WSSL: A Fluent Calculus-based Language for Web Service Specifications

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. In order to effectively discover and invoke a Web service, the provider must supply a complete specification of its behavior, with regard to its inputs and outputs, preconditions and effects. Devising such complete specifications comes with many issues that have not been adequately addressed by current service description efforts, such as WSDL, SAWSDL, OWL-S and WSMO. These issues involve the frame, ramification and qualification problems, which deal with the succinct and flexible representation of non-effects, indirect effects and preconditions, respectively. We propose WSSL, a novel specification language for services, based on the fluent calculus, that is expressly designed to address the aforementioned issues. Also, a tool is implemented that translates WSSL specifications to FLUX programs and allows for service validation based on user-defined goals.

Keywords: service specification, frame problem, ramification problem, qualification problem, service validation

1 Introduction

Service description deals with specifying all the information necessary in order to access and use a service. The description should be rich, containing both functional and non-functional aspects while it may also contain information on the internal processes of the service, depending on whether the service provider or owner decides to expose such information. Service descriptions should be written in a formal, well-defined specification language that allows for automated processing and validation of the documents produced.

Formal specifications are indispensable in engineering systems based on Service Oriented Architecture (SOA) since they can be used as a basis to construct a service or to check that an existing service meets a set of requirements. Furthermore, they can assist in auditing processes that check third party or legacy code conformance to specifications, while also playing a major role in validation and verification techniques, as well as in the evaluation of the results of service adaptation or service evolution. Specifications are also beneficial in service composition, as they can assist in deducing the composability of a set of services, by detecting inconsistencies among their specifications.

Service specifications rely on the expression of conditions that should hold before and after service execution. Such specifications are prone to a family of problems, known in the AI literature as the frame, ramification and qualification problems. While research in other fields such as programming specifications or reasoning about action and change has come up with satisfying solutions to these problems, research in Web services has largely ignored them, at the same time ignoring their effects.

To address the aforementioned issues in service description, we propose the Web Service Specification Language (WSSL), designed with the explicit purpose of describing Web service behavior by means of complete specifications. WSSL's foundation is the fluent calculus [23], a specification language and system for robots that offers solutions to the frame, ramification and qualification problems, while resource and data representation is inspired by the Web Service Modeling Language (WSML) [24], a recent large-scale effort in service description.

It is important to stress the fact that WSSL is independent of service design models. The language design was driven by traditional SOAP-based Web services as well as Semantic Web services, but WSSL specifications can describe the behavior of any service-based system or application, including RESTful Web services [19], and could be exposed in a straightforward way as Linked Services [18].

To demonstrate some initial capabilities that can be achieved through service specifications, a WSSL validation tool prototype was implemented. This tool translates WSSL specifications to FLUX [22] programs and allows user-defined goals to be validated against the specifications. Evaluation of the prototype indicates promising results with respect to scalability.

The rest of this paper is organized as follows. Section 2 offers a short description of the frame, ramification and qualification problems as well as a motivating example illustrating their effects in service specification. Section 3 provides an overview of the fluent calculus and a detailed definition of WSSL in terms of its syntax and semantics. Section 4 extends WSSL in order to support solutions to the ramification and qualification problems and also discusses complexity, decidability and applicability issues. Section 5 analyzes the implementation and evaluation of the WSSL validation tool. Section 6 offers a brief description of related work and Section 7 concludes and points out topics for future work.

2 Motivation

In this section, we present a motivating example to illustrate the need for a service specification language that addresses the frame, ramification and qualification problems. The example is based on the E-Government case study of the European Network of Excellence S-Cube [7]. In this case study, citizens submit applications to request some government-related service, such as obtaining government-issued documents, following the process in Fig. 1. Users log into the system and fill in forms regarding their request as well as payment details, which are then simultaneously processed before the payment process can begin. Certification of the final documents is optional and depends on user preferences.

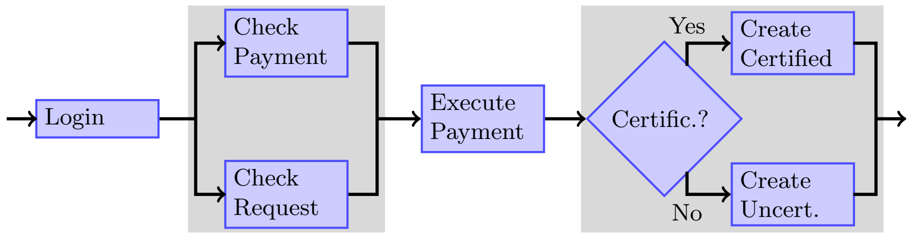

Fig. 1. Composite process of the motivating example

For our purposes, we can assume that the individual tasks described above are implemented as Web services. A possible first-order logic specification of these services, in terms of their inputs, outputs, preconditions and postconditions is offered in Table 1.

