Hindawi
Scientific Programming
Volume 2018, Article ID 4094951, 20 pages
https://doi.org/10.1155/2018/4094951

Research Article

An Improved Abstract State Machine Based Choreography Specification and Execution Algorithm for Semantic Web Services

Shahin Mehdipour Ataee and Zeki Bayram

Eastern Mediterranean University, Famagusta, Northern Cyprus, Mersin 10, Turkey

Correspondence should be addressed to Shahin Mehdipour Ataee; shahin.mpa@gmail.com

Received 13 June 2017; Revised 20 September 2017; Accepted 4 October 2017; Published 24 January 2018

Academic Editor: Mario Alviano

Copyright © 2018 Shahin Mehdipour Ataee and Zeki Bayram. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

We identify significant weaknesses in the original Abstract State Machine (ASM) based choreography algorithm of Web Service Modeling Ontology (WSMO), which make it impractical for use in semantic web service choreography engines. We present an improved algorithm which rectifies the weaknesses of the original algorithm, as well as a practical, fully functional choreography engine implementation in Flora-2 based on the improved algorithm. Our improvements to the choreography algorithm include (i) the linking of the initial state of the ASM to the precondition of the goal, (ii) the introduction of the concept of a final state in the execution of the ASM and its linking to the postcondition of the goal, and (iii) modification to the execution of the ASM so that it stops when the final state condition is satisfied by the current configuration of the machine. Our choreography engine takes as input semantic web service specifications written in the Flora-2 dialect of F-logic. Furthermore, we prove the equivalence of ASMs (evolving algebras) and evolving ontologies in the sense that one can simulate the other, a first in literature. Finally, we present a visual editor which facilitates the design and deployment of our F-logic based web service and goal specifications.

1. Introduction

The idea of using service oriented architecture (SOA) to form an IT infrastructure for carrying out business-to-business interactions has gained a lot of attention in the last 15 years. In this context, service composition has been studied and analyzed in many researches. Two important and complementary aspects of service composition are *service orchestration* and *service choreography*.

Service orchestration is the process of coordinating two or more services for the purpose directing them toward the accomplishment of a specific task in a centralized way. Service choreography however (as pointed out in [1–3] as well) does not have a unique understanding among researchers. In the Business Process Modeling (BPM) community, choreography is known as a general predefined collaboration scenario that should be agreed upon and adhered to by two or more web services in order to accomplish a business goal, without the presence of a central coordinator (unlike orchestration). The choreography engine checks whether the participants are passing proper messages at the right time and in the correct order specified by the choreography designers [4]. In this paradigm, choreography is considered as a global collaboration, rather than bidirectional interaction between a service requester and a service provider.

The global collaboration view forms the basis of modeling languages such as WSCI [5], WS-CDL [6], BPEL4Chor [7, 8], Let's Dance [9], Multi-Agent Protocol (MAP) [10], and BPEL^{gold} [11]. The dominant common features of this type of choreography languages are

(i) being process-driven: a service is modeled as a process composed of series of milestones,

(ii) having no role for goals: there is no concept of a service requester,

(iii) staticity: the overall sequence of events is specified at design time,

(iv) being nonsemantic: no ontology is used, which is the main feature of a semantic system [12],

(v) not having any inferencing capability.

In contrast, the concept of choreography among semantic web service developers is understood as the behavioral interface of a *single* web service when it is interacting with its client (so-called *goal*), which results in an automatic, flexible conversation (dialog) between the two. In other words, it is the implicit communication protocol between two (and only two) counterparts that should be dynamically carried out in order to realize a conversation. The role of choreography engine is to dynamically control the conversation and see whether it is successful or not. This concept has been named *choreography interface* in [13].

In the rest of this article, we use the term *choreography* only in the semantic web sense. Our work also falls strictly in the semantic web view of choreography and consequently is not directly comparable to choreography languages adapted by the BPM community.

To fully automate service choreography, there is the need for unambiguous, computer processable semantics that can be used for automated reasoning [13]. A well-known semantic web service framework is Web Service Modeling Ontology (WSMO) [14]. In WSMO, the specification and behavior of the service provider (web service) and the service requester (goal) are described using a rich semantic notation. WSMO *choreography* is a component of WSMO *interface* that deals with choreographing of WSMO-based web services and goals.

Although WSMO based its choreography algorithm on the well-founded theory of *Abstract State Machines* (ASMs) [15, 16], the algorithm is less than perfectly suited for the job at hand. In fact, our literature search has failed to reveal any choreography engine that implements it exactly as specified. For example, current implementations (such as WSMX [17] and IRS-III [2]) do not fully adhere to the ASM theory. Our own investigation into the algorithm has revealed certain important shortcomings which make it unsuitable for driving the correct interaction between goal and web service choreographies, and helps explain its lack of proper adaptation in existing choreography engine implementations.

In previous work [18], we used F-logic [19] for the WSMO *capability* specification of web services and goals. A capability involves *pre-* and *postconditions* of web services and goals. Capabilities are used in the *service discovery* stage. In the current work, we focus on WSMO *interfaces*, which mainly include choreography specification and are used in the *service interaction* stage. Our main contributions here can be summarized as (i) rectifying the original ASM-based choreography algorithm, (ii) proposing an F-logic specification of WSMO goal and web service choreographies as an effective alternative to the current specifications in WSML [20] and OCML [21], (iii) implementing the rectified choreography algorithm in Flora-2 [22, 23] with novel technics that adhere to theory of ASMs (missing in other implementations), (iv) validating the implemented Flora-2 engine through several realistic scenarios, (v) developing a visual editor to facilitate the design and deployment of semantic web services in a subset of the Flora-2 language that we adopted as our specification language, and (vi) proving the equivalence of ASMs (also known as *evolving algebras* [24]) and WSMO choreography specifications (commonly called *evolving ontologies* [25]) by providing appropriate mappings between them.

The rest of this article is organized as follows. In Section 2, the preliminaries are briefly explained, including ASM theory, F-logic, Flora-2, and WSMO choreography concepts, which are needed to understand the subsequent sections. In Section 3, we give the existing WSMO choreography execution algorithm, point out its weaknesses, and present a rectified version. In Section 4, the general form of goal and web service specifications in F-logic is given, as well as the implementation details of the improved algorithm, which works on the specifications. In Section 5, we provide a realistic choreography example and explain it. Section 6 is the discussion, where we describe how our choice of Flora-2 as the specification language helps resolve the data granularity mismatch problem that can occur between the goal and web service. Section 7 contains related work on choreography specification, matching, and execution, with appropriate comparison with our approach. Finally, in Section 8 we have the conclusion and future research directions. In Appendix A in the Supplementary Materials, we illustrate the developed visual editor for semantic web service specification in a subset of Flora-2. In Appendix B in the Supplementary Materials, we give the grammar of the web service and goal specifications in EBNF [26] notation. Appendix C in the Supplementary Materials contains seven more choreography scenario examples, demonstrating the capabilities of the proposed choreography specification and engine. Appendix D in the Supplementary Materials contains a description of the predicates used in the implementation, which itself is available for download at [27]. Lastly, in Appendix E in the Supplementary Materials we describe a scheme for converting JSON content into its Flora-2 equivalent, a step that will be useful in grounding Flora-2 specified semantic web services into RESTful web services.

2. Preliminaries

In this section, essential concepts that have been used in this work are briefly reviewed.

2.1. Abstract State Machine (ASM). ASMs or evolving algebras were first introduced by Gurevich [15, 16]. ASM theory says that every algorithm can be modeled as a step-by-step evolving system containing two main components: a state signature which represents the current status of the system and a finite set of transition rules which determine the next state of the system based on the current one. Finite State Machines (FSMs) [28] can be seen as a specific instance of ASMs. An important point about ASMs is that transition rules are applied in parallel at each evolution step and they are categorized into three types: *if-rule(-else)*, *forall-rule*, and *choose-rule* [29].

ASMs are generally categorized into Basic ASMs and Multiagent ASMs. The discussion here is based on Basic ASMs, similarly to WSMO which configured and extended

Basic ASMs to model choreographies. In WSMO, evolving ontologies (ontologized ASM [30]) are used to represent the state of the choreography, instead of the evolving algebra of ASMs. It turns out that evolving ontologies are equivalent to evolving algebras, and we prove this in Section 3.2. For further reading about ASMs, the reader is referred to [24, 31, 32].

2.2. WSMO and WSMO Choreography. Web Service Modeling Ontology (WSMO) is a comprehensive framework that has the aim of enabling automatic service discovery, invocation, and composition [14]. It identifies four major concepts in semantic service oriented architecture: *ontology* (provides the common terminology between goals and web services), *web service* (models the functionality of the web service at a high, semantic level), *goal* (models the request of client at a high, semantic level), and *mediator* (resolves different types of possible incompatibilities, including process and terminological, between goals and web services).

The concept of web service in turn contains *nonfunctional properties*, *ontologies*, *mediators*, *capability*, and *interface* elements. The functionality of a WSMO web service is described by the *capability* element which contains *precondition, postcondition*, *assumption*, and *effect*. *Precondition* specifies what the web service requires from the goal before it can start execution. *Postcondition* specifies what the web service can provide to the client (i.e., goal) upon successful completion of its execution. A*ssumption* is the state of the world which must hold true before the web service can be called. *Effect* is the change(s) caused to the state of the world through the execution of the web service. A WSMO web service guarantees its postcondition and effect if its precondition and assumption are true. This feature of WSMO web services is used for automatic discovery purposes.

Choreography is part of web service *Interface* element which specifies the behavioral interaction of the web service with its client.

2.3. F-Logic. Introduced by Kifer and Lausen, Frame Logic [19] or in short F-logic is a formalism that integrates first-order logic and the object-oriented paradigm [19]. Equipped with predicate calculus [33], F-logic can easily model concepts, facts, rules, and specially ontologies in a very declarative fashion and is considered a rule-integrated ontological language [34]. In F-logic, classes and subclasses are modeled as concepts and subconcepts, respectively. Also, data members are represented by attributes and their assigned values. Detailed discussion about F-logic can be found in [35–37].

2.4. Flora-2. Flora-2 is a powerful, integrated system based on F-logic, HiLog [38, 39], and Transaction Logic [40]. It offers syntax similar to F-logic and by using the XSB inferencing engine [41], it can do reasoning on the facts and knowledge represented in F-logic or HiLog. The variety and multiplicity of logic and predicate operators make Flora-2 a powerful reasoning system. Moreover, Flora-2 is being continuously extended and developed [42, 43] and it can be integrated with Java through the provided APIs [44]. We use the latest version of Flora-2 [22] to implement a new semantic web service choreography engine.

