SmartProducts

D.2.1.3: Final Version of the Conceptual Framework

WP 2 – Semantic Modelling and Management of Proactive Knowledge

Deliverable Lead: OU

Contributing Partners:
OU, CRF, VTT

Delivery Date: 01.02.2011

Dissemination Level: Public

Version 1.0

Deliverable Lead		
Name	**Organisation**	**e-mail**
Andriy Nikolov	OU	a.nikolov@open.ac.uk

Contributors		
Name	**Organisation**	**e-mail**
Mathieu d'Aquin	OU	m.daquin@open.ac.uk
Marina Giordanino	CRF	marina.giordanino@crf.it
Elena Vildjiounaite	VTT	Elena.Vildjiounaite@vtt.fi

Internal Reviewer		
Name	**Organisation**	**e-mail**
Victoria Uren	USFD	v.uren@dcs.shef.ac.uk
Oliver Kasten	SAP	oliver.kasten@sap.com

Disclaimer

Table of Contents

List of Figures

List of Tables

Executive Summary

This deliverable presents the second version of the SmartProducts conceptual framework implemented as a set of ontologies covering various aspects of smart products functionalities. The deliverable particularly focuses on several parts of the conceptual framework which have been introduced or revised since the release of the initial framework described in [D2.1.2]:

- One major addition to the initial set of knowledge models is the new *task model* aimed at supporting proactive behaviour of smart products and collaborative interaction of products within networks. The interaction is based on explicit modelling of tasks, methods handling them, and capabilities of smart products.
- Another direction of work presented in the deliverable is the development of a generic model to support the *access control* mechanism to semantic data expressed using OWL ontologies.
- The user model and the product model were extended to support the user interaction mechanisms developed in WP5.
- Other models (in particular, the location model and domain models) have been extended and revised to support the implementation work on the use case scenarios.

1 Introduction

The goal of this deliverable is to provide an overview of the extended set of knowledge models developed to support the functionalities of smart products. In [D2.1.2] we presented the initial version of the set of ontologies. In this initial set of ontologies we focused on defining the structure to represent data relevant to the SmartProducts use cases. In this deliverable, we describe the changes which have been made to this set of ontologies.

The main focus of our work on ontology development during the second year of the project was development of the knowledge models to support the proactive behaviour of smart products within ambiances. Ambiance represents a network of smart products which are able to interact with each other: e.g., a smart kitchen or a car containing several smart products represent such ambiances. Cooperation between smart products in an ambiance is based on using their capabilities to contribute to on-going tasks. Thus, the *task model* represents the core component in achieving proactive behaviour on the part of products. Based on that, smart products are defined in terms of their capabilities, and the extended product model provides formal structure for these descriptions of smart products.

In addition to this, initial models described in [D2.1.2] were revised in the course of implementation work and as a result of the evaluation stage. These revisions include, among others:

- A novel ontology for representing access rights for semantic data.
- Extensions to the user model aimed at describing user interface options. These extensions were motivated by the initial evaluation of WP5 technologies.
- Revised location ontology adapted to the needs of implementation scenarios.
- Revised model of food-related concepts developed to support the WP8 use case scenario.
- Extensions made to other models to support implementation work.

This deliverable only describes in detail the models which have been substantially changed with respect to [D2.1.2]. Other models are not described. In particular, this concerns:

- the workflow model, for which the representation based on the standard XPDL language has been chosen.
- the product life-cycle model, as the model described in [D2.1.2] has not been changed.

The rest of the deliverable is structured as follows. Section 2 provides an overview of the revised parts of the conceptual framework and the resulting set of ontologies formally

describing the conceptualisation. Section 3 describes the task model which was designed to support the proactive behaviour of smart products. Section 4 outlines the changes made to the context model, in particular, the description of the revised location model. Section 5 includes a detailed description of the new access rights model for semantic data which is aimed at complementing the techniques developed in WP4. Section 6 describes the modelling support of the user interaction mechanism. Sections 7-9 describe the domain-specific extensions made to support the use case scenarios. Finally, section 10 provides an outlook of the deliverable.

2 Overview

In [D2.1.2], the initial set of ontological models constituting the SmartProducts conceptual framework was proposed.

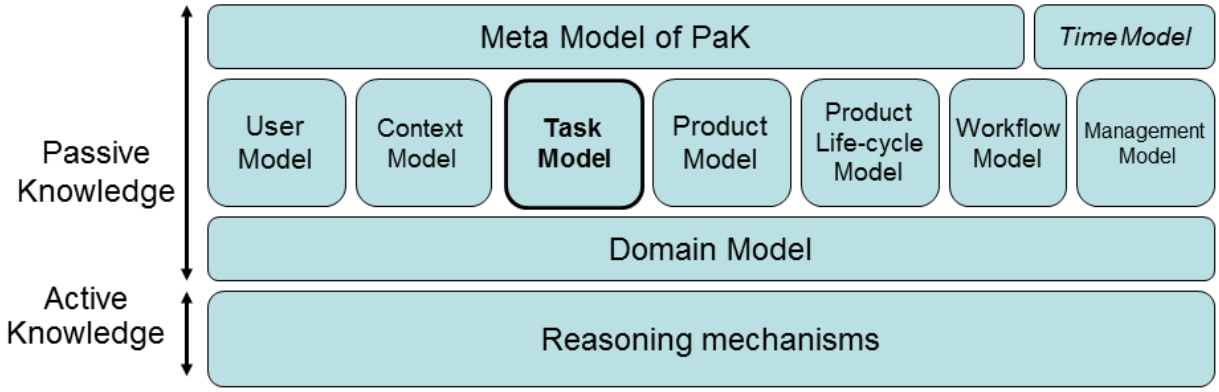

Figure 1: Overview of the main models of the conceptual framework.

