Task-Aware Location-Based Services for Mobile Environments

FP7-SME-207-1-222292-TALOS

Task Model and Authoring Tool D2.1

Deliverable lead contractor: IMIS

Anastasios Arvanitis, IMIS anarv@imis.athena-innovation.gr
John Liagouris, IMIS liagos@imis.athena-innovation.gr
Alexandros Efentakis, CTI efedakis@cti.gr
Eleni Tsigka, ISST eleni.tsigka@isst.fraunhofer.de
Stefan Pfennigschmidt, ISST stefan.pfennigschmidt@isst.fraunhofer.de

Due date: 31.3.2010
Actual submission date: 29.4.2010

Abstract

Presents a generic task modeling framework along with its properly adapted version for the purposes of TALOS. It also describes the overall management of task ontologies in TALOS along with the Task Ontology Authoring Tool (TOAT) that will be used by the SMEs in order to define tasks for the mobile users based on the established task model.

Research Academic Computer Technology Institute, Greece
Fraunhofer Gesellschaft, Institute for Software and Systems Engineering, Germany
Institute for the Management of Information Systems / Athena Research and Innovation Center in Information, Communication and Knowledge Technologies, Greece
Katholieke Universiteit Leuven, Belgium
Michael Müller Verlag, Germany
Talent SA, Greece
WiGeoGIS, Austria

Table of Contents

1 Introduction

One major reason for the difficulties in searching, finding and selecting suitable services is that User Interfaces (UIs) are currently designed from the view point of the domain. By using keywords, one has to follow the menu provided, "translate" what he/she wants to do in terms of the menu and finally reach the appropriate services [1].

A common sense already implied in the previous paragraph is that users organize their everyday lives around solving problems (tasks) and thus both services and content should be structured around tasks in order for them to be easily discovered and assimilated. This idea is strongly encouraged by the enlightening evaluation of NTT DoCoMo's[1] task-based approach conducted in 2004. As it is shown in [2], the percentage of users reached the appropriate services by employing a keyword-type search through their handsets was no greater than 16%, whereas in the existence of a task-oriented search interface the corresponding percentage grew up to about 63%. According to the same test, it is also astonishing that 50% of the latter (one out of two) reached the services within five minutes, compared to just 10% (one out of ten) of the keyword-type search users.

Precondition for providing automated task-based services is the formal description of the potential tasks. This procedure amounts to the construction of a task model or, in other words, a so-called ***Task Ontology***. Up to now, Task Ontologies have been used in various fields of Computer Science, from Artificial Intelligence and Expert Systems [3, 4] to Geographical Information Systems [5] and UI Modelling [6]. In general, the reason for their popularity lies in that they provide a very flexible way for representing problem solving procedures, mainly because they facilitate sharing and reuse of knowledge along with automated reasoning capabilities. In the context of TALOS, each Task Ontology includes the specification of the task attributes and parameters (e.g. name, input, necessary and/or sufficient conditions for accomplishing a task) and the definition of the relations between different tasks such as subsumption and temporal ordering.

The remaining document is organised as follows. Section 2 clarifies the notion of ontology and explains how Task and Domain Ontologies can be used in modelling users' tasks along with existing services and resources. Having a good picture of these issues, the requirements and specification

---

[1] NTT DOCOMO (http://www.nttdocomo.com) is Japan's premier provider of leading-edge mobile voice, data and multimedia services having more than 54 million customers.

of ToDo, a Description Language for modelling tasks, are then provided in Sections 3 and 4 respectively. Section 5 describes the overall management of task ontologies in TALOS. Section 6 presents the Task Ontology Authoring Tool (TOAT). We conclude in Section 7 where we describe the TALOS Server Database.

2 The Concept of Ontology

The term Ontology is used in a variety of fields, from Philosophy and Biology to Computer Science and Artificial Intelligence. In each of these fields, ontologies serve for different purposes and as a result they come with different meanings and definitions. For reasons that become clear in the following, this section focuses only on the notion of ontology from the view point of the **Semantic Web** [7].

In general, the realization of the Semantic Web imposes the enrichment of the current (syntactic) web with firm and "globally accepted" semantics, i.e. with application-independent metadata (data describing the data) that are based on logical formalisms like First Order Logic (FOL). Despite all the arising obstacles, this procedure is essential in that it enables software agents to share, reuse, compose and process the exchanged information automatically. Ontologies play a key role in this effort. In fact, the reason for their popularity lies in that they provide an elegant solution to the aforementioned problem.

A simplified definition of the term *Ontology* in Semantic Web terminology could be the following:

DEFINITION 1: *An ontology is a formal specification of a conceptualization.*

Each ontology of the Semantic Web consists of two parts:

- A *vocabulary* (intentional knowledge) that consists of *concepts* (aka *classes*) and *relationships* (aka *properties* or *roles*). Classes are regarded as sets of *individuals* that share at least one common attribute, while properties are sets of pairs of individuals and denote a relationship between the members of each pair.
- An *additional knowledge* (extensional knowledge) that consists of individuals, class and property *assertions*. A class assertion denotes that a specific individual belongs to a class, while a

property assertion assigns a pair of individuals to a specific property. Analogously to the terminology of the object oriented programming paradigm, individuals are also called *instances* and they usually represent resources of the World Wide Web.

A usual claim in literature is that each ontology has its own vocabulary. This can be explained easily with the following example:

Let we want to model (a) a genealogy tree and (b) our knowledge about the existing varieties of wine. In the first case, we have to use concepts such as "Father", "Grandmother" and relationships like "has descendant", "are siblings" etc. The reader can easily assume that the instances of this ontology will represent humans. However, as far as the wine domain is concerned, the previous classes and properties are obviously completely useless. Here we have to use concepts such as "Red Wine", "Sour Wine" and properties like "has flavour", "has colour" etc. Instances in this case could be specific wines such as "Zinfandel" and "Cabernet".

Concrete examples of such two (domain) ontologies can be found in [8] and [9]. Besides their XML syntax that is used for achieving platform-independence, easy storage and efficient processing by software, these ontologies can be visualized in a human-friendly way using an ontology editor like [10,11,12]. An abstract 2D graph-like representation of a simple ontology about family relationships is given in Figure 1.

Ellipses in Figure 1 denote classes, while rectangles stand for individuals. A line between two classes denotes an "IS-A" (subsumption) relationship. Such subsumption relationships form a so-called *Class Hierarchy*. Dotted lines are used to visualize class assertions. We point out that there are not property assertions in this example and that, depending on the expressivity of the ontology, subsumption relationships may also exist between two properties (e.g. "has daughter" can be defined as a subproperty of "has kid").

Although the graph-like representation gives us a good picture of the domain of interest, i.e. the family, however it conceals a very important feature of the ontology, the *axioms*. Figure 1 can be misleading as it seems to illustrate an a priori categorization. However, each complex class (e.g. Uncle), assertion and relationship between classes, properties and individuals is internally expressed in the form of axioms based on a logical formalism. From this point of view, the relationships in Figure 1 are derived from a set of axioms (the so-called Knowledge Base - KB) and can be enriched with new (implicit) axioms when the latter are discovered through an inference procedure.

The dominant family of logical formalisms for constructing ontologies for the Semantic Web is Description Logic (DL) [13]. DL is probably the most thoroughly understood logical formalism, but this is in a great extent only a consequence of its popularity. The actual reasons that led to the prevalence of DL in Semantic Web applications are the following:

- In contrast to other logical formalisms such as FOL, the syntax of DL is human-friendly and its semantics are set-theoretic and reminiscent to those of the object oriented programming paradigm.
- DL is decidable and provides efficient sound & complete algorithms (e.g. Tableau procedure [14]) for reasoning over the described knowledge. Highly optimized tools [15,16], aka Reasoners, have been developed and are already used in various applications, some of which are very similar to our work in TALOS [17].

We point out that reasoning over a DL ontology amounts to (i) check the logical consistency of its axioms and (ii) infer implicit knowledge out of the explicit one. In Section 4.5 we give representative examples of how (i) and (ii) are exploited in TALOS for checking the semantic correctness of the created Task Models and recommend context-specific services to the users.

As already mentioned, besides the abstract graph-like representation, each ontology must be expressed in a syntax that is easily processed by software. After years of research, the XML-like languages proposed for solving this problem are Resource Description Framework Schema (RDFS) [18] and Web Ontology Language (OWL) [19]. Both are already standards of the World Wide Web Consortium (W3C) [20]. The former represents axioms as triples of the form <Subject, Predicate, Object>, while the latter is based on Description Logic.

Having this basic knowledge about what an ontology stands for, we can now proceed with describing the different kinds of ontologies in the Semantic Web.

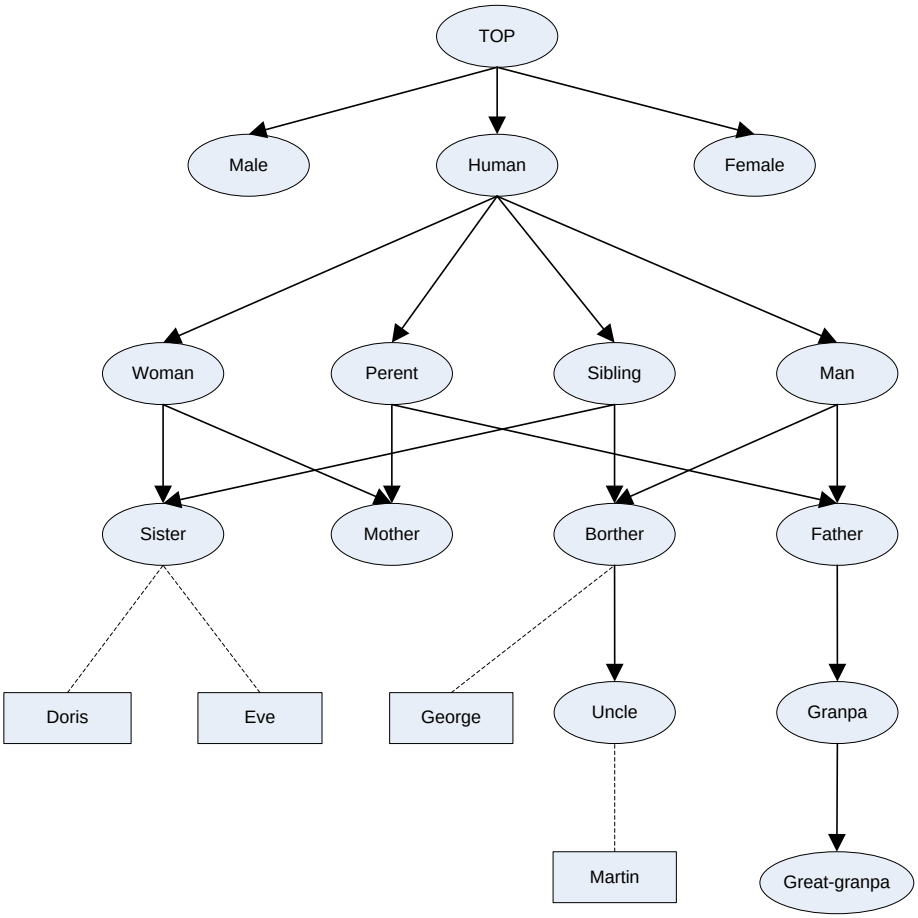

Figure 1: Family Ontology

2.1 Upper and Domain Ontologies

All previous examples refer to Domain (or Domain-specific) Ontologies. As its name disclosures, a *Domain Ontology* is a model of a specific domain (e.g. "Vehicles", "Food", "Genes", "Poker" etc.) and thus it captures concepts and relationships from the view point of the corresponding domain. For instance, assume that we have two domain ontologies, one for "Brain" and one for "Computer". A concept "Memory" in the first case could be used to classify parts of the brain that are somehow related to the human memory, while the same concept in the other ontology would probably refer to the main (RAM) and secondary (hard disk) memory of the machine.

Besides the domain-specific ones, there are also ontologies that model concepts and relationships among objects which are applicable to a wide range of domains. These are the so-called *Upper* (or *Foundation*) *Ontologies* and can be used in constructing various domain-specific ontologies. Representative examples of upper ontologies are SUMO

(Suggested Upper Merged Ontology) [21] and Dublin Core [22]. To get an idea of what an upper ontology represents, SUMO contains heterogeneous entities from "Units of Measurement" (Meter, Farad, Tesla etc.) to "Linguistic Atoms" (word, verb, noun etc.).

Before addressing Task Ontologies in the next section, we point out that some domain ontologies can also be used as foundational in case we want to express concepts and relations that belong to a specific domain but are also applicable in others (e.g. Time and Space).

2.2 Task Ontologies

Task Ontology is a newly introduced term referring to a formal model of tasks. According to the domain of interest, a task may represent a software procedure (e.g. "Sort an array of integers"), a business process (e.g. "Review the proposals") or even a simple human activity (e.g. "Cook food"). As we briefly mentioned in the introduction of this document, Task Ontologies have gained much popularity over the last years, mainly because they facilitate sharing and reuse of knowledge along with automated reasoning capabilities.

In contrast to other modeling approaches such as those followed in the Unified Modeling Language (UML) [23] and Business Process Modeling Notation (BPMN) [24], Task Ontologies go beyond the human-oriented description of the "world" by introducing formal semantics, i.e. logic underneath the model. To be more specific, both in UML and BPMN diagrams, formality is restricted in that the components used to visualize the entities of the "world" are standard. No formal semantics neither automated reasoning procedures exist up to now. As a consequence, this lack of logic can lead to unnoticed human errors, sometimes difficult to find, that violate the semantics of the model. Figure 2 provides an example of this kind between a UML Sequence Diagram and a UML State Diagram.

These diagrams describe the functionality of an ATM. In the first one (sequence diagram), the message *"ejectCard"* precedes the *"dispenseCard"* one, whereas in the second case (state diagram) the temporal ordering is the reverse. **This error does not violate the syntax of UML, but it obviously leads to semantic contradiction**.

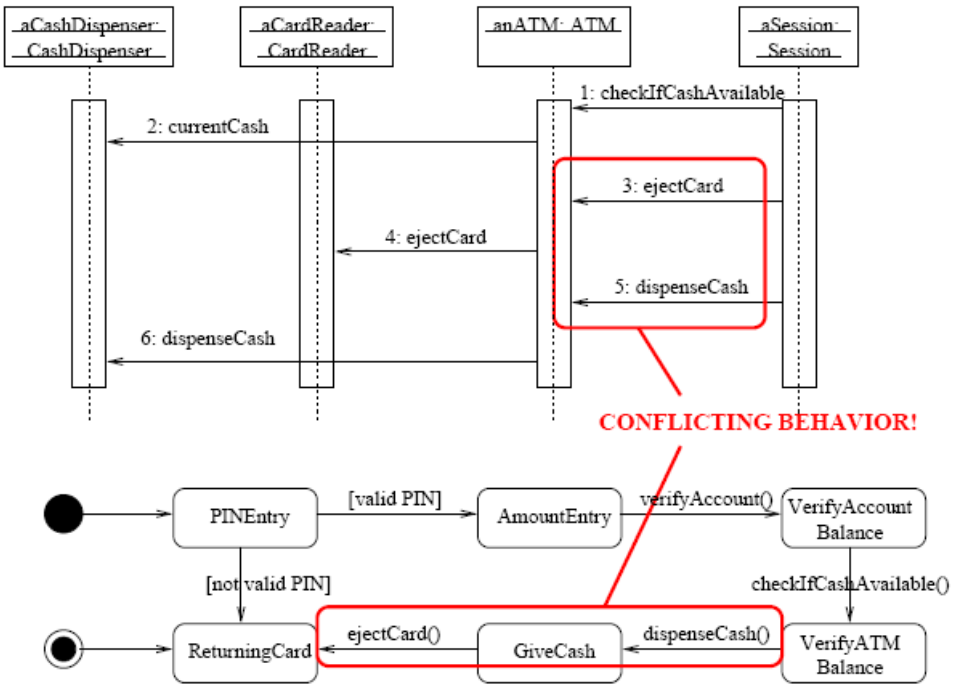

Figure 2: Inconsistent UML diagrams (from [28])

For most people that use UML or BPMN, the lack of formal semantics is actually of no great importance due to the fact that these languages serve as a means for enhancing communication among humans, not software agents. However, in other applications this may be a great disadvantage. **When we need to retrieve, process and exchange information automatically, formal data semantics are essential**. This inevitable requirement triggered numerous research efforts, one of which, the Process Ontology of the OWL-S language [25], is described below.

The Process Ontology has been recommended by W3C as a means for modeling agent processes and it is very close to the notion of the Task Ontology in TALOS. As it shown in Figure 3, the central entity of the ontology is the *Process*. Each process has a set of attributes (input, output, result, condition etc.) that describe the parameters and conditions an agent needs to validate in order to successfully execute the process. Note that the model provides constructors for defining complex processes, i.e. processes built upon a combination of others (sub-processes). This is the most important feature of the Process Ontology, as it enables agents to automatically compose new processes or decompose existing ones. In Section 4 we will show how we exploit this approach when modeling tasks in TALOS.

According to Section 2.1, the ontology in Figure 3 can be regarded as an upper ontology for modeling processes of any kind and it is expressed

in OWL. At this point, there are two things that may seem strange at first sight and need further discussion:

1. As OWL does not support temporal relations (such as sequence), how such an ontology can be expressed without adding the notion of time in Description Logic?
2. According to what we described in the beginning of Section 2, where are the classes, properties and individuals in this kind of ontology?

As far as the first issue is concerned, temporal relationships between individuals in a DL ontology can be represented by object properties like "before" and "after" (with the corresponding characteristics like transitivity, inversion etc.). For instance, let we have two processes *a* and *b*, where *a* precedes *b* in time of execution. In this case we can easily add an axiom of the form *before*(*a*,*b*). The same holds for more complex temporal relations like "overlaps".

As for the second issue, we can make the following assumption. The entity *Process* in Figure 3 is regarded as a class (i.e. a set) of individuals (i.e. specific processes) that share all the parameters and conditions shown in Figure 3 as attributes. In other words, the specific processes are represented as instances (individuals) of the class "Process" that are connected through the property "hasInput" with other individuals that belong to the class "Input" and so forth.

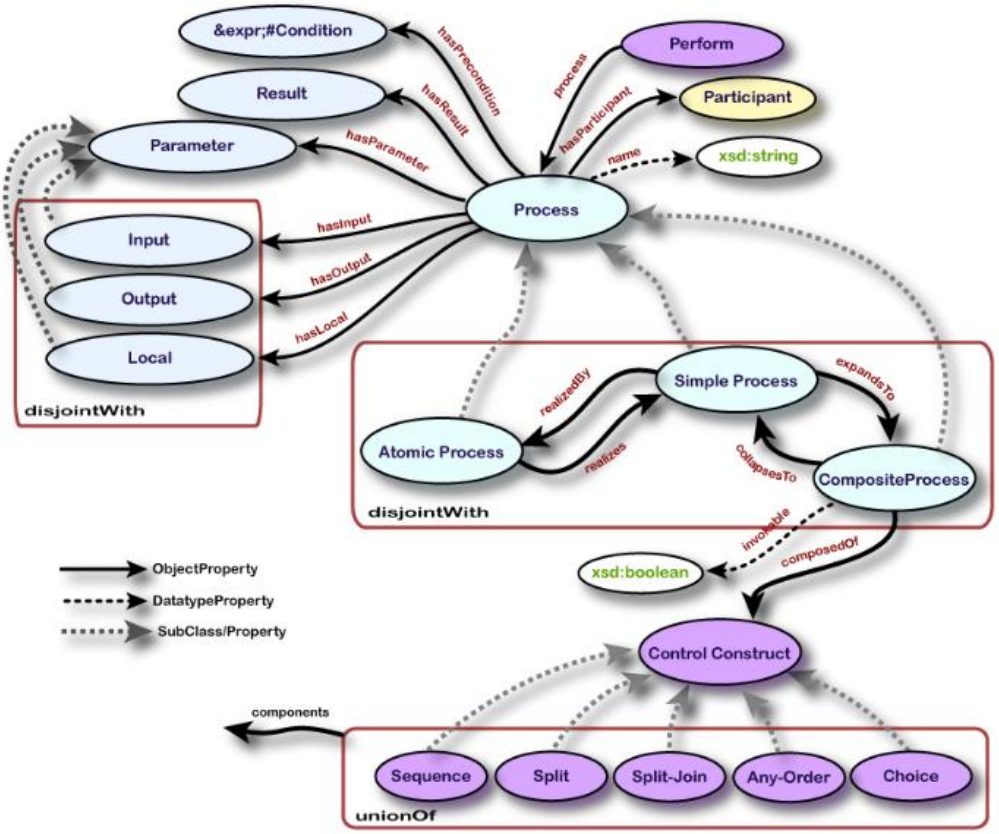

Figure 3: The Process Ontology of OWL-S

2.3 Task Ontologies in TALOS

Building a Task Ontology in TALOS amounts to modeling what the user of a mobile handset may want to do, e.g. "Go to the Theatre", "Visit a Museum" or "Eat at a Restaurant". **The basic feature of such an ontology is that complex tasks like those mentioned before are broken into simpler subtasks**. In fact, this is a human-like approach for performing a task. When someone wants for instance to go to a theatre, he/she first looks for the available plays, checks for reviews, confirms the time of the performance and finally plans his/her route to the theatre. NTT DoCoMo follows this approach in its task-based service provision system by applying task ontologies like the one shown in Figure 4.

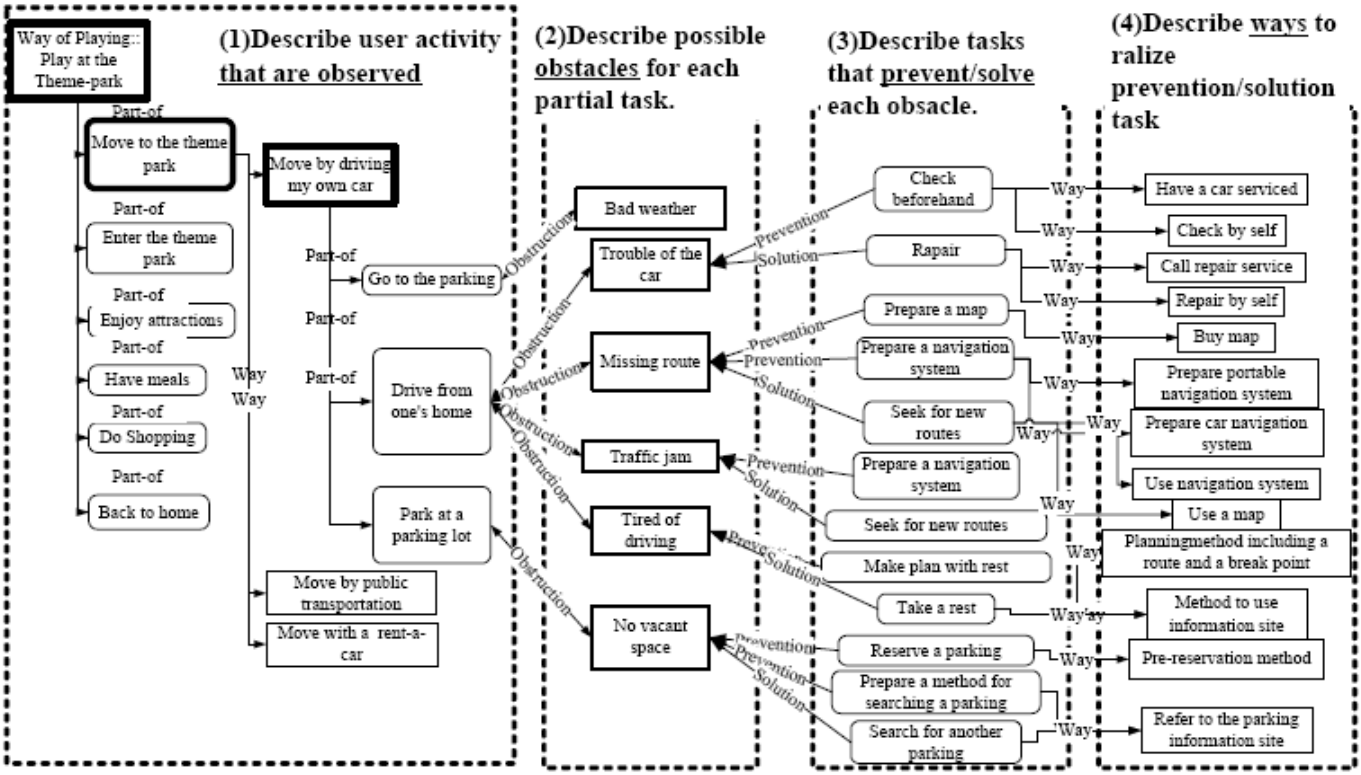

Figure 4: A Task Ontology used by NTT DoCoMo (from [26])

Although, the ontology in Figure 4 is quite different from the task ontologies we propose (see Section 4), its "philosophy" is very similar to the one in TALOS. In both approaches, **Task Ontologies serve as a task-oriented index that is used for retrieving the appropriate content while guiding users to perform a task. Thus, they serve at the same time as an abstract model of the mobile user interfaces (UIs)**. The latter is reminiscent of the well-known Model-View-Controller (MVC) architecture paradigm [27] where the view (what the user interacts with) is based on a (generic) model which changes (through the controller) according to the user's actions. In Figure 5 we give an example of how a mobile user interface based on a task ontology may look like (again from NTT DoCoMo). The three-layer cake of the MVC-like approach followed in TALOS is depicted in Figure 6.

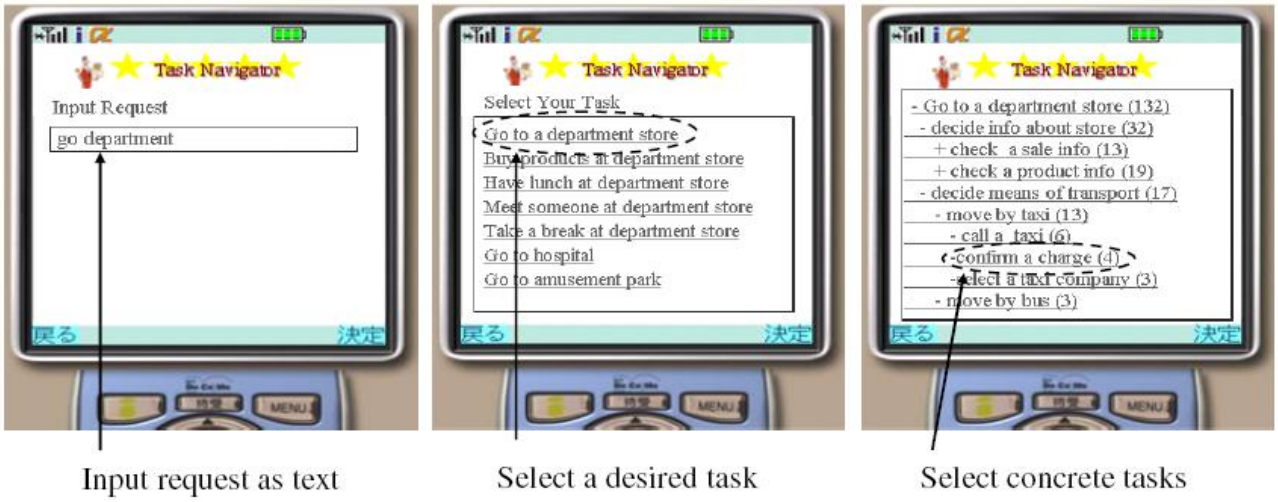

Figure 5: A task-based UI (from [2])

The MVC-like approach provides great flexibility when developing a UI. As we explain in Section 4, providing IT-illiterate people with a drawing tool for constructing Task Ontologies, both the **structure and the basic functionality** of the UIs can be automatically generated and easily updated from the XML files representing these "sketches". However, there are two issues here requiring further discussion. Task ontologies must be (a) **syntactically correct** and (b) **semantically consistent**.