Table 1. Service specifications for the motivating example

Service	**Inputs**	**Preconditions**
Login	$loginForm, user$	$Valid(loginForm, user) \wedge \neg LoggedIn(user)$
CheckRequest	$request, user$	$FilledIn(request) \wedge LoggedIn(user)$
CheckPayment	$payForm, user$	$FilledIn(payForm) \wedge LoggedIn(user)$
ExecPayment	$transaction$	$Approved(transaction, user) \wedge Valid(cCrd, user)$
CreateCertified	$payCConf$	$PayCompleted(doc, user)$
CreateUncertified	$payCConf$	$PayCompleted(doc, user)$

Service	**Outputs**	**Postconditions**
Login	$loginConf$	$LoggedIn(user)$
CheckRequest	$requestConf$	$Valid(request, user)$
CheckPayment	$payFCnf$	$Valid(payForm, user)$
ExecPayment	$payCCnf$	$\exists doc \cdot PayCompleted(doc, user)$
CreateCertified	$certifDoc$	$CertifCompleted(doc, user) \wedge$ $\exists certifDoc \cdot Delivered(certifDoc)$
CreateUncertified	doc	$Delivered(doc)$

2.1 The Frame Problem

As it was identified in [5], formal specifications that employ the pre- and postcondition notation are prone to a family of problems related to the frame problem, a long-standing issue in the field of Artificial Intelligence. The frame problem stems from the fact that including clauses that state only what is changed when preparing formal specifications is inadequate since it may lead to inconsistencies and compromise the capacity of formally proving properties of specifications.

For example, the specification of the $CheckPayment$ service includes postcondition $Valid(payForm, user)$, linking a successful execution to attributing validity to the specific payment form/user combination. Based on such a specification, we cannot answer definitively whether the execution of $CheckPayment$

affects the state of the world in any other way, e.g. attributing validity to any other payment form belonging to some other user. Similarly, it is not provable whether the postcondition $PayCompleted(doc, user)$ affects any other pair of user and document variables. Failure to address the frame problem in this example compromises any validation checks, such as answering which accounts have been affected after a successful execution of the payment process.

2.2 The Ramification Problem

The frame problem is closely related to the ramification problem, defined in [17] as the problem of adequately representing and inferring information about the knock-on and indirect effects (also known as ramifications) that might accompany the direct effects of an action. The relation of the frame and ramification problems is somewhat contradicting: an overly restrictive solution to the frame problem precludes any solution to the ramification problem.

Suppose that we want to express the fact that a side effect of postcondition $PayCompleted(doc, user)$ of the $ExecPayment$ service is the invalidation of the user's credit card under certain circumstances, e.g. when a daily spending limit has been reached. In order to do that we need a representation that not only takes ramifications into account, but also achieves that by staying consistent with the way the frame problem is solved.

Ramifications are particularly important in service compositions, as they allow the expression of relationships between effects of services participating in a composition. By making sure that these relationships are taken into account, inconsistencies between parts of a composition may be avoided. For instance, suppose that the following ramification holds for the motivating example: effects $Valid(request, user)$ and $Valid(payForm, user)$, have the knock-on effect expressed by $Approved(transaction, user)$. This ramification essentially expresses the composability of $CheckRequest/CheckPayment$ and $ExecutePayment$ and can be checked to ensure the validity of the composition of Fig. 1 before attempting an execution.

2.3 The Qualification Problem

While the ramification problem deals with the effects of an action, the qualification problem deals with the circumstances and conditions that must be met prior to the execution of an action. In the case of services, we deal with the so-called exogenous qualification problem, defined in [17] as the problem of dealing with qualifications that are outside the scope of our knowledge and result in contradicting some effects of an action due to inconsistencies.

In our motivating example, suppose that we know that both preconditions of $ExecPayment$ are true before execution. Thus, we should expect that after a successful execution, the postcondition $PayCompleted(doc, user)$ holds. If, however, due to some unforeseen circumstance, we find out that it doesn't hold, then we result in an inconsistency between the specification and the actual observed behavior. To address this, service specifications need to account for qualifications that are outside the scope of our knowledge in order to consistently model all possible behaviors of the service.

3 WSSL Syntax and Semantics

WSSL is a novel language for Web service specifications that aims to address the issues of the frame, ramification and qualification problems, by applying the solutions proposed in the definition of the fluent calculus [23] and its follow-up extensions. Employing a formalism designed for the domain of Reasoning about Action and Change (RAC) allows us to specify the effects of a service execution more accurately, by describing the state of the world before and after an execution. The rationale behind choosing the fluent calculus over other RAC formalisms such as the situation and event calculi is detailed in Sect. 6.

3.1 The Fluent Calculus

Before analyzing the syntax and semantics of WSSL, it is important to provide an overview of the fluent calculus [23]. 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. In the initial calculus definition as a specification language for robots, this change is the result of manipulation by the robot. In our case, a fluent changes value as a result of a service execution. 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. There is also the notion of the *empty state*, denoted by $\varnothing$ and of *state composition* (combining states to form a new one), denoted by the function $\circ$. An *action* represents high-level actions, i.e. executions of service operations. Finally, a *situation* is a history of action performances. The initial situation, where no actions have been initiated, is denoted by S_0.

A fluent f is said to hold in a state z, if z can be decomposed into two states, one of which is f. The macro $Holds(f, z)$ is introduced for notational convenience: $Holds(f, z) \overset{def}{=} (\exists z') \cdot z = z' \circ f$. $Holds$ formulas can be combined to create more complex state description formulas. Such a first-order formula $\Delta(z)$ is called a *state formula*, if z is the only free state variable, with states occurring exclusively in $Holds$ formulas, without any actions or situations.