Figure 1 displays the components knowledge components constituting the proactive knowledge. Two types of these components are distinguished:

- *Passive knowledge*: ontologies which provide the data structure to describe different concepts relevant for smart products.
- *Active knowledge*: procedural knowledge which guides the product in making decisions making and supports execution of activities. Active knowledge includes problem-solving methods which use reasoning to perform domain-specific tasks and task control mechanisms which trigger activities based on available information.

The set of models constituting passive knowledge is aimed at providing the data structure covering various aspects of required knowledge. In the course of implementation work and based on evaluation results, a number of revisions were made to the initial set of models. This deliverable summarises the changes made to this initial set of models in the second stage of the project. In section 2.1 we provide a brief overview of these changes. Then, section 2.2 discusses how the updated conceptual models are integrated into an interlinked set of ontologies.

2.1 Revisions made to the ontological models

2.1.1 Task model

While all other passive knowledge components have been envisaged in [D2.1.2], the task model constitutes the major extension to the initial set of models. When designing the initial conceptual framework, the bottom-up approach was taken: from the analysis of related use cases and their requirements a set of modelling requirements was produced, and, based on it,

initial conceptual models were proposed. However, this approach did not sufficiently capture the generic concepts needed to support proactive behaviour of smart products and collaborative task solving.

In the initial version of the ontologies modelling of process knowledge only involved the use of standard workflow representation languages (XPDL). This approach has several limitations, namely:

- *Rigidity of process descriptions*. All activities constituting the workflow are prescribed in the workflow definition. In the use cases involving smart products, exact capabilities of all participating products may not be known in advance (e.g., different versions of the same product can execute the same part of the workflow in different ways), which may require modifying the whole workflow when one product in the network is changed.
- *Centralised nature of workflow execution.* When executing a workflow according to the description, there is a workflow management system (the Interaction Manager component of one of the products) which plays the role of the dispatcher. While this approach is justified in many scenarios where a precise sequence of activities is known in advance (e.g., cooking a dish), it limits the possibility to exploit advanced capabilities of different products. A smart product which has to perform a task must possess a certain degree of autonomy to decide how its capabilities can be utilised in the best way.

In order to overcome these limitations, we decided to use the approach based on problem solving methods to guide the interaction between smart products. The knowledge model supporting this type of interaction was separated from the workflow model and constitutes the *task model*.

2.1.2 Management model

Another direction of major changes made to existing models involved the *management model*, in particular, the support for access control mechanisms. The challenge involves supporting access control mechanisms dealing with semantic data. On the one hand, access control mechanisms need to take into account the specific properties of semantic data and to be represented homogeneously with data. On the other hand, they must be sufficiently light-weight to be deployable on devices with limited computational capabilities. An ontology for modelling access rights was proposed to deal with these issues and describe the relations between datasets and agents.

2.1.3 User model

The motivation for extending the user model and related parts of the product model was the need to support the user interaction mechanisms developed in WP5. These interaction mechanisms need to take into account not only the user preferences regarding the interface, but also the capabilities of products included into the ambiance and the state of the context. In order to do this, ontological model describing user interface options has to be integrated with other models, in particular, the product model and the context model.

2.1.4 Domain models

In addition to these major additions to generic knowledge models, use case-specific domain models had to be extended or revised. In particular, these revisions included:

- Major revisions of the domain model for the WP8 use case. Given that the WP8 use case was chosen as the primary test scenario in the second year of the project, the ontology of food and recipes has been substantially extended and partially revised.
- Incremental changes made to other domain models in order to support corresponding use case scenarios.

2.2 Integration of models

2.2.1 Organisation of ontological modules

The SmartProducts conceptual framework is implemented as a set of OWL ontologies. The organisation of the set of OWL modules (Figure 2) does not strictly follow the distribution of models given in Figure 1. Because of the tight coupling between the generic models, separating them into different files would lead to complications in the usage and maintenance of ontologies without clear benefits: when adding or modifying ontological resources and axioms, the user would need to take into account many possible implications to different modules, and to synchronise interdependent definitions in several modules.

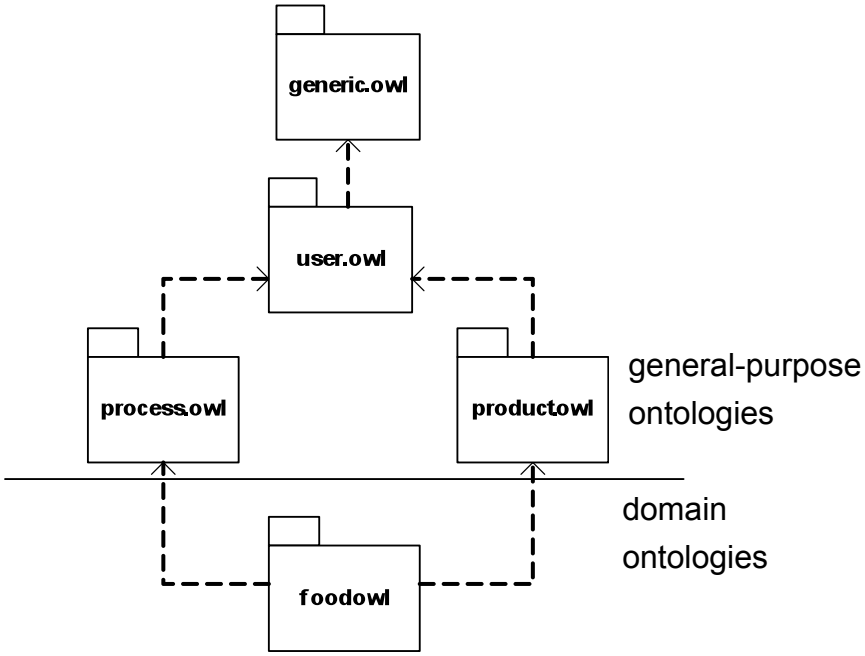

Figure 2: Organisation of ontological modules in the SmartProducts conceptual framework. Arrows reflect import relations.

