*Journal of Universal Computer Science, vol. 13, no. 9 (2007), 1157-1183*
*submitted: 15/8/06, accepted: 1/3/07, appeared: 28/9/07 © J.UCS*

# Ontology and Grammar of the SOPHIE Choreography Conceptual Framework - An Ontological Model for Knowledge Management

**Sinuhé Arroyo**
(University of Alcalá de Henares, Spain
sinuhe.arroyo@alu.uah.es)

**Abstract**: Ontologies have been recognized as a fundamental infrastructure for advanced approaches to Knowledge Management (KM) automation in SOA. Building services communicate with each other by exchanging self-contained messages. Depending on the specific requirements of the business model they serve and the application domain for which services were deployed, a number of mismatches (i.e. sequence and cardinality of messages exchanges, structure and format of messages and content semantics), can occur which prevent interoperation among a prior compatible services. Existing choreography technologies attempt to model such external visible behavior. However, they lack the consistent semantic support required to fully meet the necessities of heterogeneous KM environments. This paper describes the ontology and grammar of **SOPHIE**, a semantic service-based choreography framework for overcoming conversational pattern mismatches in knowledge intensive environments. Consequently, the paper provides an overview of the framework that depicts its main building blocks, so a good understantind of the ontology and grammar that summarize the conceptual model is gained. Such ontology allows the desing and description of fully fledged choreographies that can be used, as a result of a mediation task, to produce the mediating structures that in fact allow dynamic service-to-service interoperation. Finally, a use case centred in the telcomunications field serves as proof of concept of how **SOPHIE** is being applied.

**Keywords:** Semantic Services, Choreography
**Categories:** H.3.1, H.3.2, H.3.3, H.3.7, H.5.1

## 1 Introduction

The discipline of Knowledge Management (KM) has evolved and matured in the last decade, resulting in a considerable amount of models, tools and technologies [Sicilia06]. Nonetheless, such conceptual structures should be properly integrated into existing ontological bases, for the practical purpose of providing the required support for the development of intelligent applications. In addition, the supporting technologies for socialization, externalization, combination and internalization of knowledge are available and can be applied to build KM solutions of a diverse kind [Mohame04]. Formal ontologies [Gruber93] have been proposed and applied as the backbone of KM systems [Maedche03], and even ontologies specific to certain KM domains exist—e.g. for software development organizations [Marwick01. The new requirements in the design of knowledge [Sicilia06] intensive software systems call

for well decoupled approaches where components interoperate by exchanging self-contained messages. These systems realize their functionality by defining from a high-level point of view their dynamics. Still, components autonomously define their control flow and the message interface that allows others consuming their functionality. As the use of services is catching up, more and more interest is being place in the development of initiatives that allow their agile interoperation. With the aim of fulfilling the communication necessities, the concept of choreography, as a means to model the external visible behaviour of services withing KM environements has been sketched.

Services communicate with each other by exchanging self-contained messages, allowing them to make or to respond to requests. Upon the reception of a message, services react by executing some internal processes and possibly responding with other messages. Depending on the specific requirements of the business model they serve and the application domain for which services were deployed, a number of mismatches can occur which prevent interoperation among a prior compatible services.

- **Sequence and cardinality of messages exchanges.** Services follow different conversational patterns, which define the order and number in which messages are sent and/or received in a univocal way. A number of scenarios can be sketched that prevent interoperation:
    1. Messages being sent/received in a different order than expected (Sequence).
    2. Too many messages being sent/received that are not compliant with the expected behavior of the other party –i.e. acks, control messages, or messages being split into smaller ones– (Cardinality).
    3. Too little messages are sent/received not being compliant with the expected behavior of the other party –one message that makes up for a number of others, or no acks or no control messages– (Cardinality).
- **Structure and format of messages.** Services use different conceptualizations and naming conventions for encoding contents and characterizing messages. Even when all the information expected is enclosed in messages, the means to assert compatibility, identify and reorganize the contents and structure of messages and rename messages as appropriate, so they match the conversational model of the receiving party, need to be put in place.
- **Content Semantics.** Depending on the application domain services make use of a wide range of terminological conventions to represent concepts encoded into messages and the messages themselves. Prior to effective message exchanges the means to identify equivalent concepts needs to be put in place so messages and the pieces of data they contain can be understood by interacting services.

Due to the fact that the semantics, sequence, cardinality, structure and format followed by interacting services is wide, the means to overcome these limitations need to be put in place. Current infrastructures present an “ad-hoc” alternative for

overcoming heterogeneity which need to be improved so services are enabled to interact in a more dynamic and decoupled fashion.

**SOPHIE**[1][2] ([Arroyo, 06a][ Arroyo, 06b]), puts in place the required computational semantics that enable defining a meditation layer among the message exchanges of heterogeneous Semantic Services in knowledge intensive environments. This is avhieved regardless of the structural and behavioural models and the technological aspects used by interacting parties.

The remaining of the paper is structured as follows. Section 2 shows a high level architecture of the framework, together with the main actors involved and core ideas behind the framework. Section 3 presents the ontology of the conceptual framework that summarizes the main concepts required to model fully-fledged choreographies, providing. Section 4 Section 4 introduces the grammar of the **SOPHIE** choreography framework. Section 5 depicts the details of a use case centred in the Telecommunication industry where the principles and ideas of SOPHIE have been successfully applied. Finally, Section 6 outlines the conclusion of this research together with future direction for extending the work.

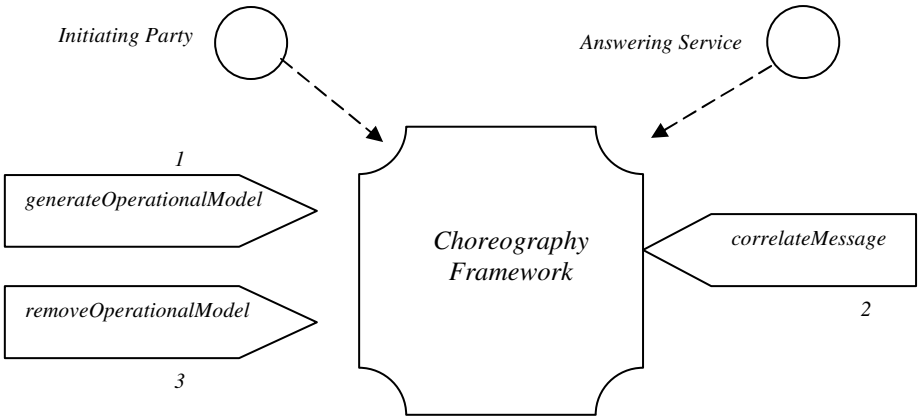


*Figure 1: Choreography service*

## 2 SOPHIE

**SOPHIE** is a knowledge management conceptual framework and architecture for a choreography service realized as a SOA. Services that use the choreography service fall into two main categories, namely, initiating parties and answering services. Both parties produce and consume messages. Additionally, initiating parties indicate the

[1] **SOPHIE** is an acronym for Semantic services chOreograPHi servIcE

[2] Notice that this paper complements a series of papers already published that go in detail into various aspects of the framework. This is why, only the relevant characteristics of **SOPHIE** are briefly covered in this work. The interested reader is encouraged to read other **SOPHIE** related publications to gain a deeper understanding of its particularities and details.

choreography framework by means of any of its constitutes correlating services that the infrastructure for the interoperation of heterogeneous message exchanges should be established.

