# A Formalism for Specifying and Executing Transactional Semantic Web Services

Francisco J. Galán and Ahmed Riveras

High Technical School of Software and Computer Engineering.
University of Seville. Seville, Spain.

**Abstract.** We propose a formalism for specifying and executing semantic web services in a transactional way. The formalism is based on the following elements: (1) the service is part of an agent with state codified by positive facts. The set of possible states during the agent's lifetime is constrained by an invariant. (2) Two types of services are considered: services which can change the agent's state (stateful service) and services which can not (stateless service). The first ones include in their specifications a transaction written in Transaction Logic and therefore operationally interpreted by sequences of states. (3) The execution of the stateful service may generate inconsistencies (states that do not satisfy the invariant). (4) We propose to repair inconsistencies by making use of a chase procedure. (5) According to transactional semantics, a service execution is undone if a repair is not possible.

Keywords: chase, invariant, pre-condition, post-condition, query, service execution, semantic web service, state, transaction.

## 1 Introduction

Service orientation is a promising paradigm for offering and consuming functionalities in the Web. Current technology based on WSDL/UDDI is rather syntactical [WSDL01]. To palliate the situation, the Web is being extended with the so-called semantic web services [SaP07,WSMO08,KLP05,JAS10,GuY10,KlK12]. A semantic web service is a web service enriched with semantics annotations that can be interpreted by machine. In this paper, the manner of achieving the post-condition of a service through particular sequences of states is a matter of interest. For example, a customer might prefer to buy products where deliveries precede payments. Such sequences of states have to be understood as a part of the specification not of the implementation. Current proposals do not consider the concept of state as a relevant aspect in the definition of the semantic web service [WSMO08,WSML08]. To palliate this situation, we propose a formalism for specifying and executing web services with transactional semantics. For illustrating it, suppose an example agent whose state, called $state_1$, specifies that $John$ is a professor, in particular, a full professor, $Cs1$ is a course and $John$ coordinates $Cs1$.

$$state_1 = \{ fullProf(John), prof(John), course(Cs1), coordinates(John, Cs1) \}$$

The agent has the following invariant:

$inv_1$ :
1. $\forall x(assistantProf(x) \Rightarrow \exists^{\downarrow} y(fullProf(y) \wedge assists(x,y)))$
2. $\forall x(fullProf(x) \Rightarrow \exists^{\downarrow} y(course(y) \wedge coordinates(x,y)))$
3. $\forall x(assistantProf(x) \Rightarrow prof(x))$
4. $\forall x(fullProf(x) \Rightarrow prof(x))$
5. $\forall x(fullProf(x) \wedge assistantProf(x) \Rightarrow false)$
6. $\forall x(prof(x) \wedge course(x) \Rightarrow false)$

As we can see, $state_1$ satisfies all rules in $inv_1$, being $\exists^{\downarrow}$ an existential quantifier bounded to the constants occurring in the agent's state.

As we said, the proposed formalism distinguishes two classes of services: services that can change the agent's state (called stateful services) and services that can not change the agent's state (called stateless services). The specification of a stateful service has four elements: a signature, a pre-condition written in First-order Logic, a formula written in Transaction Logic describing the transaction and a post-condition written in First-order Logic. The specification of a stateless service has two elements: a signature and a formula written in First-order Logic.

The example agent includes a stateful service called $addAssistantProf(x)$ whose pre-condition ($true$) refers to any of the possible agent's states, the post-condition expresses that $x$ is an assistant professor. A formula in Transaction Logic formalizes the transaction by a primitive update called add which adds a new individual $x$ to the extension of predicate $assistantProf$:

$$\begin{array}{l} \textsf{Stateful service} : addAssistantProf(x) \\ \{\textsf{Pre} :\ true\} \\ \quad addAssistantProf(x) \Leftarrow \textsf{add}(x, assistantProf) \\ \{\textsf{Post} :\ assistantProf(x)\} \end{array}$$

The agent also includes a stateless service which relates a course with its coordinator:

$$\begin{array}{l} \textsf{Stateless service} : courseCoordinatedBy(y,x) \\ \quad courseCoordinatedBy(y,x) \Leftarrow \\ \qquad\qquad course(y) \wedge coordinates(x,y) \end{array}$$

The execution of service $addAssistantProf(x)$ with $x = Sarah$ from $state_1$ will reach an inconsistency due to rules 1 and 3 in $inv_1$ are not satisfied:

$$inconsistency_1 = state_1 \cup \{\, assistantProf(Sarah)\}$$

Once reached an inconsistency, we propose to repair it by making use of the well-known chase procedure [AHV95,DNR08]. This procedure calculates universal models for database schemas $\Sigma$ from (incomplete) database instances $I$. By database schema, we refer here to a set of rules of the form $\forall x, y(\alpha(x,y) \Rightarrow \exists z(\beta(x,z)))$. We say that a rule can be applied on an instance $I$ if there are tuples of constants $t$, $u$ and $v$ such that $\alpha(t,u) \in I$ and $\beta(t,v) \notin I$. In our example, $inconsistency_1$ represents the database instance and $inv_1$ the database schema.