The resulting set of ontologies is available online[1] and contains the following modules:

1. *General-purpose ontologies*. These ontologies model common smart products-related concepts and are independent on the application domain.
    a. generic.owl – contains high-level abstract concepts and incorporates auxiliary models (such as time, location, and context).
    b. user.owl – contains the description of the user profile.
    c. process.owl – contains the OWL view of the workflow model. This model does not fully reflect the workflow descriptions expressed in XPDL, but only models information needed for workflow selection (unordered list of workflow activities and their requirements).
    d. product.owl – contains the model of devices and products.
2. *Domain ontologies*. These ontologies extend the general-purpose ontologies and describe the specific application domains.

2.2.2 Foundational concepts

The classes in the integrated set of ontologies are organised into a single class hierarchy. The high-level classes in the hierarchy represent abstract categories and do not carry actual properties describing entities.

[1] http://kmi.open.ac.uk/projects/smartproducts/ontologies/

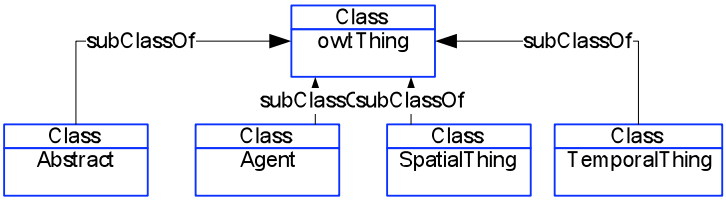

Figure 3: Top-level concepts in the SmartProducts ontology set

This has been done in accordance to a common ontology design pattern[2] which aims at identifying and categorising the most general types of things in the domain of discourse. While there exist several foundational models expressing abstract concepts such as DOLCE[3] or [Kitamura-2006], we considered these foundational models too heavy-weight for the use in smart product networks. Therefore, we used our own hierarchy of top-level concepts which covered the main categories of entities we are dealing with.

In our model, the following four high-level categories are defined (Figure 3):

- *Abstract*. This category combines all entities which cannot be positioned in space-time, such as types of substances, units of measurements, physical properties and their values, etc.
- *Agent*. This category combines all entities which can play the role of agents in some situations.
- *SpatialThing*. This category includes all entities related to space: both objects and events which can be located in space and entities describing the locations (e.g., geographic coordinates).
- *TemporalThing*. In a similar way, this category includes all entities related to time: both the objects which can be positioned in time and entities describing time points and intervals.

These categories are not mutually disjoint: e.g., physical objects usually have both temporal and spatial dimensions, time and location descriptions are abstract entities, and agents can be positioned in space and time.

[2] http://ontologydesignpatterns.org/wiki/Submissions:Types_of_entities

[3] http://www.loa-cnr.it/DOLCE.html

3 Modelling proactivity: task model

In the context of SmartProducts, proactivity is defined as a capability to initiate actions or exhibit goal-driven behaviour without an explicit request or pre-defined schedule [WN1.5]. Thus, the primary requirement of smart products is *goal-awareness* or *task-awareness*: information about the goal of the user is necessary to decide which products' activities can help achieving it. Modelling the concepts supporting this constitutes the main focus of the *task model* described in this section. The rest of the section is organised as follows. Section 3.1 gives an informal introduction to the chosen approach (extension of the problem-solving methods paradigm). Section 3.2 provides a brief overview of related research dealing with multi-agent systems and knowledge engineering. Sections 3.3 - 3.6 describe different parts of the proposed model, and section 3.8 provides a short example illustrating the model. Finally, section

3.1 Overview

The task ontology defines the core concepts supporting the task-based interaction between products, in particular, the notions of *tasks*, *methods*, *capabilities*, and *ambiances*. Tasks describe the activities to be performed in terms of their goals, inputs and outputs. Methods, in turn, represent reasoning mechanisms which either solve tasks directly or decompose tasks into subtasks. For example, the task of choosing a menu for a specific meal can be achieved by a method which implements a parametric design procedure [D2.2.1] and considers user preferences, health requirements, and ingredient availability as design constraints. Methods are associated with smart products, which are described in terms of their capabilities. Capabilities describe the product's ability either to solve a particular task or to contribute to task solving with additional information. Tasks are published within networks of products (*ambiances*) and are picked by products with appropriate capabilities. In this way, task solving is performed depending on the capabilities of products available within an ambiance. In this way, integration of devices with different level of functionalities can be achieved in a flexible way: new functionalities can be utilized in existing networks without the need to update the knowledge bases of existing products.

This paradigm provides a central point of integration for different aspects of smart products: interaction, workflow execution, and context-awareness. Traditional workflow execution mechanism based on well-defined workflow models (XPDL) is integrated into the smart products architecture as a complementary component rather than as an alternative: for the tasks where a detailed and stable description is available, workflow execution engine serves as a method which picks up the task and executes it. Information about on-going tasks in the

ambiance constitutes a part of context information. In this way it can be aggregated with other types of context (raw sensor data or situational context) and used to trigger new activities.

3.2 Related Work

When considering knowledge models supporting interaction of different autonomous agents and collaborative problem solving, three modelling aspects are particularly important:

- *Process composition*. The procedure of achieving a goal in a network of smart products may include several steps and involve collaboration of different participants. Each step, in turn, can be performed in different ways and include sub-steps. Thus, presenting process components at different levels of abstraction is necessary to support interaction between smart products.
- *Agent behaviour*. Capabilities to exhibit proactive behaviour to achieve a goal as well as respond reactively to the changes in the environment are necessary for smart products. Thus, formal modelling support of different behaviour types represents a relevant issue for our conceptual framework.
- *Declarative descriptions of devices and agents*. In order to make a link from the procedural components of the process decomposition to the actual actors (devices and software agents) performing the processes, ontological models of actors and their capabilities are needed.