On the one hand, the correctness of the XML syntax in our approach is always guaranteed as we only allow authors to **draw** the ontology, i.e. to draw a simple graph. **The translation of the resulting graphs (along with the underlying task parameters) into XML and the validation of the latter according to a specified XML Schema are automatically performed (in TALOS Server) afterwards.**

On the other, the validation of the XML syntax is not always adequate. **Each entity in the task ontology has additional features whose semantics cannot be captured by a simple XML syntax validation**. Taking into consideration that such features are translated into functionalities within the UIs[2], one can easily understand that we cannot base our UIs on inconsistent ontologies. Thus, as a solution to this problem, we propose an additional check on the semantics of the defined tasks. **This check ensures that the task ontology is consistent with respect to its semantics and it is performed (in TALOS Server) through automated reasoning and after translating the information of the task ontology into axioms of the Web Ontology**

[2] An example of a task feature that is translated into a specific functionality of the mobile application is the way data flow from one task to another when a sequence relation between two tasks is introduced.

Language (OWL). The procedure of expressing a ToDo task ontology in OWL is addressed in Section 4.4.2.

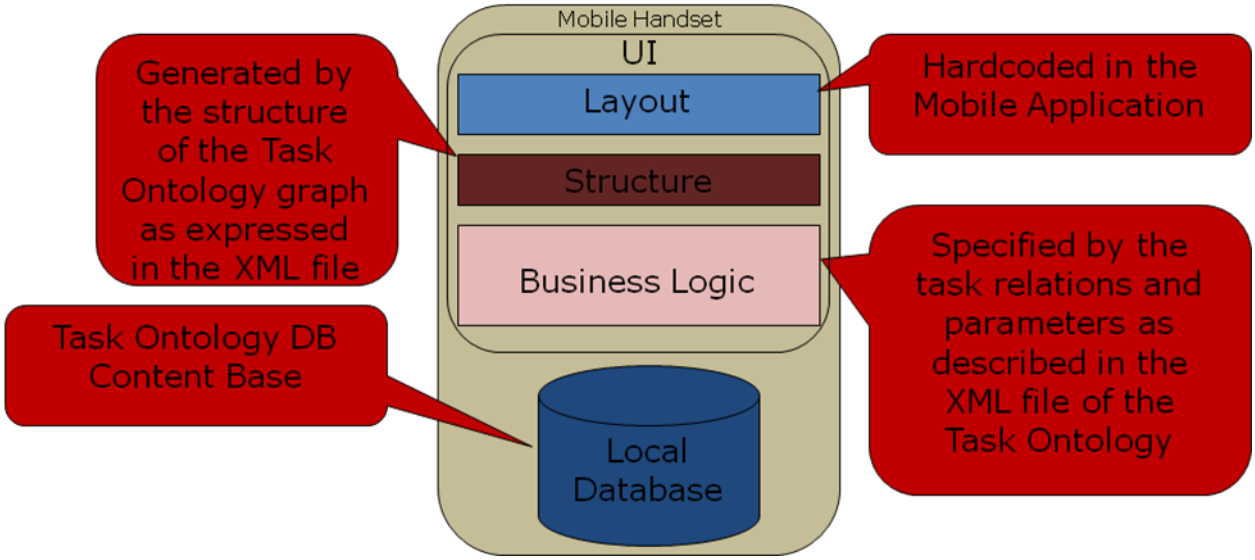

Figure 6: The three-layer cake of the task-based UI architecture in TALOS

2.4 Domain and Context Ontologies in TALOS

When defining a task, we may need to refer to one or more Domain Ontologies for concepts and definitions that are used to describe inputs, outputs, preconditions of the task and so on. We give an example here for better understanding.

Assume that we have the task "Find a Museum". Obviously, in order to perform this task, a user must be provided with a list of all possible museums in the area he/she is interested in. This list is actually a set of instances of the class *Input* (Figure 7) and can be generated from the instances of a Domain Ontology like those we described in Section 2.

Besides task parameters, tasks and the relations among them can also be expressed as a set of ontology axioms. We have already addressed the advantages of this approach. Regarding our example, taking into account that RDFS and OWL ontologies are amenable to automated reasoning, we are able to perform the following:

- ***Classification of resources***
  Instead of just providing a list of museums that may prohibitively grow huge, the classification shown in Figure 8 can be very useful and time-saving when searching for the appropriate museum.

- ***Context-aware filtering of the candidate resources***
  The resources for accomplishing a task alter dynamically according to the current context. In our case, the list of museums provided to the end-users obviously depends on the specified location. Thus, by also describing context in DL (see survey in [28]), we are able to use conjunctive query answering techniques (including both context and content) for the efficient extraction of the most appropriate resources. A similar approach is followed by NTT DoCoMo in [17].

- ***Logical consistency check***
  When constructing the task ontology for "Visit a Museum", one has to pay attention not only to the syntax, but also to the semantics of the model. The former, i.e. the syntax validation, is ensured by restricting authors to construct the ontology through a graphical interface (as a graph) and automatically interpret the graphical notations into XML. However, regardless the correct syntax, there may be declarations in the ontology that contradict one another. From our experience, allowing IT-illiterate authors to arbitrarily define their own entities (tasks, relations etc.) will definitely result in various logical contradictions or redundancies. Thus, taking into consideration that DLs adopt the "open-world assumption", we can reason over the described tasks in order to detect such problems and help authors correct them. Examples of semantic clashes in a Task Ontology are given in Section 4.

In general, instances of *Input*, *Precondition, UserPreference, Output*, *Postcondition* and *Effect* may refer to none or more Domain Ontologies.

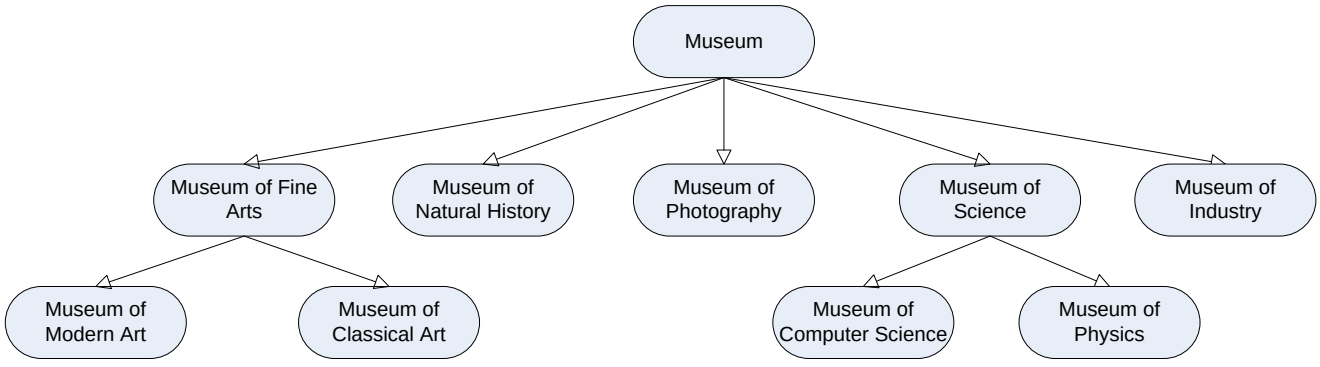

Figure 7: Museum classification in a Domain Ontology

Summarizing the previous paragraphs, ontologies in TALOS are used as illustrated in Figure 6. **Users' tasks are described in the form of task ontologies while domain and context ontologies contain *some* of the data needed for instantiating these tasks**, i.e. for capturing specific users' activities. We point out that although content from travel guides in the form of unstructured text is also used for instantiating users' tasks, however, it is not modeled using ontologies; it is just stored and retrieved into/from a relational database. The only part of information that can be maintained and retrieved using ontology-based techniques breaks into the following categories:

- ***Points of Interest*** **(e.g. museums, parks, hotels etc.)**
  POI-related information can be either **static** (retrieved from travel guides) or **dynamic** material (retrieved from the web). POIs along with their properties (e.g. addresses, operating hours, specific features etc.) are regarded as pieces of well-structured information that is extracted from the overall available (unstructured) content and thus they are stored in TALOS **Content Base**.
- ***Context-related information*** **(e.g. date, time, location, weather[3], traveler types)**
  In the proposed infrastructure, context-related information is retrieved from a TALOS-specific application named **Context Aggregator (CA)** which runs in the mobile device [29]. A part of the context that is related to time, location and user's profile can be modeled using context ontologies [28].

[3] In contrast to other context-related information, weather information is automatically retrieved by CA from the web; for instance, from the weather Web Service provided by Yahoo! (http://weather.yahoo.com/).

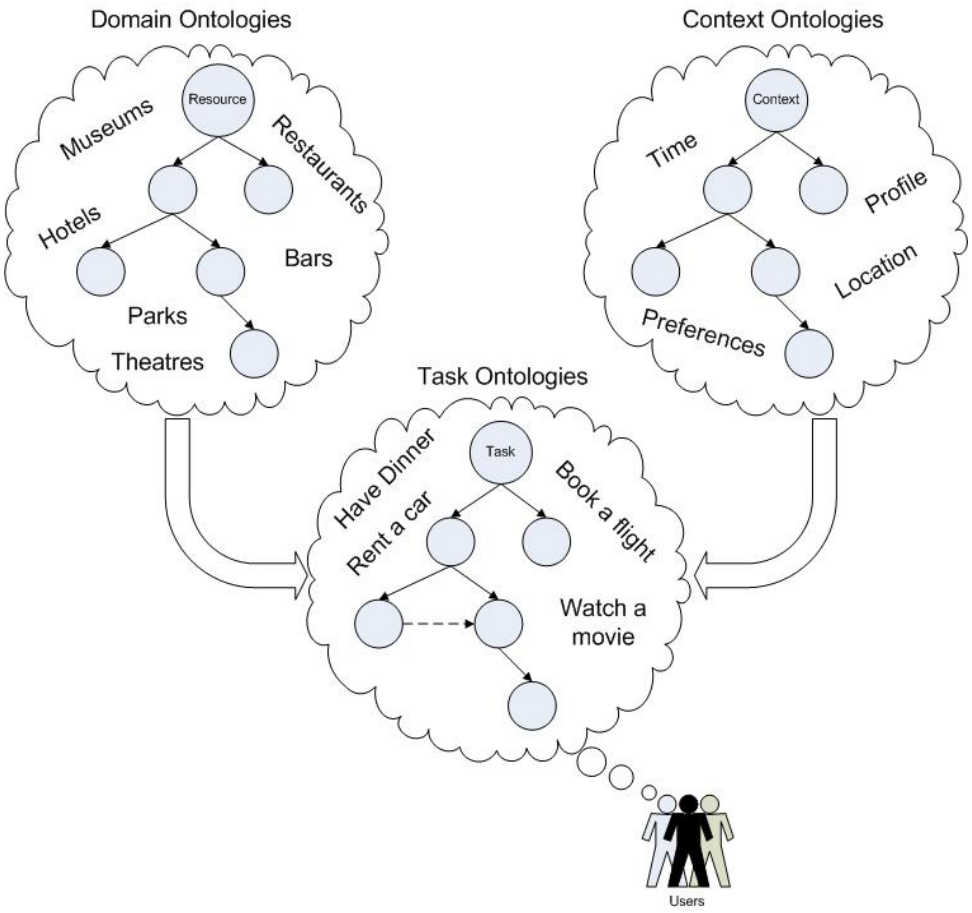

Figure 8: Ontologies in TALOS

3 ToDo Language Requirements

The proposed Description Language, ToDo, for constructing Task Ontologies comes with both a graphical model and an XML-like syntax. According to our perspectives, ToDo must meet the following needs:

- ***High Expressivity***
  All possible tasks in the travel domain and the relationships among them (hierarchy, sequence, dataflow etc.) must be described formally in ToDo.
- ***Easy Usage***
  ToDo must provide the authors of the Task Ontology with a friendly vocabulary, close to their natural language. In other words, the authors must be able to work in a high-level layer of abstraction without the need of learning neither a new formalism nor any confusing technical details.

- ***Ability to Reuse Knowledge***
  Already described knowledge about solving a task must be accessible for reuse in another task if suitable. Thus, ToDo must facilitate the ability to extract parts of solutions related to different tasks and combine them in creating solutions for new tasks. A representative example of this case is knowledge about transportation that may be needed in several Task Ontologies of different domains.
- ***Ability to Reason with Tasks***
  By defining the semantics of ToDo, it is possible to implement practical algorithms for reasoning over the described tasks. On the one hand, such reasoning algorithms could be used to inform authors of the Task Ontology about an inconsistency or redundancy in their model while, on the other, to provide quick and valuable recommendations to the end-users' requests. Guiding users when (re)organising their schedules is a representative example where reasoning over task-related knowledge can be crucial.
- ***Compatibility with current technologies***
  Besides the graphical model that only serves as a means for comprehending the Task Ontology, there must also be a machine-processable representation of it. Thus, for ToDo, we propose an XML-like syntax that is platform independent and fully compatible with the W3C standards (e.g. HTML, XML, RDFS etc.).

4 ToDo Language Specification

The proposed language specification is organised as follows. Section 4.1 introduces the concept of Task and its aspects in ToDo. Section 4.2 describes the two task categories and their meanings as defined in the current specification. Section 4.3 shows how a ToDo Task Ontology can be visualised in order to facilitate the authoring procedure. Section 4.4 introduces the XML-like syntax of the language, i.e. the tags used for denoting the entities and relations existing in a Task Ontology. Section 4.5 provides a detailed description of the logical formalism ToDo is based on. We conclude in Section 4.6 with a simple example of a Task Ontology built with ToDo.

4.1 The concept of Task in ToDo

A task reflects what an end-user wants to do in a high-level layer of abstraction, e.g. "Visit a Museum". Each task is accompanied by a set of attributes (input, output, precondition etc.) and **it is instantiated by context and content in order to become an *activity***. For example, an activity belonging to the previous task is something like "Visit the Museum of Acropolis" as shown in Figure 9. The entity "Museum of Acropolis" is a piece of well-structured content also known as **Point of Interest (POI)**. POIs are stored in TALOS **Content Base** along with unstructured content in the form of text, images etc. (see Section 7). From the application perspective, **the dynamic context corresponding to a specified activity is always retrieved by a module called Context Aggregator (CA)** which runs in the mobile device (see [29]).

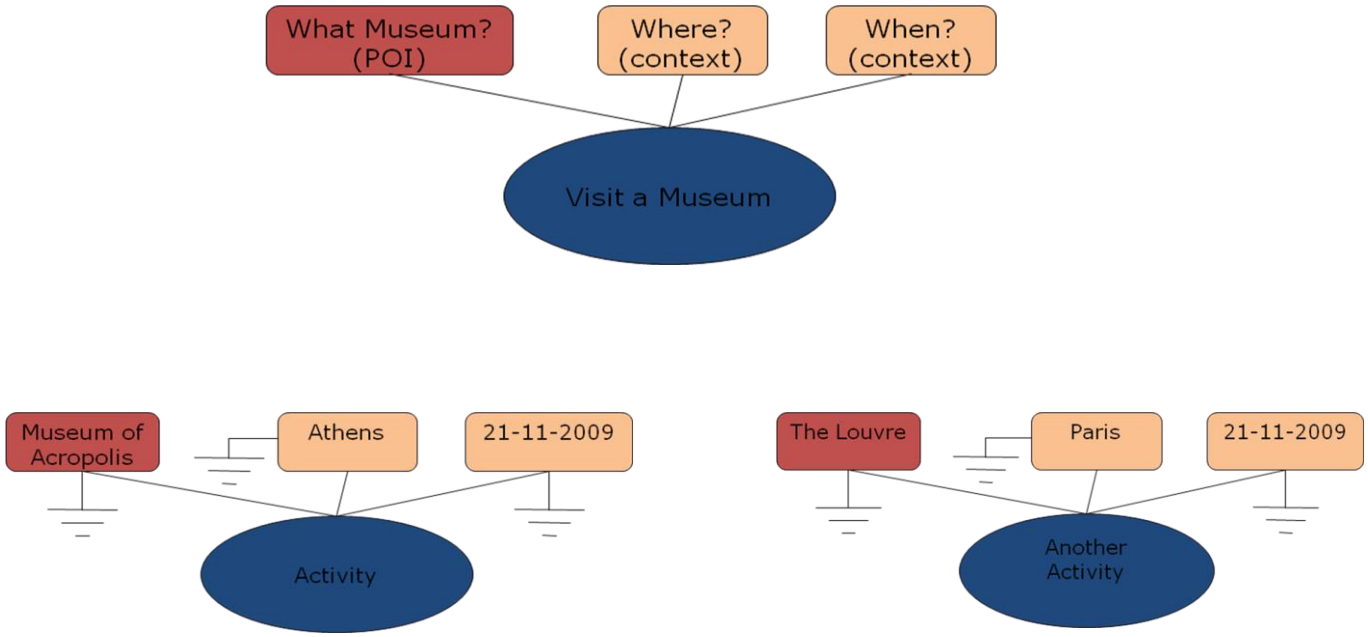

Figure 9: The task "Visit a Museum" as a class of activities that share common types of attributes

Besides the dynamic activity-related context (whether, time, location), there is also a part of user-related context known as **User Profile** which includes a number of features that refer to the traveller type (e.g. backpack traveller, family traveller, business traveller etc.). This part of context is also managed by the CA.

The different types of task attributes in ToDo are depicted in Figure 10. When someone creates a task, he/she actually creates instances of the template illustrated in Figure 10 and thus the proposed structure serves as the bottom layer of the abstract "nodes" in the constructed Task Hierarchy (see Section 4.3).

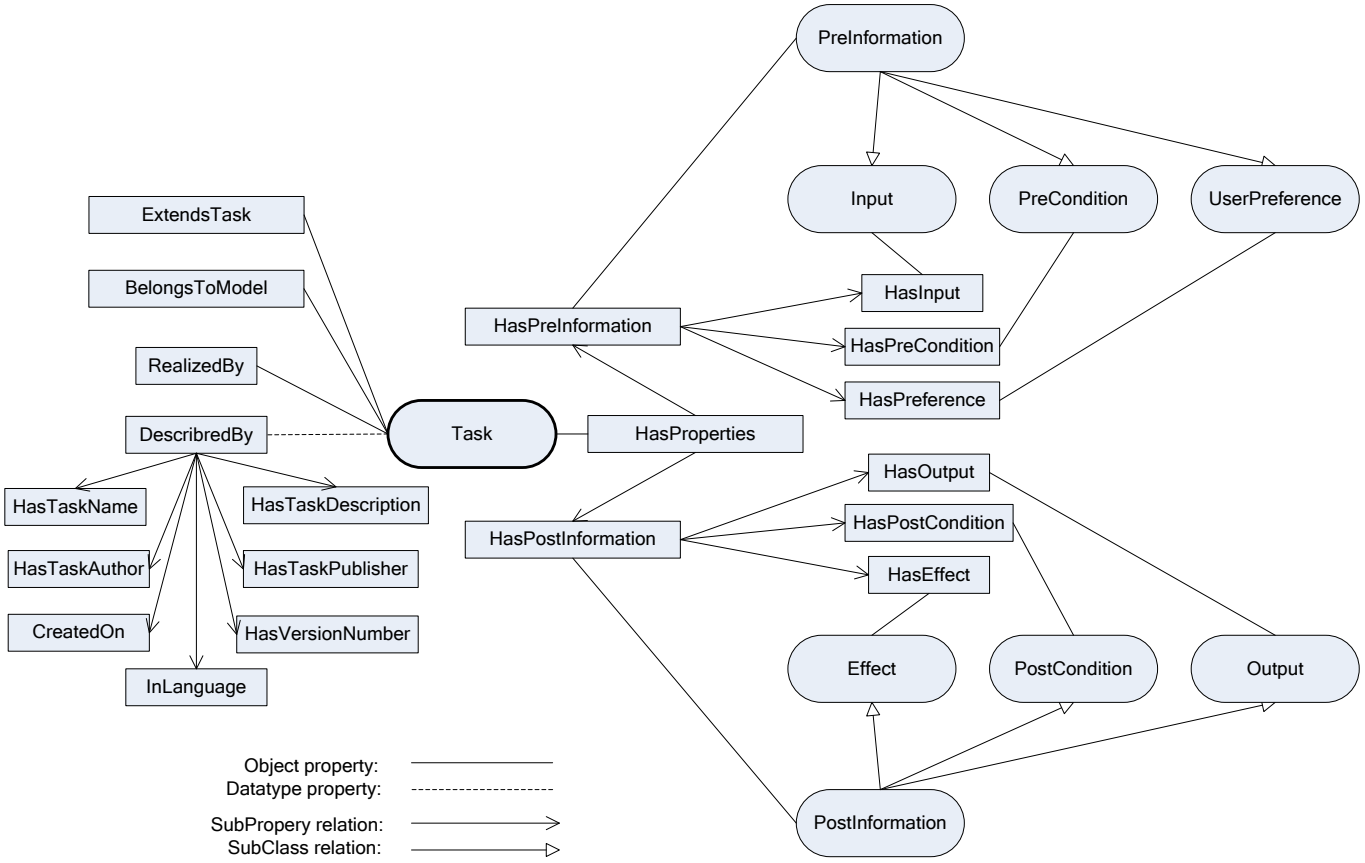

Figure 10: Task structure in ToDo

ToDo is task-oriented, i.e. the basic entity of the language is the ***Task***. Each task has a set of properties that are explained below:

- ***RealizedBy***
  This property specifies that the task is successfully realized by a service. Each service is identified by a unique URI and each task may be realized by more than one services. Web Services in TALOS are mainly used for realizing general tasks, i.e. tasks that break down into simpler ones but can optionally be accomplished by simply redirecting the user to the available Web Service. The task "Book a Hotel Room" is a representative example of this kind.

- ***BelongsToModel***
  This property specifies the Task Ontology a task belongs to. Each Task Ontology is identified by a unique URI (Universal Resource Identifier). A task is defined in only one Task Ontology, but it can be imported to other Task Ontologies for reuse. This is specified by the following object property.
- ***ExtendsTask***
  This property specifies that the task extends another task that is defined in another Task Ontology. The imported task is denoted by its name, version, and the URI of the ontology it belongs to. For instance, the task "Move to the Station" may extend the general task "Move from A to B" as defined in the "Transportation" Task Ontology. **All functional properties** (*Preinformation* and *PostInformation*) **of the imported task are inherited by the task that is extending it**.
- ***DescribedBy***
  This property has several subproperties which are used to describe **non-functional** aspects of a task. In other words, all of the following properties[4] are used for providing task-related information to authors:
  - *HasTaskName*
    This property is used for naming the task, e.g. "Go to the theatre".
  - *InLanguage*
    This property specifies the language in which the task is described, e.g. "English".
  - *HasTaskDescription*
    This property is used for summarizing text information about what the task can perform under what conditions etc. A simple description of the task "Go to theatre" can be like "This task is for helping users to find a play, search for critics about the play, find the theatre, get information about operating hours, and move to the theatre".
  - *HasTaskAuthor*
    This property is used for specifying the name(s) of the task author(s), e.g. "John Liagouris".
  - *HasTaskPublisher*

[4] Except HasVersionNumber that is also used by the TALOS system in updates.

This property is used for specifying the name of the task publisher, e.g. "IMIS/RC Athena".

- *CreatedOn*
  This property is used for specifying the date a task was created, e.g. 2009-09-14. This attribute is automatically generated by the first time a task is specified.
- *HasVersionNumber*
  This property is used for specifying the version of a task. When an author creates or updates a task, its version number is generated automatically. As explained in Section 5.2, versions are needed for updating tasks and for supporting a collaborative authoring environment.

- ***HasProperties***
  There are two types of **functional** properties:

  - Properties for describing information needed for a task before executing it.
  - Properties for describing information after executing the task.

  The first one includes classes of *Input*, *PreCondition* and *UserPreference* which are subclasses of the class *PreInformation*. The second one includes *Output*, *PostCondition* and *Effect* which are subclasses of the class *PostInformation*. All these classes are explained below:

  - *Input*
    An input represents information required for performing a task. Depending on the activity-specific parameters, inputs may be optional. For instance, the task "Find a Restaurant" may take as alternative inputs the exact location of the user in the form of longitude and latitude, as long as an abstract location in the form of the city or neighbourhood name he/she is located in. Task input parameters are classified under (a) **context-related attributes**, e.g. the user's location, and (b) **POI-related attributes**, e.g. a the name (or id) of the mall in the task "Get info about the Mall".
  - *PreCondition*
    A precondition represents conditions that must hold in order for a task to be performed successfully.

Precondition parameters can be (a) **logical expressions applying to context**, and (b) **simple notices** in the form of unstructured text that is used for informing end-users. Regarding the task "Move By Bus", a representative precondition example of the first case is "Current_Time<00.00". A simple precondition notice could be something like "Booking a flight ticket requires a credit card".

- *UserPreference*
  A preference describes a preferred property of the task output. User preferences are **logical expressions applying to** (a) **task-specific input parameters** and (b) **POI-related attributes** that exist in the user's local database. An example of a user's preference in the task "Find a Flight" may be something like "$Flight_Time> 8.00 && $Flight_Time<15:00". Regarding the second case, the preference "POI.Rank=5" can be used in filtering the hotels retrieved from the user's local database [30] when searching for a luxurous one. Such logical expressions are evaluated on-the-fly in order to act as an **optional filter** when the users search for the most appropriate resources or services. From the UI perspective, a *UserPreference* parameter indicates the existence of a screen where the user can specify his/her preferences on the available task inputs and/or POI attributes.

- *Output*
  An output describes information returned after performing a task. Similarly to the case of inputs, task outputs are classified under (a) **context-related attributes**, e.g. type of weather produced from the task "Get Weather Forecast", (b) **POI-related attributes**, e.g. the name and type of a POI that matches the specified input parameters, and (c) **content** in the form of unstructured text that is retrieved either from the **travel guide (static)** or from the **web (dynamic)**. An output of a task may be used as input in other tasks of the Task Ontology. **In the case of TALOS, the default output of a task is the unstructured content retrieved from the travel guide.**

- *PostCondition*
  A postcondition represents conditions that must hold after performing a task. Postconditions are **logical**

expressions which apply to (a) **context-related parameters** and (b) **task-specific output parameters**. An example of a postcondition regarding the task "Move to the Park" could be something like "CA.Position=$POI_Coordinates" where CA.Position is a context-related parameter refering to the user's position (managed by CA) while POI_Coordinates is the position of the POI (i.e. the specified park) defined as an output of the task. From the mobile application perspective, such a postcondition is useful when helping the user (re)organise the schedule of tasks to perform during his/her trip.

- *Effect*
  An effect describes actual events that occur after performing a task, e.g. "The boarding pass is delivered at your email". Effects are **simple notices** in the form of text used for informing users about the effects of the task they performed.