The chase procedure is iterative. In the first iteration, rules 1 and 3 in $inv_1$ can be applied to $inconsistency_1$ resulting the following repair:

$$repair_1 = inconsistency_1 \ \cup \ \{\, assists(Sarah, John), prof(Sarah)\,\}$$

As we can see, the resulting repair reaches a fixpoint which satisfies all rules in $inv_1$. We can therefore consider $repair_1$ as a new agent's state.

The repair process may generate *labeled nulls* [DNR08]. Suppose another agent with the following state:

$$state_2 = \ \{\, product(TvSony), product(TvPhilips)\,\}$$

and the following invariant:

$inv_2$ :
1. $\forall p(product(p) \wedge sold(p) \Rightarrow \exists^{\downarrow\uparrow} c(customer(c) \wedge delivery(p, c) \wedge invoice(p, c)))$
2. $\forall x(product(x) \wedge customer(x) \Rightarrow false)$

As we can see, $state_2$ satisfies $inv_2$, being $\exists^{\downarrow\uparrow}$ an existential quantifier which prioritizes quantification over the constants occurring in the agent's state.

The agent includes a stateful service called $anonymousSell(p)$. The precondition in $anonymousSell$ states that $p$ is a product and its postcondition that $p$ has been sold:

$$\begin{array}{l}\textsf{Stateful service}: anonymousSell(p)\\ \{\textsf{Pre}:\ product(p)\}\\ \quad anonymousSell(p) \Leftarrow \textsf{add}(p, sold)\\ \{\textsf{Post}:\ sold(p)\}\end{array}$$

The execution of $anonymousSell(p)$ with $p = TvSony$ from $state_2$ will reach the following inconsistency:

$$inconsistency_2 = \ \{\, product(TvSony), product(TvPhilips), sold(TvSony)\,\}$$

Rule 1 in $inv_2$ can then be applied to $inconsistency_2$ resulting the following repair:

$$repair_2 = inconsistency_2 \cup \{\, customer(c_1), delivery(TvSony, c_1), invoice(TvSony, c_1)\,\}$$

being $c_1$ a labelled null which acts as a new customer.

A second execution of $anonymousSell(p)$ with $p = TvPhilips$ from $repair_2$ will reach the following state due to the use of $\exists^{\downarrow\uparrow}$:

$$\begin{aligned} state_3 = \{\, & product(TvSony), product(TvPhilips), customer(c_1), delivery(TvSony, c_1),\\ & invoice(TvSony, c_1), delivery(TvPhilips, c_1), invoice(TvPhilips, c_1)\} \end{aligned}$$

It is important to note that in certain situations, we may reach inconsistencies without any possible repair. For instance, the execution of $addAssistantProf(John)$ from $state_1$ will fire rule 5 in $inv_1$ deriving $false$. According to transactional semantics, the execution of $addAssistantProf(John)$ has to be undone restoring the agent's state to that before execution (that is, $state_1$).

Finally, in certain situations, the repair can generate what we call a union state. For instance, considering a new invariant in the example agent:

$$inv_3 = inv_1 \cup \forall x(prof(x) \Rightarrow fullProf(x) \vee assistantProf(x))$$

the repair of the inconsistency $\{prof(John)\}$ will produce the union state:

$$\begin{array}{l} \{fullProf(John), prof(John), course(c_1), \\ \quad coordinates(John, c_1)\} \\ \qquad\qquad\qquad\qquad \cup \\ \{assistantProf(John), prof(John), course(c_1), \\ \quad fullProf(p_1), assists(John, p_1), prof(p_1), \\ \quad coordinates(p_1, c_1)\} \end{array}$$

The rest of the paper is organized as follows. Section 2 is devoted to preliminary definitions. In Section 3, we begin by formalizing the concepts agent, invariant and state and then we define the concept of semantic web service with transactional semantics. In Section 4, we explain the manner of controlling repairs. This point is interesting when designing complex services. Finally, we establish conclusions and future work.

## 2 Preliminaries

In the rest of the paper, we will assume that the reader is familiarized with Logic programming (LP) [Llo87].

A *Datalog rule* is an expression of the form $A \Leftarrow B_1, ..., B_n$ where $A$ (the head) and $B_1, ..., B_n$ (the body) are atoms and each variable occurring in $A$ must occur in some $B_i$ with $i = 1..n$. In Datalog, all rules with empty body are ground. A rule with no head is called a goal. Datalog does not allow the use of function symbols. Semantically, Datalog programs have finite models.

A *Datalog*$^\neg$ *rule* is an expression of the form $A \Leftarrow L_1, ..., L_n$ where $A$ is an atom and each $L_i$ in the body is a literal.