Figure 1 shows a high level architecture of the conceptual framework. Informally, initiating parties indicate that want to communicate with an answering service by means of "*generateOperationalModel*" (1). Once an operational model that allows the interoperation among the heterogeneous message exchanges has been created, parties can start submitting messages by means of the "*correlateMessage*" (2) primitive. Messages will go through the designated operational model, forwarding the framework the message(s) to the receiving party according to its choreography. Finally, when the conversation is finished, either party indicates that the operational model for a given conversation can be put off line, by means of the primitive "*removeOperationalModel*".

### 3.1 Conversations

A *conversation* represents the logical entity that permits to group a set of related message exchanges among parties. Conversations are composed of a number of building blocks.

*Elements* represent elementary unit of data that build up *documents*. Documents are complete, self-contained groups of elements that are transmitted over the wire within *messages*. Messages characterize the primitive piece of data that can be exchanged among parties. As messages are exchanged, a variety of recurrent scenarios can be played out. *Message Exchange Patterns (MEP)*, identify placeholders for messages, that allow to model its sequence and cardinality, defining the order on which parties send a receive messages. A set of messages sent and received among parties, optionally following Message Exchanged Patterns are referred as *message exchange*. They characterize a well defined part of a conversation. A *conversation* can be thus defined as a set of message exchanges among parties with the aim of fulfilling some goal. Every conversation need to rely on top of some communication facility referred as *communication network*.

### 3.2 Choreographies

A *choreography* describes the behavior of the answering service from the initiating party point of view [Roman04a]. It governs the message exchanges among parties in a conversation. A choreography as presented in this work is based on the Finite State Machines (FSMs)[3] formalism. FSMs allow specifying the sequences of states the choreography goes through during its lifetime, together with its responses to events. In this sense the main building blocks of abstract state machines are up to some extent redraw.

A *state* is a situation during the lifetime of a choreography during which it waits for some event or satisfies some condition. Conditions are modeled as *booleanExpression*. Boolean expressions are expressions evaluated to "*true*" or

[3] Activities, entry actions and exit actions have been deliberately left out of the scope of the work, as they are not required for purpose of the thesis.

"*false*" as in any programming or ontology language. An action represents the atomic task of sending a message to a party. *Events* represent occurrences of stimulus. They do not specify state transitions. Transitions among states are defined by *guarded transitions*. Guarded transitions allow modeling the relations between two states by means of *events*, *actions* and *guard conditions*. Guarded conditions are rules that specify a target state. *Parts* permit to relate guarded transitions and message exchanges, defining the message exchange in terms of state transitions. Finally, a choreography comprises a set of parts that define a conversation.

### 3.3 Logic Boxes

The atomic building blocks that permit to solve the mismatches among interacting parties are referred as *logic boxes*. A logic box facilitates the means to reorganize the content of documents, its mapping to messages, and the order and cardinality of messages, enabling the interoperation among heterogeneous message exchanges. Additionally, and depending on the type of box, the differences in the vocabulary used to describe the application domain can be overcome. Currently the specification defines five different types of logic boxes, namely: *refiner box, merge box, split box, select box, add box*. Logic boxes are grouped into *logic diagrams*. Logic diagrams permit to model the relation among the message exchange pattern followed by the initiating party, and the one used by the answering service. Logic diagrams are assimilated, for implementation purposes, to *correspondence tables*. A correspondence table is a logical structure, similar to routing tables, which defines relations among incoming and outgoing messages as a realization of a logic diagram. A number of logic diagrams defining a conversation are referred as *logic group*.

### 3.4 Ontologies

Ontologies define the semantics of the framework. They facilitate a formal and consensual [Gruber93] vocabulary as data and information machine-processable semantics for the shared and common understanding of a domain [Fensel01] that can be mediated for the understanding of interacting parties. *Domain ontologies* supply the general vocabulary to describe the application domain of parties. *Choreography ontologies* make available the terminology that describes the choreography of parties. In doing so they define the different entities (concepts) taking part in a choreography. *Ontology mappings* characterize the conceptual entity that allows to link similar ontological concepts and instances.

### 3.5 Related Technologies

In the following different technologies that are related to the definition of a conceptual framework for choreography are concisely reviewed. In doing so, their core characteristics are presented, their drawbacks identified and the main ideas reused in this research are summarized.

| | | ***layered model*** | | |
|---|---|---|---|---|
| | | ***no*** | | ***yes*** |
| ***relation with communication framework*** | ***tight*** | Business Process Languages | Choreography Languages | |
| | ***loose*** | Choreography Languages | Semantic-driven choreography initiatives | SOPHIE |
| | | ***no*** | ***yes*** | |
| | | ***semantic support*** | | |

*Table 1: A first cut in classifying related languages*

Table 1 presents a preliminary classification based on a three dimension exam. The first dimension depicts the relation with the underlying communication framework, differentiating among tight and loose. The second one addresses the semantic support provided. Finally, the third one discriminates them depending on whether or not they follow a layered model. Based on these depiction four main categories of languages are distinguished:

- Technologies with a tight relation to the underlying communication framework, lacking of a layered model and no support for semantics, such as BPEL4WS
- Technologies with a tight relation to the underlying communication framework, that follow a layered model and no support for semantics, such as WS-CDL
- Technologies with a loose relation to the underlying communication framework, lacking of a layered model and no support for semantics, such as WSCI
- Technologies with a loose relation to the underlying communication framework, with support for semantics but lacking of a layered model, such as WSMO-Choreography

In [Arroyo06b] a detailed overview of the related languages detailed in Table 1 is presented.

## 4 Ontology of the Conceptual Model

In the following the ontology that summarizes the ideas and concepts of SOPHIE is presented. In doing so, the different conceptual models are briefly reviewed and the fragments of the ontology where such concepts are modeled are provided.