Table 1 summarizes the features of the aforementioned task parameters. From the application perspective, the instantiation of a parameter is done on-the-fly (when the application runs on the mobile handset) with only exception the case where a task author has already specified the parameter value when defining the task (within the Task Ontology Authoring Tool - TOAT). **In ToDo, the symbol "$" is used to denote task-specific variables defined as inputs or outputs of a task. The variables used for capturing the activity-related context (Weather, Location, DateTime, User Profile) are managed by the Context Aggregator and they are denoted by the prefix "CA". Finally, the variables that will be created (by the application) in order to store the user-provided values are denoted by the prefix "Usr".**

Parameter	Applies to	Instantiated By	Example
Input	Context	Context Aggregator	User's Location (longitude , latitude)
	POI	Application	Acropolis Museum
PreCondition	Task Input	Context Aggregator	CA.Weather_Type = 'Sunny'
	Notice	Task Author (a priori)	"You need a credit card to proceed" (text)
UserPreference	Task Input	End-User (through the UI)	Usr.BeginDate <= $Flight_Date $Flight_Date <= Usr.EndDate
	POI		POI.Style=Usr.Style
Output	Context	Context Aggregator	Weather Forecast (object)
	Content	Task Author (a priori)	City History (content)
	POI	Application	List of Restaurants nearby (POI object)
PostCondition	Context	Context Aggregator	CA.Position = $POI_Location
	Task Output	Application	
Effect	Notice	Task Author (a priori)	"Book receipt is delivered by email" (text)

Table 1: Task Parameters in ToDo

4.2 Task Categories in ToDo

In ToDo we define two abstract task categories as shown in Figure 11. Intuitively, the following classification of tasks is based on whether **the activities a user needs to perform for accomplishing the task can break into one or more logically interrelated groups**:

- ***Simple Task***
  A simple (or base) task is directly completed by one and only one group of activities. A representative example of this category is the task "Find a Pharmacy".
- ***Complex task***
  A complex task consists of two or more simple tasks and thus it is accomplished by more than one group of activities. An example of a complex task is a task of planning a trip which consists of several tasks like flight booking, hotel booking, car renting etc. **Complex Tasks collapse to simple ones or other complex tasks**. From the TALOS application perspective, a complex task is regarded as a "blank node" in the overall Task Hierarchy which enclosures the specific tasks a user performs.

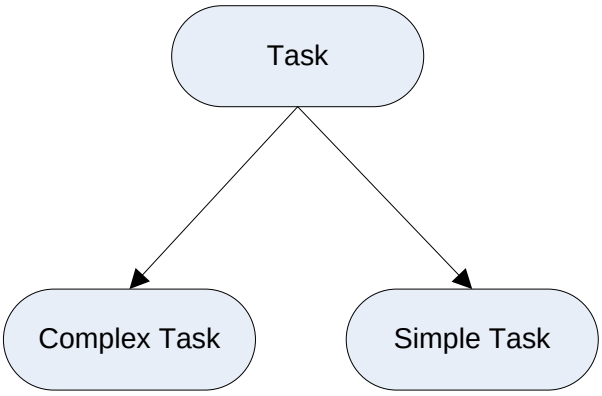

Figure 11: Task Categories in ToDo

4.3 The graphical model of ToDo

In this section we provide the reader with the graphical notations used to represent relations between tasks. As shown in the following figures, **a ToDo Task Ontology is represented in 2D as an abstract directed acyclic graph (DAG)**. In such a graph, nodes represent tasks, while edges represent relations among the latter. ToDo supports four different kinds of relationships between tasks.

4.3.1 SubTaskOf

Tasks in ToDo can be decomposed into (simpler) subtasks. In terms of functionality, this is very helpful when guiding a user. An example of a

subtask relation for a task *C*, let *"Plan a Weekend Trip"*, that breaks into tasks C_1 *"Find Accommodation"* and C_2 *"Plan Sightseeing"* is given in Figure 12. Note that **the following subtasks can be accomplished in any order**.

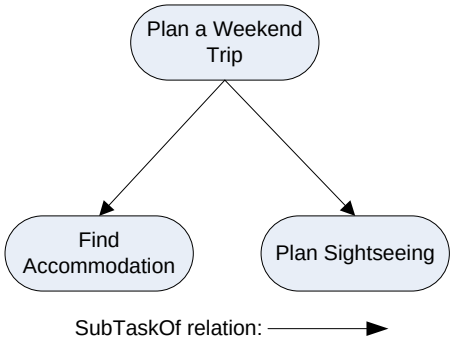

Figure 12: A task *C* broken into two subtasks C_1 and C_2

4.3.2 Sequence

Let C_1 and C_2 be the tasks *"Find a Hotel"* and *"Learn about Facilities"* respectively. It is a common sense that a person must first specify a hotel and then learn about the facilities provided. In this case, the temporal order is represented as shown in Figure 13. Note that **every *Sequence* relation between two tasks introduces a number of parameter bindings from the first task to the other**.

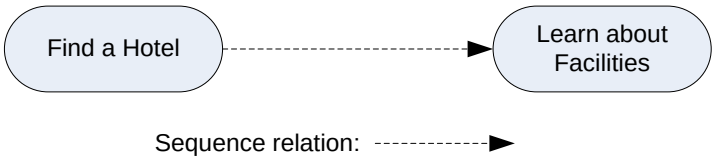

Figure 13: Task C_1 precedes task C_2

4.3.3 OR

The *OR* construct defines an optional relation between two or more tasks with respect to their common parent. Figure 14 provides an example of a task *C* that is accomplished by at least one of the tasks C_1 and C_2. In

this case, C could be the task *"Get info about a Mall"*, while C_1 and C_2 could be the tasks *"Find Shops"* and *"Find Restaurants"* (in the Mall).

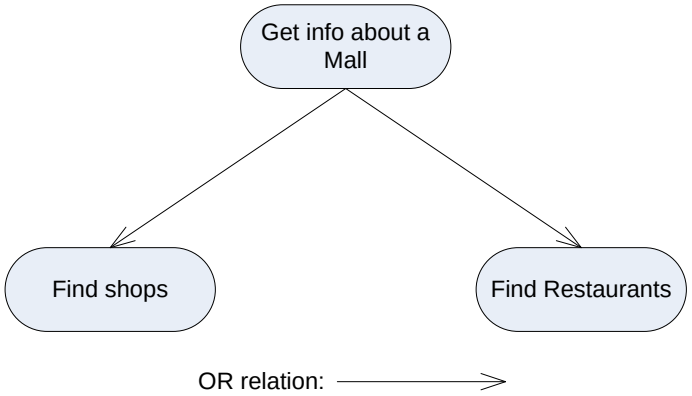

Figure 14: Task *C* is accomplished by at least one of the tasks C_1 and C_2

4.3.4 CHOICE

Similarly to the previous one, the *CHOICE* construct describes an exclusive option between two or more tasks with respect to their common parent. For example, let we have the tasks C_1 "Move to the Theatre", C_2 "Move by Bus" and C_3 "Move by Train". If we assume that a person cannot combine both bus and train in order to reach the theatre, we represent this knowledge as shown in Figure 15.

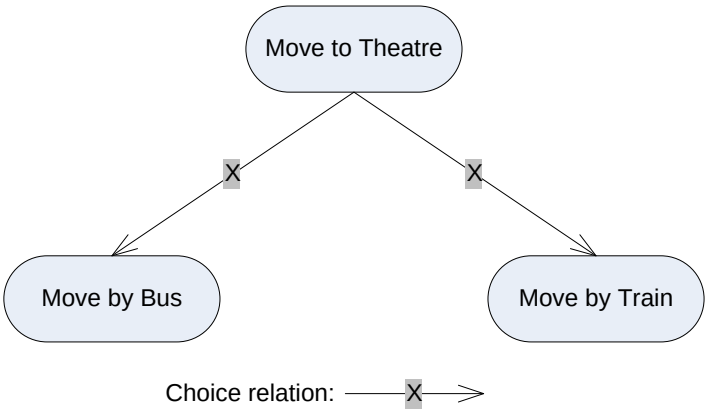

Figure 15: Task *C* is accomplished by exactly one of the tasks C_1 and C_2

4.3.5 GROUP

As its name disclosures, the *GROUP* notation is used for defining groups of tasks. It is just syntactic sugar and it does not introduce an additional relation between tasks. **A *Group* construct is equivalent to an anonymous task which is the parent of all tasks included in the group**. Besides the plain *SubTaskOf* relationship, the children-tasks may also share an *OR* or a *CHOICE* relationship with respect to their anonymous parent and thus we define three different types of *GROUP*s: *SUB*, *OR* and *CHOICE* groups. *GROUP*s can also be nested, i.e. a group may have another group as child which may also have another one as child and so forth.

GROUP constructs are included in ToDo just for helping authors describe complex tasks without the need of drawing complicated graphs. We make this clear in the following examples.

Let a task named *"Sightseeing"* that breaks into four subtasks: "Find a Site", "Learn about Events", "Get Ticket Prices", and "Move to Site". Assume that the output of the first subtask (the specified site) is used as input in all other subtasks. In this case, in order to avoid drawing three different *Sequence* relations between the corresponding pairs of tasks as shown in Figure 16, the author can easily group the three latter tasks and draw only a sequence relation between the first task and the group as shown in Figure 16.

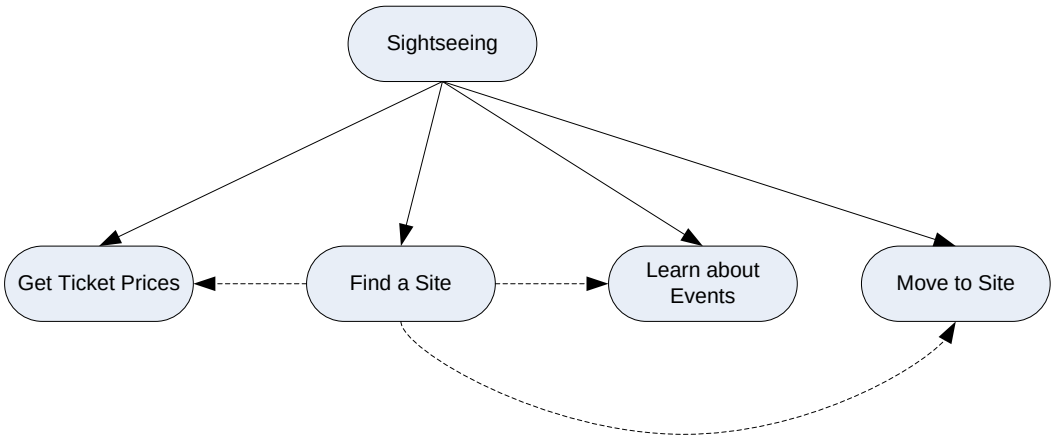

Figure 16: An example of multiple *Sequence* relations

Note that, although in this example the output of the task *"Find a Site"* goes as input to all tasks included in the group, however, this is not mandatory. In general, **a *Sequence* between a task and a group introduces at least one parameter passing from the first task to at least one of the tasks included in the group.** The same "at least"-restriction holds also in the reverse case where the group precedes a task

in time of execution. We emphasize that the parameter passing, i.e. which tasks take as inputs what outputs of other tasks, is not visualized in the 2D graph. In the context of TALOS, this binding of parameters is defined by the authors through a simple form provided in the Task Authoring Tool (see section 6.2).

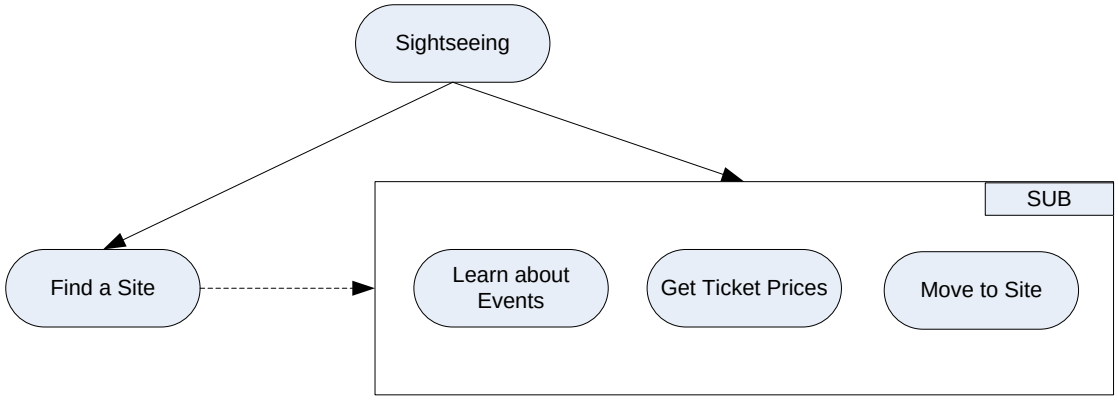

Figure 17: Defining multiple *Sequences* using the *GROUP* construct

Consider another example. Let a task *C "Watch a Movie"* that is accomplished by either accomplishing tasks C_1 *"Find a Cinema"* and C_2 *"Move to the Cinema"* (where C_1 is prior to C_2) or one of the tasks C_3 *"Find a Video Club"* and C_4 *"Learn TV Programme"*. In this case, we can easily describe the complex task hierarchy by using the *GROUP* notation as shown in Figure 18. The left group is of type *SUB*, while the right one is of type *CHOICE*.

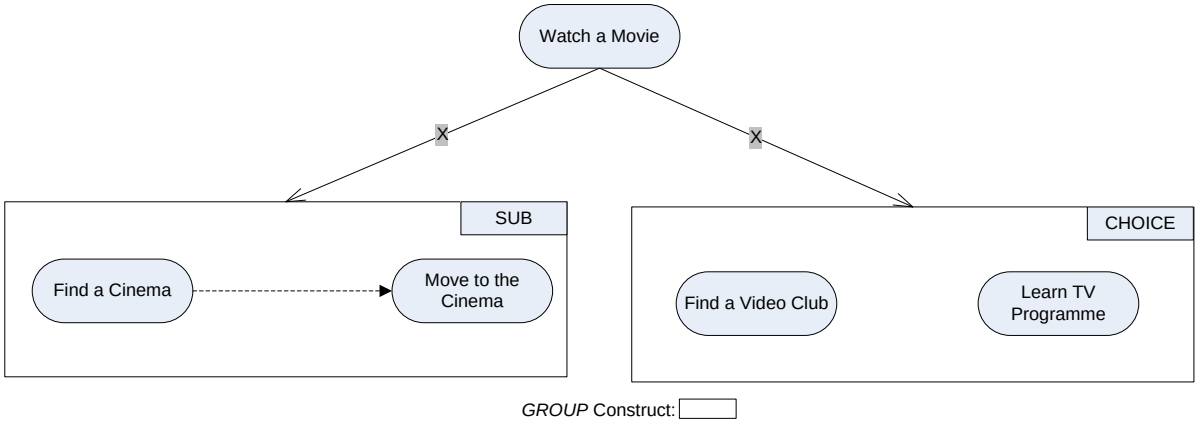

Figure 18: A complex task that breaks into two groups of subtasks

Note that the previous complex task (*"Watch a Movie"*) can also be described using only the *CHOICE* and *SubTaskOf* relationships but in this case the author must explicitly define two new parent-tasks (*"Go to*

Cinema" and *"Watch at Home"*), i.e. a new level in the overall task hierarchy, as shown in Figure 19.

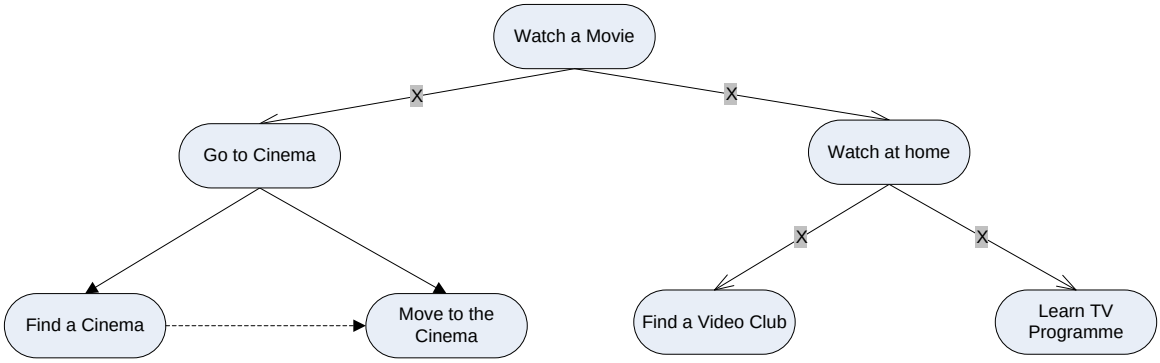

Figure 19: The previous example without GROUP constructs

Before continuing with the XML syntax of ToDo, we emphasize that although each ellipse in the previous figures corresponds to a task template as described in Figure 7, **the underlying task parameters are not visualized in 2D because the resulting DAG would be quite difficult to handle**.

4.4 The XML syntax of ToDo

This section introduces the tags used for denoting the entities of a ToDo Task Ontology in XML format. **By the time a ToDo ontology is downloaded and stored in the mobile device, it is only regarded as a template from which the task-oriented UI is generated.** Thus, for simplifying this process, we propose the following simple XML syntax.

We point out that, in the context of TALOS, all task parameters are defined by the authors through a visual interface (a simple form) provided in the Task Authoring Tool (see Section 6.2).

4.4.1 The `<todo:model>` tag

This tags enclosures the whole body of the ToDo task ontology.

4.4.2 Tags denoting the Task Attributes

The tag used for defining a task is the <todo:task>. Each task has an auto-generated ID and a user-provided Name which are unique for the ontology a task belongs to. For example, the task *"Find a Bar"* is defined as:

```
<todo:task ID="1" name="Find a Bar">
  ...
</todo:task>
```

The non-functional properties (attributes) of a task are denoted by the following tags which are always under a <todo:task> node:

- <todo:description>
  This tag denotes the description of a task in the form of unstructured text. For instance, the description of the task *"Book a Hotel Room"* could be something like the following:

```
<todo:task ID="2" name="Book a Hotel Room">
  ...
  <todo:description>
      This task is for helping users who want to
      find and book a hotel room
  </todo:description>
  ...
</todo:task>
```

- <todo:author>
  This tag denotes the name of the task author. If a task has more than one authors, then their names are given within different <todo:author> tags. For example:

```
<todo:task ID="1" name="Something">
  ...
  <todo:author>John Liagouris</todo:author>
```

```
   ...
</todo:task>
```

- <todo:publisher>
  This tag denotes the name of the task publisher. For example:

```
<todo:task ID=”1” name=”Something”>
   ...
   <todo:publisher>IMIS</todo:publisher>
   ...
</todo:task>
```

- <todo:createdOn>
  This tag denotes the date a task was created. The date is given in YYYY-MM-DD format. For example:

```
<todo:task ID=”1” name=”Something”>
   ...
   <todo:createdOn>2009-11-13</todo:createdOn>
   ...
</todo:task>
```

- <todo:version>
  This tag denotes the version of a task. From the application perspective, the version number is automatically generated by the time a task is created or updated. For example:

```
<todo:task ID=”1” name=”Something”>
   ...
   <todo:version>1.0</todo:version>
   ...
</todo:task>
```

- <todo:lang>
  This tag denotes the language in which a task and all of its parameters are given. Languages are given in abbreviations. The following example stands for "English":

```
<todo:task ID="1" name="Something">
   ...
   <todo:lang>EN</todo:lang>
   ...
</todo:task>
```

4.4.2.1 The <todo:realizedBy> tag

This tag is optional and it is used to denote the URL of the Web Service that realizes the task. It is always under a <todo:task> node. A task may be realized by none or more than one Web Services. In the latter case, the distinct URLs are given within different <todo:realizedBy> tags. For example, the task *"Book a Flight"* can be accomplished within two different Web Services that are denoted as follows:

```
<todo:task ID="34" name="Book a Flight">
   ...
   <todo:realizedBy>
        http://www.airtickets.com
   </todo:realizedBy>
   <todo:realizedBy>
        http://www.travelplanet24.com
   </todo:realizedBy>
   ...
</todo:task>
```

4.4.3 Tags denoting the Task Parameters

4.4.3.1 The <todo:preInformation> tag

This tag enclosures the tags <todo:input>, <todo:preCondition> and <todo:preference> that denote functional properties of a task.

4.4.3.2 The <todo:input> tag

This tag denotes the inputs of a task which are given within <todo:param> tags. Inputs may be optional. The latter is denoted by the optional attribute in the <todo:param> tag. Each input parameter is accompanied by its name (<todo:pName>), XSD type (<todo:type>), an optional description (<todo:pDescription>), and an optional <todo:instantiatedBy> statement. For example:

```
<todo:task ID="1" name="Something">
   ...
   <todo:preInformation>
      <todo:input>
         <todo:param ID="1" optional="false">
            <todo:pName>Date/Time</todo:pName>
            <todo:type>dateTime</todo:type>
            <todo:pDescription>
               The date and time (CET) of the visit
            </todo:pDescription>
            <todo:instantiatedBy>
               <todo:module>CA</todo:module>
               <todo:var>DateTime</todo:var>
            </todo:instantiatedBy>
            <todo:instantiatedBy>
```

```
                    <todo:module>User</todo:module>
                    <todo:var>UsrDateTime</todo:var>
                </todo:instantiatedBy>
            </todo:param>
        </todo:input>
        ...
    </todo:preInformation>
    ...
</todo:task>
```

4.4.3.3 The `<todo:instantiatedBy>` tag

The `<todo:instantiatedBy>` tag denotes the module from which the corresponding input parameter will be instantiated during a normal operation of the mobile application. As we explained in the previous sections, we adopt a MVC-like architecture where the ToDo model (expressed in XML) serves as the basis from which (a) the hierarchical structure and (b) the basic functionality of the mobile UI are generated. In order to achieve the latter, we distinguish three different TALOS-specific instantiating modules:

- **Context Aggregator (CA)**
  The CA [29] manages the user context (current location, date/time, traveller type), the weather information, and some other application-specific attributes. In our approach, the author can specify that an input parameter is instantiated by the corresponding method of the CA. We distinguish four such methods[5] as described in the following. Note that the datatypes of the variables (given within `<todo:var>` tags) which store the outputs of these methods are defined in the CA.xsd file provided in the Appendix of this document.

[5] These methods correspond to those supported in the current version of the Context Aggregator module.

- *getUserLocation()*: This method returns the user current location in the form of {Longitude, Latitude}. An input parameter that is instantiated by this method is defined along with the following `<todo:instantiatedBy>` statement:

```
<todo:instantiatedBy>
  <todo:module>CA</todo:module>
  <todo:var>Location</todo:var>
</todo:instantiatedBy>
```

- *getWeather(place,dateTime)*: This method returns the weather information in the form of {Temperature, Weather Type}. The default value of the *place* and *dateTime* arguments are the location of the user and the current date and time respectively. However, these parameters can also be given manually by the user through the UI. An input parameter that is instantiated by this method is defined along with the following `<todo:instantiatedBy>` statement:

```
<todo:instantiatedBy>
  <todo:module>CA</todo:module>
  <todo:var>Weather</todo:var>
</todo:instantiatedBy>
```

- *getTravellerType()*: This method returns the type of the traveller. This is (optionally) given by the user when planning a trip. In the context of TALOS we have defined [30] 5 different types of travellers: {Backpack Traveller, Business Traveller, Traveller with Family, Disabled Traveller, ALL}. An input parameter that is instantiated by this method is defined along with the following `<todo:instantiatedBy>` statement:

```
<todo:instantiatedBy>
  <todo:module>CA</todo:module>
  <todo:var>TravellerType</todo:var>
```

```
</todo:instantiatedBy>
```

- *getDateTime()*: This method returns the current date and time. An input parameter that is instantiated by this method is defined along with the following `<todo:instantiatedBy>` statement:

```
<todo:instantiatedBy>
  <todo:module>CA</todo:module>
  <todo:var>DateTime</todo:var>
</todo:instantiatedBy>
```

- **Application (App)**
  An input parameter can be instantiated with data retrieved from the local database. These data are retrieved through an SQL SELECT query which is defined by the author. **Note that such an SQL query can include a user-provided values (denoted by the prefix 'Usr') from a preference parameter, as long as a task input parameter (denoted with the symbol '$')**. For example, an SQL statement that retrieves all available restaurants in a specified city is defined as follows:

```
<todo:instantiatedBy>
  <todo:module>App</todo:module>
  <todo:var>
     Select POI.ID from POI where POI.type = "Restaurant"
     and POI.City_ID in (Select City_ID from City where
     Name=$City)
  </todo:var>
</todo:instantiatedBy>
```

- **End-User (Usr)**
  An input parameter can be instantiated manually by the end-user through an input field in the UI. This is denoted as follows:

```
<todo:instantiatedBy>
```

```
    <todo:module>Usr</todo:module>
    <todo:var>ParameterName</todo:var>
  </todo:instantiatedBy>
```

Note that the `ParameterName` defined within the `<todo:var>` tags is the name of the variable (arbitrarily given by the author) that will be created by the controller in order to store the user-provided value.

4.4.3.4 The `<todo:pGroup>` tag

In case a task takes as inputs a number of parameters some of which must be instantiated all together, then the author can define groups of parameters using the `<todo:pGroup>` tag. **The distinct groups under a** `<todo:input>` **node define an exclusive instantiation, whereas the parameters within the same** `<todo:pGroup>` **tag must be instantiated all together**. For example, let a task *"Find a Restaurant"* that takes as input the location of the user and a list of POIs (restaurants). In case the user location is given either in the form of a city name or in the form of longitude and latitude, and only one of the previous parameter formats along with the list of restaurants are necessary and sufficient for accomplishing the task, then this is denoted in ToDo as follows:

```
<todo:task>
  <todo:ID>12</todo:ID>
  <todo:name>Find a Restaurant</todo:name>
  ...
  <todo:preInformation>
    <todo:input>
      <todo:param ID="1" optional="false">
        <todo:pName>City</todo:pName>
        <todo:type>city</todo:type>
         <todo:pDescription>
```

```
        The location of the restaurant in the form of
        a city name, e.g. “Berlin”.
    </todo:pDescription>
    ...
</todo:param>
<todo:param ID=”2” optional=”false”>
  <todo:pName>Coordinates</todo:pName>
    <todo:type>coords</todo:type>
    <todo:pDescription>
        The longitude and latitude of the user
        location that will be used for finding
        restaurants nearby.
    </todo:pDescription>
    <todo:instantiatedBy>
        <todo:module>CA</todo:module>
        ...
    </todo:instantiatedBy>
</todo:param>
<todo:param ID=”3” optional=”false”>
  <todo:pName>Restaurants</todo:pName>
    <todo:type>POI</todo:type>
    <todo:pDescription>
        A list of restaurants
    </todo:pDescription>
    <todo:instantiatedBy>
        <todo:module>App</todo:module>
        ...
    </todo:instantiatedBy>
</todo:param>
<todo:pGroup>
```

```
            <todo:pMember>1</todo:pMember>
            <todo:pMember>3</todo:pMember>
        </todo:pGroup>
        <todo:pGroup>
            <todo:pMember>2</todo:pMember>
            <todo:pMember>3</todo:pMember>
        </todo:pGroup>
    </todo:input>
    ...
    </todo:preInformation>
    ...
</todo:task>
```

4.4.3.5 The `<todo:preCondition>` tag

This tag is used for defining a precondition, i.e. a condition that must hold in order for the task to be accomplished successfully. Preconditions are divided into: (a) logical expressions over the task inputs, denoted by the argument `type="expr"`, and (b) simple notices to the end-users, denoted by the argument `type="msg"`. **Note that any variable included in a precondition of the first type (logical expression) must always be an input of the corresponding task**. For instance, the task "Visit an open Market" may require good weather conditions. This precondition is specified in ToDo as follows:

```
<todo:task ID="11" name="Visit an open Market">
    ...
    <todo:preInformation>
        <todo:input>
            <todo:param ID="1" optional="true">
                <todo:pName>Weather</todo:pName>
                <todo:type>weather</todo:type>
```

```
                    <todo:pDescription>
                      The type of the weather for a specified
                      place, date and time.
                    </todo:pDescription>
                    <todo:instantiatedBy>
                          <todo:module>CA</todo:module>
                          <todo:var>Weather</todo:var>
                          </todo:instantiatedBy>
               </todo:param>
                ...
          </todo:input>
        <todo:preCondition type="expr">
            CA.Weather='Sunny'
        </todo:preCondition>
        ...
</todo:task>
```

Consider another example. The task "Book a Flight" obviously requires a credit card. Hence, an author can include this notice as a precondition when defining the task just like in the following example:

```
<todo:task ID="17" name="Book a Flight">
    ...
    <todo:realizedBy>
    http://www.air-tickets.com
    </todo:realizedBy>
    ...
    <todo:preInformation>
      <todo:preCondition type="msg">
           You need a credit card in order to proceed
      </todo:preCondition>
```

```
        ...
    </todo:task>
```

4.4.3.6 The <todo:preference> tag

This tag is used for denoting a user preference, i.e. a parameter that is optionally taken into account when guiding a user to accomplish a task. Preferences break into logical expressions over (a) the task inputs and (b) the POI-related attributes as defined in the user's database, i.e. the database of the mobile device [30]. For example, in case the author wants to give the users the ability to search for an affordable restaurant, then this is specified in ToDo as follows:

```
<todo:task ID="1" Name="Find a Restaurant">
    ...
       <todo:preInformation>
          <todo:input>
           ...
          </todo:input>
          <todo:preference>
                POI.Price <= Usr.Price
          </todo:preference>
          ...
</todo:task>
```

Note that the the name of the variable (arbitrarily given by the author) that will be created by the controller in order to store the user-provided value is denoted with the prefix `Usr`.