Let $P$ be a Datalog$^\neg$ program, the precedence graph $G_P$ of $P$ [AHV95] is the labeled graph whose nodes are the relation symbols occurring as heads in clauses of $P$ and its edges are the following:

– If $r_1(u) \Leftarrow ...r_k(v)...$ is a rule in $P$, then $(r_k, r_1)$ is an edge in $G_P$ with label $+$.
– If $r_1(u) \Leftarrow ...\neg r_k(v)...$ is a rule in $P$, then $(r_k, r_1)$ is an edge in $G_P$ with label $-$.

A Datalog$^\neg$ program is *stratifiable* if and only if its precedence graph $G_P$ has no cycle containing a negative edge [AHV95].

A Datalog$^\neg$ program $P$ is *stratified* if there exists a mapping $\sigma$ from relation symbols in $P$ to $[1..n]$ such that

– For every rule of the form $r_1(u) \Leftarrow ...r_k(v)...$, then $\sigma(r_k) \leq \sigma(r_1)$.
– For every rule of the form $r_1(u) \Leftarrow ...\neg r_k(v)...$, then $\sigma(r_k) < \sigma(r_1)$.

An *allowed goal* is a formula $\Leftarrow q$, being $q$ a conjunction of literals and every variable that occurs in $q$ occurs in a positive literal in $q$ [Llo87].

*Transaction Logic* (denoted by $\mathcal{T_R}$) is an extension of predicate logic which accounts for the phenomenon of state change in logic programs and databases [BK95,BK98]. $\mathcal{T_R}$ provides a syntax and a semantics in which queries and updates can be logically combined to build complex transactions. The execution of a transaction is formally described using statements called executional entailments:

$$P, <D_0, ..., D_n> \models \psi \qquad (1)$$

being $P$ a set of $\mathcal{T_R}$ formulas, $D_i$ a state and $\psi$ a $\mathcal{T_R}$ formula. Intuitively, $P$ is a set of transaction definitions, $\psi$ is a transaction invocation and $<D_0, ..., D_n>$ is a sequence of states. In formal terms, (1) means that $\psi$ is true wrt $P, <D_0, ..., D_n>$. One of the goals underlying the design of $\mathcal{T_R}$ is to make it general enough to deal with a variety of types of states and formulae. Queries and updates can be combined in $\phi$ by using logical connectives. A new connective $\otimes$ (serial conjunction) is considered [Bon97]. The formula $\varphi \otimes \psi$ denotes the composite transaction consisting of transaction $\varphi$ followed by transaction $\psi$. Roughly speaking, $\varphi \otimes \psi$ means "do $\varphi$ and then do $\psi$".

## 3 Description of Services

This section begins with the conceptual definition of transactional semantic web service (TS in the following). Once established the conceptualization, we formalize the TS by making use of two languages: First-order Logic for aspects that have a static nature and Transaction Logic for aspects that have a dynamic nature. We first formalize the concepts agent, invariant and state and then formalize the concept of semantic web service with transactional semantics. To repair states is a key aspect in the formalism. We extend the transaction language with two (non-logical) primitives for enabling and disabling respectively the repair procedure during service's execution.

### 3.1 Conceptualization

Conceptually, an agent is a container of TSs, invariant and state. A state is a set of positive facts defined on a signature and constrained by an invariant. The description of the TS consists of pre-condition, transaction and post-condition. The pre-condition is a condition that has to be satisfied at the beginning of the transaction. The post-condition is a condition that has to be satisfied at the end of the transaction. A TS invocation is allowed only if its pre-condition is satisfied at the current state. If, after transaction execution, the post-condition or invariant is not satisfied then such execution is undone restoring the agent's state to that before executing the transaction.

### 3.2 Agents, States and Invariants

An *agent* is a tuple $(TS_1, TS_2, ..., TS_n, inv, state)$ where $TS_i$ denotes the $i$-th TS service, *inv* the invariant and *state* the current state.

Function symbols are not allowed in the formalization of states. We distinguish two types of states: simple states and union states. A *simple state* is a finite set of positive ground facts. For instance, $state_1$ in Section 1 is a simple state whose formalization is:

$$\begin{aligned} state_1 = \{ & fullProf(John) \Leftarrow, prof(John) \Leftarrow, \\ & course(Cs1) \Leftarrow, coordinates(John, Cs1) \Leftarrow \} \end{aligned}$$

A *union state* is a union of simple states.

An *invariant* is a set of rules. A *rule* is a formula of the form $\forall x, y(\alpha(x, y) \Rightarrow \exists^* z(\beta(x, z))$ where $x$, $y$ and $z$ are tuples of variables, $\exists^*$ represents a kind of existential quantification, $\alpha$ is a conjunction of atoms (equality is excluded) and $\beta$ can be one of the following: (a) *false*, (b) equality ($=$), (c) conjunction of atoms or (d) disjunction of atoms. The kind of existential quantification can be: (a) $\exists^* = \exists^{\downarrow}$ (quantification over constants occurring in the agent's state), (b) $\exists^* = \exists^{\uparrow}$ (quantification over constants not occurring in the agent's state, also known as labelled nulls in database community), or (c) $\exists^* = \exists^{\downarrow\uparrow}$ (operational mixture of (a) and (b) but giving priority to quantification over constants occurring in the agent's state). See examples of invariants ($inv_1$, $inv_2$ and $inv_3$) in Section 1.