### 4.1 Conceptual Model

The conceptual model presented of **SOPHIE** describes the structure, behavior, operation and ontologies of conceptual framework for choreography as separate concerns. The semantic model details the semantic support. The structural concern provides the grounding pillars of the framework. The behavioral concern permits to model the conduct of the structural model. Finally, the operational concern facilitates the means to allow the interoperation of different behavioral models.

This clear separation of concerns facilitates a straight mechanism to extend the different models, for example Petri nets, temporal logic or transaction logic can be used instead of Finite State Machines (FSMs) for the behavioral model.

The work presented here defines the behavioral model as FSMs. Still, any other suitable paradigm can be easily plugged-in. The terms used, in particular the terms choreography, state and guarded transition, follow the formalism proposed in [Roman04b] and [Booch99]. Furthermore, the semantic model is currently based on WSML. Nonetheless, the design allows to easily extending the grammar and ontology of **SOPHIE** to accommodate any other ontology language.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"

namespace {         _"http://www.example.org/ontologies/sophie#",
                dc  _"http://purl.org/dc/elements/1.1#",
               xsd  _"http://www.w3.org/2001/XMLSchema#",
              wsml  _"http://www.wsmo.org/wsml-syntax#"
           }

ontology _"http://www.deri.org/ontology/sophie#"

nonFunctionalProperties
   dc#title hasValue "Choreography Conceptual Model"
   dc#creator hasValue "Sinuhé Arroyo"
   dc#description hasValue "an ontology for describing the concepts of
                            SOPHIE"
   dc#publisher hasValue "DERI International"
   dc#contributor hasValue "Jos de Bruijn"
   dc#date hasValue "2005-04-11"
   dc#type hasValue "http://www.deri.org/2005/#ontology"
   dc#format hasValue "text/html"
   dc#language hasValue "en-us"
   dc#rights hasValue "http://deri.at/privacy.html"
   version hasValue "$Revision 0.1$"
endNonFunctionalProperties

concept sophie#conceptualModel
   nonFunctionalProperties
      dc#description hasValue "root of the conceptual model"
   endNonFunctionalProperties
```

```
      structuralConcern ofType (0 1) structural
      behavioralConcern ofType (0 1) behavioral
      operationalConcern ofType (0 1) operational
   ontologies ofType (0 1) sophie#ontologies

concept sophie#entity
   nonFunctionalProperties
      dc#description hasValue "entities of the choreography service"
   endNonFunctionalProperties
   name ofType (1 1) sophie#name
   URI ofType (1 1) sophie#uri

concept sophie#name
   nonFunctionalProperties
      dc#description hasValue "name or key value of an entity"
   endNonFunctionalProperties

concept sophie#handler
   nonFunctionalProperties
      dc#description hasValue "address and protocol of an entity"
   endNonFunctionalProperties

concept sophie#uri
   nonFunctionalProperties
      dc#description hasValue "handler, entity and identifier"
   endNonFunctionalProperties
   handler ofType (1 1) sophie#handler
   entity ofType (1 1) sophie#entityType
   identifier ofType (1 1) sophie#name

concept sophie#entityType
   nonFunctionalProperties
      dc#description hasValue "type of an entity"
   endNonFunctionalProperties

axiom sophie#allowedEntityTypes
   definedBy
      !- ?x memberOf entityType and naf ?x = sophie#party
                                and naf ?x = sophie#element
                                and naf ?x = sophie#document
                                and naf ?x = sophie#message
                                and naf ?x = sophie#mep
                                and naf ?x = sophie#messageExchange
                                and naf ?x = sophie#conversation
                                and naf ?x = sophie#state
                                and naf ?x = sophie#action
```

```
                    and naf ?x = sophie#booleanExpression
                    and naf ?x = sophie#event
                    and naf ?x = sophie#guardCondition
                    and naf ?x = sophie#guardedTransition
                    and naf ?x = sophie#part
                    and naf ?x = sophie#choreography
                    and naf ?x = sophie#logicBox
                    and naf ?x = sophie#logicDiagram
                    and naf ?x = sophie#logicGroup
                    and naf ?x = sophie#correspondenceTable
                    and naf ?x = sophie#correspondenceModel
                    and naf ?x = sophie#domainOntology
                    and naf ?x = sophie#choreographyModel
                    and naf ?x = sophie#choreographyOntology
                    and naf ?x = sophie#ontologyMapping.

concept sophie#role
   nonFunctionalProperties
      dc#description hasValue "specifies whether a party is an initiating
                               party an answering service or a correlating
                               service"
      endNonFunctionalProperties

axiom sophie#allowedRoles
   definedBy
            !- ?x memberOf role and naf ?x = sophie# initiatingParty
                                and naf ?x = sophie# answeringService
                                and naf ?x = sophie# correlatingService.

concept sophie#formalism
   nonFunctionalProperties
      dc#description hasValue "formalism"
   endNonFunctionalProperties

concept sophie#concern
   nonFunctionalProperties
      dc#description hasValue "model of the choreography service "
   endNonFunctionalProperties
```

### 4.1.1 Structural Model

The structural model deals with the provision of a reusable collection of entities following different levels of abstraction that facilitate the basis for the description of a conceptual model. Table 2, enumerates the entities that allow the structural model to be defined.

| |
|---|
| *element = [name, type, value]* |
| *document = [name, URI, elements*[4]]* |
| *message = [name, URI, from[?5], to[?], documents*]* |
| *messageExchangePattern = [name, URI, description[?]]* |
| *message exchange = [name, URI, mep[?], messages*]* |
| *conversation = [name, URI, messageExchanges*]* |

*Table 2: Structural model*

*Conversations* are the outer most entity of the structural model. They represent the logical entity that permits to group a set of related message exchanges among parties. Conversations are composed of a set of building blocks. *Elements* describe elementary units of data that define a name, a type[6] and a value that build documents. Documents are complete, self-contained groups of elements. Documents are transmitted over the wire within *messages*. Messages characterize pieces of information that can be exchanged among parties. As messages are exchanged, a variety of recurrent scenarios can be played out as defined by *Message Exchange Patterns (MEP)*. A MEP defines a minimal contract among parties. They allow the sequence and cardinality of messages to be modeled, defining the order in which parties send and receive messages. The constituent *description* is a *part* that depicts the behavior of the pattern. A set of messages sent and received among parties optionally following a Message Exchanged Pattern that account for a well defined part of a conversation, is referred as a *message exchange*. A *conversation* can thus be defined as a set of message exchanges among parties, optionally following message exchange patterns to model their behavior. Every conversation need to rely on top of some communication facility, referred to as a *communication network*.