Two relevant research communities which specifically focus on knowledge modelling covering these issues are:

1. *Distributed AI* and *multi-agent systems* specifically targeted the problems of collaboration between distributed agents possessing reasoning capabilities. Several knowledge modelling approaches were proposed in the community and several important aspects were analysed. In particular, the analysis of behaviour types discussing the notions of proactivity and reactivity was conducted in the agent-based research community.
2. *Knowledge engineering and management* community studied the problems of formalising problem-solving knowledge and organising it into reusable components. The research on *problem-solving methods* in particular focused on the issues concerning automated task solving using a composition of methods. The most relevant contribution are the ontologies developed in the community. Our knowledge model in many respects follows this line of research.

3.2.1 Distributed AI and multi-agent systems

Approaches to formal specification of distributed reasoning systems were studied in the distributed AI community since 1970s, and they targeted all three main relevant aspects listed above. In particular, the generic patterns of agent behaviour including *proactivity* (or *pro-activeness*) and *reactivity* have been formulated. At the high level, the weak notion of agency as formulated in [Wooldridge-1995] includes the properties of autonomy (which assumes a degree of control over one's actions), social ability (capability to interact with other agents and possibly humans), reactivity, and pro-activeness. The reactivity property was formulated as the capability of agents to "perceive their environment … and respond in a timely fashion to the changes that occur in it". The pro-activeness property assumed that "agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative". These properties are not mutually exclusive, and the behaviour of an actual agent includes the degree of both reactive (or receiving) and proactive (or discovering) behaviour [Botti-1999].

It is clear that intended capabilities of smart products conform to these properties of agency. Analysis of the different variants of pro-activeness and reactiveness was conducted in [Jonker-1997] where a logical verification framework was proposed for validating these properties. In particular, pro-activeness and reactiveness were considered as attitudes of an agent to observation, communication (information provision and requesting), and reasoning, and a logical framework was proposed to verify which combinations of behaviours of different agents could lead to a successful completion of task (e.g., that at least one agent needs to be proactive with respect to communication in order to support information interchange).

In order to provide common platforms for implementing multi-agent systems, a number of agent architectures have been proposed. Two major groups of approaches formed in the community [Wooldridge-1995]:

- *Deliberative architectures* based on some symbolic model of the world.
- *Reactive architectures* which do not include any kind of central symbolic world model and do not use complex symbolic reasoning. Implemented reactive architectures instead implement various non-symbolic AI techniques such as activation spreading networks [Maes-1991] or situated automata [Kaelbling-1991].

The approach taken in SmartProducts is derived from the first paradigm. Because of the need to operate with heterogeneous information of different types (such as context data, user profile, and domain knowledge) some of which is symbolic by its nature, the need to integrate this information using a common representation framework is a necessity.

Proposed knowledge models for deliberative agent architectures (such as GRATE* [Jennings-1993] or IRMA [Bratman-1988]) include explicit representation of an agent's beliefs (knowledge about the world), desires (goals), and intensions (plans for achieving a goal). One formal specification framework, DESIRE [Brazier-1995], [Brazier-2002] includes a methodology for designing multi-agent systems. This framework includes specification of the following five types of knowledge:

1. *Task decomposition*: a task hierarchy together with specification of input and output signatures.
2. Agent *task control* knowledge, which guides activations of agents and subtasks.
3. *Information links* between agents, which describe information flow.
4. *Task-knowledge allocation* specifying the domain knowledge structures required by each task.
5. *Task allocation*: distribution of tasks between agents.

Other proposed methodologies for multi-agent systems such as ADELFE [Bergenti-2004], Gaia [Wooldridge-2000], and PASSI [Cossentino-2004] also allow modelling these types of knowledge with different degree of detail.

All these types of knowledge are relevant when we consider networks of smart products and must be covered by the conceptual framework. However, when considering smart products, several specific features of the scenario can be outlined. While task decomposition and task-knowledge allocation can be pre-defined by a developer, the other types of knowledge need to be generic in order to support the openness of smart product environments:

- Information links prescribing all information interchanges between agents cannot be defined in advance. This is necessary to ensure that smart product networks remain open systems able to incorporate new products with advanced functionalities and new kinds of tasks. Thus, descriptions of information links should be generic and independent of a specific product or ambiance type.
- Task allocation should be performed at run-time. In this way, advanced functionalities of new products can be utilised without the need to update knowledge possessed by existing products.
- Agent task control knowledge should be independent on the task at hand. In this way, description of tasks can be decoupled from the knowledge about agents (products) included in the network and rely on a common procedure for activating different products and handling tasks.

3.2.2 Knowledge engineering and knowledge-based systems

Within the knowledge engineering community, the one particularly relevant area of research studies problem-solving methods. Problem-solving methods (PSM) represent reasoning components that can tackle specific tasks and can be reused across applications [Fensel-2001]. Both research communities dealing with multi-agent systems and problem-solving methods considered formal specifications of complex reasoning systems, and formal models proposed by both these communities to a large degree cover overlapping domains. The difference is largely in the focus of the analysis. The multi-agent system architectures are more oriented towards the multi-agent perspective: identifying and describing agents performing tasks and allocating the tasks between these agents. In the problem-solving methods research, the focus is largely on the task perspective: analysing the tasks and knowledge required to solve them and organising this knowledge into interacting modules. In case of smart products, taking this view is relevant because of the need to decouple task knowledge from the knowledge about agents.