Every state during the agent's lifetime has to satisfy the invariant. If the state is a union state, each simple state in the union state has to satisfy the invariant. We say that an agent is in *consistent state* if and only if its state satisfies the invariant.

The execution of a stateful TS may reach an *inconsistency* (a set of positive ground facts that does not satisfy the invariant). In order to *repair* an inconsistency we apply the procedure known as *chase* [DNR08]. Formally, a rule $\forall x, y(\alpha(x, y) \Rightarrow \exists^* z(\beta(x, z))$ *applies* on an inconsistency *inconsistency* if there are constants $t, u$ in *inconsistency* such that $inconsistency \models \alpha(t, u)$ yet there is no $v$ in *inconsistency* such that $inconsistency \models \beta(t, v)$. The addition of $\beta(t, v)$ to *inconsistency* constitutes what we call a repair. The following example illustrates a repair:

Agent's state: $state_1 = \{course(Cs1)\}$
Agents's invariant: $inv_1$ (Section 1)
Transaction: $\psi \equiv \mathsf{add}(John, fullProf)$
Execution of $\psi$:
(before repairing) $state_2 = \{fullProf(John) \Leftarrow, course(Cs1) \Leftarrow\}$
(after repairing) $state_2 = \{fullProf(John) \Leftarrow, course(Cs1) \Leftarrow$
$prof(John) \Leftarrow, coordinates(John, Cs1) \Leftarrow\}$

As we said, it is not always possible to find a repair. The following example illustrates this situation:

Agent's state: $state_1 = \{\}$
Agents's invariant: $inv_1$ (Section 1)
Transaction: $\psi \equiv \mathsf{add}(John, fullProf) \otimes \mathsf{add}(Cs1, course)$
Execution of $\psi$:
(after executing $\mathsf{add}(John, fullProf)$) $state_2 = \{fullProf(John) \Leftarrow\}$
$state_2$ can not be repaired due to the impossibility of satifying rule 2 in $inv_1$.

A *homomorphism* from a set of ground atoms $I$ to a set of ground atoms $J$, denoted by $I \rightarrow J$, is a mapping $h$ on the constants and labeled nulls in $I$ such that (a) preserves constants ($h(c) = c$ for every constant $c$) and (b) preserves relationships (for every atom $a(t_1, ..., t_k) \in I$, we have $a(h(t_1), ..., h(t_k)) \in J$). The chase produces a sequence of repairs $repair_0 \rightarrow repair_1 \rightarrow ....$ We say that a repair exists if the chase terminates.

A sufficient condition for avoiding the construction of infinite repairs is to characterize the agent's invariant as a set of weakly acyclic rules [DNR08]. Informally, a set of rules is weakly acyclic if it does not allow for cascading

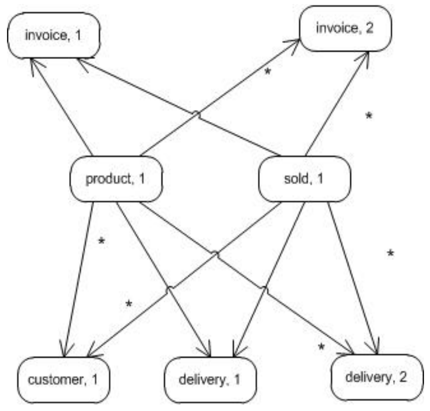

**Fig. 1.** Dependency graph for $inv_2$.

of labeled null creation during the chase [FKM05]. We can decide about weak acyclicity by constructing a directed graph called *dependency graph*. Fig.1 shows the dependency graph for $inv_2$ in Section 1. As we can see, each node in the graph identifies an argument in a predicate. For instance, (*product*, 1) denotes the argument 1 in predicate *product*. The arcs of the dependency graph are defined in the following manner. For a given rule $\forall x, y(\alpha(x, y) \Rightarrow \exists^* z(\beta(x, z))$, with $\exists^* = \exists^{\uparrow}$ or $\exists^* = \exists^{\downarrow\uparrow}$, the source nodes are arguments belonging to $x$ that occur in some atom in $\alpha$. There are two kinds of arcs: the so-called arc $*$ and the standard arc. The arc $*$ connects each source node in the $\alpha$-part of the rule with all nodes $z$ in the $\beta$-part of the same rule. The standard arc connects each source node in the $\alpha$-part of the rule with each node $x$ in the $\beta$-part of the same rule. For a given rule $\forall x, y(\alpha(x, y) \Rightarrow \exists^* z(\beta(x, z))$, with $\exists^* = \exists^{\downarrow}$, all arcs are standard. The standard arc connects each source node in the $\alpha$-part of the rule with each node $x$ in the $\beta$-part of the same rule and also with each node $z$ in the $\beta$-part of the same rule.