```
concept sophie#structural
   subConceptOf {concern}
   nonFunctionalProperties
      dc#description hasValue "structural model"
   endNonFunctionalProperties
   formalisms ofType (0 *) sophie#formalism

axiom sophie#allowedStructuralFormalisms
  definedBy
     !- ?x memberOf entityType and naf ?x = sophie#MEPConversational.

concept sophie#MEPConversational
   subConceptOf {formalism}
```

[4] The symbol "*" represents that there can exist zero or more instances of the attribute

[5] The symbol "?" represents zero or one instances of the attribute

[6] Element types, belong to a limited set of types as defined by the standard XSD [**Fehler! Verweisquelle konnte nicht gefunden werden.**]

```
   nonFunctionalProperties
      dc#description hasValue "MEP structural formalism"
   endNonFunctionalProperties
   party ofType (1 1) sophie#party
   elements ofType (0 *) sophie#element
   documents ofType (0 *) sophie#document
   messages ofType (0 *) sophie#message
   messageExchanges ofType (0 *) sophie#messageExchange
   MEPs ofType (0 *) sophie#mep
   conversations ofType (0 *) sophie#conversation
   suffcientElements ofType (0 *) sophie#sufficientSet
   sufficientMessages ofType (0 *) sophie#sufficientSet

concept sophie#type
   nonFunctionalProperties
      dc#description hasValue "XSD type subset"
   endNonFunctionalProperties

axiom sophie#allowedTypes
   definedBy
      !- ?x memberOf type and naf ?x = xsd#string
                          and naf ?x = xsd#decimal
                          and naf ?x = xsd#integer
                          and naf ?x = xsd#float
                          and naf ?x = xsd#boolean
                          and naf ?x = xsd#date
                          and naf ?x = xsd#time.

concept sophie#value
   nonFunctionalProperties
      dc#description hasValue "value of the element"
   endNonFunctionalProperties
   representation ofType _string
   type ofType sophie#type

concept sophie#element
   subConceptOf {entity}
   nonFunctionalProperties
      dc#description hasValue "piece of data either supplied or consumed
                               by  parties"
   endNonFunctionalProperties
   sophie#value ofType (1 1) sophie#value

concept sophie#sufficientSet
   subConceptOf {entity}
   nonFunctionalProperties
```

```
        dc#description hasValue "sufficient set of entities"
      endNonFunctionalProperties
      entities ofType (0 *) sophie#entity
      sophie#hasValue ofType (1 1) sophie#value

concept sophie#party
    subConceptOf {sophie#entity}
  nonFunctionalProperties
      dc#description hasValue "active entities inside or outside the
                               choreography framework"
  endNonFunctionalProperties
  role ofType (1 1) sophie#role
  URI ofType (1 1) sophie#role

  concept sophie#document
    subConceptOf {entity}
    nonFunctionalProperties
      dc#description hasValue "grouping of a number related elements"
    endNonFunctionalProperties
    elements ofType (0 *) sophie#element

  concept sophie#message
    subConceptOf {entity}
    nonFunctionalProperties
      dc#description hasValue "minimal unit that can be exchanged
                               among parties"
    endNonFunctionalProperties
    from ofType (0 1) sophie#party
    to ofType (0 1) sophie#party
    documents ofType (0 *) sophie#document

  concept sophie#mep
    subConceptOf {entity}
    nonFunctionalProperties
      dc#description hasValue "placeholder for message exchanges"
    endNonFunctionalProperties
    description ofType (0 1) sophie#part

  concept sophie#messageExchange
    subConceptOf {entity}
    nonFunctionalProperties
      dc#description hasValue "piece of a conversation optionally
                               following a mep"
    endNonFunctionalProperties
    MEP ofType (0 1) sophie#mep
    messages ofType (0 *) sophie#message
```

```
concept sophie#conversation
  subConceptOf {entity}
  nonFunctionalProperties
    dc#description hasValue "set of message exchanges"
  endNonFunctionalProperties
  messageExchanges ofType (0 *) sophie#messageExchange
```

### 4.1.2 Behavioral Model

The behavioral model cares for the description of the dynamic interaction among the entities defined in the structural model. As presented in this work, the behavioral models are based on the formalism presented by Finite State Machines (FSMs) [Wagner06]. Nevertheless, any other formalism such as Petri nets, temporal or transactional logic can be easily modeled and plugged-in in to the behavioral model. In doing so, it makes use of the entities enumerated in Table 3.

| |
|---|
| *booleanExpression = [name,URI, expression?]* |
| *state = [name, URI, subStates*]* |
| *action = [name, URI, task?]* |
| *task = [party.message]* |
| *event = [name, URI, booleanExpression?]* |
| *guardCondition = [name, URI, rule?]* |
| *rule = [**if** booleanExpression **then** state]* |
| *guarded transition = [name, URI, events*, guardCondition?, actions*]* |
| *part = [name, URI, messageExchange?, guardedTransitions*]* |
| *choreography = [name, URI, conversation?, parts*]* |

*Table 3: Behavioral model*

A *choreography* represents the outer most entity in the behavioral model. It describes the behavior of the answering service from the initiating party point's of view [Roman04a]. It governs the message exchanges among parties in a conversation.

*States*, *actions* and *events* and *guard conditions* represent the same concepts as defined by FSMs. However, the scope of events and actions has been narrowed. Particularly, actions represent the atomic task of sending a message, and events can not trigger a state transition, since do not specify a target state, but just a *booleanExpression*. Additionally, activities, entry actions and exit actions have been deliberately left out of the scope of the work, as they are not required for our purposes. Finally, the concept of guarded transitions, parts and choreography have been added.

A *guarded transition* defines the relationship between states by means of events, guard conditions and actions. In a nutshell, a guarded transition defines events and conditions, which when satisfied, perform certain actions and trigger the state transition as defined in the guarded condition. *Parts* permit guarded transitions and message exchanges to be related, defining the message exchange in terms of state transitions according to the logic of the application. Finally, a *choreography* can be

defined as a set of parts, which govern the message exchanges among parties in a conversation.