4.4.3.7 The <todo:postInformation> tag

This tag enclosures the tags `<todo:output>`, `<todo:postCondition>` and `<todo:effect>` that denote functional properties of a task.

4.4.3.8 The <todo:output> tag

This tag denotes the outputs of a task which are given within <todo:param> tags. As in the case of inputs, outputs may be optionally produced according to the current context and content. Each output parameter is accompanied by its name (<todo:pName>), XSD type (<todo:type>), and an optional description (<todo:pDescription>). For example:

```
<todo:output>
   <todo:param ID="1" optional="false">
      <todo:pName>POIs</todo:pName>
      <todo:type>POI</todo:type>
      <todo:pDescription>
         A list of available points of interest
         matching the input criteria
      </todo:pDescription>
   </todo:param>
</todo:output>
```

4.4.3.9 The <todo:postCondition> tag

This tag denotes a postcondition that must hold after the task is accomplished. Postconditions are logical expressions over (a) context-related parameters and (b) task-specific output parameters. For instance, the user's location when arrived at a park must be equal to the location of the park. **Note that the location of the park in the following example must have been defined as output of the respective task.**

```
<todo:task ID="1" Name="Move to the park">
            ...
         <todo:postInformation>
            <todo:output>
```

```
            ...
        </todo:output>
        <todo:postCondition>
            CA.Position = $Park.Location
        </todo:postCondition>
        ...
    </todo:task>
```

4.4.3.10 The <todo:effect> tag

This tag denotes the effect of a task. Effects are simple notices to the end-users specified a priori by the authors. For example:

```
<todo:effect>
   Confirmation receipt is delivered by email
</todo:effect>
```

4.4.4 The <todo:extends> tag

This tag is optional and it is used only in case the author wants to import and use in a new Task Ontology a task that is already defined in an existing Task Ontology. **A task may extend at most one task of another Task Ontology**. For example, if the task *"Move from A to B"* is defined in the *"Transportation"* ontology and the author wants to import it in a new ontology named *"Culture"* in order to reuse it, then this is done by adding the <todo:extends> tag as follows:

```
<todo:task ID="18" name="Move to the Museum">
  <todo:version>1.0</todo:version>
  ...
  <todo:extends>
    <todo:impName>Move from A to B</todo:impName>
    <todo:impVersion>1.0</todo:impVersion>
```

```
      <todo:model>Transportation</todo:model>
    </todo:extends>
    ...
  </todo:task>
```

Note that the task *"Move to the Museum"* inherits all parameters specified in the definition of the task *"Move from A to B"*. The imported task is defined by its name (`<todo:impName>`) and version (`<todo:impVersion>`), and also by the name of the ontology it belongs to. The latter is denoted by the `<todo:model>` tag. Although the author is not able to modify the inherited parameters, however, he/she is able to extend the imported task if needed by simply adding new parameters under `<todo:preInformation>` and/or `<todo:postInformation>` tags.

We point out that, from the application perspective, when a user clicks on an imported task, the mobile application (a) instantiates the task with the input parameters (if any) and (b) guides the user according to the task hierarchy that is defined in the ontology the imported task belongs to. In other words, **an imported task is accomplished by its subtasks (if any) as defined in the original ontology, i.e. the one from which it is imported**. Therefore, the only additional tasks imported along with the imported one are its children. On the other, the only relations imported along with the imported task are those between the latter and its children (e.g. subsumption), and also the relations among these children (e.g. sequence). In case the imported task (resp. any of its subtasks) has a relation with another task (resp. with a task that is not defined to be a child of the imported task) in the original ontology, then this relation is omitted. A concrete example of this kind is provided in Section 4.6.

4.4.5 Task Relations

The tags used for denoting relations between tasks are depicted in Table 2. In a ToDo Task Ontology we have four types of relations between tasks (SubTaskOf, Sequence, OR, and CHOICE). As mentioned in the previous section, the *GROUP* construct (described in Section 4.5.2.5) is just syntactic sugar. It does not introduce a new kind of relationship between tasks.

4.4.5.1 The <todo:subTaskOf> tag

This tag defines a subsumption relationship between two or more tasks. The ID of the parent-task is included in the <todo:subTaskOf> tag, while the ID(s) of the children-tasks are given within <todo:member> tags under the <todo:subTaskOf> node in any order. For example, the axiom "*Task 1 and Task 2 are subtask of Task 3"* is expressed as:

```
<todo:subTaskOf ID="3">
    <todo:member>1</todo:member>
    <todo:member>2</todo:member>
</todo:subTaskOf>
```

or

```
<todo:subTaskOf ID="3">
    <todo:member>2</todo:member>
    <todo:member>1</todo:member>
</todo:subTaskOf>
```

4.4.5.2 The <todo:sequence> tag

This tag defines a sequence between two tasks. It is always under a <todo:chain> tag that is explained in the following. The two task IDs of a sequence are given within <todo:member> tags in a specific order from the first task to the second one. The passed parameters are given with <todo:sourceParam> and <todo:targetParam> tags under the <todo:dataflow> node. For example, if "*Task 1 is prior to Task 2"* and the output *"Cinema"* of the first task is passed as input *"POI"* into the second task, then this relation is expressed as:

```
<todo:sequence>
    <todo:member>1</todo:member>
    <todo:member>2</todo:member>
    <todo:dataflow>
```

```
            <todo:sourceParam>Cinema</todo:sourceParam>
            <todo:targetParam>POI</todo:targetParam>
        </todo:dataflow>
    </todo:sequence>
```

Note that *"Cinema"* and *"POI"* are parameter names corresponding to tasks 1 and 2 respectively. These parameters have already been specified by the author(s); possibly along with other input and output parameters of the two tasks which are not included in a dataflow. We emphasize that the relation between *Source* and *Target* parameters within a specific `<todo:dataflow>` tag is bijective (1:1) and that a sequence between two tasks may introduce more than one dataflows which in this case are given within different `<todo:dataflow>` tags.

One (or more) sequences between two (or more) tasks belong to a so-called *sequence chain*. More specifically, taking into account that tasks in ToDo are reusable, each sequence between two tasks belongs to at least one chain and, therefore, different chains may have "intersections", i.e. they may include common tasks. Each sequence chain has a unique ID that is denoted in the `<todo:chain>` tag. For instance, according to the previous example, in case there is also a sequence between the task 2 and another task, let 3, and both these sequences belong to the same chain, then this is specified in ToDo as follows:

```
<todo:chain ID="13">
    <todo:sequence>
        <todo:member>1</todo:member>
        <todo:member>2</todo:member>
        <todo:dataflow>
            <todo:sourceParam>Cinema</todo:sourceParam>
            <todo:targetParam>POI</todo:targetParam>
        </todo:dataflow>
    </todo:sequence>
    <todo:sequence>
        <todo:member>2</todo:member>
        <todo:member>3</todo:member>
```

```
        <todo:dataflow>
            <todo:sourceParam>Coords</todo:sourceParam>
            <todo:targetParam>XY</todo:targetParam>
        </todo:dataflow>
    </todo:sequence>
</todo:chain>
```

4.4.5.3 The `<todo:or>` tag

This tag defines an *OR* relationship between two or more tasks with respect to their (common) parent. The ID of the parent-task is always included in the `<todo:or>` tag. The IDs of the children-tasks are given within `<todo:member>` tags in any order. For example, the axiom *"Task 1 is accomplished by either accomplishing Task 2 or Task 3 or both"* is expressed as:

```
<todo:or ID="1">
  <todo:member>2</todo:member>
  <todo:member>3</todo:member>
</todo:or>
```

or

```
<todo:or ID="1">
   <todo:member>3</todo:member>
   <todo:member>2</todo:member>
</todo:or>
```

4.4.5.4 The `<todo:choice>` tag

This tag defines a *CHOICE* relationship between two or more tasks with respect to their (common) parent. The ID of the parent-task is always included in the `<todo:choice>` tag. The IDs of the children-tasks are given within `<todo:member>` tags in any order. For example, the axiom

"Task 1 is accomplished by accomplishing one of the Tasks 2 and 3" is expressed as:

```
<todo:choice ID="1">
  <todo:member>2</todo:member>
  <todo:member>3</todo:member>
</todo:choice>
```

or

```
<todo:choice ID="1">
  <todo:member>3</todo:member>
  <todo:member>2</todo:member>
</todo:choice>
```

4.4.5.5 The `<todo:group>` tag

This tag defines a *GROUP* construct including two or more tasks. As already mentioned, this construct is just syntactic sugar and it is used for helping authors easily describe complex relationships between tasks (see Section 4.3). *GROUP* defines a task hierarchy by implicitly introducing an anonymous parent-task whose ID is always included in the `<todo:group>` tag. The IDs of the children-tasks are given within `<todo:member>` tags. The relationship among the grouped tasks is denoted in the `<todo:group>` tag and it can be one of the previously described relationships (SUB, OR, and CHOICE). For example, the axiom *"Task 1 is accomplished by at least Task 2 or one of the Tasks 3 and 4"* is expressed as:

```
<todo:group ID="0" type="choice">
  <todo:member>3</todo:member>
  <todo:member>4</todo:member>
</todo:group>
<todo:or ID="1">
  <todo:member>2</todo:member>
  <todo:member>0</todo:member>
</todo:or>
```

where ID=0 is the ID of a new anonymous task that is implicitly defined as the parent of Tasks 3 and 4.

Tag	Description
<todo:subTaskOf>	Defines a subtask relation between two or more tasks. The task whose ID is included in the <todo:subTaskOf> tags is the parent-task. The IDs of the children-tasks are given within <todo:member> tags under the <todo:subTaskOf> node in any order.
<todo:sequence>	Defines a time ordering between two tasks. Each sequence belongs to at least one chain denoted by the <todo:chain> tag. The IDs of the tasks in the sequence are given within the <todo:member> tags under the <todo:sequence> node in a specific order, i.e. from the first task to the second one. Each distinct parameter passing is denoted within <todo:sourceParam> and <todo:targetParam> tags under a <todo:dataflow> node.
<todo:or>	Defines an optional relation between two or more tasks with respect to their parent. The ID of the parent-task is always included in the <todo:or> tag. The IDs of the other (two or more) tasks are given within <todo:member> tags under the <todo:or> node in any order.
<todo:choice>	Defines an exclusive option between two or more tasks with respect to their parent. The ID of the parent-task is always included in the <todo:choice> tag. The IDs of the other (two or more) tasks are given within <todo:member> tags under the <todo:choice> node in any order.
<todo:group>	Defines a group of two or more tasks that is captured by an anonymous task. The ID of the anonymous parent-task is always included in the <todo:group> tag along with the type of the group (SUB, OR, CHOICE). The IDs of the (two or more) children-tasks are given within <todo:member> tags under the <todo:group> node.

Table 2: XML tags for denoting Task Relations in ToDo

4.5 The logical formalism underneath ToDo

We have already made clear that we do not want just a modeling language such as UML, but a graphical language with firm theoretical roots. Automated reasoning over task related knowledge requires the existence of firm semantics or, in other words, the existence of logic within the model. In this section we describe the logical formalism ToDo is based on, i.e. Description Logic[6].

4.5.1 A Description Logic Knowledge Base

A knowledge database (KB) in DL comprises of two components: the TBox and the ABox [31]. The TBox introduces the terminology, i.e. the vocabulary for representing the domain of knowledge, while the ABox consists of assertions about named constants in terms of this vocabulary.

As far as the vocabulary of knowledge is consented, there are three basic "ingredients": the concepts, the relations and the constants. Formally, in Description Logics, the initial concepts considered in modeling the domain, called atomic concepts, are unary predicates. Analogously, binary predicates called atomic roles represent the initial relations. The idea of constants is captured by the individuals. Intuitively, the notion of a concept is considered to semantically group a number of constants with common properties, i.e. it corresponds to a set of individuals. In order to represent the properties a set of individuals has, the notion of roles is introduced. A role corresponds to a set of pairs of individuals which are related with each other by this role. Apart from representing and storing both terminologies and assertions, the Description Logic systems also involve inference mechanisms that reason about the base elements.

Among the building blocks of description logic, atomic concepts are denoted by capital *A*, atomic roles by capital *R* and individuals by *a* and *b*. Complex descriptions can be built from them using concept constructors. The concepts defined using such constructors are denoted by capitals *C* and *D*.

All description languages in DL are defined as extensions of the basic description language *AL* and are distinguished by the constructors they provide. In this document, we present the *ALCQ* language whose features are adequate for understanding the following sections.

---

6 A similar idea of translating UML models into Description Logic in order to exploit the automated reasoning procedures of the latter is introduced in [32] and [33].

Concept descriptions in *ALCQ* are formed according to the following syntactic rule:

$$C, D \rightarrow A \mid \top \mid \bot \mid \neg C \mid C \sqcap D \mid \forall R.C \mid \exists R.C \mid \exists R_{\leq n}.C \mid \exists R_{\geq n}.C$$

where:

- $\top$ is the universal concept of the world described, i.e. the concept that contains all the individuals of the world.
- $\bot$ is the bottom concept of the world described, i.e. it contains no individuals.
- $\neg C$ is the description constructed by the *negation* of the (complex) concept *C*. Intuitively, concept $\neg C$ captures all individuals of the knowledge base that do not belong to *C*.
- $C \sqcup D$ is the concept defined using disjunction. It consists of the individuals that are grouped by concept *C* or *D*. For example, the concept *Male* $\sqcup$ *Female* contains all the individuals that are males or females. Intuitively the previous concept could group all humans.
- $C \sqcap D$ is the concept defined using intersection. It consists of the individuals that are grouped by concept *C* and also by *D*. For example, the concept *Human* $\sqcap$ *Female* contains all the individuals that are humans and also females. Intuitively the previous concept could group all the women individuals.
- $\forall R.C$ is the *value restriction* constructor that defines a new class containing the instances whose all participations in the role *R* are with instances grouped by class *C*. For example, the concept $\forall$ *hasChild*.*Male* contains those individuals that when they participate in pairs of individuals grouped by role *hasChild*, the second individual is always corresponding to concept *Male*. Intuitively the example denotes the individuals that have children which are all males, i.e. individuals that have only sons. If *C* is the top concept, we write $\forall R$ and not $\forall R.\top$.

- $\exists R.C$ is the *full existential quantification* constructor that defines a class whose instances have participation in a role *R* with at least one instance of class *C*. For example, the concept $\exists R.hasChild.Parent$ consists of those individuals that are participating in *hasChild* role with at least one individual of concept *Parent*, i.e. individuals that have at least one child which is also a parent (which means complex expression denoting grandparents). If *C* is the top concept, we write $\exists R$ and not $\exists R.\top$.
- $\exists R_{\leq n}.C$ is the *at most n qualified number restriction* constructor that defines a class containing the instances that participate at most n times in role *R* with instances of the class *C*. For example, the concept $\exists R_{\leq 5}.hasChild.Human$ consists of the individuals that have 5 or less children that are humans.
- $\exists R_{\geq 5}.C$ is the *at least n qualified number restriction* constructor that defines a class containing the instances that participate at least n times in role *R* with instances of the class *C*. For example, the concept $\exists R_{\geq 5}.hasChild.Human$ consists of the individuals that have 5 or more children that are humans.

All the above explanations are actually the definitions of the semantics of *ALCQ* constructors in terms of natural language. In order to proceed with the formal definitions of these semantics we must introduce the notion of interpretations.

An interpretation I consists of a structure $(\Delta^I, \cdot^I)$ where Δ^I is the domain of interpretation and $\cdot^I$ is the interpretation function. The interpretation function associates a simple concept *C* with a set $C^I \subseteq \Delta^I$ and an atomic relation *R* with a binary relation $R^I \subseteq \Delta^I \times \Delta^I$. The interpretation of the top concept is the whole domain, i.e. $\top^I = \Delta^I$. The interpretation function is extended to concept descriptions by the following inductive definition:

$\top^I = \Delta^I$

$\bot^I = \emptyset$

$(\neg C)^I = \Delta^I \setminus C^I$

$(C \sqcup D)^I = C^I \cup D^I$

$(C \sqcap D)^I = C^I \cap D^I$

$$(\forall R.C)^I = \{a \in \Delta^I \mid \forall b.\, (a, b) \in \mathrm{R}^I \rightarrow b \in C^I\}$$

$$(\exists R.C)^I = \{a \in \Delta^I \mid \exists b.\, (a, b) \in R^I\}$$

$$(\exists R_{\leq \mathrm{n}}.C)^I = \{a \in \Delta^I \mid \{ b \in C^I \mid (a, b) \in R^I \} \leq \mathrm{n}\}$$

$$(\exists R_{\geq \mathrm{n}}.C)^I = \{a \in \Delta^I \mid \{ b \in C^I \mid (a, b) \in R^I \} \geq \mathrm{n}\}$$

We point out that the above set-theoretic semantics of DL are known in literature as Tarski-style semantics. The problem of automated reasoning with respect to these semantics is addressed in Section 4.4.3.

Having this basic knowledge about what a DL knowledge base is, we can now proceed with expressing ToDo features in Description Logic.

4.5.2 Expressing a ToDo Task Ontology in Description Logic

In order to express ToDo in Description Logic, tasks are interpreted as classes grouping objects that share a common set of attributes (input, output etc.). Objects in our case represent the actual *activities* of the users. Analogously to the terminology of DL, an object of type *C* is called *individual* or *instance* of the task *C*.

In terms of functionality, task instances, i.e. activities, are created according to the user profile (context) and preferences (context and content) as specified from his/her interactions with the interface of the mobile device. To be more specific, an example of an instance of the task "Visit a Museum" is something like "Visit the Museum of Acropolis". In this sense, when a user decides the museum he/she wants to visit, activities like "Visit the Louvre" or "Visit the Guggenheim Museum in NY" are created and classified under the class "Visit a Museum".

In addition, task attributes are represented in DL by object and datatype properties. For instance, if the input for the task "Find a Museum" is a list of museums (i.e. individuals) from a DL-based domain ontology like the one in Figure 8, this attribute is represented by an object role having as domain a set of activities classified under the class "Find a Museum" and as range a set of individuals that represent museums. On the other hand, attributes such as *TaskName* are represented by datatype properties connecting the instance (activity) and its attribute (its name). Obviously, in this case, the range of the property is an XML datatype (string).

Temporal relations between tasks are denoted through relations between the corresponding instances. For this purpose we introduce the following (reserved) atomic roles:

- R_{bef} which denotes that an instance is prior in time of execution to another one. Considering the tasks "Choose Pizza" and "Order Pizza", we can write $R_{bef}(a,b)$ where a and b are the corresponding instances.
- R_{aft} which denotes that an instance follows another one in time of execution. Note that R_{aft} is the inverse of the role R_{bef} and thus, continuing with the previous example, we can write $R_{aft}(b,a)$.
- R_{syn} which denotes that two instances are concurrently executed. Every pair of instances drawn from the tasks "Boil Pasta" and "Make Sauce", as defined in Section 4.3, is connected through R_{syn}. R_{syn} is a symmetric role which means that it is inverse with itself.

All of the above roles are disjoint one another and also transitive. The former means that if two task instances are connected through one of these roles, then the same (ordered) pair cannot be connected through any of the remaining two. The latter means that if, for example, both $R_{syn}(a,b)$ and $R_{syn}(b,c)$ exist in a knowledge base, then we can also infer $R_{syn}(a,c)$.

Such roles that capture generic temporal knowledge are defined in OWL-Time ontology [34] and thus, when translating a ToDo task ontology in OWL, we can represent these temporal relations by importing the corresponding roles from OWL-Time.

The representation of *Or* and *Choice* relations in DL requires that we also introduce the following atomic role:

- R_{acc} which denotes that an instance of a task is indirectly accomplished by an instance of another task. In our previous example, an instance, let a, of the task "Get info about a Mall" is accomplished by an instance b of the task "Get info about Discounts" and thus we can write $R_{acc}(a,b)$.

We are now able to present how complex tasks of ToDo are represented formally in DL:

- Axiom C_1 *precedes* C_2 is transformed into: $\top \sqsubseteq (\forall R_{bef}.C_2) \sqcap (\forall R_{aft}.C_1)$
- Axiom C_1 *is syn with* C_2 is transformed into: $\top \sqsubseteq (\forall R_{syn}.C_2) \sqcap (\forall R_{syn}.C_1)$
- Axiom $C_1 \equiv C_2$ *OR* C_3 is transformed into: $C_1 \equiv \exists R_{acc} \sqcap \forall R_{acc} (C_2 \sqcup C_3)$
- Axiom $C_1 \equiv C_2$ *XOR* C_3 is transformed into: $C_1 \equiv \exists_{\leq 1} R_{acc} \sqcap \forall R_{acc} (C_2 \sqcup C_3)$

In the previous *OR* and *XOR* constructs, tasks C_2 and C_3 cannot have a temporal relationship. Thus, in each case, the following three axioms are also added in the knowledge base:

$(\forall R_{bef}.C_2) \sqcap (\forall R_{aft}.C_3) \sqsubseteq \bot$

$(\forall R_{bef}.C_3) \sqcap (\forall R_{aft}.C_2) \sqsubseteq \bot$

$(\forall R_{syn}.C_2) \sqcap (\forall R_{syn}.C_3) \sqsubseteq \bot$

We have already mentioned that each task attribute is represented in DL through an atomic property. In ToDo we distinguish two kinds of task properties:

- *Object Properties*
  Each object property of a task is represented by an object role. If the attribute defined by the property is classified in a Domain Ontology, then this is expressed in DL as follows:
  - $\top \sqsubseteq (\exists R_{prop}) \sqcap (\forall R_{prop}.D)$, where R_{prop} is the role representing the object property (e.g. HasInput) and D is a class of a Domain Ontology including the corresponding object. (e.g. Museum).
- *Datatype properties*
  Each datatype property of a task is represented by a datatype role. If the attribute defined by the property is of a specific datatype, then this is expressed in DL as follows:
  - $\top \sqsubseteq (\exists R_{prop}) \sqcap (\forall R_{prop}.AttributeType)$, where R_{prop} is the role representing a datatype property (e.g. HasTaskAuthor) and *AttributeType* is the XML data type of the property (e.g. string).

Obviously, in both of the above cases, if the property defines at most one attribute, then we pose the corresponding *at most one* restriction in the $\exists R_{prop}$ clause of the previous axioms.

As far as the subtask relation is concerned, we follow exactly the same notations as in DL. For instance, if a task C_1 subsumes another one, let C_2, then we write $C_2 \sqsubseteq C_1$. However, we point out that, in ToDo, breaking a task into multiple subtasks is regarded as a complete subsumption. For

example, if a task C_1 breaks into subtasks C_2 and C_3, then we assume that in order for C_1 to be accomplished both of its subtasks must be accomplished. Thus, we write $C_1 \sqsubseteq (\exists R_{acc}.C_2) \sqcap (\exists R_{acc}.C_3)$.

4.5.3 Reasoning with Knowledge Bases

In this section we provide the reader with two representative examples of how the automated reasoning capabilities of Description Logic can be used in the context of TALOS.

Example 1: Identify inconsistencies within a task model

The model in Figure 15 defines a task C which precedes task C_3 in time of execution. The same holds for task C_4 that is defined as prior to task C_2. However, tasks C_2 and C_4 are subtasks of C and C_3 respectively and thus they inherit the temporal relationship established between their "parents". According to what we described in the previous section, if *a* is an instance of C_2 and *b* an instance of C_4, then the following two axioms are added in the knowledge base:

1. $R_{bef}(a,b)$
2. $R_{aft}(a,b)$

Obviously, these axioms contradict one another as *a* cannot be prior to *b* and at the same time posterior. This is a representative example of how a syntactically correct task model may "hide" semantic inconsistencies. Such inconsistencies can be easily detected using a Description Logic reasoner like Pellet [15].

We point out that an inconsistent Task Model as the one in Figure 20 may result in undesirable functionality of the user's interface. Sequence between tasks is usually accompanied with a flow of data. In our case, the output of task C_4 may be used as input for the task C_2. However, as C is executed before C_3, when a user tries to perform task C_2, which is one of the subtasks of C, the application may enter a non-stop loop where C_2 waits for data from C_4 and reverse.

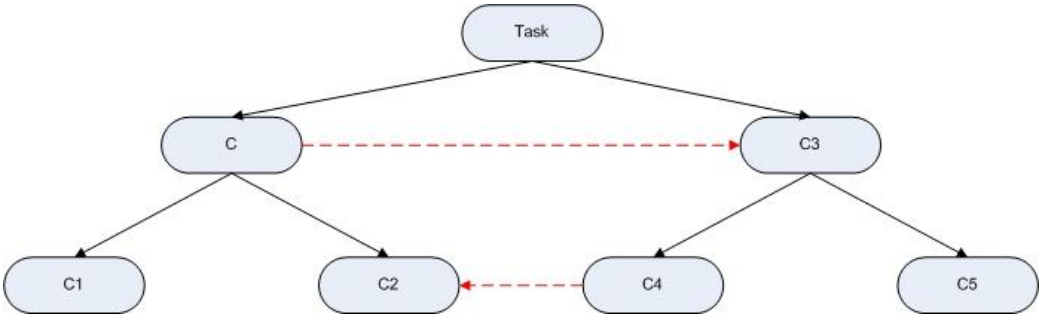

Figure 20: Inconsistent Model

Example 2: Recommendation of services and resources

Assume that we want to build a high-quality recommendation service that takes into account various aspects of the available information; from user's location, profile and preferences to weather reports and popularity of the supported services. Instead of developing algorithms for evaluating complex conditions, we are able to describe both context and content in Description Logic and take full advantage of a so-called declarative approach.

In the proposed framework, the recommendation service is regarded as an "intelligent" system that comprises of two basic modules:

1. A DL knowledge base that stores information about context and content in the form of axioms.
2. A reasoner able to answer queries over this set of axioms. Answers to these queries are the actual recommendations, i.e. a number of resources fulfilling the constraints defined in the query.