We say that a set of rules is *weakly acyclic* if and only if its dependency graph does not contain any cycle having some arc $*$. For instance, invariants shown in Section 1 are all weakly acyclic.

In the proposed formalism, every repair is terminating because it takes as inputs a state and an invariant formalized as a set of weakly acyclic rules. Theorems in [FKM05] and [DNR08] prove that every chase sequence generated from a finite instance with a set of weakly acyclic rules is always finite.

### 3.3 Services

The formalism distinguishes two kinds of TSs: stateless TSs and stateful TSs. A *stateless TS* is a TS whose execution can not change the agent's state. A *stateful TS* is a TS whose executions can change the agent's state.

A *stateless TS* consists of a signature containing name and arguments and a specification composed of clauses of the form $service(x) \Leftarrow \exists z B(x, z)$, where $x$ and $z$ are tuples of variables (the empty tuple is allowed) and $B$ is a conjunction of literals. See for instance, the stateless TS called *courseCoordinatedBy* in Section 1. Given an agent, the set formed out from the set of its stateless service specifications and its current state is a Datalog$^\neg$ program [AHV95]. For instance, the following Datalog$^\neg$ program results from putting together stateless service specification *courseCoordinatedBy* and state $state_1$ (Section 1):

$$\begin{array}{l} fullProf(John) \Leftarrow \\ prof(John) \Leftarrow \\ course(Cs1) \Leftarrow \\ coordinates(John, Cs1) \Leftarrow \\ courseCoordinatedBy(y, x) \Leftarrow course(y) \wedge coordinates(x, y) \end{array}$$

Let $P$ be the Datalog$^\neg$ program resulting from putting together the agent's state and all its stateless service specifications. We require $P$ to satisfy the following properties: (1) $P$ is stratified, (2) every clause in $P$ is admissible and (3) every clause in the definition of a predicate symbol occurring in a positive literal in the body of a clause in $P$ is allowed. The purpose of (1) is to ensure the existence of minimal Herbrand model for stateless services [Llo87]. The purpose of (2) and (3) is to avoid floundering [Llo87] during stateless service's execution. A program $P$ having these properties is also known as a program with *stratified semantics*. This semantics can be computed in polynomial time with respect to the agent's state [AHV95].

A *stateful TS* consists of a signature containing name and arguments, a FOL formula called pre-condition whose free variables are contained in the arguments, a transaction written in an executable version of $\mathcal{TR}$ and a FOL formula called post-condition whose free variables are also contained in the arguments. See for instance, the stateful TS called *addAssistantProf* in Section 1.

Again, we recall that the transaction is part of the service's specification not of its implementation.

The executable version of $\mathcal{T_R}$ we refer to in this paper restricts transactions to non-recursive clauses of the form $service \Leftarrow \phi$ where *service* is an atom and $\phi$ is a serial conjunction of transactional steps [Bon97]. Operationally, a formula $service \Leftarrow \phi$ means: "to execute *service*, it is sufficient to execute $\phi$" [BK95].

A *transactional step* can be one of the following resources: (1) query, (2) primitive update, (3) non-primitive update, (4) stateful service invocation, (5) confluent conjunction.

A *query* $q$ is formalized as an allowed goal. We intend to execute queries by means of SLDNF-resolution.

A *primitive update* is one of two following expressions: (a) $\mathsf{add}(t, r)$ or (b) $\mathsf{del}(t, r)$, being $t$ a tuple of constants and $r$ a predicate symbol.

A *non-primitive update* is one of two following expressions: (a) $\forall x(q(x) \Rightarrow \mathsf{add}(x, r))$ or (b) $\forall x(q(x) \Rightarrow \mathsf{del}(x, r))$, being $q(x)$ a query.

A *stateful service invocation* for a transaction $service(x) \Leftarrow \phi(x)$ is an atom of the form $service(t)$ where $t$ is a tuple of constants.

A *confluent conjunction* is a conjunction of transactional steps. Each of these steps can be either a query or a primitive update or a non-primitive update. The main property of the confluent conjunction is that any execution order of its transactional steps leads to the same final state.

We show an example of query, primitive update, non-primitive update, stateful service invocation and confluent conjunction.

Query : $courseCoordinatedBy(c, p) \wedge \neg fullProf(p)$
Primitive update : $\mathsf{add}(Sarah, assistantProf)$
Non-primitive update : $\forall p(assistantProf(p) \Rightarrow \mathsf{add}(p, John, assists))$
Stateful service invocation : $addAssistantProfessor(Sarah)$
Confluent conjunction : $\mathsf{add}(Sarah, assistantProf) \wedge \mathsf{add}(Sarah, John, assists)$

The semantics of a transaction $\phi$ is always defined in the context of an agent $(TS_1, ..., TS_n, inv, state_0)$ and formalized by executional entailments of the form:

$$(TS_1, ..., TS_n, inv, state_0), <state_0, ..., state_n> \models \phi$$

In situations where is clear the agent we refer to, we can relax executional entailments to expressions of the form:

$$<state_0, ..., state_n> \models \phi$$

As a notational convention, we will use $<state_1, ...>$ to denote an execution which starts from a particular state (i.e. $state_1$ is the starting state) and $<..., state_n>$ to denote an execution which ends in a particular state (i.e. $state_n$ is the ending state).