2.4.1. Flora-2 Syntax Summary. In Table 1, some of the more prominent constructs of Flora-2 syntax, which we use in the choreography engine implementation, are given.

2.4.2. Flora-2 Main, Built-In, and User Modules. Flora-2 keeps knowledge in logical storage places named *modules*. By default, all the information and knowledge is stored in the *main* module. The user can create an arbitrary number of independent modules for organizing knowledge (referred to as *user-defined* modules). Flora-2 also has *built-in* modules which contain predefined predicates, such as the `\prolog` module where useful Prolog [46] utility predicates are kept.

3. ASM-Based Choreography in WSMO

The state-based model of WSMO choreography is inspired by ASM theory. Choreography working group has chosen ASM because of the following features [47]:

(i) Minimality: ASMs are based on a small assortment of modeling primitives.

(ii) Expressivity: ASMs can model arbitrary computations.

(iii) Formality: ASMs provide a formal framework to express dynamics.

Moreover, steps of evolution in ASM match the stepwise nature of interaction between a web service and its users.

To apply ASM theory in practice, WSMO choreography authors modified Basic-ASM concepts in several aspects. The concept of signature in ASM has been replaced by the concept of WSMO ontology, which involves *concepts*, *attributes*, *relations*, and *axioms*. The concept of dynamic functions in ASM has been replaced by dynamic changes of instances and their attribute values, effectively, replacing the concept of evolving algebra [15] by the concept of evolving ontology [25]. This replacement, however, has not been formally justified so far, so in Section 3.2, we prove the equivalence of *evolving algebras* and *evolving ontologies*, filling this gap.

3.1. Specification of Choreographies in WSMO. In WSMO, the choreography concept has four components: *nonFunctionalProperties*, *stateSignature*, *state*, and *transitionRules*. *nonFunctionalProperties* refers the nonfunctional properties of the choreography, such as its author, date of creation, and other metainformation about the choreography, described in detail in [48].

3.1.1. The Modes of Concepts. Modes are used to define precise access rights on instances of concepts to be exercised by the environment (the client in this context) and the machine (the web service in this context). Five modes are defined in WSMO choreography, namely, *static*, *controlled*, *in*, *shared*, and *out*, which control reading and writing access of the machine and the environment. Table 2 summarizes the encapsulation effect of each mode [14].

3.1.2. State. State component of WSMO choreography represents the dynamic part of the ongoing choreography instance and consists of actual objects, that is, instances of concepts,

Table 1: Summary of Flora-2 syntax [45] used in the implementation.

Flora-2 syntax	Meaning
`concept[|attribute => type|].`	Defines a concept, its attributes, and their types
`object[attribute -> value].`	Specifies the value of an instance attribute
`subconcept::concept.`	Defines inheritance between two concepts
`object:concept.`	Instance declaration
`${...}`	Reifies any kind of object in Flora-2
`~`	Metaunification operator
`\object`	Top-most object in Flora-2
`\if...\then...\else`	If-then-else formula
`Predicate(parameters)`	A tabled predicate
`%Predicate(parameters)`	A nontabled predicate
`?variable`	Logic variable
`?_`	Don't care logic variable
`?_name`	Don't care identifiable logic variable
`,(\and) ;(\or) \+`	Logical AND, Logical OR, Negation as Failure
`L :- R`	Clause definition
`!`	The cut operator
`\prolog`	Module containing Prolog predefined predicates
`\btp`	Module containing embedded base-types or predicates
`setof{?X | any formula containing variable ?X}`	Generates the list of all X's such that the formula where it occurs as a free variable is true
`//comment /*comment*/`	Comments

Table 2: The modes of concept in WSMO choreography.

If the concept is	Static	Controlled	In	Shared	Out
Web Service can	Read	Read/Write	Read	Read/Write	Read/Write
Goal can	Read	-	Read/Write	Read/Write	Read

as well as instances of relations. The state is changed through the insertion of new instances, deletion of instances, or the update of attribute values in concept instances.

3.1.3. Transition Rules. In WSMO, transition rules are in the form of the following expressions:

(i) **If** guard **then do** rules.

(ii) **Forall** variables **with** guard **do** rules.

(iii) **Choose** variables **with** guard **do** rules.

In (i), guard should be an arbitrary logic formula without free variables; if the guard is true, then the rules on its right-hand side are executed. In (ii), the list of variables after *Forall* should be free in the guard and the scope of these variables extend to the rules on the right-hand side. For every value of the variables such that the guard becomes true, the actions on the right-hand side are executed in parallel. In (iii), for only one instantiation of the free variables in the guard (chosen at random), the actions on the right-hand side are executed [49].

In accordance with the original ASM definition, all rules at the top level, as well as rules on the right-hand side, are meant to be executed in parallel. Note that, in the case of a *Forall* rule, an extra level of parallelism is introduced through the different instantiations of the variables listed after the **Forall** keyword.

Rules on the right-hand side, also called *actions*, are categorized into three basic update functions as follows:

(i) Adding a fact

(ii) Deleting a fact

(iii) Updating a fact (changing the values of the attributes).

3.2. Relationship between Evolving Algebras (ASMs) and Evolving Ontologies. In ASM theory, functions can be partial and can evolve as time passes. For a function, not only can its previous range change, but also members of the domain that were not mapped to values in the codomain under the function can be mapped to a value at a later stage. For example, if `f(a)` is `1`, `f(b)` is `2`, and `f(c)` is undefined, after a while (based on the transition rules) `f` might change in a way that `f(a)` remains `1`, but `f(b)` is mapped to `3` and `f(c)` is mapped to `4`.

Evolving ontologies deal with objects, attributes, and values of attributes, as well as relations. The state of the system at a given moment is determined by the objects that exist

at that moment, the specific values of the attributes of each existing object, and instances of relations. Ontologies evolve through the insertion/deletion of objects and relations, as well as updates to the values of object attributes.

It turns out that evolving algebras and evolving ontologies are in fact equivalent to each other in the sense that, through appropriate mappings, a choreography engine can simulate ASM, and vice versa. Below, we give a brief formal definition of Abstract State Machines, evolving ontologies, and the mappings between the two that allow each one to simulate the other.

Definitions [50]. In an ASM state, *domains* (also called *universes*) contain data, with functions defined over the domains. The *superuniverse* is the union of all domains. Relations are treated as Boolean valued functions, and domains are used interchangeably with characteristic functions (e.g., if $b \in A$, then $A(b)$ = True). A *vocabulary* Σ is collection function names. Nullary function names (those with zero parameters) are called *constants*. The pair $(f, (a_1, \ldots, a_n))$, where f is a function name and $(a_1, \ldots, a_n)$ are parameters that the function can be applied to, is called a *location*. Every ASM vocabulary is assumed to contain the static constants *undef*, *True*, and *False*. A *state* U of the vocabulary Σ consists of (i) the *superuniverse* of U (which we shall call X or $|\mathbf{U}|$) and (ii) *interpretations* of the function names in Σ. For any n-ary function name f in Σ, its interpretation $\mathbf{f}^{\mathbf{U}}$ is a function from X^n into X. If $\mathbf{c}$ is a constant of Σ, $\mathbf{c}^{\mathbf{U}}$ is an element of X. *undef* is the default value X and represents and undetermined object. The notions of *terms*, *formulas*, *substitutions*, *quantifiers*, *logical connectives*, and *interpretations* of terms and formulas are exactly the same as in first-order logic. Each *state* is an *algebra* in the mathematical sense of the word, with the exception that, for f in Σ, $f(v_1, \ldots, v_n) = undef$ is permitted (i.e., functions can be partial). Furthermore, as the ASM is executing its transition rules, function interpretations can change over time, leading to the term *evolving algebras*. We should note, however, that the update rule in ASMs is of the form $f(t_1, \ldots, t_n) := t_{n+1}$, where both the arguments and result of the function application are *terms*, not *values* in $|\mathbf{U}|$. From a logic programming point of view, where we use terms to represent data, we can reasonably assume that $|\mathbf{U}|$ is nothing more than H_∞, the Herbrand universe (i.e., the set of all ground terms) [51], and functions really map (tuples of) terms to other terms in H_∞.

On the ontology side, we have *concepts* (*classes* in programming language parlance), *instances* (also called *individuals*) that are members of concepts, *attributes* that are used to describe properties of instances, *relations* that relate instances to one another, and *axioms* (logic statements that say what is true in the domain of application). $\mathbf{C}$ is the set of concepts, $\mathbf{T}$ is the set of all terms, $\mathbf{I} \subseteq \mathbf{T}$ is the set of object identifiers denoting instances, $\mathbf{R}$ is the set of relation names, and $\mathbf{A}$ is the set of attributes. The term *evolving ontologies* comes about because update rules are used to change the values of object attributes and add/delete relation or concept instances to/from the *working memory* to obtain a modified working memory. The complete contents of the working memory represent a state.

3.2.1. Simulation of Choreography Engine Execution via ASM. One way to view members of $\mathbf{A}$ is as functions with domain $\mathbf{I}$ and codomain $\mathbf{T}$; that is, $\mathbf{a} \in \mathbf{A} : \mathbf{I} \rightarrow \mathbf{T}$. Since $\mathbf{I} \subseteq \mathbf{T}$, it is also true that $\mathbf{a} \in \mathbf{A} : \mathbf{T} \rightarrow \mathbf{T}$, with the provision that *if* $\mathbf{e} \in (\mathbf{T} - \mathbf{I})$ *then* $\mathbf{A}(\mathbf{e}) =$ **undef**; that is, $\mathbf{A}$ is not defined for elements of $\mathbf{T}$ that are not in $\mathbf{I}$. For any ontology with attribute set $\mathbf{A}$, the actions of the choreography execution engine can be simulated by an ASM with vocabulary $\Sigma = \mathbf{R} \cup \mathbf{A}$. Table 3 gives the action to be performed by an ASM that simulates the choreography engine execution. Remember that predicates in ASMs can be represented as Boolean valued functions.

Definition 1. An ASM state S_A is said to *correspond to* an ontological state S_C *iff*

(i) whenever $b \in I$ and $a \in A$ and `b[a->c]` is true in S_C, then $a \in \Sigma$ and $a(b) = c$ in S_A,

(ii) whenever $r \in R$ and $r(a_1, \ldots, a_n)$ is true in S_C, then $r \in \Sigma$ and $r(a_1, \ldots, a_n)$ = True in S_A.