```
concept sophie#behavioral
    subConceptOf {concern}
    nonFunctionalProperties
       dc#description hasValue "behavioral model"
    endNonFunctionalProperties
    hasFormalism ofType (0 *) sophie#formalism

axiom sophie#allowedBehavioralFormalisms
  definedBy
        !- ?x memberOf entityType and naf ?x = sophie#FSM.

concept sophie#FSM
   subConceptOf {formalism}
   nonFunctionalProperties
      dc#description hasValue "Finite State Machine behavioral formalism"
   endNonFunctionalProperties
   states ofType (0 *) sophie#state
   actions ofType (0 *) sophie#action
   booleanExpressions ofType (0 *) sophie#booleanExpression
   events ofType (0 *) sophie#event
   guardConditions ofType (0 *) sophie#guardCondition
   guardTransitions ofType (0 *) sophie#guardTransition
   parts ofType (0 *) sophie#part
   choreography ofType (0 1) sophie#choreography
   sufficientActions ofType (0 *) sophie#sufficientSet
   sufficientBooleanExpressions ofType (0 *) sophie#sufficientSet

concept sophie#state
   subConceptOf {entity}
   nonFunctionalProperties
      dc#description hasValue "condition or situation during the lifetime of a
                               choreography"
   endNonFunctionalProperties
   subStates ofType (0 *) sophie#state

concept sophie#task
   nonFunctionalProperties
      dc#description hasValue "Party and message to be sent"
   endNonFunctionalProperties
   party ofType (0 1) sophie#party
   message ofType (0 1) sophie#message
```

```
concept sophie#action
   subConceptOf {entity}
   nonFunctionalProperties
      dc#description hasValue "action of sending a message"
   endNonFunctionalProperties
   task ofType (0 1) sophie#task

concept sophie#booleanExpression
   subConceptOf {entity}
   nonFunctionalProperties
      dc#description hasValue "boolean expression"
   endNonFunctionalProperties

concept sophie#event
   subConceptOf {entity}
   nonFunctionalProperties
      dc#description hasValue "occurrence of an stimulus that has a location
                               in time and space"
   endNonFunctionalProperties
   booleanExpression ofType (0 1) sophie#booleanExpression

concept sophie#rule
   nonFunctionalProperties
      dc#description hasValue "defines a rule"
   endNonFunctionalProperties
   booleanExpression ofType (0 1) sophie#booleanExpression
   state ofType (0 1) sophie#state

concept sophie#guardCondition
   subConceptOf {entity}
   nonFunctionalProperties
      dc#description hasValue "defines transitions among states by means of
                               rules"
   endNonFunctionalProperties
   rule ofType (0 1) sophie#rule

concept sophie#guardTransition
   subConceptOf {entity}
   nonFunctionalProperties
      dc#description hasValue "defines relations among states by means of
                               events, guardConditions and actions"
   endNonFunctionalProperties
   events ofType (0 *)  event
   guardCondition ofType (0 1) sophie#guardCondition
   actions ofType (1 *) sophie#action
```

```
concept sophie#part
   subConceptOf {entity}
   nonFunctionalProperties
      dc#description hasValue "allows establishing the link among a set of
                               guarded transitions and a message
                               exchange"
   endNonFunctionalProperties
   messageExchange ofType (0 1) sophie#messageExchange
   guardTransitions ofType (0 *) sophie#guardTransition

concept sophie#choreography
   subConceptOf {entity}
   nonFunctionalProperties
      dc#description hasValue "set of parts that govern the message
                               exchange among parties in a conversation"
   endNonFunctionalProperties
   hasConversation ofType (0 1) sophie#conversation
   hasParts ofType (0 *) sophie#part
```

### 4.1.3 Operational Model

The operational model facilitates the means to allow the interoperation among different behavioral models. Table 4, enumerates the entities that allow an operational model to be defined.

| |
|---|
| *logicBox = [name, URI, type, inputMessages*, inputMep?, outputMep?, ontologyMapping?]* |
| *logicDiagram= [name, URI, inputMessageExchange?, outputMessageExchange? logicBoxes*]* |
| *logicGroup = [name, URI, conversation?, logicDiagrams*]* |

*Table 4: Entities of the operational model*

Logic boxes constitute the key entity of the operational model. The outer most entities of the operational model are logic groups.

```
   concept sophie#operational
   subConceptOf {concern}
   nonFunctionalProperties
      dc#description hasValue "operational model"
   endNonFunctionalProperties
   formalism ofType (0 *) sophie#formalism

axiom sophie#allowedOprationalFormalisms
  definedBy
        !- ?x memberOf entityType and naf ?x = sophie#FSM.
```

```
concept sophie#logicBoxCorrespondence
   subConceptOf {formalism}
   nonFunctionalProperties
      dc#description hasValue "Logic box behavioral formalism"
   endNonFunctionalProperties
   logicBoxes ofType (0 *) sophie#logicBox
   logicDiagrams ofType (0 *) sophie#logicDiagram
   logicGroups ofType (0 *) sophie#logicGroup
   correspondenceTables ofType (0 *) sophie#correspondenceTable
   correspondenceModels ofType (0 *) sophie#correspondenceModel

concept sophie#logicBoxType
   subConceptOf {entity}
   nonFunctionalProperties
      dc#description hasValue "type of the logic box "
   endNonFunctionalProperties

axiom sophie#allowedBoxTypes
   definedBy
      !- ?x memberOf logicBoxType and naf ?x = sophie#refineBox
                                  and naf ?x = sophie#mergeBox
                                  and naf ?x = sophie#splitBox
                                  and naf ?x = sophie#selectBox
                                  and naf ?x = sophie#addBox.

concept sophie#logicBox
   subConceptOf {entity}
   nonFunctionalProperties
      dc#description hasValue "allows to solve a number of heterogeneities
                               in the messages exchanged by parties"
   endNonFunctionalProperties
   type ofType (1 1) sophie#logicBoxType
   inputMessages ofType (0 *) sophie#message
   inputMEP ofType (0 1) sophie#mep
   outputMep ofType (0 1) sophie#mep
   ontologyMapping ofType (0 1) sophie#ontologyMapping

concept sophie#logicDiagram
   subConceptOf {entity}
   nonFunctionalProperties
      dc#description hasValue "set of interconnected logic boxes that model
                               the relation among the message exchanges
                               used by interacting parties"