In order to formalize preconditions, a predefined predicate named *Poss* is introduced. Given an action $A(x)$, a situation s and a situation formula $\Pi_A(s)$ with free variables among x, s, an *action precondition axiom* is: $Poss(A(x), s) \equiv \Pi_A(x, s)$ with the semantics that action A is possible at situation s if and only if Π_A is true. There are also two predefined functions: *Do* maps an action and a situation to the situation after performing the action, while *State* maps a situation to the state of the environment in that situation. States are governed by a set of foundational axioms, detailed in [23].

Solving the Frame Problem. The fluent calculus bases its solution to both the representational and inferential aspects of the frame problem on the notion of states. Change is modeled as the difference between two states. Actions are deterministic and result in a bounded number of direct effects and can be positive or negative, with fluents added to or removed from a state, respectively. Postconditions are expressed as *state update axioms* in the following form: $Poss(A(x), s) \to (\exists y)(\Delta(s) \wedge State(Do(A(x), s)) = State(s) + \theta^{+} - \theta^{-})$, with the following semantics: provided that an action A is possible at a situation s, then executing A at s results in a successor state which is defined by the previous state ($State(s)$) if we add fluents that have been made true (*positive effects* θ^{+}) and we subtract fluents that have been made false (*negative effects* θ^{-}), under possible additional conditions expressed by formula $\Delta(s)$. Under the assumption that θ^{+} and θ^{-} are disjoint, state update axioms are a provably correct solution to the representational aspect of the frame problem (see Theorem 7 in [21]).

Representing Inputs and Outputs. While service preconditions and postconditions are directly associated with action precondition axioms and state update axioms of the fluent calculus, the translation is not as direct in the case of inputs and outputs. We propose to represent the properties of having an input or producing an output as fluents. To that end, we define two reserved unary function symbols corresponding to fluents, namely $HasInput$ and $HasOutput$. $HasInput$ denotes that the associated variable is available to the service as an input while $HasOutput$ denotes that the associated variable is produced as a service output. We define an *input formula* in z as a state formula $I(z)$ with just one free state variable z, which is composed exclusively of $HasInput$ fluents. An *output formula* $O(z)$ is defined equivalently.

3.2 WSSL Abstract Syntax

A *WSSL specification* is a 5-tuple $\mathcal{S} = \langle \mathbf{service}, \mathbf{input}, \mathbf{output}, \mathbf{pre}, \mathbf{post} \rangle$ where:

- **service** is a set of identifiers offering general information about the service (e.g. service or operation name, invocation information),
- **input** is a set of WSSL logical expressions defining *input formulas*,
- **output** is a set of WSSL logical expressions defining *output formulas*,
- **pre** is a set of WSSL logical expressions defining *action precondition axioms* that describe the service preconditions,
- **post** is a set of WSSL logical expressions defining *state update axioms* that represent the service postconditions.

Each of the input, output, precondition and postcondition logical expressions is associated with an IRI [8] that acts as an identifier. WSSL identifiers can be any Unicode character sequence, provided that it represents a valid and absolute IRI. Preconditions and postconditions of a service are not limited to functional conditions but can represent the full spectrum of conditions that define a complete behavior specification of a service. It goes without saying that this includes specification of non-functional conditions. Since WSSL supports IRI sequences, any ontology-based QoS models can be employed, such as OWL-Q [13].

Apart from IRIs, data can also be expressed in WSSL. As in WSML, data values can either be *elementary*, corresponding to the three primitive XML Schema datatypes (integer, decimal or string) or *constructed*, created using a *datatype wrapper*, which consists of an IRI corresponding to any non-primitive datatype and a set of arguments, which can be elementary data values or *variables*. WSSL variables must start with a question mark (?) to identify them from other symbol sequences, especially in logical expressions.

WSSL Logical Expressions. A WSSL logical expression is defined using a first-order fragment of the fluent calculus, as defined in [23]. The alphabet consists of the following sets of symbols:

- A countable set $\boldsymbol{S}$ of *sorts*. $S = FLUENT, STATE, ACTION, SIT, BOOL$ with $FLUENT < STATE$. The sorts represent the four fundamental entities of the fluent calculus, as well as boolean values.
- *Logical connectives*: $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), $\equiv$ (equivalence), $\top$ (true), $\perp$ (false).
- *Quantifiers and Equality*: For every sort $s \in S$, $\forall_s$ (for all), $\exists_s$ (exists), $=_s$.
- *Variables*: For every sort $s \in S$, a countably infinite set of variables $\boldsymbol{V_s}$. The family of sets V_s is denoted by $\boldsymbol{V}$.
- *Nonlogical symbols*: A set $\boldsymbol{L}$ consisting of a countable, nonempty set $\boldsymbol{FS}$ of *function symbols* containing at least Do, $State$, $HasInput$ and $HasOutput$, a countable, nonempty set of *constants* $\boldsymbol{C}$, containing at least the empty state $\varnothing$ and the initial situation S_0 and a countable, nonempty set $\boldsymbol{PS}$ of *predicate symbols*, containing at least $Poss$.