Definition 2 (move of an ASM). $\boldsymbol{\sigma}$ ~> $\boldsymbol{\rho}$ denotes one step move of an ASM when it goes from *abstract* state $\boldsymbol{\sigma}$ to *abstract* state $\boldsymbol{\rho}$. $\boldsymbol{\sigma}$ ~>$^{\mathbf{n}}$ $\boldsymbol{\rho}$ denotes the fact that an ASM goes from *abstract* state $\boldsymbol{\sigma}$ to *abstract* state $\boldsymbol{\rho}$ in $\mathbf{n}$ moves.

Definition 3 (move of a choreography engine). $\boldsymbol{\mu}$ => $\boldsymbol{\beta}$ denotes one step move of the choreography engine when it goes from *ontological* state $\boldsymbol{\mu}$ to *ontological* state $\boldsymbol{\beta}$. $\boldsymbol{\mu}$ =>$^{\mathbf{n}}$ $\boldsymbol{\beta}$ denotes the fact that the choreography engine goes from *ontological* state $\boldsymbol{\mu}$ to *ontological* state $\boldsymbol{\beta}$ in $\mathbf{n}$ moves.

Theorem 4 (simulation of choreography engine execution by ASM actions). *Let σ be a state of an ASM and μ be an ontological state. Let $\Sigma = \mathbf{R} \cup \mathbf{A}$. If $\boldsymbol{\sigma}$ corresponds to $\boldsymbol{\mu}$ and $\boldsymbol{\mu}$ =>$^{\mathbf{n}}$ $\boldsymbol{\beta}$ then $\boldsymbol{\sigma}$ ~>$^{\mathbf{n}}$ $\boldsymbol{\rho}$ and $\boldsymbol{\rho}$ corresponds to $\boldsymbol{\beta}$, provided that each action of the choreography engine is replaced by its corresponding action as specified in Table 3.*

Proof. Straightforward induction can be done on the number of moves performed by the choreography engine (omitted). □

3.2.2. Simulation of ASM Execution via Choreography Engine. The execution of any ASM can be simulated by a choreography engine using evolving ontologies. The requirement is that functions need to be represented somehow in the ontology. The reverse of the mapping given in the previous Section 3.2.1 (i.e., whenever $a(b) = c$ in S_A then $b[a$ -> $c]$ is true in S_C) does not always work, since a function may be n-ary, where $n > 1$. One possibility is to map functions of the ASM to relations of the ontology. Specifically, every n-ary function of the ASM can be considered an $(n + 1)$-ary predicate of the ontology. Another possibility is to have an object called *func*, with locations as attribute names, and the value associated with the location as the value of the attribute. For example, if $f(a, b, c) = d$ in the ASM, then $func[f(a, b, c)$ -> $d]$ is its representation in the ontology. We use the second approach, since we can update objects in an atomic manner, but relation instances are not updatable in one step.

Table 3: Simulating a move of the choreography engine with an ASM.

Choreography engine action	ASM action
Insertion of `b[a->c]`	`a(b) ≔ c`
Deletion of `b[a->c]`	`a(b) ≔ undef`
Update of the `a` attribute of `b` to value `c`	`a(b) ≔ c`
Insertion of relation instance $r(a_1, \dots, a_n)$	$r(a_1, \dots, a_n)$ ≔ `True`
Deletion of relation instance $r(a_1, \dots, a_n)$	$r(a_1, \dots, a_n)$ ≔ `False`

Table 4: Simulating a move of the ASM with a choreography engine.

ASM action	Choreography engine action
$f(b_1, \dots, b_n) \coloneqq b_{n+1}$ ($b_{n+1} \neq$ `undef`)	Update the `func` object's $f(b_1, \dots, b_n)$ attribute to b_{n+1} (if the attribute $f(b_1, \dots, b_n)$ does not exist for `func`, it is created in the update procedure)
$f(b_1, \dots, b_n) \coloneqq$ `undef`	Delete `func`$[f(b_1, \dots, b_n)$`-> ?_`$]$, where `?_` is a free variable (if $f(b_1, \dots, b_n)$ does not exist in `func`, nothing is done in the delete procedure)

Definition 5. An ontological state S_C is said to *correspond to* an ASM state S_A *iff* whenever $f \in \Sigma$ and $f(b_1, \dots, b_n) = b_{n+1}$ in S_A then $func \in \mathbf{I}$ and $f(b_1, \dots, b_n) \in \mathbf{A}$ and $func[f(b_1, \dots, b_n) \text{ -> } b_{n+1}]$ is true in S_C.

Theorem 6 (simulation of ASM execution by choreography actions). *Let μ be an ontological state and σ be a state of an ASM. Let $A = \{f(a_1, \dots, a_n) \mid (f, (a_1, \dots, a_n))$ be a location of the ASM$\}$. Let $I = \{func\}$. If $\boldsymbol{\sigma} \sim>^{\mathbf{n}} \boldsymbol{\rho}$ and $\boldsymbol{\mu}$ corresponds to $\boldsymbol{\sigma}$ then $\boldsymbol{\mu} =>^{\mathbf{n}} \boldsymbol{\beta}$ and $\boldsymbol{\beta}$ corresponds to $\boldsymbol{\rho}$, provided that each action of the ASM is replaced by its corresponding action as specified in Table 4.*

Proof. Straightforward induction can be done on the number of moves performed by the ASM (omitted). □

3.3. Choreography Matching Algorithm in WSMO. The original algorithm of WSMO choreography is about validation of a choreography interface run. As such, it provides only indirect operational semantics of how a choreography engine should run. For the sake of completeness, it is given in Algorithm 1, exactly as it is in [49]. In the algorithm, an update set is *not consistent* if it contains an insertion and a deletion of the same data simultaneously; otherwise it is *consistent*. A run is *terminated* if either (a) there is an update set U associated with S_n, O, and T such that U is not consistent with S_n with respect to O, or (b) there is an update set U associated with S_n, O, and T such that $S_n = U(S_n)$, where $U(S_n)$ denotes the state obtained after the actions specified in the update set U are implemented.

3.4. Problems with the WSMO Choreography Algorithm. The algorithm presented in Section 3.3 verifies whether the choreography run is valid without addressing the role of the client of the web service. Indeed, by definition, choreography should model the interactive behavior of a web service from the client's point of view, which is not truly addressed by the algorithm.

This algorithm has three major missing ingredients that make it an incomplete way of specifying client interaction with the web service.

(i) Choreography specification of the goal is completely ignored.

(ii) It is not clear what the initial state (S_0) should be. In the context of choreography, it should be state of the world, together with what the client can provide as input through its precondition component.

(iii) Its termination condition happens either when there are no more valid actions to take, or when it reaches a stable state, that is, no more changes are possible to the current state. A terminated run, however, does not allow one to draw any conclusions regarding the suitability of the web services for a given client, since it is possible that the client requirements are not satisfied by the state in which the run terminated (whatever the reason for termination is). On the other hand, suppose a run is infinite, but at the same time an intermediate state reached in the execution is satisfactory from the client's point of view, at which point the execution should actually stop. This highlights the fact that the present algorithm overlooks the obvious link between the *capability* required by the goal and web service choreography.

Inability of the present algorithm to determine whether the client and web service are compatible can be summarized with the observation that no formal relationship is established between what the client provides and the initial state of the choreography, as well as the requirements of the client and the termination condition.

These deficiencies in the algorithm are rectified in Section 3.5 by (i) taking into account the client choreography specification in addition to the web service choreography specification, (ii) establishing the connection between the initial state of the world plus the input provided by the client and initial state, and (iii) defining what a *successful run* is by taking into account the requirements of the client.

3.5. Improved Choreography Execution Algorithm. In this section, a modified algorithm for choreography execution is presented that rectifies the deficiencies identified in the original algorithm given in Algorithm 1. The rectified algorithm

A choreography interface run ρ is defined as a sequence of states $(S_0, \ldots, S_n)$.
Given a choreography interface $CI = (O, T, S)$ such that S is consistent with O, a choreography interface run $\rho = (S_0, \ldots, S_n)$ is *valid* for CI iff

(i) $S_0 = S$,

(ii) for $0 \leq i \leq n-1$,

  (a) $S_i \neq S_{i+1}$,

  (b) $U = \{\mathbf{add}(a) \mid a \in S_{i+1} \setminus S_i\} \cup \{\mathbf{delete}(a) \mid a \in S_i \setminus S_{i+1}\}$ is an update set associated with S_i, O and T,

  (c) S_{i+1} is consistent with O, and

(iii) the run is *terminated*.

CI: Choreography Interface O: Ontology
T: Set of Transition-rules S: Original State-signature

Algorithm 1: The original WSMO choreography algorithm [49].

A choreography interface run ρ is defined as a sequence of states $(S_0, \ldots, S_n)$.
Given a choreography interface $CI = (O_{common}, T_{webservice}, T_{goal}, S_{web_service}, S_{goal})$ such that both $S_{web\ service}$ and S_{goal} are consistent with O_{common}, a choreography interface run $\rho = (S_0, \ldots, S_n)$ is *successful* for CI iff:

(i) $S_0 = S_{web\ service} \cup S_{goal} \cup O_{common}$

(ii) for $0 \leq i \leq n-1$,

  (a) $S_i \neq S_{i+1}$,

  (b) $U = \{\mathbf{add}(a) \mid a \in S_{i+1} \setminus S_i\} \cup \{\mathbf{delete}(a) \mid a \in S_i \setminus S_{i+1}\}$ is a **consistent** update set associated with S_i, O_{common} and $T_{webservice} \cup T_{goal}$,

  (c) S_{i+1} is consistent with O_{common}

(iii) $S_n \vDash goal.post$

(iv) For all $k < n$, $\neg (S_k \vDash goal.post)$

CI: Choreography Interface
O_{common}: Common Ontology, possibly containing concept definitions, instances and axioms
$T_{webservice}$: Web Service Choreography Transition rules
T_{goal}: Goal Choreography Transition-rules
$S_{web\ service}$: Local state of the web service, possibly consisting of instances and axioms contributed to the common *working memory (WM)* of the choreography execution engine
S_{goal}: Instances implied by the goal pre-condition that are contributed to the common *working memory* of the choreography execution engine

Algorithm 2: Rectified choreography matching algorithm.

(Algorithm 2) takes into account the pre- and postconditions of the goal, making it stop when the goal can be satisfied with the current state. The original implicit style has been kept in order to highlight the differences better.