The key feature of such a DL-based system is that all reasoning tasks are performed through the *same algorithm*, named *Tableau* procedure. In addition, as highly-optimized Tableau-based implementations already exist, the only thing needed to provide such an infrastructure is to describe the existing data in Description Logic, i.e. in OWL. For a similar approach followed by NTT DoCoMo, see [17].

In this section we do not go further with describing the Tableau procedure. The following example is given just to show how a recommendation service can be reduced in answering complex (conjunctive) queries over a Description Logic knowledge base.

Assume that someone is on a business trip in Barcelona and looks for a traditional restaurant to have dinner with his/her colleagues. In case there is an appropriate DL-based domain ontology featuring the "Places to eat", when he/she asks for recommendation, the following (simplified) SPARQL [35] query is posed to the system:

```
SELECT ?name
WHERE
  { ?x hasName ?name .
    ?x typeOf Restaurant .
    ?x locatedIn Barcelona .
    ?x suitableFor y .
    ?y typeOf BusinessMeeting .
    ?x serves z .
    ?z typeOf TraditionalFood }
```

The result-set of the above query will contain the names of all available traditional restaurants (captured by the variable x) located in Barcelona and rated as suitable for business meetings.

We point out that the previous query does not imply any reasoning procedures. In fact, reasoning services are performed on TALOS server and thus they are only provided in the existence of internet access. However, the interesting thing here is that all necessary reasoning tasks can be done on TALOS server (every time the ontology is updated) and the resulting serialized ontology can then be stored (in RDF format) for offline querying in the mobile device. As for the latter, the sceptic reader can refer to [36] and [37] that present and evaluate an interesting approach for storing and retrieving millions of RDF triples using the iPhone.

4.6. Example

Figure 19 provides an example of a simple Task Ontology built with ToDo graphical notations. The presented ontology is for guiding users who want to find accommodation and/or a place to entertain themselves. The XML representation of the model is given in APPENDIX II. We have already pointed out that the underlying task parameters are not visualized in 2D. Instead, they are managed throught the Task Properties panel of TOAT (see Section 6.4). The groups in this example are used in order to avoid drawing many OR and Sequence relations between tasks.

From the application perspective, each task of the following model is instantiated by context and content (i.e. becomes an activity) while the user interacts with the interface of the mobile device. For example, when a user clicks to find a hotel, the list provided by default will only include hotels close to him/her. In such a case, the location of the user is captured by the Context Aggregator Module (see [29]).

The following model also defines the functionality of the UI. As shown in the XML file in the APPENDIX II, the task "Find out Prices" takes as input the output of the task "Find a Restaurant". When the user clicks on the task "Find out Prices" without having specified a restaurant first, the interface can redirect him/her to the task "Find a Restaurant" that is defined as prior to the selected one. In terms of functionality, this implies that we want a fully operational interface. Following another approach, we could enable tasks only in case they do not depend on others that are not already instantiated.

Figure 21: An example of a ToDo Task Ontology

5 Task Ontology Lifecycle

This section provides a clear description of the Task Ontology Lifecycle, i.e. **the distinguished states to which a Task Ontology comes while being used by the participants in the TALOS system**. We point out that our intention is not to analyze thoroughly all the potential activities of each participant and thus we only address those that are adequate to define, alone or in relation with others, a state in the ontology lifecycle.

Before continuing with the detailed description of the diagram provided in Figure 22, we first clarify some terms that are essential for understanding the following. These terms are:

- ***Task***
  As already mentioned, a task reflects what an end-user wants to do in a high-level layer of abstraction, e.g. "Eat at a restaurant". Each task is accompanied by a set of attributes (input, output, precondition etc.) and is instantiated by content (and context) in order to become an *activity*. For example, an activity related to the previous task is something like "Eat at TGI Fridays in Athens".
- ***Task Ontology***
  A Task Ontology amounts to a formal specification of a user's tasks. In other words, it is a model of a user's tasks with firm syntax and semantics. For more information about the notion of Task Ontology in TALOS please refer to Section 2.2.13.
- ***Task Ontology Database (TODB)***
  Each Task Ontology can be expressed in XML-like syntax and stored in a relational database. The TODB is the central database in TALOS Server that keeps **all versions** of every Task Ontology.
- ***Task Ontology Author (TOA)***
  A TOA is the person who designs the Task Ontology using the graphical tools provided in a TALOS-specific editor called *Task Ontology Authoring Tool (TOAT)*.
- ***Content Manager (CM)***
  A CM is the person who manages the content and decides which resources (text or POIs) are appropriate for specific tasks. Besides the content existing in a travel guide, other resources can be **dynamically derived from the web (through scrapping) and assigned to the tasks of a Task Ontology**

using an editor called *Annotation Tool (AT)*. We point out that the corresponding web content is also stored in CB.

- ***Content Base (CB)***
  Similarly to the case of Task Ontologies, content can also be structured with XML and stored in a relational database. The CB is the central database in TALOS Server that stores (a) **unstructured content** in the form of text, maps, images etc., (b) **geo-referenced (structured) content as Points of Interest (POIs)**, and (c) **context-related information** that is used for filtering the corresponding resources in CB according to a set of context attributes).
- ***Expert***
  An expert is either a TOA or a CM.
- ***End-User***
  An end-user is the user of the mobile handset which, in our case, is either the iPhone or an e-book reader.
- ***Idle Task Ontology (ITO)***
  An ITO is a Task Ontology uploaded on TALOS Server with no content assigned to its tasks. The ITO cannot be used by the end-users until a CM assigns content to its tasks.
- ***Operational Task Ontology (OTO)***
  When a CM assigns content to the tasks of a Task Ontology and uploads it on TALOS server, the corresponding Task Ontology is called operational. From the view point of the end-user, **an OTO is what he/she follows in order to reach the appropriate content and services**.

The UML Activity Diagram [23] provided in Figure 22 depicts the major activities of the participants in TALOS system. Thus, the diagram is divided into four parts, swimlanes in UML terms: one for the Author of the Task Ontology, one for TALOS Server, one for the Content Manager, and one for the End User.

The presented diagram illustrates only a normal use case scenario and does not include any of the potential problems (e.g. user authentication failure, database crash etc.). In addition, we emphasize that the specified activities are based on the assumption that we deal with **a real-world environment where there are many Task Ontology Authors, Content Managers, and End Users**. The role of each distinct participant is described in the following:

- Author of the Task Ontology

The TOA is responsible for performing the following tasks:

- Design a new Task Ontology
- Edit an existent Task Ontology (ITO or OTO)
- Upload a Task Ontology on TALOS Server

As shown in Figure 22, editing an existing Task Ontology requires a check for new versions in TALOS Server. This is necessary due to the existence of many TOAs. We have already mentioned that our analysis is focused on the development of a collaborative environment where many TOAs can work on the same or different Task Ontologies. Following this general idea, we can assume that when an author uploads a Task Ontology on TALOS Server, there may also be other authors who want to download the ontology, review it or make changes on it. Thus, in order to**synchronize TOAs' work**, we have to implement a check-in/check-out control mechanism before every update in the local version of a Task Ontology. Note that in case **a TOA downloads an OTO and changes it, by the time he/she uploads it back on TALOS server, the Task Ontology is then considered as ITO and not OTO**. This means that it cannot be used by end-users until a CM assigns content to its tasks. Although it may seems strange, this approach is imposed by the difficulty in handling the evolution of Task Ontologies. We come back to this issue in the following.

- TALOS Server

TALOS Server keeps all versions of every Task Ontology in Task Ontology Database (TODB). We argue that all versions for Task Ontologies must be stored in TALOS Server due to the following reasons:

- ***Compatibility with the OTOs used by the end-users***
  When downloading an OTO, the user of the mobile device may decide to include only a part of the assigned content and not all of it. In such a case, the rest of the content can be downloaded gradually during the use of the Task Ontology. Thus, a downloaded OTO cannot be overwritten on TALOS Server until all end-users have updated their local copies.
- ***Ability to rollback to older versions***
  A version history of the experts work can be very useful when, for some reason, one needs to refer to an older version of a project.

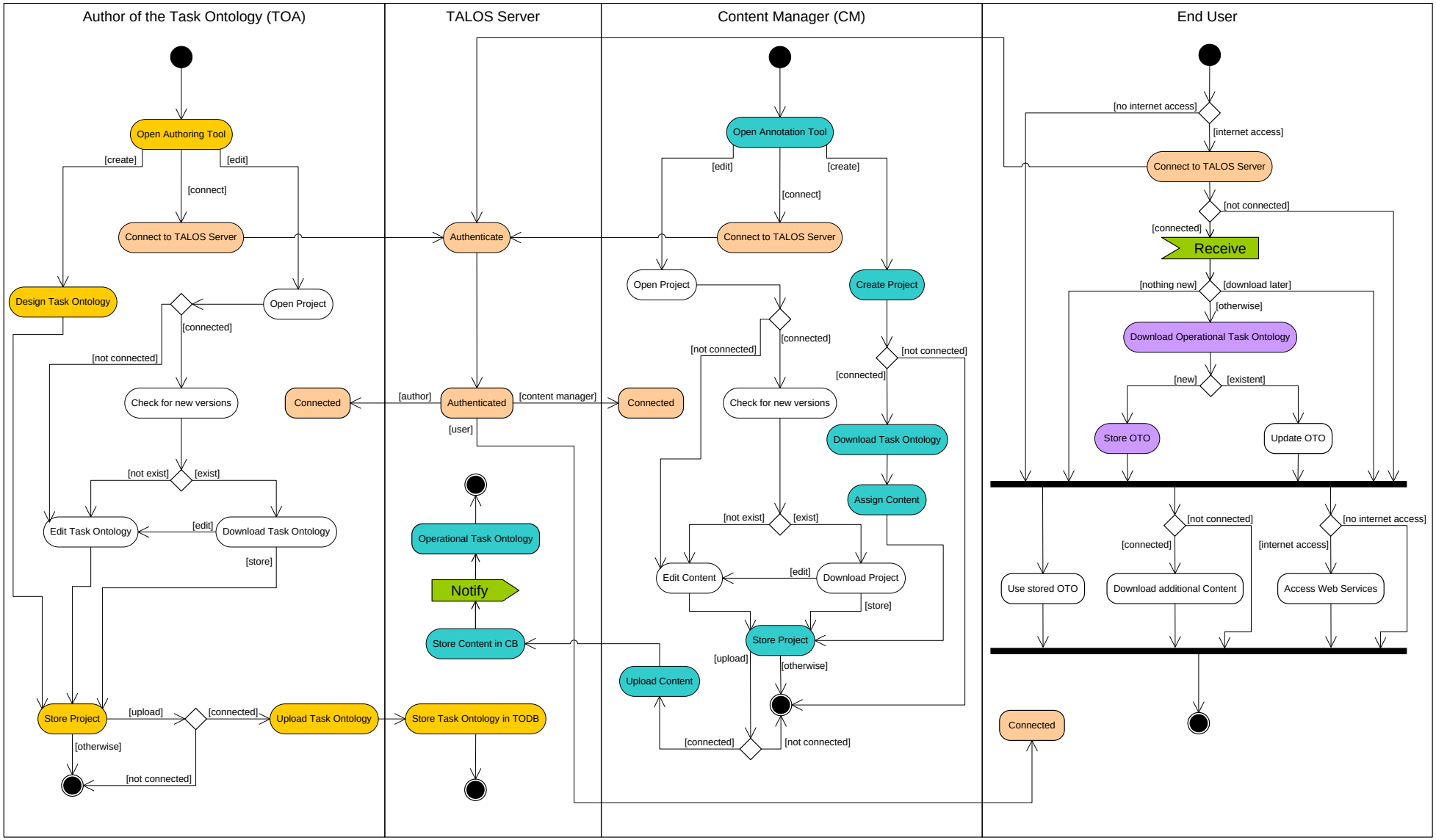

Figure 22: TALOS Activity Diagram

- Content Manager

The CM is responsible for performing the following tasks:

- Organize content
- Assign content to tasks of an existent Task Ontology
- Upload content on TALOS server

Organizing content is a CM's task that does not affect the state of a Task Ontology and thus it is not included in Figure 22.

As far as the editing of an existing CM's project is concerned, the procedure is similar to the one described in the previous about editing Task Ontologies locally. In this case and due to the fact that **CMs always need to download the Task Ontology first in order to assign content to its tasks**, checks must be done not only for new versions of content, but also for new versions of Task Ontologies.

- End User

The user of the iPhone can perform the following actions:

- Download an OTO
- Download additional content
- Access Web Services

The user is automatically notified for the existence of a new version of an OTO by the time the latter is published on TALOS Server.

We point out that **the end-users may not download all content assigned to an OTO from the start, but decide to download only a part of it and retrieve the remaining content gradually during the use of the application**.

Colors in Figure 22 are used to denote the distinct "paths" that are followed when bringing a new Task Ontology in TALOS system. The sequence of the required tasks is the following:

1. A TOA creates a new Task Ontology and uploads it on TALOS Server
2. A CM downloads the corresponding Task Ontology, assigns content to it and uploads it on TALOS Server
3. The end-user is notified that an OTO exists on TALOS Server
4. The end-user downloads the OTO

Obviously, in all cases, precondition for connecting to TALOS server is the successful authentication by the system.

As shown in Figure 23, in a normal use case scenario and after the first two of the above steps are performed, a Task Ontology is considered to be operational and amenable only to updates (white "paths" of TOAs and CMs in Figure 22).

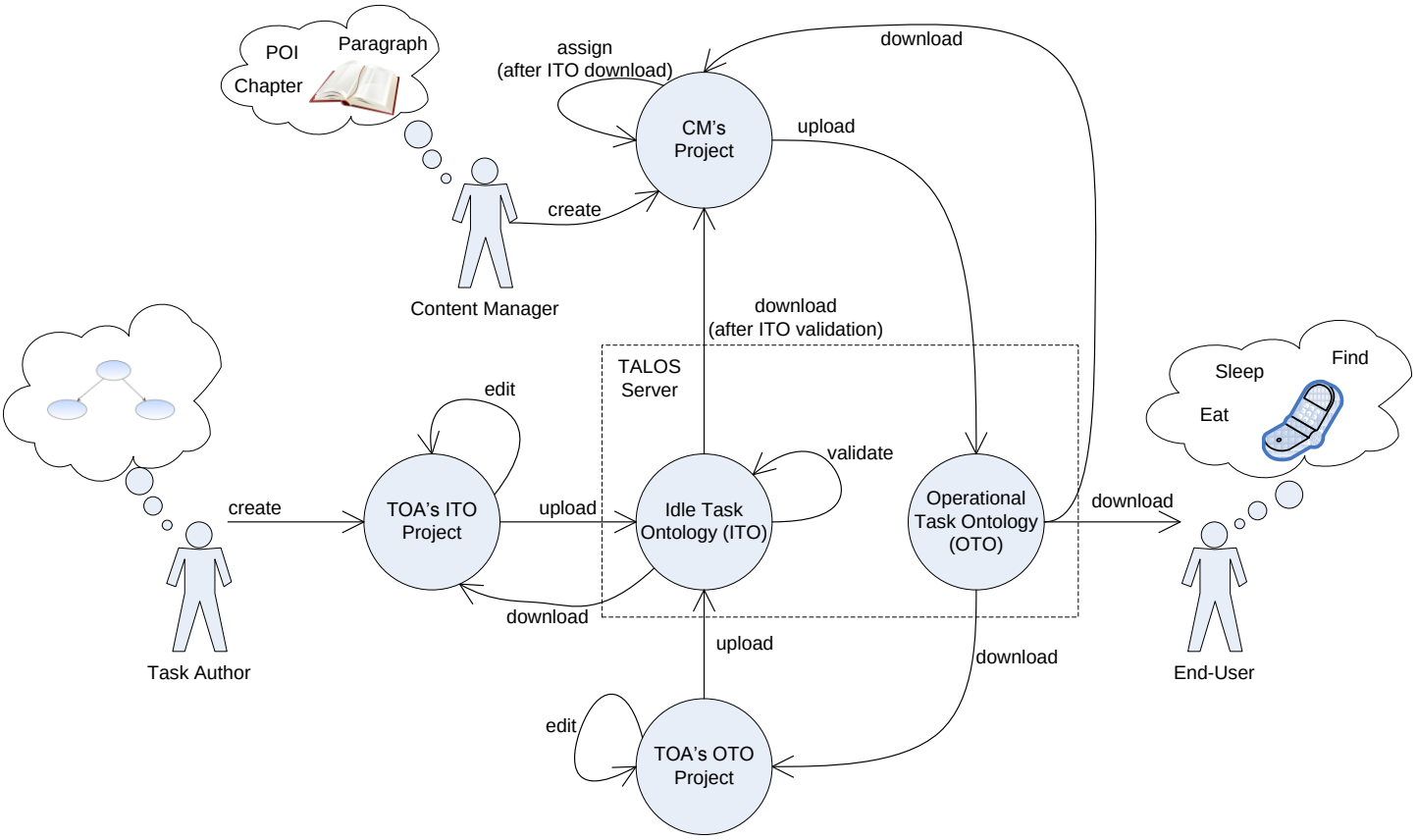

Figure 23: Task Ontology Lifecycle in TALOS

6 Task Ontology Authoring Tool

6.1 Introduction

This section describes the Task Ontology Authoring Tool. The Task Ontology Authoring Tool (TOAT) is a specially designed software tool that support authors in the design and manipulation of task models in an intuitive manner.

The remaining section is organized as follows. In Section 6.2 we present the user requirements for TOAT. Section 6.3 discusses some issues regarding the design decisions we made for TOAT development. Finally, Section 6.4 provides an overview of TOAT functionality.

6.2 User requirements

The overall purpose of the Talos project is to design, develop and evaluate a complete framework that will enable the task-aware provision of content to mobile travellers. Prototype mobile travel guide applications, incorporating this framework, will be developed. These applications will thus make use of task-oriented functionalities and a task-based user interface.

The Task Ontology Authoring Tool (TOAT) is developed as a means to create the task hierarchies/ontologies that will afterwards be used in the prototype applications. These ontologies will namely define the structure of the user interface for the mobile travel guide, and, possibly, the basic functionality. With the TOAT, the task models for the prototype applications will thus be generated.

The TOAT is an authoring environment for SME's, enabling them to define tasks for mobile users based on the established task model. The intended users of the TOAT are thus, firstly, non-expert users, namely SME authors. They should be able to design and manipulate task models in an intuitive manner. Task ontologies have to be easily expandable and specialized with more tasks and subtasks for various use cases and needs.

The TOAT interface is a Microsoft Visio-like tool, allowing users to create task 'bubbles' and connecting these with arrows.

We have collected the following set of user requirements:

- **User-friendliness**: TOAT should be an easy to use environment for task authoring both for expert and non-expert users, like the SME task authors. Therefore, it should provide an intuitive interface for those users, like a graphical Visio-like tool. Section 4.3 provides a thorough discussion about the graphical notation of the TODO language we use to model tasks in TALOS project. We have experimented with several visualizations of TODO for TOAT. In the following we give some examples:

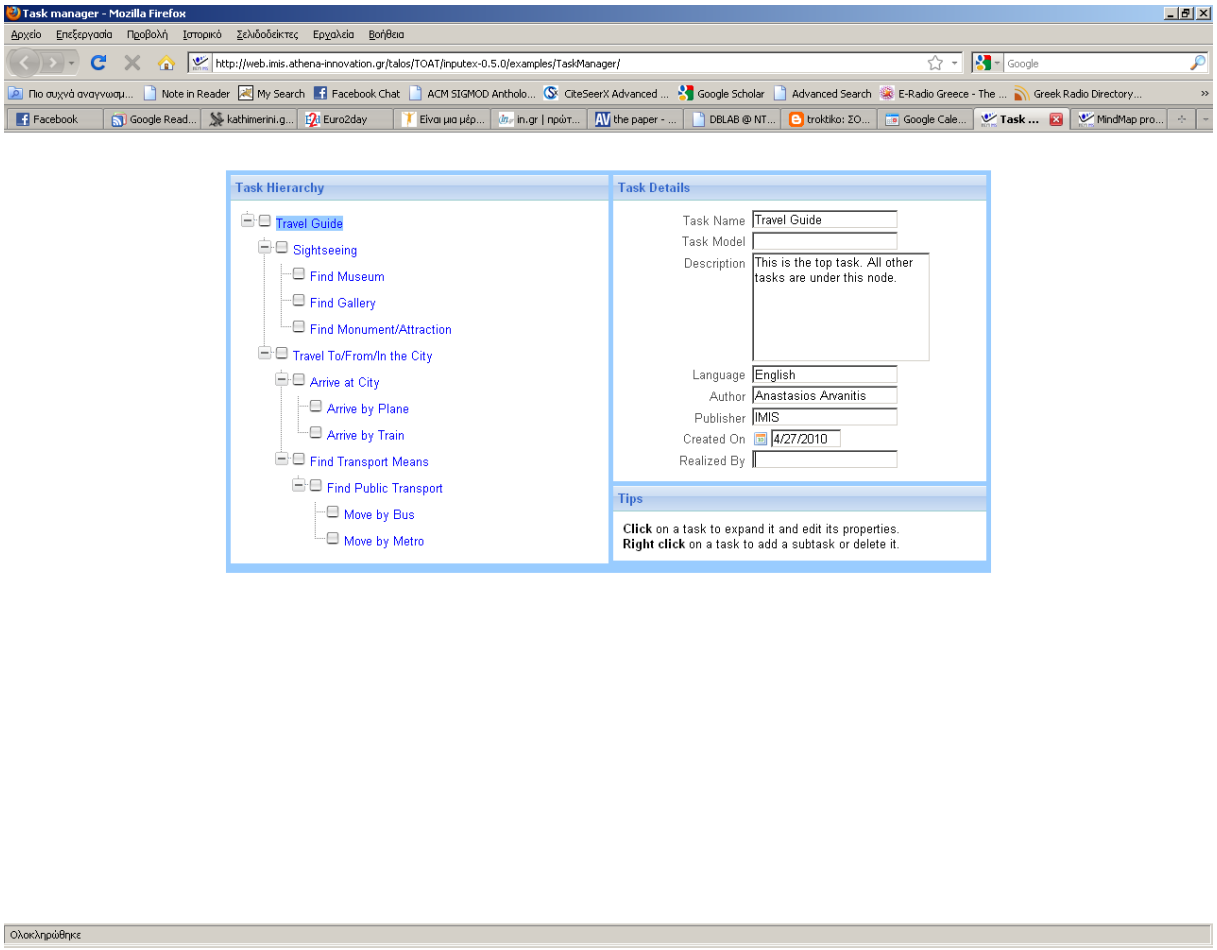

Figure 24: Tree-like Interface

The tree-like prototype interface provides a very easily understood presentation of task hierarchies. Further, we have implemented a task details panel for adding textual details (task parameters) relevant for each task. However, it can easily capture large hierarchies and especially those that are represented as graphs and not as trees or DAGs.

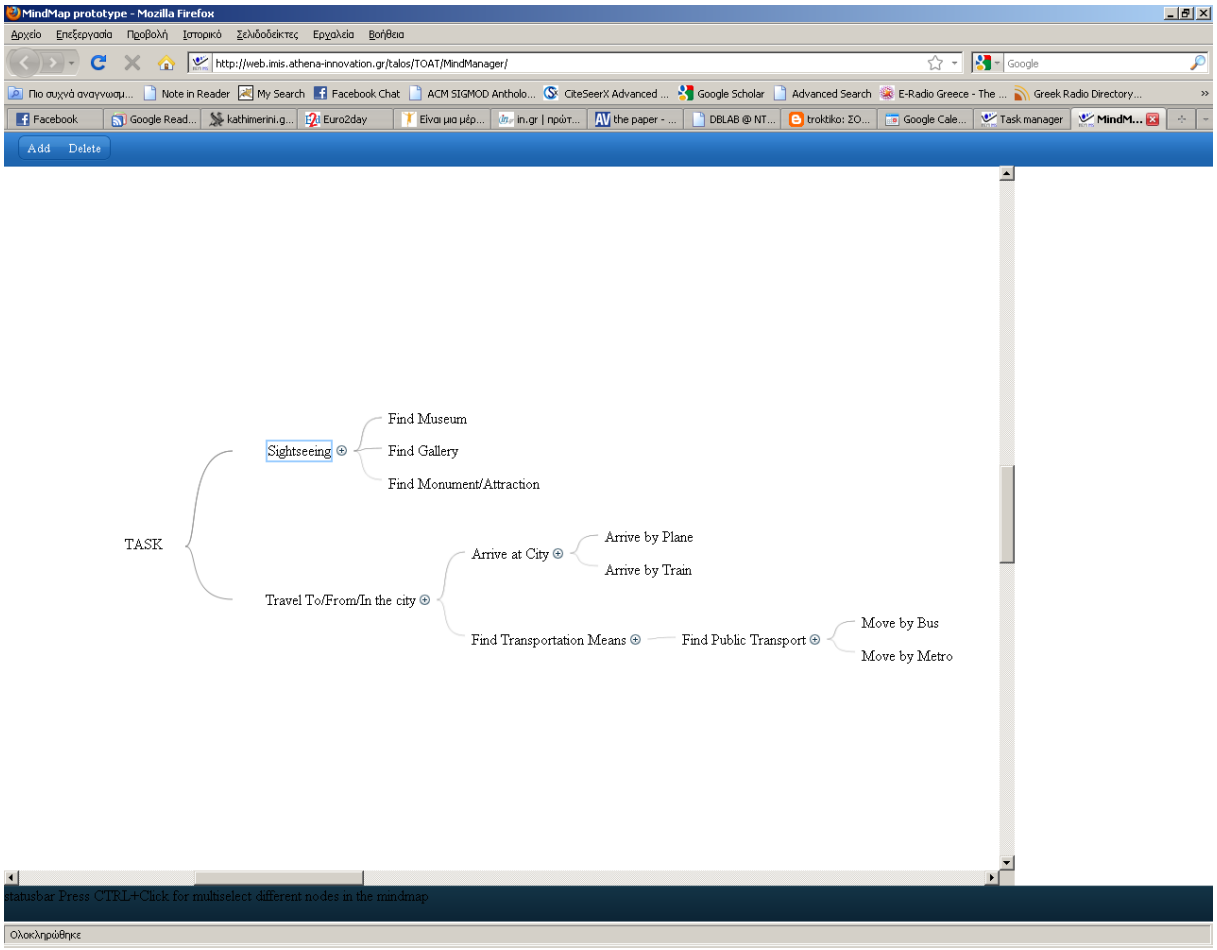

Figure 25: Mindmap-like Interface

The mindmap-like prototype interface combines a very structured presentation of task hierarchies with a text-based way of creating the hierarchies. However, it can still have limitations in regards to the representation of large task hierarchies or graph-based models.