The main modelling aspect from the problem-solving method research related to the SmartProducts model concerns defining meta-level description of procedural knowledge. Generic model describing tasks and procedures to achieve them is necessary to support proactive behaviour: a product must be aware of an on-going task and be able to make decisions about contributing to it. One of the earlier knowledge modelling approaches which describes a knowledge-based system in terms of tasks and methods was proposed in [Chandrasekaran-1992]. In an informal way, “a task defines *what* needs to be achieved (a declarative functional specification), and a PSM *how* it has to be achieved (an operational specification)” [Benjamins-1996]. Such decoupling between tasks and methods allows specifying alternative ways to solve the same problem by introducing several methods for a single task. Methods can define complete procedures for achieving a task's goal (primitive methods) or introduce several lower-level subtasks, which in turn can be performed by their own methods, thus leading to task-subtask hierarchies (task decomposition methods). Libraries of methods can be used in two ways:

- At design time, when a system designer constructs a knowledge-based system for a specific domain.
- At run time, when a system automatically selects and invokes appropriate methods to perform a task in a specific context.

While initially many approaches primarily focused on the first of these options ([Schreiber-2000], [Orsvarn-1998]), in our case both are relevant. Although the product developer can envisage usage scenarios in which the product can participate and the ways the products can contribute, the actual problem-solving procedure is composed at run-time based on capabilities

of available products. Automated selection and invocation of methods at run time requires them to be described formally in terms of their applicability to tasks and input-output data flow. There are several proposed problem-solving method libraries described in the literature, which use formal descriptions of methods and their assumptions to guide the method selection. Here we list a few of them.

EXPECT [Swartout-1999] uses a description logic-based representation called Loom to represent methods' capabilities. Capability descriptors define the actions of a method and its input and output roles. Role-filler objects are specified using a shared domain ontology. The ontology defines hierarchical relations between concepts and allows the system to match a generic method with a specific goal.

PRODIGY [Fink-2004] introduces statistical method invocation making use of methods' past performance. The PRODIGY system deals with the problem of choosing an appropriate search engine for a given problem. Thus, the time cost of the chosen method is crucial for the task and defines an important teleological assumption: in order to be valuable, the goal must not just be achieved but be achieved within a reasonable time frame. The selection mechanism estimates for each method the expected gain, based on both the expected reward for solving the task at hand and the past performance of the method (percentage of failures and time cost), and then selects a method on the basis of these metrics.

TMDA (Task-Method-Domain-Application) [Motta-1999] defines an ontology describing task-method structures and allows the specification of applicability conditions in the form of logical expressions, which must hold for the method. Both problem-solving method descriptors and domain knowledge are expressed using ontologies and complex reasoning can be performed to validate the conditions.

UPML (Unified Problem-solving Method description Language) [Fensel-2003] describes a language and a specific ontology for describing problem-solving method libraries. Special adaptor structures are used to refine the problem-solving method descriptions and match them to tasks. The system allows the flexible refinement of a method's description depending on the task at hand.

Later, the models for problem-solving methods libraries were used as a basis for ontologies describing Semantic Web Services, such as WSMO[4] and OWL-S[5]. Our task model is primarily based on the TMDA modelling framework proposed in [Motta-1999]. Tasks represent generic domain-independent problem specifications which describe the problem in terms of input and output roles. Methods provide abstract descriptions of the problem-solving procedures. The common task control procedure is used to select appropriate methods for the tasks and invoke them. However, the original framework does not cover all required aspects of collaborative task solving and had to be substantially revised and extended. This is discussed in more detail in section 3.3.

Another relevant line of research includes modelling devices and their functionalities. Here, several solutions have been proposed. Because a review of related work for product models was provided in [D2.1.2], here we only mention several relevant ones. Among them, the FIPA ontology[6] provided a model for representing devices and their hardware and software components. This ontology, however, primarily focuses on specific devices which have visual output capabilities while does not provide a detail model of capabilities and functions. The CoDAMoS project [Preuveneers-2004] provided a light-weight ontology for the representation of devices and their context. This ontology has been reused in other research projects using semantic technologies in mobile environments (e.g., [Cadenas-2009]). However this ontology does not cover many required aspects needed for smart products (e.g., detailed descriptions of relations between devices and tasks they can handle). In [Kitamura-2006], a high-level ontological analysis of modelling artefacts and their functionalities is presented. The model proposed by authors includes the function decomposition tree which utilises the relations between a macro-function (task) and the ways of achieving it (method). In this way, it overlaps with our approach based on decoupling the description of the goal from the description of the procedure of achieving it.