Our algorithm starts with an initial state consisting of the facts and instances implied by the goal precondition, facts, instances, and axioms contributed by the local state of the web service, as well as the facts, instances, and axioms contained in the common ontology. Significantly, we note that the concept of a *valid* choreography interface run is replaced by a *successful* choreography interface run. At each iteration, the update set is computed, and provided that it is consistent[1], the next state of the system is obtained through the application of the actions in the update set. The execution of the choreography engine terminates successfully at the earliest state which logically implies the goal postcondition. Any other termination signifies failure and can happen if (i) the execution engine reaches a stable state (i.e., the state remains unchanged by the application of the transition rules) which does not logically imply the goal postcondition, or (ii) the update set is not consistent at any stage, or (iii) a state is reached that is not consistent with the common ontology.

The differences between the improved algorithm and the original algorithm stand out: the concept of a *valid* choreography interface run, which says nothing about the actual suitability of the web service to satisfy the goal demand, is abandoned in favor of a *successful* choreography interface run, which does give useful information regarding such suitability. The initial state of the system is linked directly to the input provided by the goal precondition, reflecting the actual state of affairs in the real world, and the final state is linked directly to the postcondition of the goal. Consequently, it becomes possible to show not only that a web service can satisfy the demands of the goal, but

```
goalName:Goal.
goalName[
   importOntology -> address of local ontology,
   capability -> ${
       pre -> ${ Conjunction of frames and predicates },
       post -> ${ F-logic expression }
   },
   gRule(R01):ForallRule -> ${
       \if ( F-logic expression )
       \then ( Actions ) },
   gRule(R02):ChooseRule -> ${
       \if ( F-logic expression )
       \then ( Actions ) },
   ⋮
].
```

Box 1: General form of goal specification in Flora-2.

also that there is an actual interaction sequence between the client (i.e., goal) and the web service which results in such satisfaction. Furthermore, both goal and web service choreography specifications participate in the choreography execution run.

Given the semantic specifications of a goal and web service, as well as imported ontologies, the job of the choreography engine is to determine if a *successful* run is possible.

4. Implementing the Improved Choreography Algorithm in Flora-2

In this section, we present the specification of semantic choreographies and implementation of the improved choreography algorithm in Flora-2. It has been tested on Flora-2 Reasoner 1.2, which is available since 2017-01-30 (rev: 1258b) [22], running on Microsoft Windows 7 (64-bit).

The terms below are used in explanations in the following sections:

(i) *Working memory (WM)* is the main place for storing the state of the choreography. It keeps the whole knowledge produced by the choreography run in real time. The knowledge can be shrunk, expanded, and altered.

(ii) *Choreography round* is the sequence of actions: (i) starting with the current state of WM, (ii) determining which rules are applicable to this current state, (iii) determining the changes to WM that the application of these rules will cause, and (iv) in case there is no contradiction in the changes (to be explained later), actually implementing those changes in the current WM, leading to a new WM.

(iii) *Delta Working Memory (ΔWM)* is a temporary storage place for actions to be carried out on WM at each choreography round.

4.1. Semantic Specification of Goal and Web Service in Flora-2. As in WSMO, our Flora-2 specifications for goal and web service are composed of the elements *ontology*, *capability*, and *choreography*. *Ontology* contains frames and relations that represent knowledge used by the web service and goal. *Capability* element encloses two subelements, *pre* and *post*, which represent pre- and postconditions. Precondition of the goal can contain conjunctions of nonnegated frames and relations. Postcondition of the goal can be an F-logic expression (including all the logical connectives). Postcondition of the web service can contain conjunctions of nonnegated frames and relations. Precondition of the web service can be an F-logic expression (including all the logical connectives). Note the similarity between the precondition of the goal and the postcondition of the web service, as well as the similarity between the postcondition of the goal and the precondition of the web service. *Choreography* element is modeled by a set of transition rules. Each rule is specified by **`ruleId:ruleType -> ruleBody`**. Goal's **`ruleId`** is in form of `gRule(OID)` and web service's **`ruleId`** is in form of `wsRule(OID)`, where `OID` is any Flora-2 object identifier and is used as a label for the rule. **`ruleType`** can be either `ForallRule` or `ChooseRule`. **`ruleBody`** is a reified Flora-2 implication shown by an *if-then(-else)*[2] statement. The implication antecedent (we refer to it as left-hand side) can be an F-logic expression, and the implication consequent (we refer to it as right-hand side) contains a set of update functions (actions). Box 1 depicts the general form of a goal specification. Specification of web services is also similar.

Appendix B in the Supplementary Materials contains the EBNF grammar of goal and web service specifications.

4.2. Proposed Architecture. In this section, we describe the architecture of our choreography engine.

4.2.1. Modules of the System. Our Flora-2 solution for implementing the choreography algorithm utilizes the *main* module and two Flora-2 user modules: *WM* for keeping the current state signature of the choreography and *DeltaWM* for keeping the actions for modifying the current state into a new state. The *main* module contains the declaration of concepts,

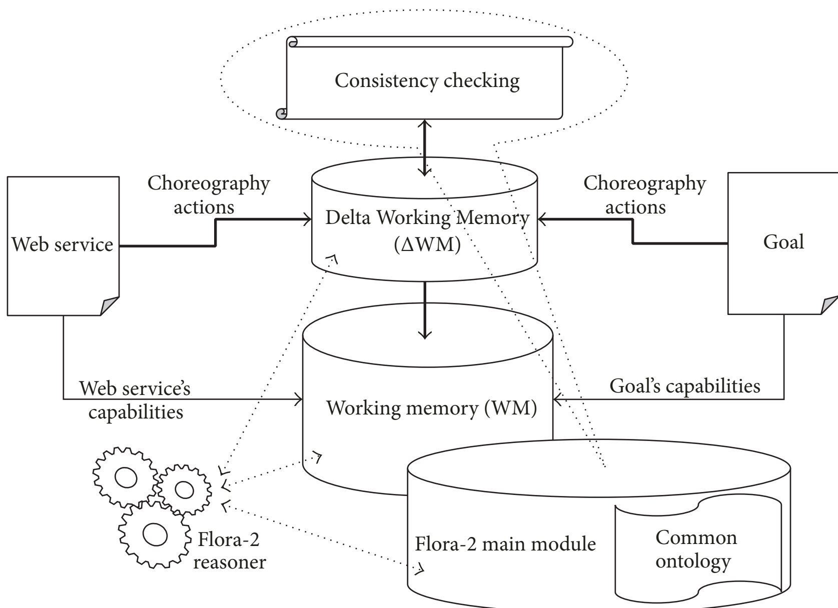


Figure 1: Architectural view of the choreography engine.

instances, and their modes provided by web service and goal, as well as the code for the choreography engine. These three modules (i.e., *main*, *WM*, and *DeltaWM*) are interconnected, as shown in Figure 1.

4.2.2. Delta Working Memory (ΔWM): Realizing ASM Parallelism. An auxiliary and transient user module named Delta Working Memory (ΔWM) is used to temporarily keep single choreography round updating actions that should be applied to the main knowledge-base (WM). In each choreography round, all the updates are aggregated into ΔWM and then ΔWM is checked whether there exists any contradiction among the requested actions (explained below). If no contradiction is detected, then all the updates are carried out on the WM, evolving it into a new conflict-free state.

4.2.3. Deterministic Choreography and Contradiction. As mentioned before, the transition rules must be applied in parallel. In the absence of any consistency checking, the rules can do contradictory actions which should be prevented.

The three kinds of contradiction that can occur in a choreography round are as follows [1]:

(i) Inserting and deleting/updating the same knowledge simultaneously

(ii) Updating knowledge which does not exist

(iii) Deleting knowledge which does not exist.

It is clear that such contradictory actions must be detected and the choreography execution must be stopped. In the case of nonexistent knowledge, it makes no sense to remove or modify it. In the case of simultaneous insertion/deletion of the same knowledge, if the choreography run is continued, then it becomes nondeterministic in an unintended way: even though theoretically updates are done in parallel, in reality they have to be done in a serial fashion, and the order in which they are carried out leads to different WM states.

```
(1) %start(?goal,?WS) :-
(2)    %initializations,
(3)    %preProcessCheckings(?goal,?WS),
(4)    %prepareModule(WM),
(5)    %prepareModule(DeltaWM),
(6)    %importOntology(?goal,WM),
(7)    %importOntology(?WS,WM),
(8)    %insertGoalPre(?goal,WM),
(9)    %runChoreography(?goal,?WS).
```

Listing 1: The top-level predicate of the choreography engine.

In each choreography round, in addition to testing for the above-mentioned issues, checking for any violation on the modes of concepts (explained in Section 3.1.1) is performed as well. If one of the participants wants to do an action on a concept which violates the concept's access mode, that action is prevented, leading to choreography execution failure.

4.3. Major Predicates of the Choreography Engine[3]. The main process is started through a call to `%start` predicate which is shown in Listing 1. The choreography engine execution begins with a call to the `%initializations` predicate on line (2), which currently resets the seed of the random number generator that is used to process the **choose** rule type. On line (3), goal and web service rules are checked for mode violations before the choreography rounds start. On lines (4) and (5), the modules *WM* and *DeltaWM* are created if they do not already exist. On lines (6) and (7), local ontologies of the goal and web service are loaded into WM. On line (8),

```
(1) %runChoreography(?goal, ?WS) :-
(2)    %proveGoalPost(?goal), !,
(3)    %watchln(['Success! '-?goal-'and'-?WS-'are'-'choreographed!']).
(4)    %runChoreography(?goal, ?WS):-
(5)    %eraseModule(DeltaWM),
(6)    %runWsRules(?WS),
(7)    %runGoalRules(?goal),
(8)    ( (%contradictory(WM,DeltaWM), !,
(9)       %watchln('Choreography failed due to CONTRADICTION.') ) \or
(10)     ( \+ %deltaMakesAChange(WM,DeltaWM), !,
(11)       %watchln('Choreography failed due to NO CHANGE.') ) \or
(12)      ( %mergeDeltaIntoWM,
(13)        %runChoreography(?goal,?WS) )).
```

Listing 2: `%runChoreography` predicate.

```
(1) %runWsRules(?WS) :-
(2)     ?_Temp = setof{ ?ruleID |
(3)     ?WS:WebService[wsRule(?ruleID):ForallRule -> ?ruleBody],
(4)     %invoke(WEBSERVICE, ?ruleBody)},
(5)     ?_Temp2 = setof{ ?ruleID |
(6)     ?WS:WebService[wsRule(?ruleID):ChooseRule -> ?ruleBody],
(7)     %invokeChoose(WEBSERVICE,?ruleBody)}.
/*----------------------------------------------------*/
(8) %runGoalRules(?goal) :-
(9)     ?_Temp = setof{ ?ruleID |
(10)     ?goal:Goal[gRule(?ruleID):ForallRule -> ?ruleBody],
(11)     %invoke(GOAL,?ruleBody)},
(12)     ?_Temp2 = setof{ ?ruleID |
(13)     ?goal:Goal[gRule(?ruleID):ChooseRule -> ?ruleBody],
(14)     %invokeChoose(GOAL,?ruleBody)}.
```

Listing 3: Running the goal and web service rules.

the goal precondition is loaded into WM, becoming part of the initial state of the choreography run, followed by a call to the predicate `%runChoreogarphy` on line (9) which tries to satisfy the goal postcondition through repeated application of goal and web service transition rules.