Note that the explicit mention of the sort in symbols such as existential and universal quantifiers can be omitted if the particular sort can be otherwise derived. All elements of sets V, FS, PS are represented by IRIs. Finally, the macro $Holds$, as well as adding and subtracting fluents from states can also be used in WSSL logical expressions. Terms and atomic formulae are defined as in many-sorted first-order logic. Table 2 offers a WSSL specification of the payment services in the motivating example (with the poss clauses omitted and with $?z\_in = State(?s\_in)$ and $?z\_out = State(Do(A(?x), ?s\_in)$ for each service).

3.3 WSSL Semantics

The semantics for WSSL is defined based on the standard model theory for classical first-order logic, augmented by the semantics for IRIs as defined in WSML [24]. The main additional aspect brought on by the need to interpret IRIs is the notion of abstract and concrete domains. An abstract domain gives us flexibility in interpreting IRIs as any kind of abstract object, while a concrete domain allows for the interpretation of elementary data values. A concrete domain needs to support all three elementary data types used by WSSL (integer,

Table 2. Example WSSL specifications

Service	**Inputs**
CheckPayment	$Holds(HasInput(?payForm), ?z\_in) \wedge$ $Holds(HasInput(?user), ?z\_in)$
ExecPayment	$Holds(HasInput(?payForm), ?z\_in)$

Service	**Preconditions**
CheckPayment	$Holds(LoggedIn(?user), ?z\_in)$
ExecPayment	$Holds(Valid(?payForm, ?user), ?z\_in) \wedge$ $Holds(LoggedIn(?user), ?z\_in) \wedge Holds(Valid(?cCrd, ?user), ?z\_in)$

Service	**Outputs and Postconditions**
CheckPayment	$?z\_out = ?z\_in + HasOutput(?payFCnf) + Valid(?payForm, ?user)$
ExecPayment	$\exists ?doc \cdot ?z\_out = ?z\_in + HasOutput(?payCCnf) +$ $+ Valid(?payForm, ?user) + PayCompleted(?doc, ?user)$

decimal and string). For instance, it can be equal to the union of the sets of integer numbers, finite-length sequences of decimal digits (preceded or not by the minus symbol) and finite-length sequences of Unicode characters.

A *WSSL interpretation* is a 6-tuple $\mathcal{I} = \langle U, D, \mathcal{I}_{\mathcal{C}}, \mathcal{I}_{\mathcal{F}}, \mathcal{I}_{\mathcal{P}}, B \rangle$ where:

- U is the abstract domain of interpretation, a non-empty countable set used to interpret IRIs.
- D is the concrete domain of interpretation, a non-empty set disjoint from U, used to interpret elementary data values.
- $\mathcal{I}_{\mathcal{C}}, I_F, I_P$ are mappings from individual constants, function symbols and predicate symbols, respectively, to elements of U and D.
- B is an assignment from a variable to an element of $U \cup D$

The interpretation of constants depends on whether the constant is an IRI or an elementary data value. In the former case, $\mathcal{I}_{\mathcal{C}}(c) = u \in U$, while in the latter case $\mathcal{I}_{\mathcal{C}}(c) = d \in D$. A similar distinction applies to function symbols. If the function symbol represents a datatype wrapper function (that creates a constructed data value), it is interpreted as a function over the concrete domain: $\mathcal{I}_{\mathcal{F}}(f)^i = D^i \rightarrow D$, with i denoting the arity. Otherwise, we work on the abstract domain and $\mathcal{I}_{\mathcal{F}}(f)^i = U^i \rightarrow U$. Predicate symbols are interpreted as a subset of both domains: $\mathcal{I}_{\mathcal{P}}(p) \subseteq D \cup U$.

In order to define the interpretation of terms, we first need to handle variables. For each variable, there can be a number of variable assignments B, assigning each variable v to an individual $v^B \in U \cup D$. A variable assignment in the concrete domain $v^B \in D$ is called a concrete variable while a variable assignment in the union of the domains $v^B \in U \cup D$ is called an abstract variable. Given that definition, if a term is a variable with assignment B, then $t^{\mathcal{I},B} = t^B$, while if a term is a function $f(t_1, ..., t_n)$, then $t^{\mathcal{I},B} = \mathcal{I}_{\mathcal{F}}(f)(t_1^{\mathcal{I},B}, ..., t_n^{\mathcal{I},B})$.

Due to lack of space, we omit a detailed presentation of formula interpretation. Satisfaction and entailment follow the definition of classical first-order logic, slightly modified to include variable assignment.

4 WSSL Extensions

4.1 Solving the Ramification Problem

At the heart of the solution to the ramification problem in the fluent calculus lie causal relationships (see Chap. 9 in [23]). A ramification is always linked to the direct effect (or another ramification) that brings it about, hence that relationship needs to be modeled. In order to do that, the fluent calculus is extended with a new predefined predicate named *Causes*, with 6 state variables and one situation variable as arguments. The semantics of $Causes(z, p, n, z', p', n', s)$ is the following: in situation s, the occurred positive and negative effects p and n possibly cause an automatic update from state z to state z', with positive and negative effects p' and n'. Essentially, the Causes predicate is a relation between two state-effect triples with respect to a situation. The general form of a causal relationship allows for expressing separately the set of conditions Γ that drive such a relationship: $(\forall)(\Gamma \rightarrow Causes(z, p, n, z', p', n', s)$.