   endNonFunctionalProperties
   inputMessageExchange ofType (0 1) sophie#messageExchange
```

*outputMessageExchange* ***ofType*** *(0 1) sophie#messageExchange*
*logicBoxes* ***ofType*** *(0 *) sophie#logicBox*

***concept*** *sophie#logicGroup*
***subConceptOf*** *{entity}*
***nonFunctionalProperties***
*dc#description* ***hasValue*** *"conceptual entity that allows to put together a number of logic diagrams that model a conversation"*
***endNonFunctionalProperties***
*conversation* ***ofType*** *(0 1) sophie#conversation*
*logicDiagrams* ***ofType*** *(0 *) sophie#logicDiagram*

***concept*** *sophie#correspondenceTable*
***subConceptOf*** *{entity}*
***nonFunctionalProperties***
*dc#description* ***hasValue*** *"structure containing the operational model"*
***endNonFunctionalProperties***
*inputMessages* ***ofType*** *(0 *) sophie#message*
*outputMessagese* ***ofType*** *(0 *) sophie#message*
*sophie#value* ***ofType*** *(0 *) sophie#value*

***concept*** *sophie#correspondenceModel*
***subConceptOf*** *{entity}*
***nonFunctionalProperties***
*dc#description* ***hasValue*** *"disjunctive piece of the operational model"*
***endNonFunctionalProperties***
*initiatitingPartyCorrespondenceTable* ***ofType*** *(0 1) sophie#correspondenceTable*
*answeringServiceCorrespondenceTable* ***ofType*** *(0 1) sophie#correspondenceTable*

### 4.1.4 Ontologies

The framework differentiates among three types of ontologies, namely "*domain ontologies*", "*choreography model*" "*choreography ontologies*". Domain ontologies facilitate the general vocabulary to describe the application domain of the parties. The choreography model summarizes the SOPHIE concepts and ideas as presented in this research. Finally, choreography ontologies provide the conceptual framework and vocabulary required to semantically describe a choreography, by borrowing terminology from the domain ontology and choreography model. Additionally o*ntology mappings* put in place the mechanisms to link similar ontological concepts and instances, same for the domain and choreography ontologies.

| |
|---|
| *domainOntology = [name, URI]* |
| *choreographyOntology = [name, URI]* |
| *ontologyMapping = [name, URI, source, target]* |

*Table 5: Entities of the semantic model*

Table 5, distinguishes the different constituents of the semantic model.

```
concept sophie#semantics
        subConceptOf {aspect}
        nonFunctionalProperties
      dc#description hasValue "semantic aspects"
   endNonFunctionalProperties
   hasSemanticModel ofType (0 1) sophie#semantic

concept sophie#semantic
   subConceptOf {model}
   nonFunctionalProperties
      dc#description hasValue "semantic model"
   endNonFunctionalProperties

concept sophie#domainMapping
   subConceptOf {formalism}
   nonFunctionalProperties
      dc#description hasValue "Domain knowledge mapping formalist formalism"
   endNonFunctionalProperties
   hasDomainOntologies ofType (0 *) sophie#domainOntology
   hasChoreographyOntologies ofType (0 *) sophie#choreographyOntology
   hasChoreographyMapping ofType (0 *) sophie#ontologyMapping

concept sophie#ontology
   subConceptOf {entitiy}
   nonFunctionalProperties
      dc#description hasValue "ontology"
   endNonFunctionalProperties

concept sophie#domainOntology
   subConceptOf {sophie#ontology}
   nonFunctionalProperties
      dc#description hasValue "domain ontology"
   endNonFunctionalProperties

concept sophie#choreographyOntology
   subConceptOf {sophie#ontology}
   nonFunctionalProperties
```

```
      dc#description hasValue "choreography ontology"
   endNonFunctionalProperties