`%runChoreography` implements the improved choreography algorithm given in Section 3.5, employing recursion instead of iteration. The definition of the `%runChoreography` predicate is given in Listing 2. It attempts first to prove the goal postcondition with the current state of WM in lines (1)–(3); the cut (!) operator on line (2) prevents backtracking and a success message is shown to the user on line (3). If the goal postcondition is not satisfied, then the second definition of `%runChoreography` is called: ΔWM is emptied on line (5); web service and goal rules are applied, populating ΔWM with pending actions to be performed on the WM (lines (6) and (7)). On lines (8)–(11), ΔWM is checked for consistency (line (8)) as well as whether pending actions result in a new state of WM (line (10)). In the case of inconsistent changes or no change to WM, execution is stopped to prevent infinite recursion and a failure message is reported (lines (9) and (11)). Otherwise, the actions in ΔWM are applied to WM to obtain an updated WM (line (12)), and the process is repeated through a recursive call to `%runChoreography` (line (13)).

4.4. Running (Firing) the Rules. One of the key features of ASMs is that rules should be fired in parallel. We realize this by placing all the insertion, deletion, and update actions on the right-hand sides of the rules that match the current WM into ΔWM, checking them for consistency, and then applying them to the previous WM to get an updated WM. The predicates `%deltaInsert`, `%deltaDelete`, and `%deltaUpdate` represent tentative changes to WM, not actual ones, until they are verified to not cause any conflicts.

Listing 3 contains the predicates for running goal and web service rules. The Flora-2 `setof` operator is used to iterate over all rules in the choreography specification (lines (2), (5), (9), and (12)). In the case of **forall** rules, if the rule antecedent contains only ground facts, it models the ASM **if-then** transition rule type. If the antecedent contains free variables, it acts like the ASM **forall** transition rule type.

```
(1) %contradictory(?WM, ?DeltaWM) :-
(2)     ins_action(?A1)@?DeltaWM, del_action(?A2)@?DeltaWM,
(3)     %contained(?X1,?A1),%contained(?X2,?A2),?X1 = ?X2,!.
(4) %contradictory(?WM, ?DeltaWM) :-
(5)     del_action(?A)@?DeltaWM,
(6)     %convertReifiedObjectModule(?A, ?DeltaWM, ?WM, ?A_new),
       \+ ?A_new@?WM.
(7) %contradictory(?WM, ?DeltaWM) :-
(8)     update_action(?objOld,?objNew)@?DeltaWM,\+ ?objOld@?WM.
```

Listing 4: Contradictory cases.

In the case of a **choose** rule, the predicate `%invokeChoose` randomly selects exactly one ground instance of the free variables existing in the antecedent of the rule.

4.5. Contradictions in Applying the Rules. Checks for contradictory actions are implemented by the definition of the predicate `%contradictory` in Listing 4. In lines (1)–(3), simultaneous insertion and deletion of the same item are detected; in lines (4)–(6), deletion of a nonexistent item is detected; finally in lines (7)-(8), update of a nonexistent item is detected.

4.6. Access Control to Objects of Different Types. ASMs define access control modes for object manipulation. In the implementation, this access control has been enforced by checking whether a given manipulation is legal or not. This depends on the actor of the manipulation. For example, if an object has *in* mode, then only a goal can change its attributes. While invoking the rules belonging to the goal or web service, the legality of the access to the object is verified before the real action.

Listing 5 depicts a part of the implementation of access control for a goal. Before the rule is tested against the current WM, its concepts and user predicates on its left-hand side and right-hand side are extracted through the `%extractConcepts`, `%extractPredicates`, and `%filterOutPredicates` predicates (lines (4)–(6)), and access rights of the goal are verified for those concepts via `%checkAllFramesModes` and `%checkAllPredicatesModes` (lines (7), (10), (18), and (21)). If a concept is on the right-hand side of a rule, the goal must have write access to it. On the other hand, if it is on the left-hand side, only read access is enough.

`%checkAllFramesModes` (lines (24) to (27)) checks modes for a list of extracted frames through calls to `%checkFrameMode` (defined on lines (28) to (31)). In case of failure, error messages are generated on lines (32) to (35).

The complete list of predicates and their explanations are presented in Appendix D in the supplementary materials, and the full source code of the implementation is available in [27].

5. A Realistic Choreography Example

In this section, we give the choreography specification for interacting with an online flight reservation service. An autonomous software agent, acting on behalf of a human, is used to make the purchase. The behavior of the agent is described semantically in the form of a goal, with a choreography component. Similarly, the behavior of the online reservation service is described semantically as a web service, with its own choreography component. A person who wants to use the service can just provide the required information to the agent and leave the scene. The agent then interacts with the web service in accordance with its choreography, provided that it is compatible with the web service's choreography.

The essence of this scenario has been inspired by the online ticket reservation website of an actual airline, similar to Virtual Travel Agency used in [52].

The actual scenario between a human and a website providing flight reservation service is as follows. After opening the airline website, the user is able to set six items: departure city/airport, arrival city/airport, whether the trip is roundtrip or one-way (only roundtrips are considered in this case), departure date, return date, and the number of passengers. After the user submits this information, the reservation website offers some candidate flight numbers and their details, including date, time, airport, and price. The user has to choose one of the candidates. In the next step, the online ticket reservation site asks for passenger data, such as the full name, gender, and date of birth. After these items are provided by the user, the system asks for credit card information, including the holder's name, credit card number, and its CVV code. After the user provides the card specifications, the online ticket service queries the bank to validate the card. Depending on the outcome of the bank query, the flight reservation service completes the transaction and issues a ticket to the user.

Listings 6 and 7 are Flora-2 specifications of the user (goal) and the web services (reservation system), respectively. We simulate the conversation which should take place between the reservation service and the bank because it is a third party and not directly involved in the choreography.

In the precondition of the goal, the departure and arrival cities and days have been specified. The postcondition states that a reservation instance is demanded.

On the flight web service side, the first rule of choreography `wsRule(R01)` checks if there is a request for a flight consisting of all the necessary items, searches for a roundtrip

```
(1) %check(?gOrWs,?X) :-
(2)    ?X ~ ${\if ?Y \then ?Z}, !,
(3)    %reformatToString(?Y, ?YStr),
(4)    %extractConcepts(?YStr, [], ?conceptsInY),
(5)    %extractPredicates(?YStr, [], ?termList1),
(6)    %filterOutPredicates(?termList1, [], ?predicatesInY),
(7)     \if (\+ %checkAllFramesModes(?gOrWs,READ,?conceptsInY))
       \then
(8)          (writeln(['Error: Illegal access mode in '-?gOrWs])@\prolog,!,
(9)         \false),
(10)   \if (\+ %checkAllPredicatesModes(?gOrWs,READ,?predicatesInY))
       \then
(11)         (writeln(['Error: Illegal access mode in '-?gOrWs])@\prolog,!,
(12)        \false),
(13)   %decomposeRHS(?Z, [], ?allFsOrPs),
(14)   %reformatToString(?allFsOrPs, ?allFsOrPsStr),
(15)   %extractConcepts(?allFsOrPsStr, [], ?conceptsInZ),
(16)    %extractPredicates(?allFsOrPsStr, [], ?temp),
(17)   %filterOutPredicates(?temp, [], ?predicatesInZ),
(18)   \if (\+ %checkAllFramesModes(?gOrWs,WRITE,?conceptsInZ))
       \then
(19)         (writeln(['Error: Illegal access mode in '-?gOrWs])@\prolog,!,
(20)        \false),
(21)  \if (\+%checkAllPredicatesModes(?gOrWs,WRITE,?predicatesInZ))
      \then
(22)        (writeln(['Error: Illegal access in '-?gOrWs])@\prolog,!,
(23)        \false).
/*------------------------------------------------------------------*/
(24) %checkAllFramesModes(?gOrWS, ?reOrWr,[]).
(25)  %checkAllFramesModes(?gOrWS, ?reOrWr,[?F|?R]):-
(26)     %checkFrameMode(?gOrWS, ?reOrWr,?F),
(27)     %checkAllFramesModes(?gOrWS, ?reOrWr,?R).
/*------------------------------------------------------------------*/
(28) %checkFrameMode(GOAL,READ,?F):-
(29)      (?F:In \or ?F:Out \or ?F:Shared \or ?F:Static), !.
(30) %checkFrameMode(GOAL,WRITE,?F):-
(31)      (?F:In \or ?F:Shared), !.
(32) %checkFrameMode (GOAL,READ,?F):-
(33)     writeln(['Illegal GOAL READ action for' ,?F])@\prolog, !, \false.
(34) %checkFrameMode(GOAL,WRITE,?F):-
(35)     writeln(['Illegal GOAL WRITE action for' ,?F])@\prolog, !, \false.
```

Listing 5: Checking access mode.

on the specified days, and if this search is successful inserts a new triple into the knowledge-base containing two flight numbers and their total price. On the goal side, the rule `gRule(R01)`, which is of rule type **choose**, is responsible for checking the existence of any choice on the knowledge-base. As soon as some flight choices become available in the knowledge-base, this rule selects just one of them randomly and inserts this selection into the knowledge-base. Note the condition `(\+ trip:Trip)` which prevents the rule from being fired again.

The rest of the rules in the goal cover the answers to the general questions such as name, date of birth, and credit card information. On the flight web service side, rule `wsRule(R02)` checks the knowledge-base for any trip choice by the user; as soon as this choice becomes available, it asks for all the passenger identities. After receiving the answers from the goal, it then asks for credit card information and checks its validity by querying the bank. If it receives a positive reply from the bank, it puts a reservation into the knowledge-base which satisfies the goal postcondition and the choreography terminates successfully; otherwise it fails.