Given an agent $(TS_1, ..., TS_n, inv, state)$, the semantics of a transaction $\phi$ is defined recursively as follows:

1. Query: Let $\phi$ be a query called $q$.
   (a) Let *state* be a simple state. Then, $<state> \models q$ if and only if there is a SLDNF-refutation for $P \cup \{\Leftarrow q\}$, being $P$ the Datalog$^\neg$ program resulting from putting together *state* and all stateless service specifications in $\{TS_1, ..., TS_n\}$.
   (b) Let $state = \bigcup_{i=1..k} state^i$ be a union state. Then, $<state> \models q$ if and only if $<state^i> \models q$ for every $i \in \{1..k\}$.
2. Primitive update: Let $\phi$ be a primitive update of the form $\mathsf{del}(t, r)$, being $t$ a tuple of constants and $r$ a predicate symbol.

   (a) Let $state_0$ be a simple state. Then $<state_0, state_1> \models \mathsf{del}(t, r)$ if and only if $state_1 = state_0 - \{\, r(t) \Leftarrow \}$.
   (b) Let $state_0 = \bigcup_{i=1..k} state_0^i$ be a union state. Then, $<state_0, state_1> \models \mathsf{del}(t, r)$ if and only if $state_1 = \bigcup_{i=1..k} state_1^i$ and $state_1^i = state_0^i - \{\, r(t) \Leftarrow \}$ for every $i \in \{1..k\}$.
3. Primitive update: Let $\phi$ be a primitive update of the form $\mathsf{add}(t, r)$, being $t$ a tuple of constants and $r$ a predicate symbol.
   (a) Let $state_0$ be a simple state. Then, $<state_0, state_1> \models \mathsf{add}(t, r)$ if and only if $state_1 = state_0 \cup \{\, r(t) \Leftarrow \}$.
   (b) Let $state_0 = \bigcup_{i=1..k} state_0^i$ be a union state. Then, $<state_0, state_1> \models \mathsf{add}(t, r)$ if and only if $state_1 = \bigcup_{i=1..k} state_1^i$ and $state_1^i = state_0^i \cup \{\, r(t) \Leftarrow \}$ for every $i \in \{1..k\}$.
4. Non-primitive update: Let $\phi$ be a non-primitive update of the form $\forall x(q(x) \Rightarrow \mathsf{del}(x, r))$.
   (a) Let $state_0$ be a simple state. Then, $<state_0, state_1> \models \forall x(q(x) \Rightarrow \mathsf{del}(x, r))$ if and only if $< state_0 > \models q(x)$ and $state_1 = state_0 - \{\, r(x)\theta \Leftarrow \}$ for every computed answer $\theta$ for $P \cup \{\Leftarrow q(x)\}$, being $P$ the Datalog$^\neg$ program resulting from putting together $state_0$ and all stateless service specifications in $\{TS_1, ..., TS_n\}$.
   (b) Let $state_0 = \bigcup_{i=1..k} state_0^i$ be a union state. Then, $<state_0, state_1> \models \forall x(q(x) \Rightarrow \mathsf{del}(x, r))$ if and only if $< state_0 > \models q(x)$ and $state_1 = \bigcup_{i=1..k} state_1^i$ and $state_1^i = state_0^i - \{\, r(x)\theta_i \Leftarrow \}$ for every $i \in \{1..k\}$ and computed answer $\theta_i$ for $P_i \cup \{\Leftarrow q(x)\}$, being $P_i$ the Datalog$^\neg$ program resulting from putting together $state_0^i$ and all stateless service specifications in $\{TS_1, ..., TS_n\}$.
5. Non-primitive update: Let $\phi$ be a non-primitive update of the form $\forall x(q(x) \Rightarrow \mathsf{add}(x, r))$.
   (a) Let $state_0$ be a simple state. Then, $<state_0, state_1> \models \forall x(q(x) \Rightarrow \mathsf{add}(x, r))$ if and only if $< state_0 > \models q(x)$ and $state_1 = state_0 \cup \{\, r(x)\theta \Leftarrow \}$ for every computed answer $\theta$ for $P \cup \{\Leftarrow q(x)\}$, being $P$ the Datalog$^\neg$ program resulting from putting together $state_0$ and all stateless service specifications in $\{TS_1, ..., TS_n\}$.
   (b) Let $state_0 = \bigcup_{i=1..k} state_0^i$ be a union state. Then, $<state_0, state_1> \models \forall x(q(x) \Rightarrow \mathsf{add}(x, r))$ if and only if $< state_0 > \models q(x)$ and $state_1 = \bigcup_{i=1..k} state_1^i$ and $state_1^i = state_0^i \cup \{\, r(x)\theta_i \Leftarrow \}$ for every $i \in \{1..k\}$ and computed answer $\theta_i$ for $P_i \cup \{\Leftarrow q(x)\}$, being $P_i$ the Datalog$^\neg$ program resulting from putting together $state_0^i$ and all stateless service specifications in $\{TS_1, ..., TS_n\}$.
6. Stateful service invocation: Let $\phi$ be a stateful service invocation of the form $service(t)$ where $service$ is a stateful service with precondition $pre(service)$, transaction $service(x) \Leftarrow \phi(x)$ and post-condition $post(service)$ and $t$ a tuple of constants.
   (a) Let $state_0$ be a simple state. Then, $<state_0, ..., state_j> \models service(t)$ if and only if $state_0 \models pre(service)$, $state_j \models post(service)$ and $< state_0, ..., state_j > \models \phi(t)$.
   (b) Let $state_0 = \bigcup_{i=1..k} state_0^i$ be a union state. Then, $< state_0, ..., state_j > \models service(t)$ if and only if (1) $state_0^i \models pre(service)$ for some $i \in \{1..k\}$, (2) for every $i \in \{1..k\}$ such that $state_0^i \models pre(service)$, $state_j^i \models post(service)$ and $< state_0^i, ..., state_j^i > \models \phi(t)$ and (3) every $state_0^i$ such that $state_0^i \not\models pre(service)$ remains frozen (that is, without any change) along the transaction execution.
7. Serial conjunction: Let $\phi$ be a serial conjunction of the form $step \otimes \phi$ where $step$ a transactional step.