concept sophie#ontologyMapping
   subConceptOf {sophie#ontology}
   nonFunctionalProperties
      dc#description hasValue "domain or choreography ontology mapping"
   endNonFunctionalProperties
      hasSource ofType (0 *) sophie#ontology
      hasTarget ofType (0 *) sophie#ontology
```

## 5 Grammar of the Conceptual Model

A symbol between square brackets ([]) is not required to occur (may occur zero or one time). A symbol between curly brackets is not required and may occur zero or more times. A symbol not enclosed in square or curly brackets is required and may only occur one time. The bar (|) stands for an exclusive choice. All keywords are represented in boldface. [de Bruijn04].

**conceptualModel** ::= 'ConceptualModel (' **structure behavior operation semantics** ')'
**party** ::= 'Party (' **name URI role** ')'
**name** ::= **identifier**
**URI**[7] ::= **handler** '/' **entity** '/' **identifier**
**handler** ::= **identifier**
**entity** ::= 'party' | 'element' | 'document' | 'message' | 'mep' | 'messageExchange' | 'conversation' | 'state' | 'action' | 'booleanExpression' | 'event' | 'guardCondition' | 'guardedTransition' | 'part' | 'choreography' | logicBox' | 'logicDiagram' | 'logicGroup' 'correspondenceTable' | 'correspondenceModel' 'ontology' | 'domainOntology' | 'choreographyOntology' | 'choreographyModel' | 'sufficientSet'
**role** ::= 'initiatingParty' | 'answeringService' | 'correlatingService'
**sufficientSet** ::= 'SufficientSet (' **name URI** { **entity** } ')'
**formalism** ::= 'Formalism (' **nameFormalism** { **sufficientSet** }')'
**nameFormalism** ::= ::= 'MEPConversational' | 'FSM' | 'logicBoxCorrespondence'
**structure** ::= 'Structure (' { **formalism** } ')'
**conversation** ::= 'Conversation (' **name URI** { **messageExchange** } ')'

[7] The definition of the URI has been narrowed to better accommodate the requirements of the work.

**messageExchange** ::= 'MessageExchange (' **name URI** [ **mep** ] { **message** } ')'
**mep** ::= 'mep (' **name URI** [ **description** ] ')'
**message** ::= 'Message (' **name URI** [ **from** ] [ **to** ] { **document** } ')'
**from** ::= **party**
**to** ::= **party**
**document** ::= 'Document (' **name URI** { **element** } ')'
**element** ::= 'Element (' **type name value** ')'
**type** ::= 'xs:string'
| 'xs:decimal'
| 'xs:integer'
| 'xs:float'
| 'xs:boolean'
| 'xs:date'
| 'xs:time'

**value** ::= **valueSpace**[8]
**description** ::= **part**
**behavior** ::= 'behavior ('{ **formalism** } ')'
**choreography** ::= 'Choreography (' **name URI** [ **conversation** ] { **part** } ')'
**part** ::= 'Part (' **name URI** [ **messageExchange** ] { **guardedTransition** } ')'
**guardedTransition** ::= 'GuardedTransition (' **name URI** { **event** } [ **guardedCondition** ] { **action** } ')'
**guardedCondition** ::= 'GuardedCondition (' **name URI** [ **rule** ] ')'
**rule** ::= 'Rule ( if' **booleanExpression** 'then' **state** ')'
**event** ::= 'Event (' **name URI** [ **booleanExpression** ] ')'
**booleanExpression** ::= '**BooleanExpression (' name URI LogicalExpresion** ')'
**logicalExpression** ::= **expr**[9]
**action** ::= 'Action (' **name URI** [ **party** ] [ **message** ] ')'
**state** ::= 'State (' **name URI** { **subState** } ')'
**subState** ::= 'SubState (' **state** ')'
**operation** ::= 'Operation ('{ **formalism** } ')'
**logicGroup** ::= 'LogicGroup (' **name URI** [ **conversation** ] { **logicDiagram** } ')'
**logicDiagram** ::= 'LogicDiagram (' **name URI** [ **inputMessageExchange** ] [ **outputMessageExchange** ] { **logicBox** } ')'
**logicBox** ::= 'LogicBox (' **name URI boxType** { **inputMessage** } [ **inputMep** ] [ **outputMep** ] [ **ontologyMapping** ]')'

**correspondenceTable** ::= 'correspondenceTable (' **name URI** { **inputMessageExchange** } { **outputMessageExchange** } **value** ')'

[8] XSD value-space for a given data type
[9] WSML expression as defined in [**Fehler! Verweisquelle konnte nicht gefunden werden.**]

**correspondenceModel** ::= 'correspondenceModel (' **name URI**
[ **initatingPartyCorrespondenceTable** ]
[ **answeringServiceCorrespondenceTable** ] ')'
**boxType** ::= 'refineBox' | 'mergeBox' | 'splitBox' | 'selectBox' | 'addBox'
**inputMep** ::= **mep**
**outputMep** ::= **mep**
**inputMessageExchange** ::= **messageExchange**
**outputMessageExchange** ::= **messageExchange**
**initatingPartyCorrespondenceTable** ::= **correspondenceTable**
**answeringServiceCorrespondenceTable** ::= **correspondenceTable**
**semantics**::= 'Semantics (' { **domainOntology** } { **choreographyOntology** }
[ **choreographyMapping** ] ')'
**ontology** ::= 'Ontology ('**name URI** ')'
**domainOntology** ::= 'DomainOntology ('**ontology** ')'
**choreographyModel** ::= 'ChoreographyModel (' **ontology** ')'

**choreographyOntology** ::= 'ChoreographyOntology (' **ontology** ')'
**ontologyMapping** ::= 'OntologyMapping ('**name URI** { **source** } { **target** } ')'
**choreographyMapping** ::= **ontologyMapping**
**source** ::= **ontology** | **ontologyMapping**
**target** ::= **ontology**
**identifier** ::= **letter** { **letter** | **digit** } | **digit** { **letter** | **digit** }
**letter** ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' |
'q' | 'r' | s' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | '_' | '-' | '.' | '/' | ' :' | '@'
**digit** ::= '0' | '1' | '2' | '3' | '4' '5' | '6' | '7' | '8' |

## 6 Use case: BT Wholesale KM Operational Support System

**SOPHIE**[10] has provided the paradigm used to define the choreographies of the trading partners and BT as Wholesale Provider in the context of the DIP project[11]. Using the previously defined domain ontology and the choreography ontology model, it is possible to specify a choreography that depicts the structural and behavioral models. To do so, it imports from both (domain and model choreography) ontologies.

Figure 2 shows the MEPs followed by both interacting parties. If the service provider were to use a different MEP and terminology than BT Wholesale, then the operational model of **SOPHIE** can be applied. In this case, the Service Provider uses the message exchange *tPontTestRequest* following the MEP *request-response* while BT Wholesale makes use of the message exchange *eCoTestRequest* following the *In-Multi-Out* one. More concretely, the Service provider starts the message exchange with the *MRequest* message, while BT Wholesale expects the message *testRequest*. Additionally, BT provides two different response message (*failure* and *success*),

[10] The evaluation focuses at the message exchange level as it is considered enough to prove the consistency of the model. In later versions of the work, probably full conversations will be exemplified.

[11] http://dip.semanticweb.org/

indicating whether the test was accepted or rejected, and if accepted, the result of the test, while the *Service Provider* awaits the reception of a single message named *MCompleted* accounting for both of them.

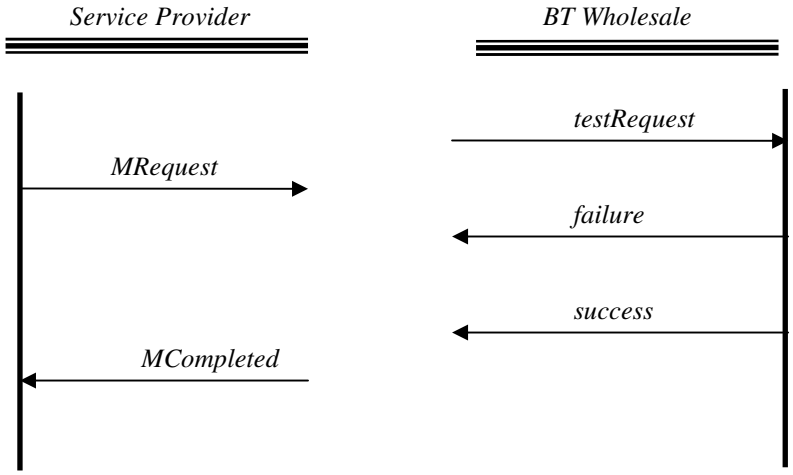


*Figure 2: Request-Response and In-Multi-Out Message Exchange Pattern*

| | |
|---|---|
| ***Elements*** | *testPerformer (testPerformerCustomerID, testPerformerPartyID), testConductor (testConductorAgencyID, testConductorPartyID), testReference (testIdentifier, testDate), testCategory, testParam1, testParam2, …, testParamn, testResult1, testResult2,…, testResultn, testOK* |
| ***Documents*** | *DRequest, DCompleted* |
| ***Messages*** | *MRequest, MCompleted* |