Table 5 shows what new knowledge is added to WM in each choreography round of a successful choreography, effectively tracing the execution of the choreography engine.

6. Discussion

Flora-2 has been used not only as the specification language of semantic web service capability and interface components, but also as the implementation language of the choreography

```
/* Local ontology stored in a separate file
Name('Peter').
DateOfBirth(19830622).
Gender('Male').
CreditCardNo('1234432156788765').
CreditCardHolder('PETER JACKSON').
CreditCardCVV(123).
*/
myGoal:Goal[
   importOntology -> '../Flight/GoalsOntology.flr',
   capability -> ${
       pre -> ${ myRequest:RequestFlight[
                   From->'Paris',
                   To->'Chicago',
                   Departure->23,
                   Return->30] },
       post -> ${ (?R:Reservation[?X->?Y])} },
   gRule(R01):ChooseRule -> ${
      \if ( tripChoice(?fl_dep,?fl_ret,?P),
          (\+ trip:Trip) )@WM
      \then (
        %deltaInsert(${trip:Trip[
                          Dep->?fl_dep,
                          Ret->?fl_ret]}) ) },
   gRule(R02):ForallRule -> ${
      \if ( (?Q:QuestionByWS[
               Name->?X,
               DateOfBirth->?Y,
               Gender->?Z])@WM,
              (Name(?N), DateOfBirth(?DoB), Gender(?G))@WM )
      \then (
          %deltaInsert(${answer:AnswerByGoal[
                              Name->?N,
                              DateOfBirth->?DoB,
                              Gender->?G]} ) ) },
   gRule(R03):ForallRule -> ${
     \if ( (?Q:QuestionByWS[
               CreditCardNo->?X,
               CreditCardHolder->?Y,
               CreditCardCVV->?Z])@WM,
              ( CreditCardNo(?CCN),
                CreditCardHolder(?CCH),
                CreditCardCVV(?CCCVV) )@WM
     )
     \then (
        %deltaInsert(${answer:AnswerByGoal[
                          CreditCardNo->?CCN,
                          CreditCardHolder->?CCH,
                          CreditCardCVV->(?CCCVV)]}) ) }
].
```

LISTING 6: Goal (the user).

execution engine itself. This choice gives the choreography developer a concise, frame based logical syntax to work with, as well as all the functionality of the underlying Flora-2 system in terms of its built-in predicates and reasoner. This is in contrast to WSML, the class of languages developed for WSMO, which has a verbose syntax, and must rely on external reasoners for all semantic computing activities, including choreography execution.

Our choice of Flora-2 as both the specification and implementation language also helps us in dealing with the granularity mismatch problem [17, 53]. As explained in [17], data granularity can be a barrier to reach a successful

```
/* Local ontology stored in a separate file
flight(F100,Paris,Chicago,23,250).
flight(F101,Paris,Chicago,23,350).
flight(F102,Paris,Chicago,25,400).
flight(F103,Chicago,Paris,29,150).
flight(F104,Chicago,Paris,30,200).
flight(F105,Chicago,Paris,30,150).
*/
FlightReservationService:WebService[
   importOntology -> '../Flight/WebServicesOntology.flr',
   capability -> ${
       pre -> ${ ?Req:RequestFlight[?X1->?Y1] },
       post -> ${ (?Res:Reservation[?X2->?Y2]) } },
   wsRule(R01):ForallRule -> ${
     \if ( (?R:RequestFlight[
                 From->?X,
                 To->?Y,
                 Departure->?Z,
                 Return->?W])@WM,
                 (flight(?fl_dep,?X,?Y,?Z,?priceDep))@WM,
                 (flight(?fl_ret,?Y,?X,?W,?priceRet))@WM,
                 (%sum(?priceDep,?priceRet,?priceTot)) )
     \then (
        %deltaInsert( ${tripChoice (?fl_dep,?fl_ret,?priceTot)} ) ) },
   wsRule(R02):ForallRule -> ${
     \if ( ?T:Trip[
             Dep->?fl_dep,
             Ret->?fl_ret] )@WM
     \then (
        %deltaInsert(${question:QuestionByWS[
                          Name->?X,
                          DateOfBirth->?Y,
                          Gender->?Z]}) ) },
   wsRule(R03):ForallRule -> ${
     \if ( ?A:AnswerByGoal[
              Name->?X,
              DateOfBirth->?Y,
              Gender->?Z] )@WM
     \then (
        %deltaInsert(${question:QuestionByWS[
                          CreditCardNo->?XX,
                          CreditCardHolder->?YY,
                          CreditCardCVV->?ZZ]}) ) },
   wsRule(R04):ForallRule -> ${
       \if ( ?A:AnswerByGoal[
              CreditCardNo->?X,
              CreditCardHolder->?Y,
              CreditCardCVV->?Z] )@WM
      \then (
        %deltaInsert(${validation:CreditCardValidation[
                            Number->?X,
                            Holder->?Y,
                            CVV->?Z]}) ) },
   wsRule(R05):ForallRule -> ${
      \if ( (BankYesNoAnswer('Yes'))@WM,
             (trip:Trip[
              Dep->?fl_dep,
              Ret->?fl_ret])@WM )
     \then (
```

Listing 7: Continued.

```
                %deltaInsert(${reservation:Reservation[
                              Number->11100,
                              Flight1->?fl_dep,
                              Flight2->?fl_ret]}) ) },
       wsRule(Bank_R01):ForallRule -> ${
           \if ( (?R:CreditCardValidation[
                    Number->?X, Holder->?Y, CVV->?Z])@WM,
                    (DB_CreditCard(?X,?Y,?Z) )@WM
           )
           \then (
             %deltaInsert(${BankYesNoAnswer('Yes')}) ) }
   ].
```

Listing 7: Web services (the reservation system and the bank).

choreography. Authors of [2] demonstrate the data granularity mismatch issue with an example: one web service requires credit card details to be sent one at a time, whereas another requires that all details are sent in single message. Frame structures in Flora-2 intrinsically solve this type of granularity issue. For example, a credit card can be defined as follows:

```
joeCreditCard:CreditCard[
    number -> "1234-5678-9012-3456",
    name -> "Joe Brown",
    CVV -> 123].
```

Internally, however, such a frame is represented as the composition of its data members, and each data member of a frame can be referred to individually, without the need to refer to other data members at the same time. Also, a frame can be built up incrementally through the addition of its data members. Consequently, the granularity level with which frames of a certain concept are handled by the goal or web service becomes insignificant: the web service or goal can provide the constituents of a frame either in piecemeal fashion in any order or as a whole at once, and its counterpart can consume it under both conditions.

7. Related Work

Semantic web service frameworks such as WSMO and OWL-S [54] use rich semantic reasoning systems to realize semantic web service choreography. They model the interaction between the client and the service as a bidirectional conversation, with implementations such as WSMX [1, 17, 55], WSMO Studio (a visual editor for WSML) [56], WSMO4J API [57], IRS-III [2, 58], and OWL-S tools [59].

OWL-S is not well aligned with the WSMO framework. It "does not provide an explicit definition of choreography, but instead focuses on a process based description of how complex web services invoke atomic web services" [2]. In [60], WSMO and OWL-S are compared in detail and the author concludes that "WSMO presents some important advantages when compared to OWL-S." Here, we point out some general issues about OWL-S:

(i) OWL-S does not properly decouple the viewpoint of service requester and service provider.

(ii) OWL-S service profile mixes the information of WSMO goal, WSMO capability, and nonfunctional properties.

(iii) In OWL-S, the requester has to formulate its request based on the descriptions of profiles.

(iv) OWL-S does not clearly define how logical expressions are used to describe conditions and results.

(v) In spite of its incompleteness, WSMO choreography provides ASM as its formal model, whereas a formal semantic OWL-S process model is still missing.

WSMX is known as the reference prototype implementation of WSMO [55]. WSMX offers a flexible architecture that can accept different components as its plug-ins. The project has been implemented in Java, can handle service and goal specifications that are written in WSML [20, 61], and uses WSML2Reasoner [62], which converts WSML into the internal representation of external reasoning engines in order to do the reasoning tasks. KAON2 [63] is the external reasoner used to deal with choreography reasoning tasks [30].

We have thoroughly investigated WSMX using publicly available documents, including published papers and source code [64, 65]. We have found that

(i) the implementation of choreography in WSMX was started but not completed,

(ii) the implementation does not support parallelism and consequently inconsistency checking is not even an issue,

(iii) the implementation does not support intentional nondeterministic behavior necessitated by the *Choose* rule type,

(iv) in the case of *if-then* rules, if more than one left-hand side (antecedent) is satisfied by the current ontology state, right-hand sides of all matching rules are executed sequentially, without any consistency check of the actions performed, resulting in behavior that depends on the order of the rules.

Table 5: Items added to WM at each choreography round.

Round 0	Name('Peter'). DateOfBirth(19830622). Gender('Male'). CreditCardNo('1234432156788765'). CreditCardHolder('PETER JACKSON'). CreditCardCVV(123). flight(F100,Paris,Chicago,23,250). flight(F101,Paris,Chicago,23,350). flight(F102,Paris,Chicago,25,400). flight(F103,Chicago,Paris,29,150). flight(F104,Chicago,Paris,30,200). flight(F105,Chicago,Paris,30,150). DB_CreditCard('876543212345678','PAUL BROWN',123). DB_CreditCard('1234432156788765','PETER JACKSON',123).
Round 1	myRequest:RequestFlight[From->Paris] myRequest:RequestFlight[To->Chicago] myRequest:RequestFlight[Departure->23] myRequest:RequestFlight[Return->30]
Round 2	tripChoice(F100,F104,450) tripChoice(F100,F105,400) tripChoice(F101,F104,550) tripChoice(F101,F105,500)
Round 3	trip:Trip[Dep->F101] trip:Trip[Ret->F105]
Round 4	question:QuestionByWS[Name-> _h592309] question:QuestionByWS[DateOfBirth-> _h592309] question:QuestionByWS[Gender-> _h592309]
Round 5	answer:AnswerByGoal[Name->Peter] answer:AnswerByGoal[DateOfBirth->19830622] answer:AnswerByGoal[Gender->Male]
Round 6	question:QuestionByWS[CreditCardNo-> _h592309] question:QuestionByWS[CreditCardHolder-> _h592309] question:QuestionByWS[CreditCardCVV-> _h592309]
Round 7	answer:AnswerByGoal[CreditCardNo->1234432156788765] answer:AnswerByGoal[CreditCardHolder->PETER JACKSON] answer:AnswerByGoal[CreditCardCVV->123]
Round 8	validation:CreditCardValidation[Number->1234432156788765] validation:CreditCardValidation[Holder->PETER JACKSON] validation:CreditCardValidation[CVV->123]
Round 9	BankYesNoAnswer(Yes)
Round 10	reservation:Reservation[Number->11100] reservation:Reservation[Flight1->F101] reservation:Reservation[Flight2->F105]

Furthermore, in the last version of WSML2Reasoner, which is used by WSMX to translate WSML logical expressions to the native language of the used reasoner, there is no translation of the *Forall* or *Choose* rule types, confirming our findings. It is clear that several of the most fundamental features of ASMs remain unimplemented in WSMX.