   (a) Let *step* be either an update or a stateful service invocation. Then, $< state_0, ..., state_j > \models step \otimes \phi$ if and only if $< state_0, state_1 > \models step$ and $<state_1, ..., state_j> \models \phi$.
   (b) Let *step* be a query. Then, $< state_0, ..., state_j > \models step \otimes \phi$ if and only if $<state_0> \models step$ and $<state_0, ..., state_j> \models \phi\theta$ for some computed answer for *step*.
8. Confluent conjunction: Let $\phi$ be a confluent conjunction of the form $\phi_1 \wedge ... \wedge \phi_n$. Let $\phi_{\pi(1)} \otimes ... \otimes \phi_{\pi(n)}$ be the serial conjunction which results from a permutation $\pi$ of the constituents of $\phi$.

   $<state_1, state_{n+1}>_{inv} \models \phi$ iff for every $\pi$,
   $<state_1, ...>_{inv} \models \phi_{\pi(1)} \otimes ... \otimes \phi_{\pi(n)}$ and $<..., state_{n+1}>_{inv} \models \phi_{\pi(1)} \otimes ... \otimes \phi_{\pi(n)}$

## 4 Enabling and Disabling Repairs

As we have seen, whenever an inconsistency is reached due to the execution of a service, a repair process must be done in reverse. In the proposed formalism, we can control the moment at which a repair is allowed to proceed. To do it, we extend the vocabulary of symbols in the formalism with two predefined (and non-logic) propositions: disableRepair and enableRepair. The execution of the first one disables the repair process and the execution of the second one enables. Neither of two propositions causes state changes. The two propositions are assumed to be true in any state. These propositions are useful to hide undesirable intermediate states when executing complex transactional services. The following example clarifies the effect of these propositions.

Agent's state: $state_1 = \{\}$
Agent's invariant: $inv_1$ (Section 1)
Transaction:
  $\phi \equiv \mathsf{disableRepair} \otimes \mathsf{add}(John, fullProf) \otimes \mathsf{add}(Cs1, course) \otimes \mathsf{enableRepair}$
Execution of $\phi$:
  (after executing disableRepair, repairs are disabled)
  (after executing add(John, fullProf)) $state_2 = \{fullProf(John) \Leftarrow\}$
  (after executing add(Cs1, course)) $state_3 = \{fullProf(John) \Leftarrow, course(Cs1) \Leftarrow\}$
  (after executing enableRepair, repairs are enabled)
  (after repairing) $state_3 = \{fullProf(John) \Leftarrow, course(Cs1) \Leftarrow$
  $prof(John) \Leftarrow, coordinates(John, Cs1) \Leftarrow\}$

As we can see, the use of propositions disableRepair and enableRepair can change the semantics of a transaction. Note the difference between this example and its corresponding one in Subsection 3.2.