| ***Documents-Elements mapping*** | *DRequest.{ testPerformer, testConductor, testReference, testCategory, testParameters}*<br>*DCompleted.{ testReference, testCategory, testOK, testResults }* |
| ***Messages-Documents mapping*** | *MRequest.{ DRequest }*<br>*MCompleted.{ DCompleted }* |

*Table 6: Service Provider structural model*

### 6.1 Structural Model

The service provider follows the message exchange pattern "*request-response*" based on the structural model[12] *tPontStr*, detailed in Table 6.

The test conductor follows the message exchange pattern "*in-multi-out*" which uses the structural model *eCoStr*, detailed in table 7.

| | |
|---|---|
| ***Elements*** | *performer (customerID, performerPartyID),*<br>*conductor (agencyID, conductorPartyID),*<br>*reference (identifier,date),*<br>*category,*<br>*parameter1,parameter2, …, parametern,*<br>*result1, result2,…,resultn,*<br>*accepted* |
| ***Documents*** | *requestDoc, successDoc, failureDoc* |
| ***Messages*** | *testRequest, failure, success* |
| ***Documents-Elements mapping*** | *requestDoc.{ performer, conductor,*<br>*reference, category,*<br>*parameters }*<br>*successDoc.{ reference, category,*<br>*accepted, results }*<br>*failureDoc.{ reference, category, accepted }* |
| ***Messages-Documents mapping*** | *testRequest.{requestDoc }*<br>*success.{ successDoc }*<br>*failure.{ failureDoc }* |

*Table 7: BT Wholesale structural model*

| | ***Service Provider*** | ***BT Wholesale*** |
|---|---|---|
| ***Actions*** | *sendMRequest* | *outFailure,*<br>*outSuccess* |
| ***Boolean Expressions*** | *receiveMCompleted* | *recTestRequest* |

*Table 8: Behavioral model of both parties*

[12] With the aim of facilitating the reader understanding *testPerformer*, *testConductor*, *testReference* are presented at this point as complex data structure. However, as far as SOPHIE concerns they are single elements, built as a result of the concatenation of their constituents. The same principle applies to the elements *performer*, *conductor* and *reference* of the BT Wholesale structural model.

### 6.2 Behavioral Model

As far as the behavioral model concerns, Table 8 depicts the most relevant entities required to assert interoperability among the behavioral models *tPontBhv* and *eCoBhv*.

In [Arroyo, 07] the complete ontologies that describe the domain knowledge and choreographies of both parties are carefully depicted.

## 7 Conclusion and future work

This paper has presented the ontology and grammar of **SOPHIE**, an extensible knowledge management conceptual framework that is especially suitable for supporting the fine grained interaction among services following different structural, behavioral or semantic models. **SOPHIE** elaborates on current existing initiatives trying to rise above their limitations with the addition of a computational semantics and a clear separation of concerns that help to overcoming intrinsic service heterogeneity. On the one hand, the **SOPHIE** ontology allows to semantically drescribing the conversational patterns of intereacting services, with the aim of, as a result of a reasoning task, producing the intermediate models that enable communication among heterogeneous parties in knowledge intensive settings. On the other hand, the clear separation of concerns facilitates to easily change, add or modify the different underlying formalisms, without impacting other concerns or the architecture itself.

The applicability of **SOPHIE** as semantic framework for the integration of choreographies has been demonstrated, as a solution to the compatibility of data and interchange details between parties engaged in business collaboration. **SOPHIE** can thus be considered an extension to existing languages and formalisms dealing with choreography that allows the reconciliation of divergences that are common in concrete implementation of typical scenarios. A detailed example of such capability centred in KM for the Telecomunications field.

The mapping provided in this paper can be further extended and revised for concrete application profiles, and it is essentially intended to provide a concrete realization of an existing ontology of KM [Holsapple04], thus sharing with it the objective of providing a foundation for systematic KM research study and practice.

Future work should deal with improving the abilities of the choreography service, by extending the different conceptual models with the addition of new formalisms remains open. Also, the definition of a mapping language that permits executing an operational model in traditional workflow engines represents an interesting research field. Areas of this work that definitely need to be researched are the assessment of performances, QoS and security, as the discussion about these topics is well out of the scope of this work. In addition to that, the extension to support different communication frameworks constitutes an interesting research area.

**Acknowledgements**

The work is funded by the European Commission under the project LUISA (IST – FP6-027149), and by the Comunidad Autónoma de Madrid under the project PAWSEL funded by the CAM-PRICYT program.

## References

[Arroyo, 07] Arroyo, S (2007). SOPHIE: A choreography framework for semantic services. PhD Thesis.

[Arroyo, 06] Arroyo, S., López Cobo, J.M. and Sicilia, M. (2006). Patterns of Message Interchange in Decoupled Hypermedia Systems. Journal of Networks and Computer Applications. 2006 (to appear).

[Arroyo, 06] Arroyo, S., Duke, A., López-Cobo, J. M. and Sicilia, M. A. (2006). A Model-driven Choreography Conceptual Framework. Computer Standards & Interfaces. 2006 (to appear).

[Booch, 99] Booch, G., Rumbaugh, J., and Jacobs, I.: "The Unified Modelling Language User Guide, Addison-Wesley, 1999.

[de Bruijn, 04] de Brujin, J. (editor), Polleres, A., Lara, Ruben, Fensel, D.: "D20.3 v0.1 OWL Flight" WSML Working draft, http://www.wsmo.org/2004/d20/d20.3/v0.1/20040823/, 2004.

[Fensel, 01] Fensel, D.: "Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce", Springer-Verlag, Berlin, 2001.

[Gruber, 93] Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications, Knowledge Acquisition, 5:199-220, 1993.

[Holsapple, 04] C.W. Holsapple, K.D. Joshi, A formal knowledge management ontology: Conduct, activities, resources and influences, Journal of the American Society for Information Science and Technology 55 (7) (2004) 593–612.

[Maedche, 03] A. Maedche, B. Motik, L. Stojanovic, R. Studer, R. Volz, Ontologies for enterprise knowledge management, IEEE Intelligent Systems 18 (2) (2003) 26–33.

[Marwick, 01] A.D. Marwick, Knowledge management technology, IBM Systems Journal 40 (4) (2001) 814–830.

[Mohame, 04] A. Mohame, S. Lee, S. Salim, An ontology-based knowledge model for software experience management, Journal of Knowledge Management Practice 5 (2004).

[Roman, 04] Roman, D., Scicluna, J., Feier, C., (eds.) Stollberg, M and Fensel, D.: "D14v0.1. Ontology-based Choreography and Orchestration of WSMO Services", http://www.wsmo.org/TR/d14/v0.1/, March, 2005.

[Roman, 04] Roman, D., Lausen, H. and Keller, U. (eds): Web Service Modeling Ontology. WSMO Working Draft v0.3. http://www.wsmo.org/2004/d2/v1.0/, 2004.

[Sicilia, 06] Sicilia, M. A. Lytras, M., Rodríguez, E., García-Barriocanal. (2006).Integrating descriptions of knowledge management learning activities into large ontological structures: A case study. Data & Knowledge Engineering 57 (2006) 111–121.

[Wagner, 06] Wagner, F. (2006). Modeling Software with Finite State Machines: A Practical Approach", Auerbach Publications.