IRS-III (the Internet Reasoning Service: 3rd version) [2, 58, 66] provides an infrastructure that utilizes the WSMO

framework. The IRS system is composed of three major components: *server*, *client*, and *publisher*. Choreography between a client and a web service is not done directly, but through the IRS choreography engine, which acts as a broker between the available web services and user requests. IRS takes the responsibility of service discovery, mediation, communication, and invocation of the web services and provides the result for the goal; however, clients should formulate their needs to IRS in the specific representation language of IRS [58]. IRS uses the OCML ontology representation language and its server has been implemented in Lisp [67].

IRS does not adhere to either original ASM, or WSMO choreography, because

(i) transition rules of IRS are not run in parallel. In the case that more than one transition rule applies to the current state of the choreography, only one is selected using an internal function for which no further details are available,

(ii) actions in the rules are tightly coupled with the actual messages sent to the web service, which makes the choreography specification inflexible; the actual call sequence of operations is predetermined for different kinds of requests,

(iii) goals are modeled by pre- and postconditions only and do not contain a choreography component at all. The interaction is between the IRS, acting on behalf of the goal, and the web service, using solely the choreography specification of the web service,

(iv) the concept of modes is completely absent; flexible interaction between the requester and service provider that is made possible by having modes is replaced by a rigid communication model where the *actor* which has the *initiative* can update data.

In comparison with our approach, IRS does not support parallel firing of transition rules and does not check for consistency of the updates. Consequently, the next state of the ontology is not unique and depends on the choice of the rule to be fired, leading to nondeterministic behavior. Whereas we make full use of modes and enforce their compliance, as already mentioned, IRS completely ignores them. Most importantly, it ignores the obvious connection between the initial choreography execution state and the precondition of the goal, as well as the final state of the choreography execution and the goal postcondition, relying instead on the built-in predicate *init-choreography* to start the chain of rule firing and the action *end-choreography* to terminate the choreography run. If the choreography is not designed carefully, the situation where the choreography run terminates without the goal postcondition being satisfied could arise.

There are other notable works on the analysis, formalization, and modeling of choreographies. Roman et al. in [68] argue that choreographies specified in the original ASM model become quite involved when they contain contracting and enactment (additional policies and constraints imposed by web service and goal). In [68], they extend the current model of WSMO with Concurrent Transaction Logic (CTR) [69, 70] to simplify the representation; however, the CTR implementation is still in its prototype stage [71]. Bonner and Kifer in [40] discuss the main reasoning and mediation activities required for choreography and orchestration, both in general and in the context of WSMO. SWORD authors use rule-based expert systems to "determine whether a desired composite service can be realized using existing services" [72]. This approach is similar to ours in that it uses forward-chaining reasoning to develop knowledge in a stepwise manner, but for the purpose of web service composition, and not choreography.

8. Conclusion and Future Work

In this work, we identified important weaknesses in the original ASM-based choreography execution algorithm for WSMO, which prevented it from being useful in a practical way, and improved it in order to remedy the identified weaknesses. The improved ASM-based choreography execution algorithm establishes the missing connection between the capability and interface components of WSMO. We used F-logic and Flora-2 to specify ASM-based choreographies of semantic web services in a concise and logical manner and implemented a fully functional choreography execution engine based on our improved algorithm in Flora-2. The full functionality of Flora-2 and its underlying reasoning system is available for developing ontologies and writing transition rules in the choreography specification. To the best of our knowledge, this work is the first fully functional WSMO choreography implementation that fires rules in parallel, as required in the theory of ASMs, and models the ASM **if-then(-else)**, **forall**, and **choose** rule types authentically, while enforcing access modes of concepts and relations. We demonstrated the workings of our algorithm through a real-life example, concerning a flight reservation scenario, where both web service and goal choreographies were specified in our F-logic based syntax (seven more real-life examples are provided in Appendix C in the Supplementary Materials). We also developed a visual tool that helps choreography engineers write specifications in a convenient manner, reducing the chance of mistakes in the specification.

Another important contribution of our work is that we proved for the first time the equivalence of evolving algebras (ASMs) and evolving ontologies (the basis of semantic choreography engines) through the definition of bidirectional mappings between them.

For future work, we are planning to develop our system through the addition of a grounding mechanism, as well as a mediation component. We also intend to identify and classify different types of general requests and general responses among software components and present them in the form of an ontology. Such a classification scheme will help in the development of accurate and commonly acceptable choreographic interactions.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Endnotes

1. The meaning of the term *consistent* is essentially the same as in the original algorithm, but involves accessibility checks required by the declared modes of concepts as well.
2. In Flora-2, logical implication ($p \Rightarrow q$) can be shown by either `p~~>q` or `\if p \then q`. It is equivalent to $\sim p \vee q$; however, Flora-2 also offers `\if p \then q \else r` which is equivalent to $(p \wedge q) \vee (\sim p \wedge r)$; so in this case, if p is not true, the proposition's value depends on the value of r.
3. Due to space considerations, we only discuss the major predicates of the choreography engine implementation. Appendix D in the Supplementary Materials contains the list of all predicates and their functionalities. Moreover, complete choreography engine source code is available in [27].

Supplementary Materials

Appendix A: visual editor for Flora-2 based SWS specifications (VSCHOR). Appendix B: E-BNF grammar for Flora-2 goal and web service specifications. Appendix C: more choreography examples. Appendix D: choreography engine predicate list. Appendix E: converting JSON to flora-2. *(Supplementary Materials)*

References

[1] A. Haller and J. Scicluna, "D13.9v0.1 WSMX Choreography," WSMX Working, June 2005, http://www.wsmo.org/TR/d13/d13.9/v0.1/.

[2] J. Domingue, S. Galizia, and L. Cabral, "Choreography in IRS-III - Coping with heterogeneous interaction patterns in web services," in *The Semantic Web – ISWC 2005*, vol. 3729, pp. 171–185, 2005.

[3] S. Galizia, M. Stollberg, E. Kilgarriff, and L. Henocque, "WP3: Service Ontologies and Service Description D3.5 An Ontology for Web Service Choreography, Data, Information and Process Integration with Semantic Web Services," 2006.

[4] C. Peltz, "Web Service Orchestration and Choreography: A look at WSCI and BPEL4WS," Web Search Journal 2nd Edition, 2003.

[5] A. Arkin, S. Askary, S. Fordin et al., "Web Service Choreography Interface (WSCI) 1.0," August 2002, https://www.w3.org/TR/wsci/.

[6] N. Kavantzas, D. Burdett, and G. Ritzinger, "Web Services Choreography Description Language Version 1.0," W3C, November 2005, http://www.w3.org/TR/ws-cdl-10/.

[7] G. Decker, O. Kopp, F. Leymann, K. Pfitzner, and M. Weske, "Modeling service choreographies using BPMN and BPEL4Chor," in *Advanced Information Systems Engineering*, vol. 5074, pp. 79–93, Springer, Berlin, Germany, 2008.

[8] G. Decker, O. Kopp, F. Leymann, and M. Weske, "BPEL4Chor: extending BPEL for modeling choreographies," in *Proceedings of the IEEE International Conference on Web Services (ICWS '07)*, pp. 296–303, IEEE, Salt Lake City, Utah, USA, July 2007.

[9] J. M. Zaha, A. Barros, M. Dumas, and A. ter Hofstede, "Let's Dance: A Language for Service Behavior Modeling," in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, vol. 4275 of *Lecture Notes in Computer Science*, pp. 145–162, Springer, Berlin, Germany, 2006.

[10] A. Barker, C. D. Walton, and D. Robertson, "Choreographing web services," *IEEE Transactions on Services Computing*, vol. 2, no. 2, pp. 152–166, 2009.

[11] L. Engler, *BPELgold: Choreography on the Service Bus*, Institute of Architecture of Application Systems, University of Stuttgart, 2009.

[12] S. Arroyo and A. Duke, "SOPHIE - A Conceptual Model for a Semantic Choreography Framework," in *Proceedings of the in In proceedings of the Workshop on Semantic and Dynamic Web Processes (SDWP)*, 2005.

[13] M. Stollberg, "Reasoning tasks and mediation on choreography and orchestration in WSMO," in *Proceedings of the WIW 2005 Workshop on WSMO Implementations, WIW 2005*, June 2005.

[14] J. D. Bruijn, C. Bussler, J. Domingue, and D. Fensel, "Web Service Modeling Ontology (WSMO)," June 2005, http://www.w3.org/Submission/WSMO/.

[15] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide," in *Specification and Validation Methods*, pp. 9–36, Oxford University Press, 1993.

[16] Y. Gurevich, "Sequential abstract-state machines capture sequential algorithms," *ACM Transactions on Computational Logic*, vol. 1, no. 1, pp. 77–111, 2000.

[17] M. Herold, *WSMX Documentation*, Digital Enterprise Research Institute, Galway, Ireland, 2008.

[18] S. M. Ataee and Z. Bayram, "A novel concise specification and efficient F-Logic based matching of semantic web services in Flora-2," *Lecture Notes in Electrical Engineering*, vol. 355, pp. 191–198, 2015.

[19] M. Kifer and G. Lausen, "F-logic: a higher-order language for reasoning about objects, inheritance, and scheme," in *Proceedings of the ACM SIGMOD international conference*, pp. 134–146, Portland, Ore, USA, 1989.

[20] J. de Bruijn, H. Lausen, A. Polleres, and D. Fensel, "The Web Service Modeling Language WSML: An Overview," in *The Semantic Web: Research and Applications*, vol. 4011 of *Lecture Notes in Computer Science*, pp. 590–604, Springer, Berlin, Germany, 2006.

[21] E. Motta, "An Overview of the OCML Modelling Language," in *Proceedings of the In proceedings of the 8th Workshop on Knowledge Engineering Methods and Languages (KEML '98)*, 1998.

[22] M. Kifer, "Flora-2," January 2017, https://sourceforge.net/projects/flora/files/FLORA-2/.