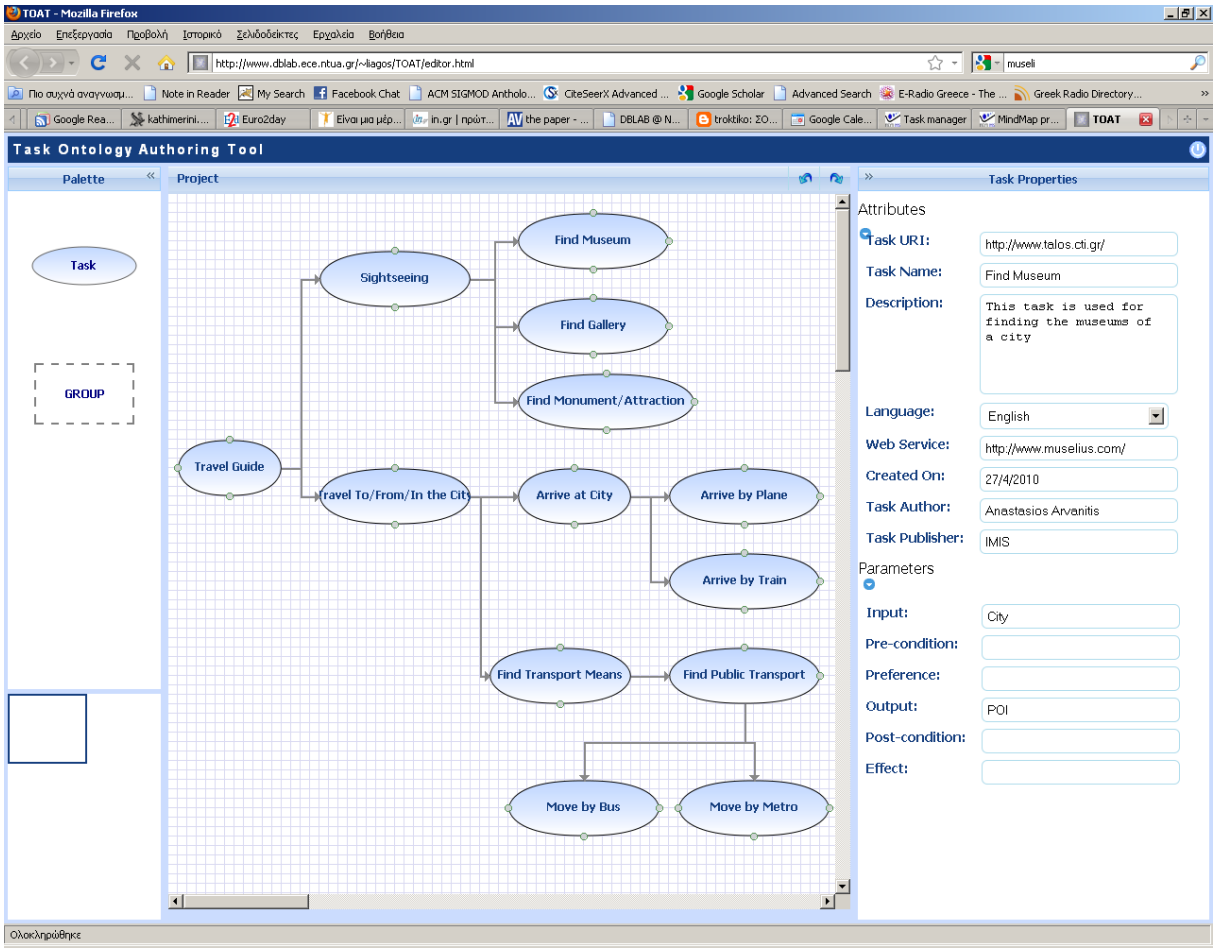

Figure 26: Graph-based interface

The graph-based prototype interface provides a highly structured presentation of task ontologies using a graph layout and drag-n-drop functionalities. Further, we have added a panel for editing task properties (task attributes and parameters). This is the interface option we have decided to use for TOAT development.

- **Minimum installation effort:** TOAT will be used by SMEs authors that are expected to be non-IT experts. Therefore, there is an additional requirement for TOAT to need minimum (or no) installation effort. For example, it should not require the existence of external libraries, frameworks, etc. like Java Runtime Environment (JRE) or .NET SDK.
- **Lightweight:** TOAT should be as light-weight as possible in order to enable editing of large task hierarchies with minimum memory impact.
- **Collaboration capabilities:** TOAT should enable importing and exporting task models from/to TALOS server, versioning of models and user authentication.
- **Open-source:** TOAT will be an intellectual property of SMEs, therefore it should be using open-source technologies.

- **Compatibility with standard technologies – Platform independence:** TOAT should operate in multiple platforms and operating systems and use standard technologies.

6.3 TOAT Overview and Design Decisions

The requirements discussed above lead us to the decision of implementing TOAT as a web-based and not as a desktop application. A web-based application has the following advantages over a desktop one:

- It has a friendly interface in which non-expert users are used to (browser)
- It is platform independent
- It needs no installation as it operates through the browser
- It is lightweight as several heavy operations are performed at the server

Further, TOAT has been developed with the following issues in mind:

- It does not require any external library such as JRE, .NET SDK, flash etc.
- Only open-source frameworks have been used. Specifically, we have used the following JavaScript libraries:
    - jQuery [38] (MIT License)
    - MooTools [39] (MIT License)
    - MooCanvas [40] (MIT License)
    - draw2d [41] (LGPL License)
- It complies with standard web technologies such as XML, DOM, JavaScript and AJAX.

Further, TOAT exhibits the following features:

- A graph-based interactive 2D representation of the task hierarchy
- Compatibility with ToDo graphical notation
- Syntactic (on the ToDo XML Schema level) and semantic validation of task models
- Mapping of graphical models to ToDo XML documents
- Import and export of task models from and into the TALOS server
- Versioning of task models
- User authentication

All graph editing functionalities, such as drag-n-drop, pan, zoom, and layout are performed in client-side using JavaScript an AJAX. More heavyweight functionalities are performed in the server-side using Java 1.6 and SQL. Backend functionalities include user authentication, storing and retrieving of XML task models and interaction with the TALOS server.

6.4 The TOAT UI

This section provides screenshots of the TOAT User Interface.

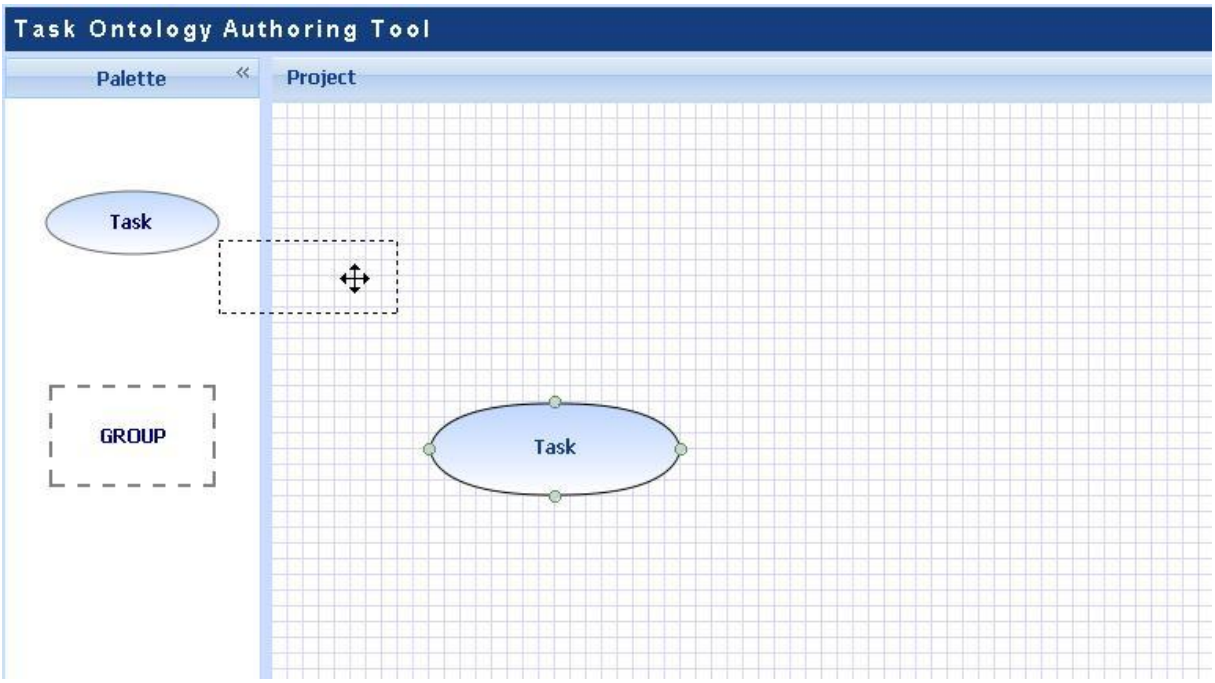

Figure 27: Drag a task from the Palette and drop it on the canvas

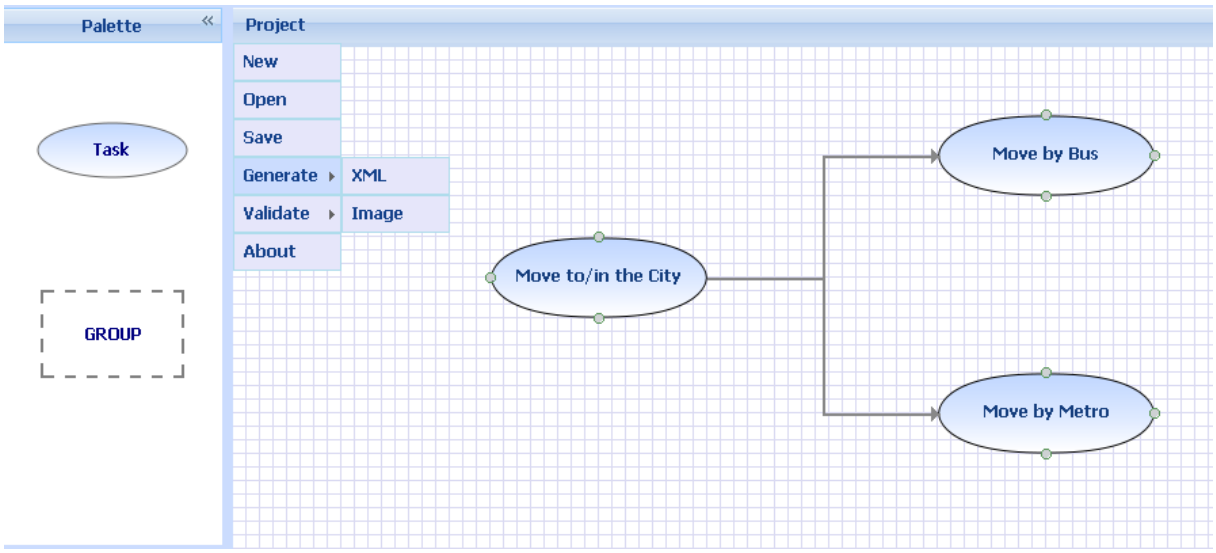

Figure 28: The TOAT Menu

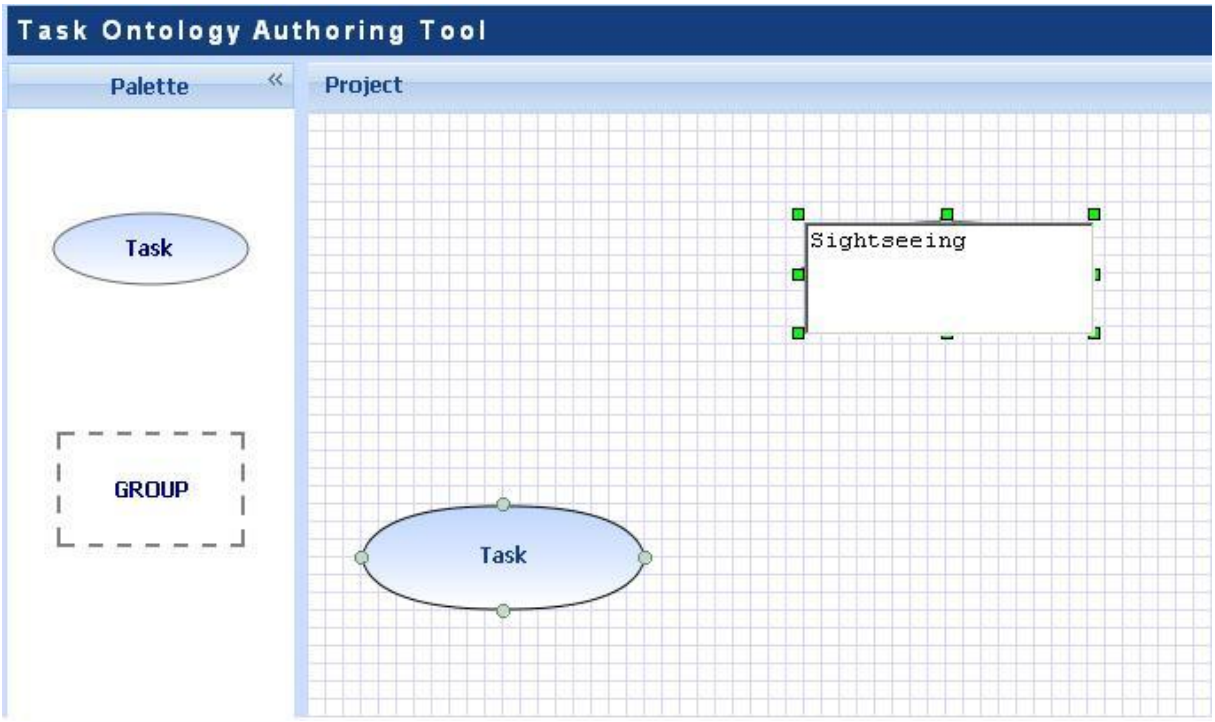

Figure 29: Editing the name of a task

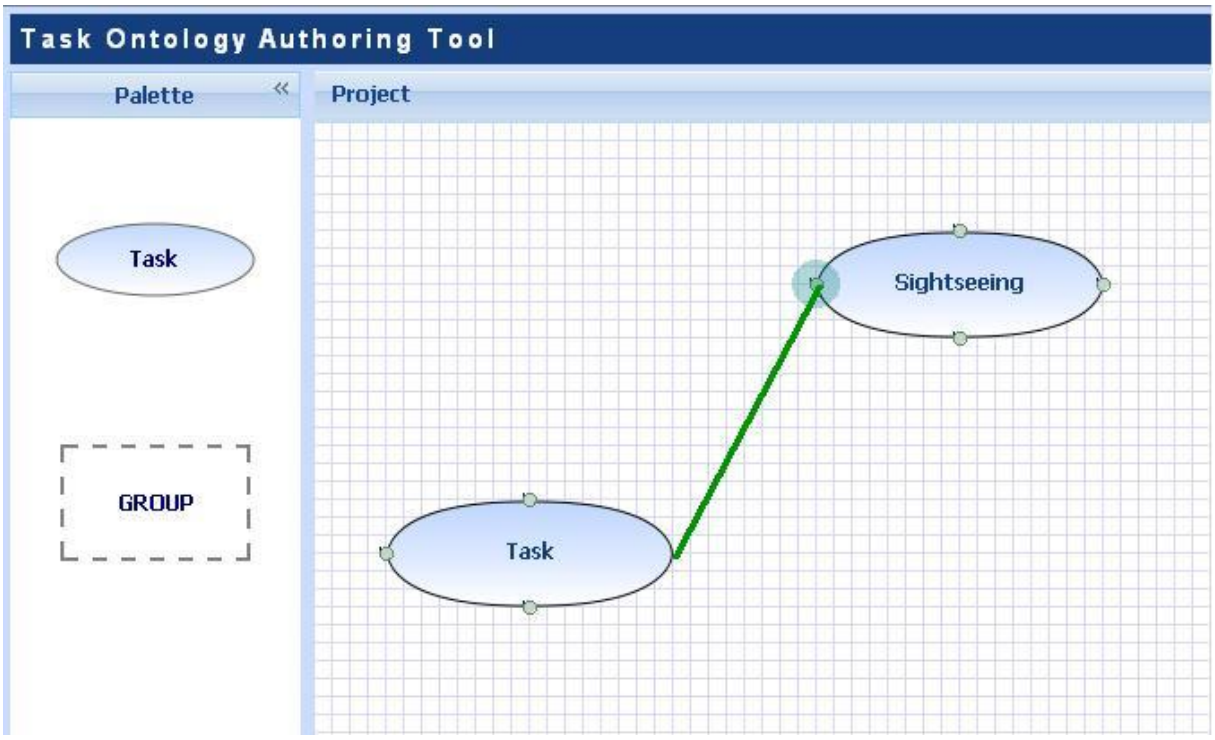

Figure 30: Adding a relation between two tasks

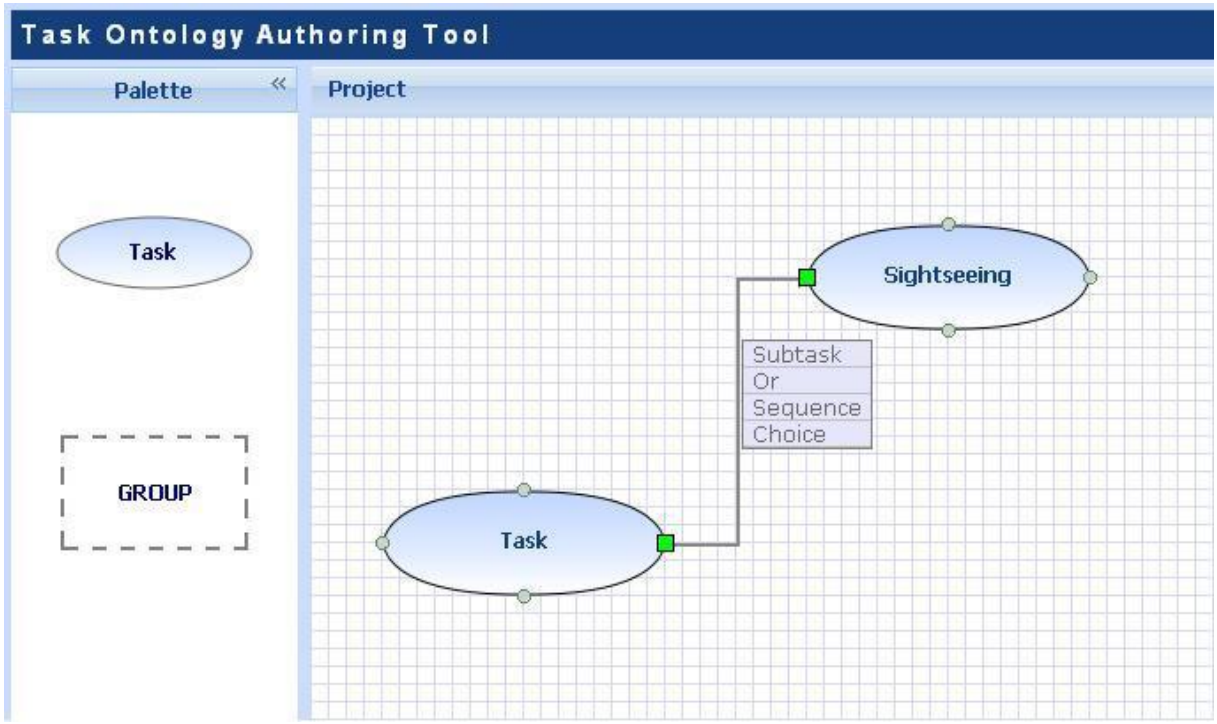

Figure 31: Changing the type of a relation

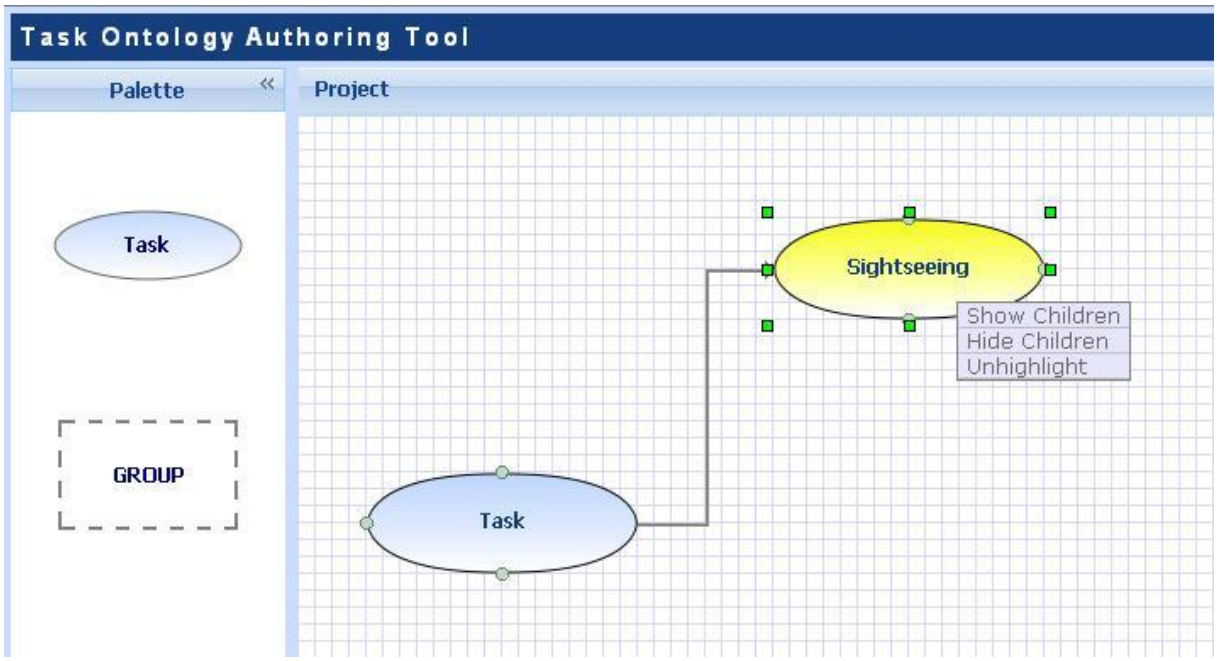

Figure 32: Highlight a task

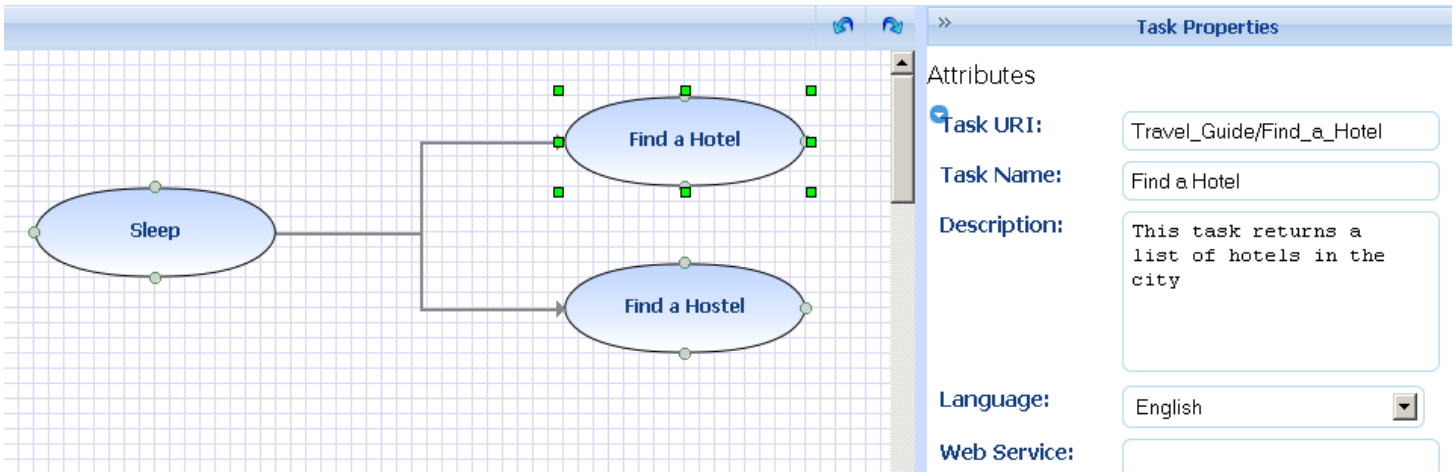

Figure 33: Editing Task Properties

7 TALOS Server Database

7.1 Introduction

This section describes the Data Model of the TALOS Server Database. As described in [42], the server database breaks into two parts: (a) a part where Task Ontologies described in ToDo are stored; this is the so-called **Task Ontology Database (TODB)**, and (b) a part that stores all content-related information; this is the so-called **Content Base (CB)**. The CB includes (a) unstructured content in the form of text, (b) geo-referenced (structured) content as Points of Interest (POIs), and (c) context-related information that is used for filtering the corresponding resources in CB according to a set of context attributes. As we explain in the following, context-related information is essential **(a) for customizing the content** a user may want to retrieve and store in his/her local database [30], i.e. the database of the iPhone, and **(b) for supporting offline context-based recommendation of both POIs and Tasks.**

The remaining section is organised as follows. Section 7.2 provides the Entity-Relationship (ER) diagram from which the Data Model is generated. Section 7.3 discusses the Data Model and illustrates the Relational Schema of the Server Database. Section 7.4 addresses the details (data types, integrity constraints etc.) of all relational tables introduced in Section 7.3.

7.2 Entity-Relationship Diagram

The Entity-Relationship diagram of the TALOS Server Database is depicted in Figure 34. The meaning of the notations used can be found in [43]. The part

referring to TODB is captured by the blue frame. The rest of the database corresponds to CB. Note that the context-related information stored in TALOS server is included in CB.

7.2.1 TODB ER Diagram

The basic entity in TODB is the Task. Each task has the following set of attributes:

- **Model**: Denotes the model to which the task belongs. This attribute is the URI of the Task Ontology. By the time a task changes, the version of the corresponding model changes too (derived property).
- **Name**: The name of the task, e.g. "Book a flight".
- **Version**: Each task has a version which is automatically generated by the time a Task Ontology Author (TOA) creates or updates a task
- **Description**: A piece of text that describes the task.
- **Author**: The name of the TOA. Each task may have more than one co-authors.
- **Publisher**: The name of the publisher, e.g. IMIS/RC "Athena".
- **Created On**: The date a task was created.
- **Language**: The language in which the task attributes are given, e.g. German.
- **Web Service**: The URL of the web service that realizes the task. Each task may be realized by more than one web services. We point out that **Web Services in TALOS are mainly used for realizing general tasks**, i.e. tasks that break down into simpler ones but can optionally be accomplished by simply redirecting the user to the available Web Service. The task "Book a Hotel Room" is a representative example of this kind.

Each task in TODB is identified by its **Name**, **Version** and the **Model** it belongs to. Thus, these three attributes serve as a Primary Key (PK) of the Task entity.