Note that in [23], causal relationships are generalized to include the transitive closure of $Causes$, in order to express the notion of arbitrary chains of ramifications. However, in the case of services, specification requirements are much simpler: since causal relationships are not always expected to be expressed by providers, some relationships can be derived given a set of services that participate in a service composition, in order to determine the consistency of the composition. Hence, there is no need to express arbitrary chains, since the derivation will yield concrete direct causal relationships between condition pairs. Taking this into account, the inference of ramifications is expressed by a simplified version of the *Ramify* macro defined in [23]: $Ramify(z, p, n, z', s) \overset{def}{=} (\exists p', n')(Causes(z - n + p, p, n, p', n', s)$.

The final step for solving the ramification problem is integrating ramifications into state update axioms, so that they offer a complete view of what is caused after the execution of an action. The state update axioms with ramifications are defined as follows: $Poss(A(x), s) \rightarrow (\exists y)(\Delta(z) \wedge Ramify(z, \theta^+, \theta^-, z', Do(A(x), s))$.

WSSL with Ramifications. In order to support specification of ramifications, the WSSL signature needs to be extended to include the $Causes$ predicate and the $Ramify$ macro, in order to be able to express causal relationships and ramification inference. Returning to our motivating example, expressing the side effect of credit card invalidation under certain circumstances yields the following causal relationship:
$DailyLimitReached(?cCrd, ?user) \rightarrow Causes(?z, Valid(?payForm, ?user) + PayCompleted(?doc, ?user), \varnothing, ?z\_r, \varnothing, Valid(?cCrd, ?user), ?s)$ which encodes the behavior that whenever the daily limit condition holds and we are in a state where the postconditions of $ExecPayment$ have been made true, then we move to a new state where the $Valid$ fluent has been removed (falsified) for the particular user and credit card.

4.2 Solving the Qualification Problem

Solving the (exogenous) qualification problem involves taking into account expectations that are otherwise assumed to be always satisfied, but which can explain unsuccessful executions when all known preconditions hold. To account for such accidents, the fluent calculus signature is further extended with a new sort $ACCIDENT$ and C, a finite set of function symbols into sort $ACCIDENT$ (see Chap. 10 in [23]). Also, a new predefined predicate named Acc is introduced, with two variables, one of sort $ACCIDENT$ and one of sort SIT. $Acc(c, s)$ carries the semantics that accident c happened in situation s.

In order to assume away accidents , a default logic theory must be employed: $\Delta = (\{\frac{:\neg Acc(c,s)}{\neg Acc(c,s)}\}, \Sigma \cup O)$, where O is a set of situation formulas that are true, called observations and Σ is the domain axiomatization. The rule is essentially a single universal default on the non-occurrence of all accidents: as long as the observations are in line with the expected effects of an action, then no accident needs to be considered as having taken place.

Accidents are then integrated into state update and action precondition axioms. To express the default case, where no accident has taken place, we include the conjunct $(\forall c)\neg Acc(c, s)$ in the right-hand side of the state update axiom. Action precondition axioms are rewritten in the following form: $Poss(A(x), s) \equiv [(\forall c)\neg Acc(c, s) \rightarrow \Pi_A(x, s)]$ meaning that an action is possible at a situation provided that no accidents have happened and the preconditions are true.

Apart from these default modifications, where no accident has happened, we may want to express the effects that are brought upon by a particular accident happening. This can be accomplished by adding a disjunct to the state update axiom of the action that expresses that connection, essentially providing more than one possible state updates for the same action, depending on whether an accident has occurred and which accident it was.

WSSL with Qualifications. In the spirit of the extensions performed to support ramifications, WSSL can be further extended to include qualifications, by introducing the new sort $ACCIDENT$ and the predefined predicate Acc. Action precondition and state update axioms also need to be rewritten to follow the guidelines expressed previously (stating preconditions and effects in case no accidents happen and optional additional effects in case an accident occurs). Furthermore, a set of defaults should be attached to every WSSL specification, expressing the occurrence or not of accidents by default.

Returning to our motivating example, in order to solve the exogenous qualification problem, we include the simplest default theory $\Delta = (\{\frac{:\neg Acc(c,s)}{\neg Acc(c,s)}\}, \Sigma \cup O)$, which assumes all accidents do not hold by default. Then, we modify the postconditions of each service to account for accidents. For instance, the postcondition for $ExecPayment$ can now express the fact that if we observe no change after a seemingly successful execution, then some accident must have happened:
$Poss(ExecPayment(?payForm), ?s) \rightarrow (\exists ?doc) \cdot ((\forall ?c)\neg Acc(?c, ?s) \wedge ?z\_out = ?z\_in + HasOutput(?payCCnf) + Valid(?payForm, ?user) + PayCompleted(?doc, ?user) \vee \exists ?c(Acc(?c, ?s) \wedge ?z\_out = ?z\_in)$

4.3 Complexity and Decidability

WSSL and its extensions were designed with high expressivity as a fundamental goal in order to be able to support solutions to the frame, ramification and qualification problems. Decidability results for the simple fluent calculus obtained by Lehmann [14] and Hölldobler [11] show that considerable restrictions must be made to both the calculus definition and the queries that can be decided. However, in [20], Schiffel and Thielscher present a complete and correct bidirectional translation process between the situation calculus and the fluent calculus. Based on this work, one can argue that any decidability results that have been obtained for the situation calculus can be applied for the fluent calculus as well, given the fact that the translation process guarantees the equivalence of the entailment procedures for the two calculi.