[23] Y. Guizhen, M. Kifer, and C. Zhao, "Flora-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web," in *Proceedings of the in Second International Conference on Ontologies, Databases and Applications of Semantics (ODBASE*, Catania, Italy, 2003.

[24] Y. Gurevich, "Abstract state machines: an overview of the project," in *Foundations of Information and Knowledge Systems*, vol. 2942 of *Lecture Notes in Computer Science*, pp. 6–13, Springer, Berlin, Germany, 2004.

[25] G. Flouris, D. Plexousakis, and G. Antoniou, "Evolving ontology evolution," in *SOFSEM 2006: Theory and Practice of Computer Science*, vol. 3831, pp. 14–29, 2006.

[26] L. M. Garshol, "BNF and EBNF: What are they and how do they work," *acedida pela última vez em*, vol. 16, 2003.

[27] S. Mehdipour and Z. Bayram, *Choreography Engine Implementation in Flora-2*, Eastern Mediterranean University, 2017, https://sourceforge.net/projects/vschore-semantic-choreography/.

[28] J. Belzer, A. G. Holzman, and A. Kent, *Encyclopedia of computer science and technology. Vol. XXV*, CRC Press, 1975.

[29] E. Börger and R. Stärk, *Abstract State Machines*, Springer, Berlin, Germany, 2003.

[30] R. Zaharia, L. Vasiliu, and C. Bădică, "Semi-automatic Composition of Geospatial Web Services Using JBoss Rules," in *Rule Representation, Interchange and Reasoning on the Web*, vol. 5321 of *Lecture Notes in Computer Science*, pp. 166–173, Springer, Berlin, Germany, 2008.

[31] C. Wallace and J. K. Huggin, *An Abstract State Machine Primer*, Computer Science Department, Michigan Technological University, 2002.

[32] J. Huggins, *Abstract State Machines*, Department of Electrical Engineering and Computer Science, University of Michigan, 2013, http://wwweb.eecs.umich.edu/gasm/.

[33] T. E. O. E. Britannica, "predicate calculus," July 1998, https://global.britannica.com/topic/predicate-calculus.

[34] M. Kifer, "Rules and Ontologies in F-Logic," in *Reasoning Web*, vol. 3564 of *Lecture Notes in Computer Science*, pp. 22–34, Springer, Berlin, Germany, 2005.

[35] M. Kifer, G. Lausen, and J. Wu, "Logical foundations of object-oriented and frame-based languages," *Journal of the ACM*, vol. 42, no. 4, pp. 741–843, 1995.

[36] o. GmbH, *How to write F – Logic - Programs*, OntoBroker, Karlsruhe, Germany, 2007.

[37] M. Kifer and G. Lausen, "F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme," in *Proceedings of the SIGMOD '89 Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, Portland, Ore, USA.

[38] W. Chen, M. Kifer, and D. S. Warren, "HiLog: a foundation for higher-order logic programming," *Journal of Logic Programming*, vol. 15, no. 3, pp. 187–230, 1993.

[39] W. Chen, M. Kifer, and D. S. Warren, "HiLog: a foundation for higher-order logic programming," in *Proceedings of the North American Conference on Logic Programming*, 1989.

[40] A. J. Bonner and M. Kifer, "An overview of transaction logic," *Theoretical Computer Science*, vol. 133, no. 2, pp. 205–265, 1994.

[41] "XSB," Computer Science Department of Stony Brook University; Universidade Nova de Lisboa; XSB Inc.; Coherent Knowlege Systems, Inc., April 2015, http://xsb.sourceforge.net/.

[42] M. Kifer, "Flora-2 (a.k.a. Ergo Lite)," 2017, http://flora.sourceforge.net/.

[43] "Ergo Suite Platform," Coherent Knowledge Systems, August 2015, http://coherentknowledge.com/product-overview-ergo-suite-platform/.

[44] "A Guide to FLORA-2 Packages Version 1.0 (Cherimoya)," 2014.

[45] M. Kifer, G. Yang, H. Wan, and C. Zhao, *Flora-2: User's Manual (Version 1.0)*, Department of Computer Science, Stony Brook University, New York, NY, USA, 2014.

[46] U. Nilsson and J. Maluszynski, *Logic, Programming and Prolog*, John Wiley & Sons, 2nd edition, 2000.

[47] D. Roman, J. Scicluna, and C. Feier, "D14v0.1. Ontology-based Choreography and Orchestration of WSMO Services," WSMO Working Draft, March 2005, http://www.wsmo.org/TR/d14/v0.1/.

[48] D. Fensel, H. Lausen, and J. D. Bruijn, "Introduction to WSMO," in *Enabling Semantic Web Services*, pp. 57–61, Springer, Berlin, Germany, 2007.

[49] D. Roman, J. Scicluna, and J. Nitzsche, "D14v0.4. Ontology-based Choreography," Working Draft, Feburary 2007, http://www.wsmo.org/TR/d14/v0.4/.

[50] R. F. Stärk, J. Schmid, and E. Börger, "Abstract state machines," in *Java and the Java Virtual Machine Definition, Verification, Validation*, pp. 15–26, Springer, Berlin, Germany, 2001.

[51] J. H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Harper & Row, New York, NY, USA, 2nd edition, 2015.

[52] http://www.wsmo.org/2004/d3/d3.3/v0.1/.

[53] D. Fensel, M. Kerrigan, and M. Zaremba, "Semantic Web Services," in *Implementing Semantic Web Services: The SESA Framework*, pp. 27–41, Springer Science & Business Media, Berlin, Germany, 2008.

[54] D. Martin, M. Burstein, and J. Hobbs, "OWL-S: Semantic Markup for Web Services," W3C, 22 November 2004, http://www.w3.org/Submission/OWL-S/.

[55] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler, "WSMX - A semantic service-oriented architecture," in *Proceedings of the 2005 IEEE International Conference on Web Services, ICWS 2005*, pp. 321–328, July 2005.

[56] "WSMO Studio," December 2008, http://www.wsmostudio.org/.

[57] DERI, "WSMO4J," Innsbruck, The Semantic Technology Institute (STI), September 2008, http://www.sti-innsbruck.at/results/tools/downloads/wsmo4j.

[58] L. Cabral, J. Domingue, S. Galizia et al., "IRS-III: a broker for semantic web services based applications," in *The Semantic Web - ISWC 2006*, vol. 4273 of *Lecture Notes in Computer Science*, pp. 201–214, Springer, Berlin, Germany, 2006.

[59] "OWL for Services (OWL-S) - Tools," 2017, http://www.ai.sri.com/daml/services/owl-s/tools.html.

[60] R. Lara, A. Polleres, H. Lausen, D. Roman, J. D. Bruijn, and D. Fensel, "A Conceptual Comparison between WSMO and OWL-S," WSMO Working Draft, 2005.

[61] J. d. Bruijn, C. Feier, U. Keller, and R. Lara, "D16.2 v0.2 WSML Reasoning Implementation," WSML Working Draft, 2005.

[62] B. Bishop, F. Fischer, U. Keller, N. Steinmetz, C. Fuchs, and M. Pressnig, "WSML2Reasoner," 2008, http://tools.sti-innsbruck.at/wsml2reasoner/.

[63] "KAON2," 2017, http://kaon2.semanticweb.org/.

[64] "Web Services Execution Environment," 2009, https://sourceforge.net/projects/wsmx/.

[65] DERI and STI2, "WSMO Publications," Web Service Modelling eXecution environment, 2017, http://www.wsmx.org/publications.html.

[66] J. Domingue, L. Cabral, F. Hakimpour, D. Sell, and E. Motta, "Demo of IRS-III: A Platform and Infrastructure for Creating WSMO-based Semantic Web Services," in *Proceedings of the 3rd International Semantic Web Conference (ISWC2004)*, Hiroshima, japan, 2004.

[67] T. C. L. Foundation, "Common Lisp," March 2015, https://common-lisp.net/.

[68] D. Roman, M. Kifer, and D. Fensel, "WSMO Choreography: From Abstract State Machines to Concurrent Transaction Logic," in *Proceedings of the The Semantic Web: Research and Applications: 5th European Semantic Web Conference, ESWC 2008*, Tenerife, Spain, 2008.

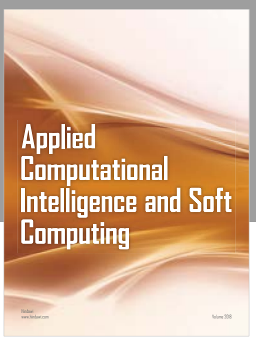
[69] D. Roman and M. Kifer, "Reasoning about the behavior of Semantic Web services with concurrent transaction logic,"

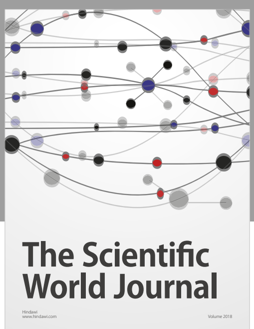
in *Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB 2007*, pp. 627–638, Vienna, Austria, September 2007.

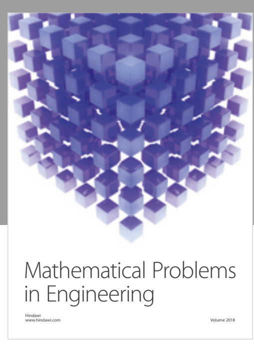
[70] A. J. Bonner and M. Kifer, "Concurrency and Communication in Transaction Logic," in *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP '96)*, 1996.

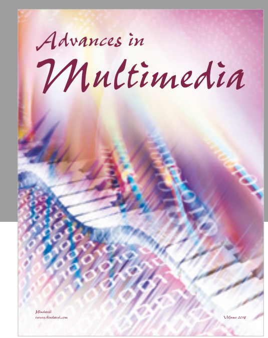
[71] A. Bonner and M. Kifer, *Concurrent Transaction Logic Prototype*, University of Toronto, 2017, http://www.cs.toronto.edu/.

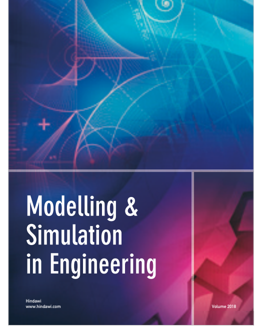
[72] S. Ponnekanti and A. Fox, "SWORD: A developer toolkit for web service composition," in *Proceedings of the 11th International WWW Conference (WWW)*, Honolulu, Hawaii, USA, 2002.











Submit your manuscripts at
www.hindawi.com