[4] http://www.wsmo.org/

[5] http://www.w3.org/Submission/OWL-S/

[6] http://www.fipa.org/specs/fipa00091/PC00091A.html

3.3 Task model structure

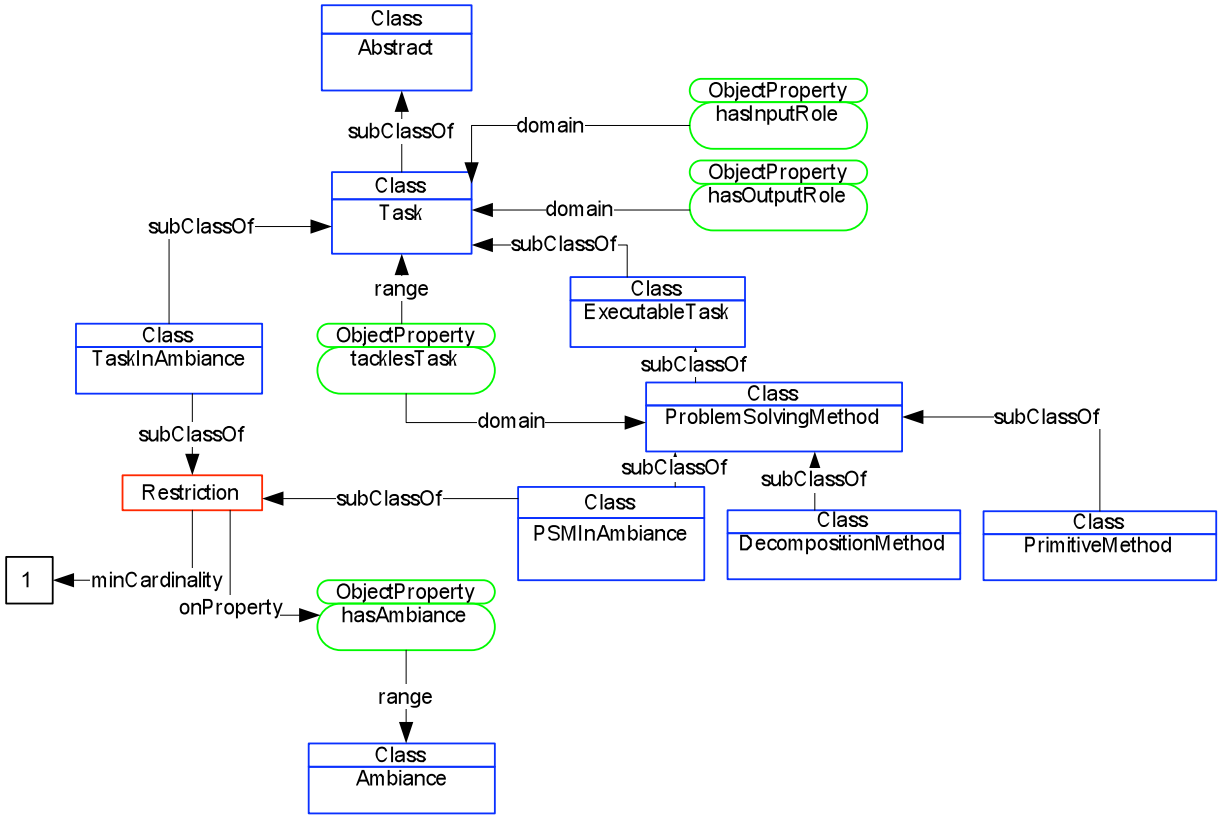

Figure 4: Hierarchy of the task description concepts (fragment)

At the core of the SmartProducts task model is the concept of *Task* (Figure 4). Its definition is borrowed from the original TMDA. Tasks are used to describe problems to be solved in terms of input and output roles which specify the information types needed to solve the task and information which has to be provided as the result. Some tasks are tightly coupled to the procedures of solving them. Such tasks are called executable tasks and are expressed by the class *ExecutableTask* in the ontology.

The ontology defines the following main properties for the class *Task*:

- *hasInputRole*. This property defines input parameters which are needed for solving the task. For example, the input roles for the meal planning task include the set of design prescriptions: various preferences and constraints originated from the user preferences, health profile, available food items, and the context of the meal.

- *hasOutputRole*. The output roles specify the type of information which has to be produced as the result of the task. For example, for the meal planning task the output role is the meal plan: a set of meal courses with assigned values.

A task is solved by a problem solving method: there can be several alternative methods tackling the same class of tasks. The concept *ProblemSolvingMethod* provides a high-level definition of the procedure for achieving the goal. A method tackling a specific task has the same input and output roles as defined in the task specification. Additional input roles can be defined for the method as well, if the method requires additional knowledge to operate.
The class *ProblemSolvingMethod* defines one main property besides those inherited from the class *Task*:

- *tacklesTask*. Establishes a link between the problem solving method and the task it handles.

Some methods represent atomic procedures perceived as "black box" at the knowledge representation level. Such methods are called primitive and are represented by the class *PrimitiveMethod*. Other methods in turn decompose the task into subtasks which can be solved using other methods. Such methods are expressed using the class *DecompositionMethod*. The class *DecompositionMethod* defines a single property to define the list of subtasks:

- *hasGenericSubtasks*. This property links the task decomposition method to the list of subtasks solving this task. In RDF, the list of subtasks is expressed as an instance of the class *rdf:Seq*.

The network of smart products (ambiance) provides an environment in which collaborative task solving takes place. One smart product can participate in different ambiances, and its behaviour in different ambiances may differ: e.g., some methods can be exposed only in a specific ambiance (e.g., kitchen), but not be available in other ambiances (e.g., supermarket). Given this, a link is needed between the tasks and methods on the one side and the corresponding ambiance on the other side. The classes *TaskInAmbiance* and *PSMInAmbiance* are used to specify these relations.