One case of decidability results in the situation calculus that is of particular interest because of its expressiveness is presented in [10]. The authors restrict situation calculus to a two-variable fragment with counting, C^2, which has been proven to be decidable in NExpTime and also add Description Logic capabilities such as the expression of TBox and RBox statements. The authors examine the expressivity of such an action language for the case of service domains, pointing out the fact that if both atomic services and properties affected by them can be expressed using only two parameters, then they can be expressed using C^2, with decidable entailment. In his thesis, Milicic [16] proves that the C^2 fragment of the situation calculus is more expressive than any of the DL fragments considered in relation to action formalisms.

As far as the extensions of WSSL are concerned, decidability is affected only by the inclusion of a default theory. In the case of defaults without prerequisites (such as the one used in the solution to the qualification problem), decidability is possible provided that consequences and axioms in the default are *predefinite variable clauses* (function-free and all the variables occurring in a positive literal also occur in a negative one).

4.4 Applicability and Practical Concerns

WSSL is designed as a language at the level of OWL-S and WSML, focusing on the completeness of service descriptions with regard to the frame, ramification and qualification problems. As such, WSSL is related to existing Web service description languages in two different ways. First, WSDL (and SAWSDL) can be employed as ground languages so that WSSL specifications can be associated with concrete realizations of services and service executions, benefitting greatly from the wide acceptance and use of WSDL as a descriptive interface to services. Second, existing OWL-S and WSML descriptions can be ported to WSSL using translation mechanisms, as well annotation facilities to fill up information that is exclusive to WSSL (such as causal relationships). Moreover, given the strong foundations provided by the fluent calculus, WSSL specifications can facilitate not only service validation and verification, as detailed in the next section, but also service composition, by applying fluent calculus-based planning.

5 WSSL Validation Tool

To exploit the expressive capabilities of WSSL, we implemented a validation tool that takes a WSSL specification as input and allows the user to express goals that are validated against the specification. The validation process involves 3 basic steps. First, the WSSL document (written in XML following the syntax defined later on) is translated into a FLUX [22] program. Then the tool expects a WSSL goal to be expressed by the user, which is also translated in FLUX (the user can also express FLUX queries directly). The final step involves automatically loading the program in the ECLiPSe Constraint Programming System [1], validating the goal and returning the results to the user.

FLUX (FLUent eXecutor) is a method that essentially offers an implementation of the fluent calculus in logic programming with constraint handling. FLUX has a restricted expressiveness that enables an excellent computational behavior, offering reasoning in linear time with regard to the size of state representation. It is expressive enough, however, to be able to support any WSSL specification. Table 3 offers a mapping from WSSL elements to FLUX clauses. Note that preconditions are combined to a single clause with a *poss* head (and postconditions to one with a *state_update* head) to preserve correct service execution semantics.

Table 3. Mapping from WSSL to FLUX

WSSL	FLUX
$Holds(HasInput(x), z)$ $Poss(A(x), z) \equiv \Pi(z)$	$poss(a(x), Z) :-$ $holds(HasInput(x), Z), \hat{\Pi}(z).$
$Holds(HasOutput(x), z)$ $Poss(A(x), z_1) \rightarrow \Delta(z_1) \wedge$ $State(Do(A(x), z_1)) = State(z_1) - \theta^- + \theta^+$ $= State(z_1) - \theta^- + \theta^+$	$state\_update(Z_1, a(x), Z_2) :-$ $\hat{\Delta}(Z_1), update(Z_1, [\theta^+,$ $HasOutput(x)], \theta^-, Z_2).$
$Ramify(z_1, \theta^+, \theta^-, z_2, Do(A(x), s))$	$ramify(Z_1, \theta^+, \theta^-, Z_2)$
$\Gamma \rightarrow Causes(z_1, p_1, n_1, z_2, p_2, n_2, s)$	$causes(Z_1, p_1, n_1, Z_2, p_2, n_2) :- \hat{\Gamma}.$
$Poss(A(x), z_1) \rightarrow \exists C Acc(C, s) \wedge ...$	$ab\_state\_update(Z_1, a(x), Z_2) :- ...$

The ˆ denotes that the particular clause has been suitably translated in FLUX. Note that for efficiency reasons, translation of postconditions omits the verification of the precondition, assuming that services are executed only in states it is possible. If the user wants to verify them, then they should be included in the query expression. Also, the situation argument is suppressed in the translation of *Ramify* and *Causes*. As far as accidents are concerned, FLUX avoids the explicit encoding of the predicate *Acc* and the default theory, by producing one `state_update` clause for the normal case, and one `ab_state_update` clause, containing all state updates that are associated with accidents in the order they are expressed in the initial representation.

The translation process can be trivially proven to be sound since it follows the soundness of the encoding of the fluent calculus in FLUX. For instance, the tool produces the FLUX clause B: $holds(f, Z)$ from a WSSL expression A: $Holds(f, z)$ for a fluent f and a state variable z. B is indeed the FLUX encoding of the fluent calculus macro $Holds$, hence the translation is sound and the same applies to all WSSL element translations in Table 3.