Besides the above attributes, each task has an additional set of parameters. The **Parameter** entity is defined as a weak entity because, from a conceptual point of view, a parameter cannot exist in the absence of a task. Thus, each parameter is identified by its **Name** along with the PK of the **Task** entity. Each parameter can also have a **Description** for the task authors explaining its meaning, purpose etc. The datatype of the value a parameter is instantiated with

is denoted by the **XSD** attribute. For the purposes of TALOS, a task parameter can only have one of the following types that are captured by the **Type** attribute:

- *Input*
- *Output*

The relations of the TODB ER Diagram provided in Figure 34 are the following:

- **SubTaskOf**: It denotes a subsumption relationship between a parent-task and its subtasks. A task may have one or more children. All tasks in the task ontology have at least one parent task. The attribute **Parent Task** denotes the super-task.
- **Sequence**: It denotes a time ordering between tasks. A task may participate in more than one sequence relations. A sequence relation may also denote a dataflow from the first task to the second one. Such a dataflow is interpreted as a parameter passing. The parameter that is passed from the first task is denoted by the attribute **Source Parameter**, while the parameter of the second task that is instantiated with the passed value is denoted by the **Target Parameter** attribute. Note that a sequence between two tasks may introduce more than one parameter passing. The attribute **Source Task** denotes which task is first in order of execution.
- **OR**: A task that is accomplished by at least one of its children is related with its subtasks through the OR relation.
- **CHOICE**: A task that is accomplished by exactly one of its children is related with its subtasks through the CHOICE relation.
- **GROUP**: Tasks in a ToDo ontology may form a group. This group is regarded as an anonymous task that is related with the corresponding tasks through the GROUP relation. Note that tasks into a group may be related one another with one of the **SubTaskOf**, **OR**, **CHOICE,** and **Sequence** relations.
- **Has Parameter**: It denotes the (weak) relationship between a task and its parameters.
- **Recommended For**: Each task is connected with a piece of context-related information. **This information can be downloaded and stored in the user's database in order to serve as a filter for an offline context-based task recommendation service.**

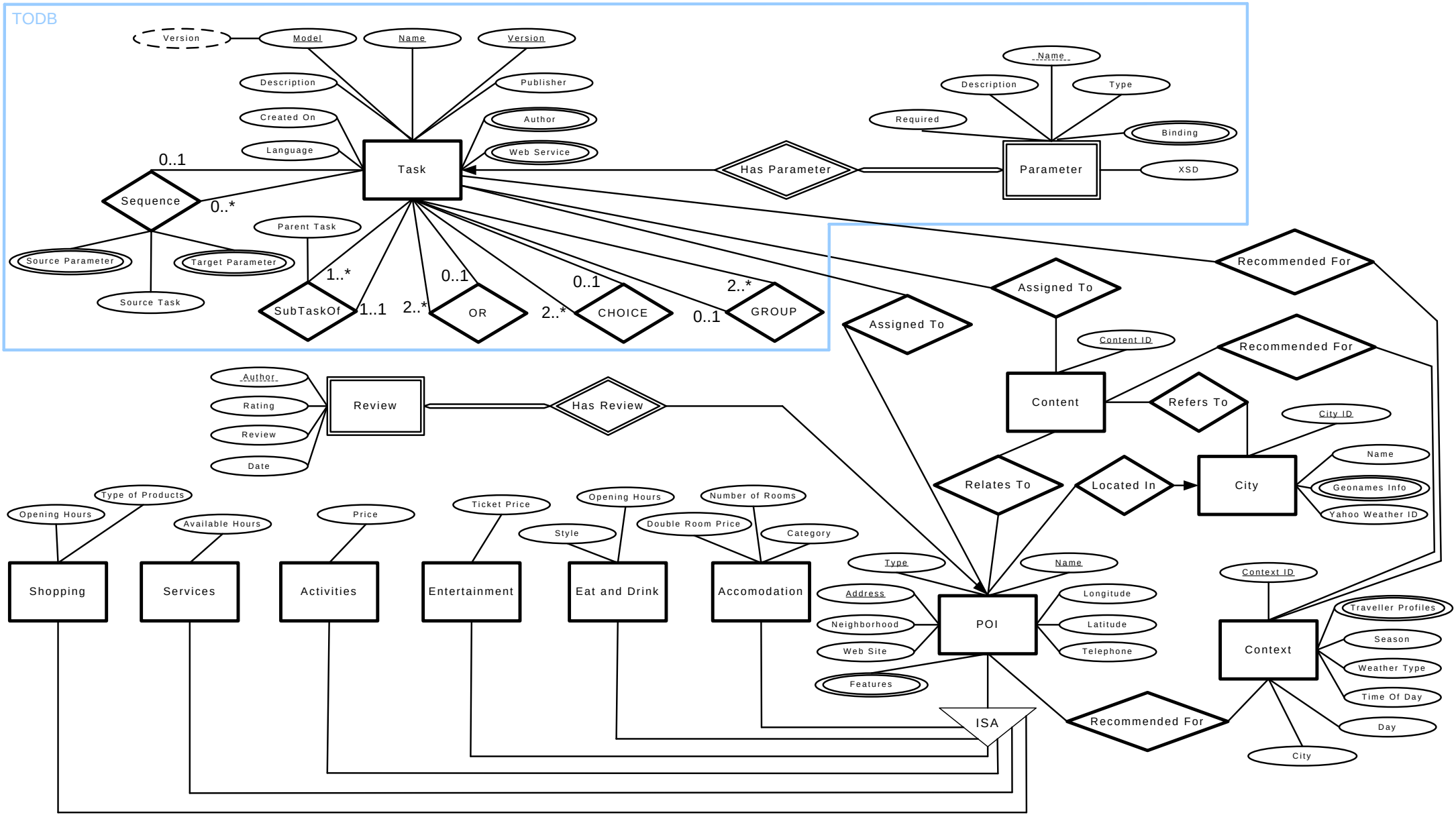

Figure 34: ER Diagram of the TALOS Server Database

- **Assigned To:** This relation connects a task in TODB with a piece of content stored in CB. It is used to capture the Content-to-Task annotation made by the Content Managers through the Task Annotation Tool. We emphasize that in case there are POIs in CB with no related content, these POIs can be associated directly with a specific task through the **Assigned To** relation.

7.2.2 CB ER Diagram

In CB there are three basic entities: **Content**, **POI** and **City**. The entity Content captures the unstructured content in the form of text (paragraphs, sections), images, etc. Each piece of content has the following attribute:

- **Content ID**: This is a unique ID used to identify the different pieces of content. By using this ID, the TALOS server is able to identify the type, version and path (URL, file, database) where the actual content is stored.

Content is related to Context through the **Recommended For** relation. This is very useful for **customizing the content retrieved from the TALOS server according to the user's profile and date of his/her trip**. Taking into account that POIs are also related to context (and content), one can easily argue that an additional relation between content and context is useless. However, there are two important factors that impose such a Content-to-Context relationship. The first one is for distinguishing content that relates to the same POIs but addresses **different types of readers** (e.g. young people, elders etc.). The second and most important reason is that a specific piece of content may refer to a **city event** that is not classified under a POI subtype. Therefore, similarly to the case of POI filtering, a user should also be able to download only the events that take place during the small period of his/her trip.

The **POI** entity captures all geo-referenced content. Each POI has the following set of attributes:

- **Name**: The name of the POI.
- **Type**: The type of the POI.
- **Address**: The physical address of the POI.
- **Longitude**: The longitude of the POI.
- **Latitude**: The latitude of the POI.

- **Neighbourhood**: The name of a neighbourhood a POI belongs to. It is used in case the neighbourhood is well-known, e.g. Thessio in Athens.
- **Web Site**: The URL of the web site of the POI.
- **Telephone**: The telephone number of the POI.
- **Features**: The specific features of the POI. Each POI may have more than one features, e.g. a hotel may have both pool and mini bar as features.

Each POI may also have a review. The **Review** entity is defined to be weak as it must always refer to a specific POI. The **Date** attribute is the date a review was submitted, **Author** is the name of the review's author, the **Rating** attribute is the POI rate given by the review, and the **Review** attribute is the actual text.

Besides the previous attributes, each distinct POI subtype has also a set of additional properties. Due to lack of space Figure 20 depicts only some of the representative properties of the corresponding POI subtypes. POI subtypes are used **(a) in case a user may want to download only specific kind of POIs from the TALOS Server and (b) for optimizing the queries related to POIs by grouping POIs of the same category into distinct entity sets and thus into different relational tables** (see Section 7.3). The provided POI subtypes are the following:

- **Accommodation**: Hotels, Motels, Rooms to Let, etc.
- **Eat and Drink**: Restaurants, Bars, Pubs, etc.
- **Entertainment**: Museums, Theatres, Cinemas, etc.
- **Activities**: Hitchhiking, Boat Excursions, etc.
- **Services**: Police Stations, Banks, Hospitals, Rent a Car etc.
- **Shopping**: Shopping Malls, Shopping Areas, Public Markets etc.

As the provided content in TALOS system refers to specific cities (e.g. Berlin, Brussels, etc.), a **City** is regarded as a distinct entity in our ER model. Each city is identified by a unique **City ID** (PK) and it has also a (unique) **Yahoo Weather ID,** a **Name** and a set of attributes taken from Geonames Database[7] (**Geonames Info)**. Each POI is located in one and only one city. This relationship is captured by the (N:1) **Located In** relation of Figure 20.

Content in CB may relate to a specific POI (Relates To) or to a specific City (Refers To). For instance, general information about the history of a city is directly assigned to this city and not to a POI. Note that POIs are also

[7] http://www.geonames.org/

associated with a piece of context captured by the **Context** entity (**Recommended For**). This information is used **(a) for filtering the POIs** according to a specified user profile (when downloading content from the server) and **(b) for recommending POIs or Tasks** to the users based on context-related information such as **Weather**, **Day**, **Time Of Day** and **Season** of their trip. The latter is achieved by downloading the corresponding part of context into the user's local database and combining it with the dynamic context retrieved by the Context Aggregator.

7.3 Data Model Overview

As mentioned in the previous, the three major entities that play a central role in the TALOS project are: **Tasks, Context and Content** (both structured and unstructured).

Tasks provide a means to organize the activities or goals of the TALOS users. In the travel guide use case scenario, tasks could be "Find nice places to eat or drink", "Get information about museums and exhibitions" or "Move to a hotel". More detailed information regarding the structure and use of tasks in the TALOS project can be found in Section 4.

Available content can be either **static** such as text, photos or maps provided by a travel guide or **dynamic** such as transportation schedules, ticket prices, events and museum exhibitions retrieved from the web. Content can be also well-structured in the form of POIs (such as hotels, restaurants, bars, metro stations etc.) each one having a set of properties (price, opening hours, music style etc.).

Finally, context is used for modelling a set of contextual attributes related to a user's situation. Its purpose is to qualify tasks, content and POIs with specific context conditions.

As shown in the ER Diagram of Section 7.3.3, the aforementioned entities are related one another. Therefore, the suggested data model provides tables for Task-to-Content, Task-to-Context, POI-to-Context, and Content-to-Context relations. Figure 35 illustrates the Relational Schema of the TALOS Server Database. The detailed description of each table follows in Section 7.4.

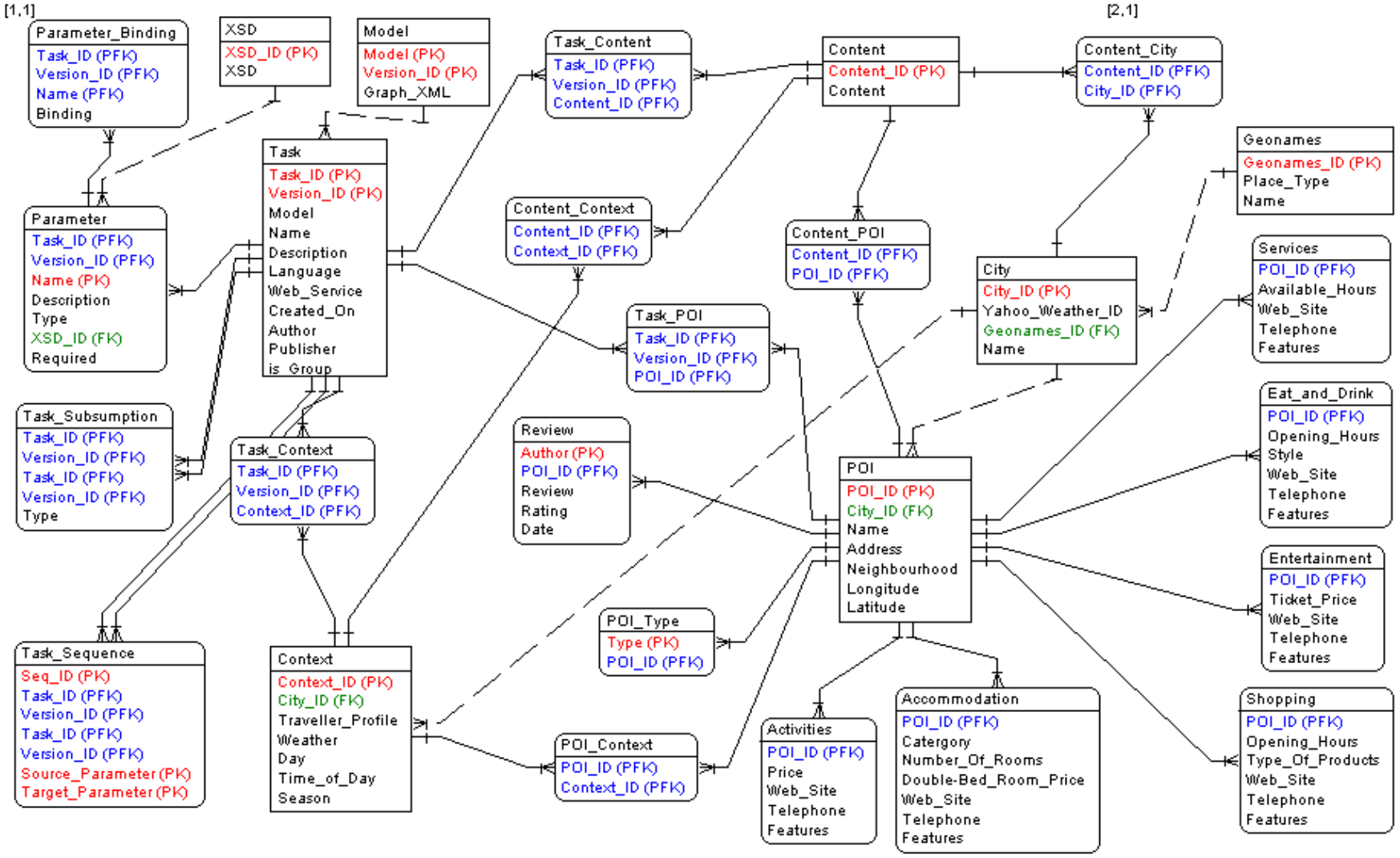

Figure 35: The Relational Schema of the TALOS Server Database

7.4 Description of Tables

In this section we provide detailed information about the properties of each table shown in Figure 21.

7.4.1 Table *Task*

Column Name	Column Type	Description
Task_ID	auto-generated ID	PK
Version_ID	varchar	PK
Model	varchar	URI of the Task Ontology
Name	varchar	
Description	varchar	
Language	char(20)	
Author	varchar	Name of the authors separated with comma (;)
Publisher	varchar	
Created_On	datetime	
Web_Service	URL	URLs separated with comma (;)
is_Group	bool	Specifies if the task is atomic or an anonymous group of tasks

Note that although the three attributes (**Model**, **Name** and **Version_ID**) can identify all together a task in the **Task** table (they are unique), we decided to create an auto-generated ID (Task_ID) in order to facilitate joins including the Task table. This ID actually encodes the first two attributes (**Model** and **Name**).

As far as the versioning problem is concerned, we can follow two different approaches. Due to the fact that **we cannot overwrite the changes in the database**, both approaches use replication and not overwriting. The naïve one is to **(a) replicate the whole Task Ontology**, i.e. all tasks belonging to the corresponding model, while the other is **(b) to replicate only the specific part of the ontology that was infected by the updates and use the remaining part as is**. We explain the advantages and disadvantages of each approach in the following:

1. **Replicating the whole model**

   Each time a TOA updates at least one Task, a new Task Ontology is inserted into TODB. Each task in this model has the same ID as the corresponding task in the previous model, but a different (higher) version number. **Note that this new version number is the same for all**

tasks in the model and thus it represents the version of the whole model. The task-to-content and task-to-context assignments are not propagated to the new Task Ontology and thus it is considered to be an idle ontology (ITO). This means that in order for a user to be able to download this new version, a Content Manager must make all assignments from scratch and publish the ontology as operational (see Section 6 for details about the TALOS Ontology Lifecycle).

2. **Replicating part of the model updated**

   In this case, each time one or more tuples in tables **Task** or **Parameter** are updated, one or more tasks are inserted into TODB (depending on the update). These new tasks correspond to new versions of the existing ones. The tuples that express a relation between the updated tasks are replicated with the new version numbers of the participating tasks. In this case, the version of the model is the higher version among all tasks belonging to this model and it is not the same for all tasks. All assignments are propagated and thus the new version is considered to be an OTO. **Note that this approach is applicable only in case a task is slightly updated, i.e. a parameter is added or an attribute changes, and not in case tasks are deleted or inserted in TODB**. In the latter case, the evolution of the ontology is quite difficult to handle and thus we have decided to follow the first approach where the whole model is replicated.

7.4.2 Table *Model*

Column Name	Column Type	Description
Model	varchar	PK (FK to Task->Task_ID)
Version_ID	varchar	PK (FK to Task->Version_ID)
Graph_XML	text	Contains an XML representation of the model graph generated by TOAT

7.4.3 Table *Task_Subsumption*

Column Name	Column Type	Description
Task_ID1	int	PK (FK to Task->Task_ID)
Version_ID1	varchar	PK (FK to Task->Version_ID)
Task_ID2	varchar	PK (FK to Task->Task_ID)
Version_ID2	varchar	PK (FK to Task->Version_ID)
Type	smallint	One of: 1 (Sub), 2 (OR), 3 (CHOICE), 4 (SubSeq) SubSeq denotes that the child-task participates in a sequence chain

The reason we decided to follow this **flat representation of the Task Hierarchy** is because the tasks are unravelled gradually as the user interacts with the interface of the mobile device (see [44]) and thus, at each step, **only the direct subtasks of the current task are needed**. In other words, there is no need to reconstruct the whole hierarchy by recursively traversing the task-nodes. The only case we may need to recursively traverse a path in the Task Hierarchy is when we have a GROUP construct as subtask. However, we argue that, in our travel guide use case scenario, sequentially nested GROUPs are unlikely to exist and thus we only need to repeat the traversal once more (for the GROUP encountered). In any case, the repeated traversal of the hierarchy can be done (as many times as needed) using the **is_Group** attribute of the retrieved task.

7.4.4 Table *Task_Sequence*

Column Name	Column Type	Description
Seq_ID	int	PK (FK to Task->Task_ID)
Source_Task_ID	int	PK (FK to Task->Task_ID)
Source_Version_ID	varchar	PK (FK to Task->Version_ID)
Target_Task_ID	varchar	PK (FK to Task->Task_ID)
Target_Version_ID	varchar	PK (FK to Task->Version_ID)
Source_Parameter	varchar	PK (FK to Parameter->Name)
Target_Parameter	varchar	PK (FK to Parameter->Name)

Each distinct sequence chain is captured by an auto-generated anonymous group task. The ID of this task is denoted by **Seq_ID**. Note that a sequence between two tasks may introduce more than one parameters passing and thus **Source_Parameter** and **Target_Parameter** are also included in the PK of the table.

7.4.5 Table *Parameter*

Column Name	Column Type	Description
Task_ID	int	PK (FK to Task->Task_ID)
Version_ID	varchar	PK (FK to Task->Version_ID)
Name	varchar	PK (The name of the parameter)
Description	text	A description of the parameter for the authors
Type	smallint	One of the six types introduced in Section 1 encoded from 1 to 6
XSD_ID	int	FK to XSD->XSD_ID
Required	bool	Indicates if a parameter is required or not

7.4.6 Table *Parameter_Binding*

Column Name	Column Type	Description
Task_ID	int	PK (FK to Task->Task_ID)
Version_ID	varchar	PK (FK to Task->Version_ID)
Name	varchar	PK (The name of the parameter)
Binding	smallint	One of: 1 (Context), 2 (User), 3 (Database)

7.4.7 Table *XSD*

Column Name	Column Type	Description
XSD_ID	auto-generated ID	PK
XSD	text	The XML datatype

7.4.8 Table *Content*

Column Name	Column Type	Description
Content_ID	auto-generated ID	PK
Content	text	The actual content

7.4.9 Table *POI*

Column Name	Column Type	Description
POI_ID	auto-generated ID	PK
Name	varchar	The name of the POI
City_ID	int	FK to City->City_ID
Longitude	double precision	
Latitude	double precision	
Address	varchar	
Neighbourhood	varchar	The name of the neighbourhood the POI belongs to

POIs are located to a city (or more explicitly to a neighbourhood of a city) represented by the **City_ID** and **Neighbourhood** attributes respectively. The **City_ID** attribute acts as a foreign key to the **City** table, which enables the application to get more information available about the city. **If a POI represents an area instead of a point in the map, (like for example parks) we use the POI center as coordinates**.

POIs are categorized into different types; we have identified 6 general type categories for POIs, i.e. **Accommodation**, **Eat and Drink**, **Shopping**, **Services**, **Activities** and **Entertainment**. POIs of different types have different additional attributes, for example a hotel has a number of rooms, whereas a museum has opening hours and ticket prices. **We have made a design decision not to include any location-irrelevant attributes inside the POI table (except Name), because this table is used in queries just for retrieving IDs of POIs that match a specified location (and maybe Type) parameter.** In addition, although our design decision regarding the relational schema may contradict at first sight with the ISA relationship of the ER model provided in Section 1, we emphasize that POI attributes such as **Features** are not common in all POI subtypes. **It just seems to be this case because our prototype implementation does not follow a Normal Form (see [43]) due to lack of space**. Therefore, in this document, the data model provides different tables for POI types which contain both common and non common attributes.

The **Type** attribute of a POI (e.g. restaurant, bar, hotel, metro station etc.) is stored in a separate table named **POI_Type** (see Section 7.4.24). Depending on the **Type** attribute, the TALOS server is responsible for extracting all available information about a POI by joining the POI table with the corresponding relational table of its type. For example, types "restaurant" and "bar" correspond to **Eat_And_Drink**, "hotel" corresponds to **Accommodation**, "metro station" correspond to **Services** table etc. In order to facilitate the integration with **Qype**

API[1], we propose that the different types of POIs should be identical to the Qype category_id values. In this way, we can also utilize Qype's detailed and sophisticated POI type hierarchy and POI database.

7.4.10 Table *Accommodation*

Column Name	Column Type	Description
POI_ID	int	PK (FK to POI->POI_ID)
Number_Of_Rooms	int	
Double-Bed_Room_Price	double precision	
Category	varchar	Stars for hotels, ranks for rooms to let
Telephone	varchar	Tel numbers separated with comma (;)
Web_Site	URL	
Features	varchar	Key features separated with comma (;)

Regarding the features of each subtype, **the Relational Schema of our prototype implementation does not follow a Normal Form** (see [43]). In other words, queries over distinct POI features cannot be always formulated using standard SQL. This also holds for all other POI tables. However, **in case the database administrator knows exactly which features belong to which POIs, then it is easy to define the corresponding features as separate fields in each table and manipulate them directly with standard SQL features**.

7.4.11 Table *Eat_and_Drink*

Column Name	Column Type	Description
POI_ID	int	PK (FK to POI->POI_ID)
Features	varchar	Key features separated with comma (;)
Opening_Hours	varchar	
Telephone	varchar	Tel numbers separated with comma (;)
Web_Site	URL	
Style	varchar	Style features (jazz music, chinese food etc.) separated with comma (;)

[1] http://www.qype.co.uk/developers/api

7.4.12 Table *Shopping*

Column Name	Column Type	Description
POI_ID	int	PK (FK to POI->POI_ID)
Type_Of_Products	varchar	
Features	varchar	Key features (eg. brands) separated with comma (;)
Opening_Hours	varchar	
Telephone	varchar	
Web_Site	URL	

7.4.13 Table *Services*

Column Name	Column Type	Description
POI_ID	int	PK (FK to POI->POI_ID)
Available_Hours	varchar	
Features	varchar	Key features separated with comma (;)
Telephone	varchar	
Web_Site	URL	

7.4.14 Table *Activities*

Column Name	Column Type	Description
POI_ID	int	PK (FK to POI->POI_ID)
Price	double precision	
Features	varchar	Key features separated with comma (;)
Telephone	varchar	
Web_Site	URL	

7.4.15 Table *Entertainment*

Column Name	Column Type	Description
POI_ID	int	PK (FK to POI->POI_ID)
Ticket_Price	double precision	
Features	varchar	Key features separated with comma (;)
Telephone	varchar	
Web_Site	URL	

7.4.16 Table *Review*

Column Name	Column Type	Description
POI_ID	int	PK (FK to POI->POI_ID)
Author	varchar	PK
Review	text	
Rating	smallint	From 1 to 5
Date	date	The date of the review

7.4.17 Table *Content_POI*

Column Name	Column Type	Description
Content_ID	int	PK (FK to Content->Content_ID)
POI_ID	int	PK (FK to POI->POI_ID)

This table associates pieces of content (paragraphs, sections, photos, map tiles etc.) with a POI.

7.4.18 Table *Content_City*

Column Name	Column Type	Description
Content_ID	int	PK (FK to Content->Content_ID)
City_ID	int	PK (FK to City->City_ID)

This table associates pieces of content (paragraphs, sections, photos, map tiles etc.) to specific cities. It is used for assigning general content (that is not related to a specific POI) with a city.

7.4.19 Table *City*

Column Name	Column Type	Description
City_ID	auto-generated ID	PK
Name	varchar	
Yahoo_Weather_ID	char(10)	The unique ID given to a city by Yahoo weather web service
Geonames_ID	int	FK to Place->Geonames_ID

Users of the mobile travel guide are expected to search for content and services related with a city. Further, many web services, like Yahoo! Weather, Geonames web services etc., take city as parameter. Therefore, we have included a City table in the database schema. Each city, apart from its **Name** and auto-generated **City_ID**, has a **Yahoo_Weather_ID** used for accessing Yahoo Weather services and a **Geonames_ID** (the ID of the city in the Geonames database).

7.4.20 Table *Geonames*

Column Name	Column Type	Description
Geonames_ID	int	PK
Place_Type	varchar	
Name	varchar	

This table represents a place entity in the Geonames database. Geonames services provide additional information about a city, such as its timezone, place hierarchy, currency etc.

7.4.21 Table *Context*

Column Name	Column Type	Description
Context_ID	auto-generated ID	PK
City_ID	int	FK to City->City_ID
Weather	varchar	
Season	varchar	
Day	varchar	
Time_Of_Day	varchar	
Traveller_Profile	varchar	

Context is used for modelling a set of contextual attributes related to a user's situation. **The purpose of context is to be used as a means to recommend personalized and thus more relevant content or tasks to the user**. In the mobile guide prototype, the contextual attributes include date and time (**Time_Of_Day**, **Day** and **Season**), weather conditions (**Weather**), and general user preferences which are grouped into predefined traveller types (**Traveller Profile**). **A specific context instance groups these contextual attributes as key-value pairs, so that they can be stored and linked with the other entities (Tasks, Content, POIs)**. **We emphasize that we do not intend to store dynamic context (e.g. changing weather conditions, user's current location etc.) in the TALOS server database**. Instead, we provide a means for task and content authors to declare that a specific activity (e.g. a boat trip) or a POI (e.g. an open-air market) are highly suggested or available only under specific context conditions (e.g. hot or calm weather). Therefore, we suggest that the values of contextual attributes should be taken from a predefined set. For example, Yahoo! Weather provides a set of 47 different codes for weather conditions; instead we propose to use only five: {Hot, Rainy, Cold, Calm, ALL} which we think are adequate for the purposes of the TALOS prototype. For the other contextual attributes we suggest the following possible values:

- **Season**: Winter, Summer, ALL
- **Day**: Monday-Friday, Weekend , ALL
- **Time of Day**: Morning, Afternoon, Night, ALL
- **Traveller Type**: Backpack Traveller, Business Traveller, Traveller with Family, Disabled Traveller, ALL

7.4.22 Table *Task_Content*

Column Name	Column Type	Description
Task_ID	int	PK (FK to Task->Task_ID)
Version_ID	int	PK (FK to Task->Version_ID)
Content_ID	int	PK (FK to Content->Content_ID)

This table stores the correspondence between pieces of content in CB and tasks in TODB. As already mentioned, the assignment of content to tasks is performed manually by the Content Manager through the Task Annotation Tool.