3.4 Defining capabilities

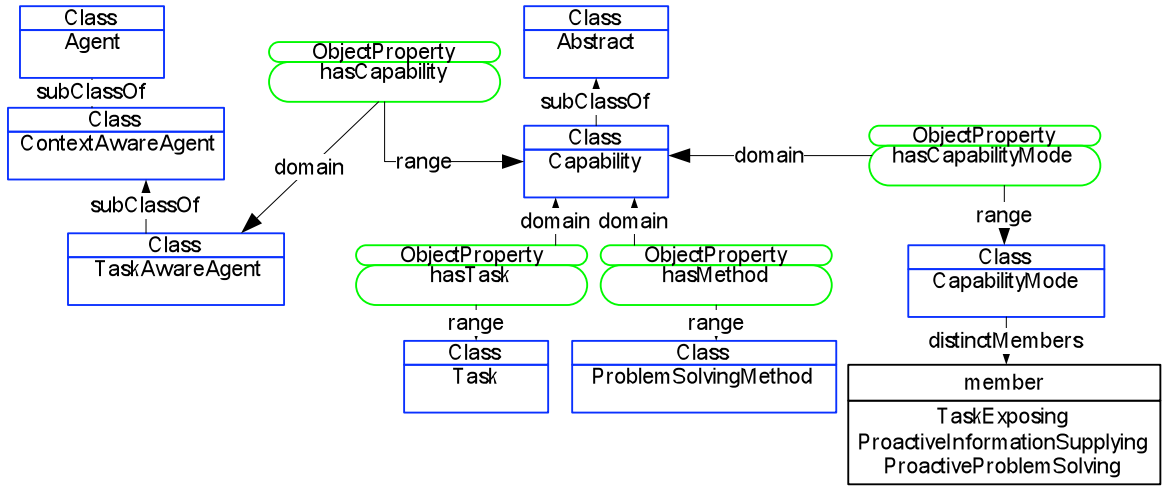

Figure 5: Modeling capabilities of agents

Matching tasks with methods and decisions about how a specific task is going to be solved in a specific situation are based on the available functionalities of agents which participate in an ambiance. The class *Capability* is used to define these functionalities (Figure 5). Unlike in the original research on problem-solving methods, collaborative tasks execution in smart product environments involves two possible ways of contributing to a task:

- *Problem solving*. In this case, the method applied to a task solves it and produces the results defined as the output roles of the task.
- *Information supplying*. In this case, the method does not solve the task but instead provides additional information relevant for solving the task. For instance, for the meal planning task the input roles include design prescriptions: various preferences and restrictions which are necessary to distinguish between valid and invalid meal plan solutions and to sort the solutions. A smart fridge can contribute to the task by providing additional design prescriptions: e.g., knowing that an ingredient is going to expire, it can add a preference for the recipes including this ingredient. In this way, the method does not fill the resulting output roles of the task, but instead modifies its input roles.

Thus, an instance of the class capability defines an object with three properties:

- *hasTask* specifying the task which the agent can contribute to solve.
- *hasMethod* specifying the method which is applied by the agent to contribute to solving the task.
- *hasCapabilityMode* specifying, in which way the agent can contribute to solving the task. The object of this property is an instance of the class *CapabilityMode*.

The class *CapabilityMode* defines an enumeration of capability modes and contains the following pre-defined members:

- *ProactiveProblemSolving*. This value specifies that the method can be applied directly to solve the task.
- *ProactiveInformationSupplying*. This value specifies that the method provides additional information by modifying the input roles of the task.
- *TaskExposing*. This value refers to the generic capability of an agent to expose a task to the ambiance and is normally not associated with any method. Some agents (products) may not possess this capability, for example, because of the ambiance joining policy: e.g., in the supermarket ambiance smart products belonging to customers should not be able to post arbitrary tasks.

3.5 Defining a smart product

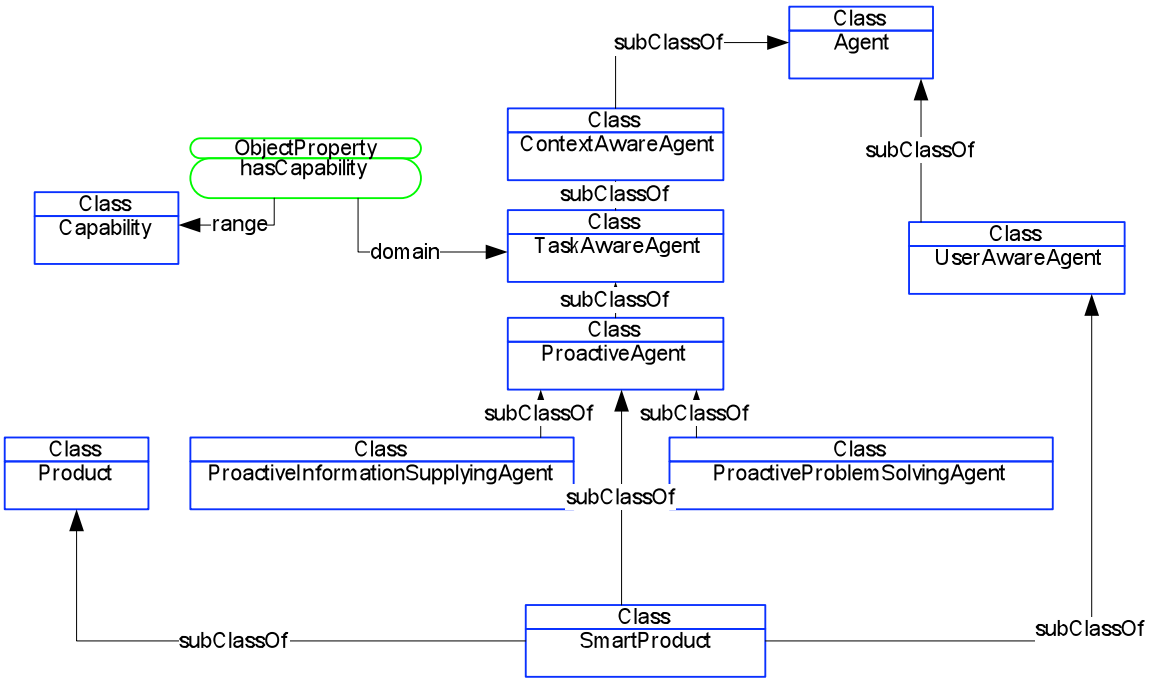

Figure 6: Hierarchy of the superclasses of smart products

The class *Agent* in our ontology defines a top-level class of entities which are able to play the role of agents in some situations. As was mentioned in [D2.5.1], there is a difference with the class *Agent* as discussed in [Welty-2001] where the assumption is that "being an agent" is a role which can be played by some entities in specific circumstances. In our ontology, we distinguish a category of entities which possess the capability of acting as an agent in some situations. The class *Agent* in our ontology describes this category of entities, and thus "being an agent" represents an inherent property of an individual.