An example FLUX query fed to the validation tool, given the service specification of *ExecPayment* in Table 2 is: `poss(execpayment(payform), Z), state_update(Z, execpayment(payform), Z1).` which would yield the result `Z=[hasinput(payform),valid(payform, user),valid(ccrd, user)]` and `Z1=[Z, hasoutput(payccnf), paycompleted(doc, user)].`

5.1 WSSL/XML

In order to provide machine readability for WSSL documents, so that they are parsed in a standardized way by the validation tool, an XML syntax for WSSL is proposed in this section, named WSSL/XML. The syntax includes the extensions covered in Sect. 4; such elements are considered optional. The XML schema for WSSL/XML can be found online at http://www.csd.uoc.gr/˜gmparg/wssl.

Translation of WSSL logical expressions is based on the XML syntaxes for WSML and RuleML. Table 4 shows some indicative cases. $expr_i$ stands for any arbitrary WSSL logical expression.

Table 4. WSSL/XML syntax for logical expressions

Expression	Abstract Syntax	XML Syntax
Negation	$tr(\neg expr_1)$	`<neg>` $tr(expr_1)$ `</neg>`
Conjunction	$tr(expr_1 \wedge expr_2)$	`<and>` $tr(expr_1)$ $tr(expr_2)$ `</and>`
Implication	$tr(expr_1 \rightarrow expr_2)$	`<implies>` $tr(expr_1)$ $tr(expr_2)$ `</implies>`
Universal Quantification	$tr(\forall(?varname) \cdot expr_1)$	`<forall> <var name="varname"/>` $tr(expr_1)$ `</forall>`
Function	$tr(func(term_1, ..., term_n))$	`<term name="func">` $tr(term_1)...tr(term_2)$`</term>`
Predicate	$tr(pred(term_1, ..., term_n))$	`<predicate name="pred">` $tr(term_1)...tr(term_n)$`</predicate>`
Holds	$tr(Holds(f, ?z))$	`<holds state="?z">` $tr(f)$ `</holds>`

5.2 Evaluation

The WSSL validation tool essentially consists of a translation mechanism from WSSL to FLUX, followed by a query execution in the ECLiPSe constraint programming system. In this section, we evaluate the overhead of the translation process in terms of runtime and memory consumption. For efficiency results with regard to running FLUX queries in ECLiPSe, refer to [22] and the FLUX website (http://www.fluxagent.org).

The tool was implemented as a Java application and can be found online at http://www.csd.uoc.gr/˜gmparg/wssl. We investigated the scalability of the translation process in terms of runtime and memory consumption by translating WSSL specifications that contained from 1 to 500 pairs of pre- and postconditions containing randomly generated, uniquely named fluents. The runtime values are an average of 10 runs, while memory consumption is presented before and after garbage collection. The evaluation was performed on an Intel® Core™ i7-740QM processor running at 1.73GHz, with 6 GB RAM.

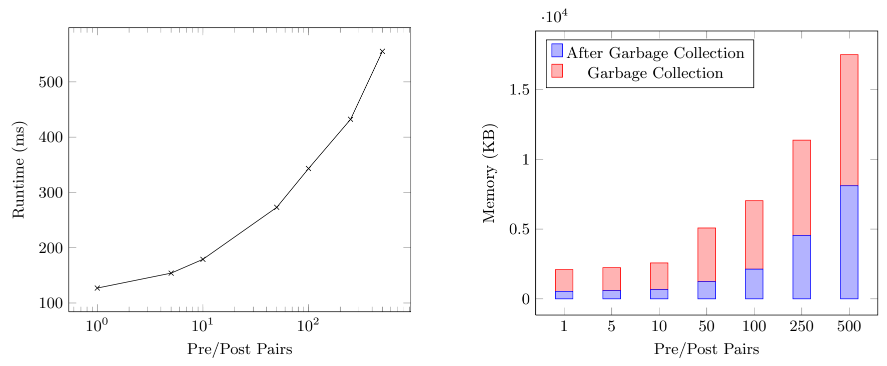

Fig. 2. Scalability evaluation of the WSSL validation tool

As we can see in Fig. 2, runtime stays less than 600ms even for specifications containing 500 pairs of pre- and postconditions, and actual memory consumption peaks at around 8000KB. Based on these results and the fact that actual service specifications contain at most tens of pre/post pairs, even in the case of service composition, we can safely assume that the overhead posed by the translation process from WSSL to FLUX is insignificant.

6 Related Work

The frame, ramification and qualification problems have been largely ignored by every service description language that has been proposed in recent years (SAWSDL [9], OWL-S [15] and WSML [24]), although it was considered (but not included) in WSML to label postconditions as complete (framed) or incomplete. WSSL employs the resource and data representation of WSML but uses the fluent calculus as a knowledge foundation, which allows for efficient solutions to the aforementioned problems, while WSML's DL foundation does not.

In previous work [3], we identified the existence of the frame problem in Web service specifications and proposed a solution that involves expressing state update axioms in OWL-S and WSML descriptions. While this solution is more compatible with existing service descriptions than WSSL (which would require either a grounding or a translation mechanism), it doesn't provide a unified solution framework that includes the ramification and qualification problems.