## 5 Related Work

The literature contains multiple proposals of different nature. Some proposals conceive semantic web services as artifacts specified by means of sequences of states but these specifications are of conceptual nature. For instance, in [KLP05] a conceptual frame is proposed for locating semantic web services. The absence of formal languages for expressing the involved concepts obstructs its formal interpretation and therefore its programming. Other proposals are incomplete in the sense that only cover certain kinds of services. For instance, in [HZB06], a formalism is proposed for describing and reasoning about stateless services only. For instance, in [LiH04], web services and service requests are modeled as description logic classes. Matchmaking is viewed as the intersection of a web service with a service request, being this intersection computed by a DL-reasoner. Such proposals fail to reason about the dynamic of web services since DL reasoners can only work with "static" knowledge bases. Other proposals are mainly interested in finding practical solutions to particular problems but their expressive capabilities are quite limited. For instance in [JAS10], the main motivation is not to specify expressive semantic web services but to make a clear distinction between semantic web services and service requests in order to define a practical semantic web service discovery. In [GuY10], the dynamics of web services is expressed in a restricted way in order to allow an effective processing by a model checker (Promela/SPIN). This restriction impedes the materialization of states and therefore the use of data invariants.

## 6 Conclusion and Future Work

We have presented a formalism for specifying and executing semantic web services in a transactional way. The formalism is original in the following sense: (1) it integrates both declarative and operational aspects in the semantic web service, (2) it allows to describe both stateless and stateful services, (3) the semantic web service is part of an agent with state. The concept of state is not a mere label as in many other approaches. Its materialization has allowed to include in the formalism the concept of invariant. In our opinion, the use of invariants approximates our specifications to reality: the set of possible states an agent may reach along its lifetime is usually constrained by invariants. To the best of our knowledge, no approaches includes the concept of invariant in the formalization of semantic web services. (4) Based on the concept of invariant, we define the concepts of inconsistency and state repair. The execution of an stateful service may reach inconsistencies (violation of the invariant) that can be repaired in a completely automatic manner. If a repair is not possible, the transaction is undone.

In relation to future work, we plan two actions: (1) the implementation of the formalism in order to gain knowledge about its practical feasibility and then (2) to face us with the so-called service discovery problem, that is, the manner of selecting mechanically a TS from a service request (goal). This question is rather complex but relevant if we want to improve semantic searching of services on the web.

## References

[AHV95] S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[BK95] A. Bonner and M. Kifer. *Transaction Logic Programming. (or, A Logic of Procedural and Declarative Knowledge).* Technical Report CSRI-323. November 1995. Computer System Research Institute. University of Toronto.

[BK98] A. Bonner and M. Kifer. A Logic for Programming Database Transactions. In J. Chomicki and G. Saake, editors, Logics for Databases and Information Systems, chapter 5. Kluwer Academic Publishers, March 1998.

[Bon97] A.J. Bonner. Transaction Datalog: a Compositional Language for Transaction Programming. In Proceedings of the Sixth International Workshop on Database Programming Languages (DBPL), Estes Park, Colorado, August 1997. Lecture Notes in Computer Science, volume 1369, Springer-Verlag, 1998, pages 373-395.

[Dev90] Y. Deville, Logic Programming. Systematic Program Development (Addison-Wesley, 1990).

[DNR08] A. Deustch, A. Nash and J. Remmel. The Chase Revisited. PODS'08 June, 2008 Vancouver, Canada.

[FKM05] Data exchange: semantics and query answering. R. Fagin, P. G. Kolaitis, R. J. Miller, L. Popa. Theoretical Computer Science 336 (2005) 89-124.

[GuY10] Service Matchmaking Revisited: An Approach Based on Model Checking. A. Gunay, P. Yolum. Journal of Web Semantics, 8 2010, pp. 292-309.

[HZB06] D. Hull, E. Zolin, A. Bovykin, I. Horrocks, U. Sattler and R. Stevens. Deciding Semantics Matching of Stateless Services. AAAI 2006. AAAI Press 2006.

[JAS10] M. Junghans, S. Agarwal and R. Studer. Towards a Practical Semantic Web Service Discovery. ESWC 2010. LNCS 6089, pp 15-29, Springer-Verlag 2010.

[KLP05] U. Keller, R. Lara, A. Polleres and D. Fensel. Automatic Location of Services. ESWC 2005. LNCS 3532 Springer 2005.

[KlK12] M. Klusch, P. Kapahnke. The iSeM Matchmaker: A Flexile Approach for Hybrid Semantic Service Selection. Journal of Web Semantics, 15 2012, pp.1-14.

[LiH04] L. Li, I. Horrocks. A Software Framework for Matchmaking based on Semantic Web Technology. Int. J. Electronic Commerce 8(4), 39-60, 2004.

[Llo87] J. W. Lloyd, Foundations of Logic Programming 2nd ed. (Springer-Verlag, 1987).

[SaP07] M. Sabou J. Pan. Towards Semantically Enhaced Service Repositories. Journal of Web Semantics, 5, 2007, pp. 142-150.

[WSDL01] Web Services Description Language (WSDL) 1.1. W3C Submission. http://www.w3.org/TR/wsdl, March, 2001.

[WSML08] The WSML working group members. WSML Language Reference. http://www.wsmo.org/wsml, 2008

[WSMO08] The WSMO working group members. Web Service Modelling Ontology (WSMO). http://www.wsmo.org/ 2008