7.4.23 Table *Task_POI*

Column Name	Column Type	Description
Task_ID	int	PK (FK to Task->Task_ID)
Version_ID	int	PK (FK to Task->Version_ID)
POI_ID	int	PK (FK to Content->POI_ID)

This table stores the correspondence between POIs in CB and tasks in TODB. It is used for assigning tasks to POIs of CB that do not have related (unstructured) content.

7.4.24 Table *POI_Type*

Column Name	Column Type	Description
Type	varchar	PK (restaurant, pub etc.)
POI_ID	int	PK (FK to POI->POI_ID)

7.4.25 Table *POI_Context*

Column Name	Column Type	Description
POI_ID	int	PK (FK to POI->POI_ID)
Context_ID	int	PK (FK to Context->Context_ID)

This table associates POIs to specific context. From the application perspective, this relation is used for filtering the provided POIs according to the user's preferences and context.

7.4.26 Table *Content_Context*

Column Name	Column Type	Description
Content_ID	int	PK (FK to Content->Content_ID)
Context_ID	int	PK (FK to Context->Context_ID)

This table is used for assigning pieces of content to specific context parameters. Its purpose is similar to that of the previous table.

7.4.27 Table *Task_Context*

Column Name	Column Type	Description
Task_ID	int	PK (FK to Task->Task_ID)
Context_ID	int	PK (FK to Context->Context_ID)
Version_ID	int	PK (FK to Task->Version_ID)

This table associates tasks with specific context parameters. It is used for filtering and/or recommending tasks according to the user's preferences and context.

8 References

[1] TALOS: Task Aware Location Based Services for Mobile Environments, Project Proposal (at TALOS wiki).

[2] Naganuma, T. and Kurakake, S., Task Knowledge Based Retrieval for Service Relevant to Mobile User's Activity. International Semantic Web Conference (ISWC), 2005.

[3] Mizoguchi, R., Van Welkenhuysen, J. and Ikeda, M., Task Ontology for Reuse of Problem Solving, Towards very large Knowledge Bases, 1995.

[4] Schreiber, G., Akkermans, H., Anjewierden, A., de Hoog, R., Shadbolt, N., Van de Velde, W. and Wielinga, B., Knowledge Engineering and Management - The Common-KADS Methodology, MIT Press, 2000.

[5] Von Hunolstein, S. and Zipf, A., Towards Task Oriented Map-based Mobile Guides. In Proceedings of the International Workshop "HCI in Mobile Guides", 2003.

[6] Van Welie, M., Task-Based User Interface Design, PhD Thesis, Vrije Universiteit Amsterdam, 2001.

[7] Berners-Lee, T., Hendler, J. and Lassila, O., The Semantic Web, Scientific American, May 2001.

[8] http://protege.cim3.net/file/pub/ontologies/generations/generations.owl

[9] http://protege.cim3.net/file/pub/ontologies/wine/wine.owl

[10] Protégé Ontology Editor, http://protege.stanford.edu/

[11] Web Protégé, http://protegewiki.stanford.edu/index.php/WebProtege

[12] OWLSight Ontology Browser, http://pellet.owldl.com/ontology-browser

[13] Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D. and Patel-Schneider, P.F., The Description Logics Handbook, Cambridge University Press, 2002.

[14] Baader, F. and Sattler, U., An Overview of Tableau Algorithms for Description Logics. In Tableux, 2000.

[15] Pellet: An open-source OWL-DL Reasoner, http://clarkparsia.com/pellet/

[16] HermiT OWL Reasoner, http://www.hermit-reasoner.com/

Naganuma, T. Luther, M., Wagner, M., Tomioka, A., Fujii,

[17] K., Fukazawa, Y. and Kurakake, S., Task-Oriented Mobile Service Recommendation Enhanced by a Situational Reasoning Engine. In Proceedings of the Asian Semantic Web Conference, 2006.

[18] Resource Description Framework (RDF), http://www.w3.org/RDF/

[19] Web Ontology Language (OWL), http://www.w3.org/TR/owl-features

[20] World Wide Web Consortium (W3C), http://www.w3.org/

[21] Suggested Upper Merged Ontology (SUMO), http://www.ontologyportal.org/

[22] Dublin Core Metadata Initiative, http://dublincore.org/

[23] Unified Modeling Language, http://www.omg.org/technology/documents/formal/uml.htm

[24] Business Process Modeling Notation, http://www.bpmn.org/

[25] OWL-S, Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/

[26] Sasajima, M., Kitamura, Y., Naganuma, T., Kurakake, S. and Mizoguchi, R., Task Ontology-Based Framework for Modeling Users' Activities for Mobile Service Navigation (poster). In Proceeding of the European Semantic Web Conference (ESWC), 2006.

[27] Model-View-Controller Architecture, http://en.wikipedia.org/wiki/Model-view-controller

[28] Arvanitis, A., Athanasiou S. and Liagouris, J., A survey on Context Models, 2009 (at TALOS wiki).

[29] Georgantas, P., D1.1 - Context Aggregation, 2010 (at TALOS wiki)

[30] Tsigka, E., Pfennigschmidt, S., Arvanitis, A. and Liagouris, J., D4.1 – Client-side Task Model, 2010 (at TALOS wiki)

[31] Haarslev, V. and Möller, R., Expressive ABox Reasoning with Number Restrictions, Role Hierarchies, and Transitively Closed Roles. In Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, 2000.

[32] Berardi, D., Calvanese, C. and De Giacomo, G., Reasoning on UML Class Diagrams using Description Logic Based Systems. In Proceedings of the Workshop on Applications of Description Logics, 2001.

[33] Straeten, R., Mens, T., Simmonds, J. and Jonckers, V., Using Description Logic to Maintain Consistency between UML Models. In Unified Modeling Language, 2003.

[34] Time Ontology in OWL, http://www.w3.org/TR/2006/WD-owl-time-20060927

[35] SPARQL Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query

[36] Weiss, C., Bernstein, B., and Boccuzzo, B., i-MoCo: Mobile Conference Guide - Storing and querying huge amounts of Semantic Web data on the iPhone/iPod Touch. In Billion Triples Challenge, International Semantic Web Conference (ISWC), 2008.

[37] Weiss, C., Karras, P. and Bernstein, A., Hexastore Sextuple Indexing for Semantic Web Data Management. In Proceedings of Very Large Data Bases Conference (VLDB), 2008.

[38] http://jquery.com/

[39] http://mootools.net/

[40] http://skyweb.hu/x/moocanvas/

[41] http://www.draw2d.org/draw2d/

[42] Arvanitis, A., Liagouris, J. and Efentakis, A., TALOS Server-side Data Model (at TALOS wiki).

[43] Silberschatz, A., Korth F., H. and Sudarshan, S., Database System Concepts, The MacGraw-Hill Companies Inc., 2002

[44] Tsigka, E., Pfennigschmidt, S., Arvanitis, A., Liagouris, J., Eyckerman, P. and Dehner, H., D4.1 – Task Computing Framework, 2010 (at TALOS wiki)

APPENDIX I - ToDo XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns="http://www.talos.cti.gr/ToDo"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.talos.cti.gr/ToDo"
        elementFormDefault="qualified">

        <xs:simpleType name="relationshipType">
                <xs:restriction base="xs:string">
                    <xs:enumeration value="SUB"/>
                    <xs:enumeration value="OR"/>
                    <xs:enumeration value="CHOICE"/>
                    <xs:enumeration value="SUBSEQ"/>
                </xs:restriction>
        </xs:simpleType>

        <xs:simpleType name="bindingType">
                <xs:restriction base="xs:string">
                    <xs:enumeration value="CA"/>
                    <xs:enumeration value="USER"/>
                    <xs:enumeration value="APP"/>
                </xs:restriction>
        </xs:simpleType>

        <!-- Input/Output parameter -->
      <xs:simpleType name="pType">
                <xs:restriction base="xs:string">
                    <xs:enumeration value="dateTime"/>
                    <xs:enumeration value="city"/>
                    <xs:enumeration value="POI"/>
                    <xs:enumeration value="location"/>
                    <xs:enumeration value="weather"/>
                    <xs:enumeration value="list"/>
                </xs:restriction>
      </xs:simpleType>
```

```
        <!-- InstantiationType -->
        <xs:complexType name="instantiationType">
           <xs:sequence>
                    <xs:element name="binding" type="bindingType"
minOccurs="1" maxOccurs="1"/>
                    <xs:element name="var" type="xs:string"
minOccurs="1" maxOccurs="1"/>
                </xs:sequence>
     </xs:complexType>

        <!-- Parameter -->
        <xs:complexType name="paramType">
           <xs:sequence>
                    <xs:element name="name" type="xs:string"
minOccurs="1"/>
                    <xs:element name="type" type="pType"
minOccurs="1"/>
                    <xs:element name="description"
type="xs:string"/>
                    <xs:element name="instantiatedBy"
type="instantiationType" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="optional" type="xs:boolean"
use="required"/>
     </xs:complexType>

     <!-- Task Input/Output -->
     <xs:complexType name="ioType">
          <xs:sequence>
                    <xs:element name="param" type="paramType"
minOccurs="0" maxOccurs="unbounded"/>
                </xs:sequence>
     </xs:complexType>

        <!-- Preinformation -->
     <xs:complexType name="preInformationType">
          <xs:sequence>
                    <xs:element name="input" type="ioType"
minOccurs="0"/>
                </xs:sequence>
     </xs:complexType>
```

```
        <!-- Postinformation -->
      <xs:complexType name="postInformationType">
           <xs:sequence>
                    <xs:element name="output" type="ioType"
minOccurs="0"/>
                </xs:sequence>
      </xs:complexType>

        <!-- Sequence -->
      <xs:complexType name="dataflowType">
           <xs:sequence>
                    <xs:element name="sourceParam" type="xs:string"
minOccurs="1" maxOccurs="1"/>
                    <xs:element name="targetParam" type="xs:string"
minOccurs="1" maxOccurs="1"/>
                </xs:sequence>
      </xs:complexType>

        <!-- SubTask -->
      <xs:complexType name="subTaskType">
                <xs:sequence>
                    <xs:element name="member"
type="xs:positiveInteger" minOccurs="1" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="ID" type="xs:positiveInteger"
use="required"/>
      </xs:complexType>

      <!-- OR -->
      <xs:complexType name="orType">
                <xs:sequence>
                    <xs:element name="member"
type="xs:positiveInteger" minOccurs="2" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="ID" type="xs:positiveInteger"
use="required"/>
      </xs:complexType>
```

```
     <!-- Choice -->
      <xs:complexType name="choiceType">
           <xs:sequence>
                    <xs:element name="member"
type="xs:positiveInteger" minOccurs="2" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="ID" type="xs:positiveInteger"
use="required"/>
      </xs:complexType>

        <!-- Group -->
        <xs:complexType name="groupType">
           <xs:sequence>
                    <xs:element name="member"
type="xs:positiveInteger" minOccurs="2" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="ID" type="xs:positiveInteger"
use="required"/>
                <xs:attribute name="type" type="relationshipType"
use="required"/>
      </xs:complexType>

      <!-- Sequence -->
      <xs:complexType name="sequenceType">
           <xs:sequence>
                    <xs:element name="member"
type="xs:positiveInteger" minOccurs="2" maxOccurs="2"/>
                    <xs:element name="dataflow" type="dataflowType"
minOccurs="1" maxOccurs="1"/>
                </xs:sequence>
      </xs:complexType>

        <!-- Chain -->
      <xs:complexType name="chainType">
           <xs:sequence>
                    <xs:element name="sequence" type="sequenceType"
minOccurs="1" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="ID" type="xs:positiveInteger"
use="required"/>
        </xs:complexType>
```

```
      <!-- Task -->
      <xs:complexType name="taskType">
            <xs:sequence>
                  <xs:element name="taskID"
type="xs:positiveInteger"/>
                        <xs:element name="versionID"
type="xs:positiveInteger"/>
                        <xs:element name="model" type="xs:string"/>
                        <xs:element name="name" type="xs:string"/>
                        <xs:element name="description"
type="xs:string" minOccurs="0"/>
                        <xs:element name="language"
type="xs:language"/>
                        <xs:element name="author" type="xs:string"
maxOccurs="unbounded"/>
                        <xs:element name="publisher"
type="xs:string"/>
                        <xs:element name="createdOn"
type="xs:date"/>
                        <xs:element name="realizedBy"
type="xs:anyURI" minOccurs="0" maxOccurs="unbounded"/>
                        <xs:element name="isGroup"
type="xs:boolean"/>
                        <xs:element name="preInformation"
type="preInformationType" minOccurs="0"/>
                        <xs:element name="postInformation"
type="postInformationType" minOccurs="0"/>
                </xs:sequence>
      </xs:complexType>


         <xs:element name="model">
                <xs:complexType>
                        <xs:sequence>
                            <xs:element name="task" type="taskType"
minOccurs="1" maxOccurs="unbounded"/>
                            <xs:element name="group"
type="groupType" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element name="subTaskOf"
type="subTaskType" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element name="chain"
type="chainType" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element name="or" type="orType"
minOccurs="0" maxOccurs="unbounded"/>
```

```
                              <xs:element name="choice"
type="choiceType" minOccurs="0" maxOccurs="unbounded"/>
                          </xs:sequence>
                </xs:complexType>
        </xs:element>

</xs:schema>
```

APPENDIX II – Example in XML

```
<?xml version="1.0"?>

<model xmlns="http://www.talos.cti.gr/ToDo"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:todo="http://www.talos.cti.gr/ToDo"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.talos.cti.gr/ToDo TODOXMLSchema.xsd">

      <todo:task>
                <todo:taskID>1</todo:taskID>
                <todo:versionID>1</todo:versionID>
                <todo:model>Travel Guide</todo:model>
                <todo:name>Task</todo:name>
                <todo:description></todo:description>
            <todo:language>En</todo:language>
            <todo:author>John Liagouris</todo:author>
            <todo:publisher>IMIS</todo:publisher>
            <todo:createdOn>2009-07-17</todo:createdOn>

     <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
                <todo:isGroup>false</todo:isGroup>
     </todo:task>
        <todo:task>
                <todo:taskID>2</todo:taskID>
                <todo:versionID>1</todo:versionID>
                <todo:model>Travel Guide</todo:model>
                <todo:name>Get a City Overview</todo:name>
                <todo:description></todo:description>
            <todo:language>En</todo:language>
            <todo:author>John Liagouris</todo:author>
```

```
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>USER</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>3</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Travel to/in the City</todo:name>
            <todo:description></todo:description>
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>USER</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
```

```
                    </todo:param>
                </todo:input>
            </todo:preInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>4</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Sleep</todo:name>
            <todo:description></todo:description>
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>USER</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>5</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Eat and Drink</todo:name>
            <todo:description></todo:description>
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>
```

```
<todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
          <todo:isGroup>false</todo:isGroup>
          <todo:preInformation>
              <todo:input>
                  <todo:param optional="false">
                      <todo:name>City</todo:name>
                      <todo:type>city</todo:type>
                      <todo:description></todo:description>
                      <todo:instantiatedBy>
                          <todo:binding>USER</todo:binding>
                          <todo:var>city</todo:var>
                      </todo:instantiatedBy>
                  </todo:param>
              </todo:input>
          </todo:preInformation>
</todo:task>
  <todo:task>
          <todo:taskID>6</todo:taskID>
          <todo:versionID>1</todo:versionID>
          <todo:model>Travel Guide</todo:model>
          <todo:name>Sightseeing</todo:name>
          <todo:description></todo:description>
     <todo:language>En</todo:language>
     <todo:author>John Liagouris</todo:author>
     <todo:publisher>IMIS</todo:publisher>
     <todo:createdOn>2009-07-17</todo:createdOn>

<todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
          <todo:isGroup>false</todo:isGroup>
          <todo:preInformation>
              <todo:input>
                  <todo:param optional="false">
                      <todo:name>City</todo:name>
                      <todo:type>city</todo:type>
                      <todo:description></todo:description>
                      <todo:instantiatedBy>
                          <todo:binding>USER</todo:binding>
                          <todo:var>city</todo:var>
                      </todo:instantiatedBy>
                  </todo:param>
              </todo:input>
```

```
            </todo:preInformation>
      </todo:task>
        <todo:task>
            <todo:taskID>7</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Shopping</todo:name>
            <todo:description></todo:description>
          <todo:language>En</todo:language>
          <todo:author>John Liagouris</todo:author>
          <todo:publisher>IMIS</todo:publisher>
          <todo:createdOn>2009-07-17</todo:createdOn>

      <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>USER</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
      </todo:task>
        <todo:task>
            <todo:taskID>8</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Entertainment</todo:name>
            <todo:description></todo:description>
          <todo:language>En</todo:language>
          <todo:author>John Liagouris</todo:author>
          <todo:publisher>IMIS</todo:publisher>
          <todo:createdOn>2009-07-17</todo:createdOn>

      <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
```

```
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>USER</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>9</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Find a Hotel</todo:name>
            <todo:description></todo:description>
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
            <todo:postInformation>
```

```
                    <todo:output>
                        <todo:param optional="false">
                            <todo:name>selected_POI</todo:name>
                            <todo:type>list</todo:type>
                            <todo:description></todo:description>
                            <todo:instantiatedBy>
                                <todo:binding>APP</todo:binding>
                                <todo:var>POI_list</todo:var>
                            </todo:instantiatedBy>
                        </todo:param>
                    </todo:output>
                </todo:postInformation>
    </todo:task>
       <todo:task>
                <todo:taskID>10</todo:taskID>
                <todo:versionID>1</todo:versionID>
                <todo:model>Travel Guide</todo:model>
                <todo:name>Find a Hostel</todo:name>
                <todo:description></todo:description>
          <todo:language>En</todo:language>
          <todo:author>John Liagouris</todo:author>
          <todo:publisher>IMIS</todo:publisher>
          <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
                <todo:isGroup>false</todo:isGroup>
                <todo:preInformation>
                    <todo:input>
                        <todo:param optional="false">
                            <todo:name>City</todo:name>
                            <todo:type>city</todo:type>
                            <todo:description></todo:description>
                            <todo:instantiatedBy>
                                <todo:binding>APP</todo:binding>
                                <todo:var>city</todo:var>
                            </todo:instantiatedBy>
                        </todo:param>
                    </todo:input>
                </todo:preInformation>
                <todo:postInformation>
                    <todo:output>
```

```
                    <todo:param optional="false">
                        <todo:name>selected_POI</todo:name>
                        <todo:type>list</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>POI_list</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:output>
            </todo:postInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>11</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Find an Appartment</todo:name>
            <todo:description></todo:description>
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
            <todo:postInformation>
                <todo:output>
                    <todo:param optional="false">
```

```
                        <todo:name>selected_POI</todo:name>
                        <todo:type>list</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>POI_list</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:output>
            </todo:postInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>12</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Find a Camping</todo:name>
            <todo:description></todo:description>
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
            <todo:postInformation>
                <todo:output>
                    <todo:param optional="false">
                        <todo:name>selected_POI</todo:name>
```

```
                        <todo:type>list</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>POI_list</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:output>
            </todo:postInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>13</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Make a Booking</todo:name>
            <todo:description></todo:description>
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>POI</todo:name>
                        <todo:type>POI</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>POI</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>14</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
```

```
            <todo:name>Find a Restaurant</todo:name>
            <todo:description></todo:description>
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
            <todo:postInformation>
                <todo:output>
                    <todo:param optional="false">
                        <todo:name>selected_POI</todo:name>
                        <todo:type>list</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>POI_list</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:output>
            </todo:postInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>15</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Find a Snack-bar</todo:name>
```

```
            <todo:description></todo:description>
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
            <todo:postInformation>
                <todo:output>
                    <todo:param optional="false">
                        <todo:name>selected_POI</todo:name>
                        <todo:type>list</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>POI_list</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:output>
            </todo:postInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>16</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Find a Cafe</todo:name>
            <todo:description></todo:description>
```

```
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
            <todo:postInformation>
                <todo:output>
                    <todo:param optional="false">
                        <todo:name>selected_POI</todo:name>
                        <todo:type>list</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>POI_list</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:output>
            </todo:postInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>17</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Find a Bar</todo:name>
            <todo:description></todo:description>
        <todo:language>En</todo:language>
```

```
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
            <todo:postInformation>
                <todo:output>
                    <todo:param optional="false">
                        <todo:name>selected_POI</todo:name>
                        <todo:type>list</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>POI_list</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:output>
            </todo:postInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>18</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Find a Club</todo:name>
            <todo:description></todo:description>
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
```

```
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>City</todo:name>
                        <todo:type>city</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>city</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
            <todo:postInformation>
                <todo:output>
                    <todo:param optional="false">
                        <todo:name>selected_POI</todo:name>
                        <todo:type>list</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>POI_list</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:output>
            </todo:postInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>19</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Get Operating Days/Hours</todo:name>
            <todo:description></todo:description>
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
```

```
            <todo:createdOn>2009-07-17</todo:createdOn>

      <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
                <todo:isGroup>false</todo:isGroup>
                <todo:preInformation>
                    <todo:input>
                        <todo:param optional="false">
                            <todo:name>POI</todo:name>
                            <todo:type>POI</todo:type>
                            <todo:description></todo:description>
                            <todo:instantiatedBy>
                                <todo:binding>APP</todo:binding>
                                <todo:var>POI</todo:var>
                            </todo:instantiatedBy>
                        </todo:param>
                    </todo:input>
                </todo:preInformation>
      </todo:task>
        <todo:task>
                <todo:taskID>20</todo:taskID>
                <todo:versionID>1</todo:versionID>
                <todo:model>Travel Guide</todo:model>
                <todo:name>Find Out Prices</todo:name>
                <todo:description></todo:description>
           <todo:language>En</todo:language>
           <todo:author>John Liagouris</todo:author>
           <todo:publisher>IMIS</todo:publisher>
           <todo:createdOn>2009-07-17</todo:createdOn>

      <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
                <todo:isGroup>false</todo:isGroup>
                <todo:preInformation>
                    <todo:input>
                        <todo:param optional="false">
                            <todo:name>POI</todo:name>
                            <todo:type>POI</todo:type>
                            <todo:description></todo:description>
                            <todo:instantiatedBy>
                                <todo:binding>APP</todo:binding>
                                <todo:var>POI</todo:var>
                            </todo:instantiatedBy>
                        </todo:param>
```

```
                </todo:input>
            </todo:preInformation>
    </todo:task>
      <todo:task>
            <todo:taskID>21</todo:taskID>
            <todo:versionID>1</todo:versionID>
            <todo:model>Travel Guide</todo:model>
            <todo:name>Make a Reservation</todo:name>
            <todo:description></todo:description>
        <todo:language>En</todo:language>
        <todo:author>John Liagouris</todo:author>
        <todo:publisher>IMIS</todo:publisher>
        <todo:createdOn>2009-07-17</todo:createdOn>

    <todo:realizedBy>http://www.talos.cti.gr</todo:realizedBy>
            <todo:isGroup>false</todo:isGroup>
            <todo:preInformation>
                <todo:input>
                    <todo:param optional="false">
                        <todo:name>POI</todo:name>
                        <todo:type>POI</todo:type>
                        <todo:description></todo:description>
                        <todo:instantiatedBy>
                            <todo:binding>APP</todo:binding>
                            <todo:var>POI</todo:var>
                        </todo:instantiatedBy>
                    </todo:param>
                </todo:input>
            </todo:preInformation>
    </todo:task>
      <todo:group ID="22" type="OR">
          <todo:member>9</todo:member>
          <todo:member>10</todo:member>
          <todo:member>11</todo:member>
          <todo:member>12</todo:member>
      </todo:group>
      <todo:group ID="23" type="OR">
          <todo:member>14</todo:member>
          <todo:member>15</todo:member>
          <todo:member>16</todo:member>
          <todo:member>17</todo:member>
```

```
            <todo:member>18</todo:member>
        </todo:group>
        <todo:group ID="24" type="OR">
            <todo:member>19</todo:member>
            <todo:member>20</todo:member>
            <todo:member>21</todo:member>
        </todo:group>
        <todo:subTaskOf ID="1">
            <todo:member>2</todo:member>
            <todo:member>3</todo:member>
            <todo:member>4</todo:member>
            <todo:member>5</todo:member>
            <todo:member>6</todo:member>
            <todo:member>7</todo:member>
            <todo:member>8</todo:member>
        </todo:subTaskOf>
        <todo:subTaskOf ID="4">
            <todo:member>22</todo:member>
        </todo:subTaskOf>
        <todo:subTaskOf ID="8">
            <todo:member>23</todo:member>
        </todo:subTaskOf>
        <todo:chain ID="1">
            <todo:sequence>
                <todo:member>9</todo:member>
                <todo:member>13</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
        <todo:chain ID="2">
            <todo:sequence>
                <todo:member>10</todo:member>
                <todo:member>13</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
```

```
            </todo:sequence>
        </todo:chain>
            <todo:chain ID="3">
            <todo:sequence>
                <todo:member>11</todo:member>
                <todo:member>13</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
        <todo:chain ID="4">
            <todo:sequence>
                <todo:member>12</todo:member>
                <todo:member>13</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
            <todo:chain ID="5">
            <todo:sequence>
                <todo:member>14</todo:member>
                <todo:member>19</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
        <todo:chain ID="6">
            <todo:sequence>
                <todo:member>15</todo:member>
                <todo:member>19</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
```

```
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>

        <todo:chain ID="7">
            <todo:sequence>
                <todo:member>16</todo:member>
                <todo:member>29</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
            <todo:chain ID="8">
            <todo:sequence>
                <todo:member>17</todo:member>
                <todo:member>19</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
        <todo:chain ID="9">
            <todo:sequence>
                <todo:member>18</todo:member>
                <todo:member>19</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
            <todo:chain ID="10">
            <todo:sequence>
                <todo:member>14</todo:member>
                <todo:member>20</todo:member>
```

```
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
        <todo:chain ID="11">
            <todo:sequence>
                <todo:member>15</todo:member>
                <todo:member>20</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
           <todo:chain ID="12">
            <todo:sequence>
                <todo:member>16</todo:member>
                <todo:member>20</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
           <todo:chain ID="13">
            <todo:sequence>
                <todo:member>17</todo:member>
                <todo:member>20</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
           <todo:chain ID="14">
            <todo:sequence>
```

```
                <todo:member>18</todo:member>
                <todo:member>20</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
           <todo:chain ID="15">
            <todo:sequence>
                <todo:member>14</todo:member>
                <todo:member>21</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
        <todo:chain ID="16">
            <todo:sequence>
                <todo:member>15</todo:member>
                <todo:member>21</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
           <todo:chain ID="17">
            <todo:sequence>
                <todo:member>16</todo:member>
                <todo:member>21</todo:member>
                <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                    <todo:targetParam>POI</todo:targetParam>
                </todo:dataflow>
            </todo:sequence>
        </todo:chain>
```

```
            <todo:chain ID="18">
             <todo:sequence>
                 <todo:member>17</todo:member>
                 <todo:member>21</todo:member>
                 <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                     <todo:targetParam>POI</todo:targetParam>
                 </todo:dataflow>
             </todo:sequence>
         </todo:chain>
            <todo:chain ID="19">
             <todo:sequence>
                 <todo:member>18</todo:member>
                 <todo:member>21</todo:member>
                 <todo:dataflow>

<todo:sourceParam>selected_POI</todo:sourceParam>
                     <todo:targetParam>POI</todo:targetParam>
                 </todo:dataflow>
             </todo:sequence>
         </todo:chain>
</model>
```