, M.C., Pastor, O. (eds.) Advanced Information Systems Engineering, LNCS, vol. 7908, pp. 256–271. Springer Berlin Heidelberg (2013)
5. Bertoli, P., Pistore, M., Traverso, P.: Automated composition of Web services via planning in asynchronous domains. Artif. Intell. 174(3-4), 316–361 (2010)
6. Bhuvaneswari, A., Karpagam, G.R.: Applying Fluent Calculus for Automated and Dynamic Semantic Web Service Composition. In: Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications (2010)
7. Chifu, V.R., Salomie, I., Harsa, I., Gherga, M.: Semantic Web Service Composition Method Based on Fluent Calculus. In: Watt, S.M., Negru, V., Ida, T., Jebelean, T., Petcu, D. (eds.) SYNASC. pp. 325–332. IEEE Computer Society (2009)
8. Duerst, M., Suignard, M.: Internationalized Resource Identifiers (IRIs). RFC 3987 (2005)
9. Farrell, J., Lausen, H.: Semantic Annotations for WSDL and XML Schema. World Wide Web Consortium, Recommendation REC-sawsdl-20070828 (August 2007)
10. Klusch, M., Gerber, A.: Semantic Web Service Composition Planning with OWLS-Xplan. In: Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web. pp. 55–62 (2005)
11. Kritikos, K., Plexousakis, D.: Requirements for QoS-based web service description and discovery. IEEE T. Services Computing 2(4), 320–337 (2009)
12. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., Sycara, K.: OWL-S: Semantic Markup for Web Services (2004)
13. OASIS: Web Services Business Process Execution Language Version 2.0. Specification (April 2007)
14. Peer, J.: A POP-Based Replanning Agent for Automatic Web Service Composition. In: Gómez-Pérez, A., Euzenat, J. (eds.) The Semantic Web: Research and Applications, LNCS, vol. 3532, pp. 47–61. Springer Berlin Heidelberg (2005)
15. Sohrabi, S., Prokoshyna, N., McIlraith, S.A.: Web Service Composition via the Customization of Golog Programs with User Preferences. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications, LNCS, vol. 5600, pp. 319–334. Springer Berlin Heidelberg (2009)
16. Studer, R., Grimm, S., Abecker, A. (eds.): Semantic Web Services. Springer, Berlin (2007), http://www.springerlink.com/content/kj5458/
17. Tang, X., Jiang, C., Zhou, M.: Automatic Web service composition based on Horn clauses and Petri nets. Expert Syst. Appl. 38(10), 13024–13031 (2011)
18. Thielscher, M.: The Fluent Calculus. Tech. Rep. CL-2000-01, Dresden University of Technology (2000)
19. Thielscher, M.: FLUX: A Logic Programming Method for Reasoning Agents. CoRR cs.AI/0408044 (2004)
20. Thielscher, M.: Reasoning Robots, Applied Logic Series, vol. 33. Springer Netherlands (2005)
21. Traverso, P., Pistore, M.: Automated Composition of Semantic Web Services into Executable Processes. In: McIlraith, S., Plexousakis, D., Harmelen, F. (eds.) The Semantic Web ISWC 2004, LNCS, vol. 3298, pp. 380–394. Springer Berlin Heidelberg (2004)
22. WSML Working Group: The Web Service Modeling Language WSML (2008), http://www.wsmo.org/wsml/wsml-syntax