To the best of our knowledge, the only efforts to use the fluent calculus in service research are those of Chifu et al. [6], followed by [4]. In these works, OWL-S service descriptions are translated to fluent calculus theories for the purpose of automated service composition. The authors, however, use the simple fluent calculus, without the extensions that solve the ramification and qualification problems. Also, their choice to represent inputs and outputs using the extension of the fluent calculus to support knowledge and sensing is invalid since Thielscher's intent for this extension was to represent incomplete state knowledge. In our work, inputs and outputs are represented in a natural way by special fluents.

The fluent calculus is not the only formalism that offers a solution to knowledge representation problems. The frame problem has been addressed in the situation and event calculi, while Kakas et al. [12] present an event calculus theory that solves the ramification and qualification problems as well. The reason for choosing the fluent over the event calculus in our work is two-fold. First and foremost, service specification requires a non-narrative-based formalism, since there is no need for an explicit notion of time: we only need to recognize the state before and the state after execution. Second, FLUX provides an efficient implementation of the complete fluent calculus, allowing us to implement more easily tools that require reasoning, such as the validation tool presented in Sect. 5.

7 Conclusions and Future Work

In this paper, we proposed WSSL, a novel specification language for Web services based on the fluent calculus, that provides solutions to representational issues caused by the frame, ramification and qualification problems. Also, a validation tool for WSSL specifications was implemented and evaluated. WSSL allows service providers to accurately express the behavior of a service, by defining conditions before and after execution, expressing or inferring causal relationships among conditions and accounting for unexpected unsuccessful executions. Such rich behavior specifications can then be exploited for the purposes of service validation, verification and composition.

Future work includes supporting compatibility with existing description languages, as detailed in Sect. 4.4, as well as developing a service composition approach for WSSL services, based on planning with FLUX. Also, support for composite service specification will be extended based on our previous work on deriving composite service specifications [2] as well as deriving ramifications based on the individual specifications of services participating in a composition.

Acknowledgments. This research was partially funded by the PaaSage (FP7-317715) EU project, as well as the Special Account for Research Grants, University of Crete (Grant No. 3742).

References

1. The ECLiPSe Constraint Programming System, http://www.eclipseclp.org/
2. Baryannis, G., Carro, M., Plexousakis, D.: Deriving Specifications for Composite Web Services. In: COMPSAC. pp. 432–437 (2012)

3. Baryannis, G., Plexousakis, D.: The Frame Problem in Web Service Specifications. In: Principles of Engineering Service Oriented Systems, 2009. PESOS 2009. ICSE Workshop on. pp. 9–12. IEEE Computer Society, Washington, DC, USA (2009)
4. 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. pp. 16:1–16:6. ISWSA '10, ACM, New York, NY, USA (2010)
5. Borgida, A., Mylopoulos, J., Reiter, R.: On the Frame Problem in Procedure Specifications. Software Engineering, IEEE Transactions on 21(10), 785–798 (1995)
6. 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)
7. Di Nitto, E., Mazza, V., Mocci, A.: IA-2.2.2: Collection of industrial best practices, scenarios and business cases. Tech. rep., S-Cube NoE (May 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. Gu, Y., Soutchanski, M.: Decidable Reasoning in a Modified Situation Calculus. In: Veloso, M.M. (ed.) IJCAI. pp. 1891–1897 (2007)
11. Hölldobler, S., Kuske, D.: The Boundary between Decidable and Undecidable Fragments of the Fluent Calculus. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNCS, vol. 1955, pp. 436–450. Springer (2000)
12. Kakas, A.C., Michael, L., Miller, R.: Modular-E and the role of elaboration tolerance in solving the qualification problem. Artif. Intell. 175(1), 49–78 (2011)
13. Kritikos, K., Plexousakis, D.: Requirements for QoS-based web service description and discovery. IEEE T. Services Computing 2(4), 320–337 (2009)
14. Lehmann, H., Leuschel, M.: Decidability Results for the Propositional Fluent Calculus. In: Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K.K., Palamidessi, C., Pereira, L., Sagiv, Y., Stuckey, P. (eds.) Computational Logic CL 2000, LNCS, vol. 1861, pp. 762–776. Springer (2000)
15. 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)
16. Milicic, M.: Action, time and space in description logics. Ph.D. thesis, Technische Universität Dresden (2008)
17. Miller, R.: Three Problems in Logic-Based Knowledge Representation. ASLIB Proceedings: New information perspectives (2006)
18. Pedrinaci, C., et al.: iServe: a Linked Services Publishing Platform (2010)
19. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly, Beijing (2007)
20. Schiffel, S., Thielscher, M.: Reconciling Situation Calculus and Fluent Calculus. In: AAAI. pp. 287–292. AAAI Press (2006)
21. Thielscher, M.: The Fluent Calculus. Tech. Rep. CL-2000-01, Dresden University of Technology (2000)
22. Thielscher, M.: FLUX: A Logic Programming Method for Reasoning Agents. CoRR cs.AI/0408044 (2004)
23. Thielscher, M.: Reasoning Robots, Applied Logic Series, vol. 33. Springer Netherlands (2005)
24. WSML Working Group: The Web Service Modeling Language WSML (2008), http://www.wsmo.org/wsml/wsml-syntax