A generic category of agents able to perceive their context (at least some of its parts) is defined by the subclass *ContextAwareAgent* of the class *Agent*. Information about on-going tasks in the ambiance constitutes one type of context information. Task-awareness represents a necessary precondition for agents to exhibit proactive behaviour aimed at achieving a certain goal. Thus, the class *TaskAwareAgent* is defined as a subclass of *ContextAwareAgent*. Since the capabilities of agents are defined in relations to tasks, they can only be specified for task-aware agents. The class *TaskAwareAgent* is characterised by the following properties:

- *hasCapability*. This property links the agent to its capabilities.
- *hasLocation*. While this property can be used for any kind of entity to specify its location, task-aware agents can be located in an ambiance. The class *Ambiance* defines an abstraction for the environment in which tasks can be shared.

The next level of functionality for a task-aware agent represents the capability to proactively contribute to tasks. Such agents are defined using the class *ProactiveAgent*. In section 3.4 two possible modes of proactive contribution to tasks were described: *problem solving* and *information supplying*. To reflect this distinction, two subclasses are defined for the class *ProactiveAgent*: *ProactiveProblemSolvingAgent* and *ProactiveInformationSupplyingAgent*. The class *ProactiveAgent* is equivalent to the union of these two subclasses.
Another dimension of the agent functionality involves user-awareness. This category of agents is defined using the class *UserAwareAgent*, which defines one property:

- *hasUser*. This property links the agent to its user.

A smart product represents an agent which is both user-aware and capable of proactive behaviour. In the ontology, the class *SmartProduct* is defined as a subclass of *Product*, *ProactiveAgent*, and *UserAwareAgent* (Figure 6).

3.6 Defining an ambiance

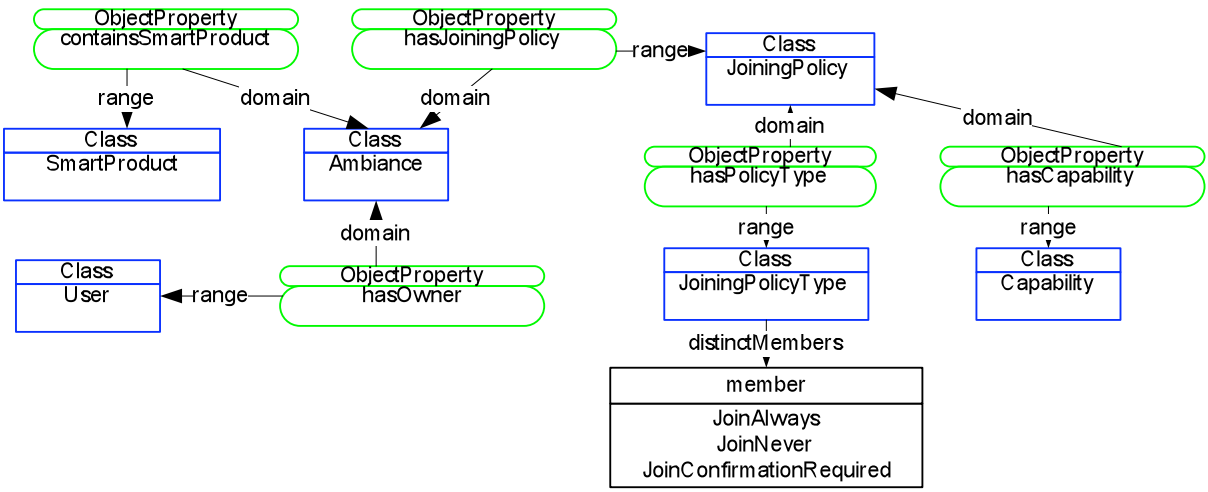

Figure 7: Modelling ambiances and joining policies

The class *Ambiance* serves in the ontology to define a network of smart products (Figure 7). All interaction between products takes place within an ambiance. The ambiance can be created by the user owning a smart product. New products joining the network can be connected to existing ambiance. The description of the ambiance has to be available at any time so that the maintenance of the ambiance does not depend on the product originally used to create it. Thus, this description must be synchronised and replicated. The following properties are defined for the class *Ambiance*:

- *containsSmartProduct*: links the ambiance to a smart product.
- *hasOwner*: links the ambiance with its owner (the user).
- *hasJoiningPolicy*: links the ambiance to a joining policy descriptor. There can be several descriptors defined for the same ambiance.

Ambiance joining policy is necessary in order to regulate the inclusion of other products into the ambiance. For example, the user might not want products belonging to non-trusted users to join her home ambiance. Moreover, the owner of the ambiance might want to restrict certain capabilities of products within it: e.g., within a supermarket ambiance, the owner would not want smart products belonging to customers (mobile devices) to advertise arbitrary tasks to other customers' products.

An ambiance joining policy can be expressed using an instance of the corresponding class *JoiningPolicy*. The policy includes the following properties:

- *applicableTo*: a pointer to the criterion distinguishing the products to which the policy is applicable. A criterion can include, e.g., a single product (by its URI), an ID or a role of the user on behalf of which a product is acting, etc.
- *hasPolicyType*: joining conditions: e.g., one of "never", "confirmationRequired", or "always".
- *hasCapability*: capabilities, which a joining product can use (e.g., only receive tasks but not initiate them).

The generic procedure of solving a task in an ambiance involves three main types of activities:

1. Exposing tasks in the ambiance.
2. Matching tasks with product capabilities.
3. Invoking methods contributing to tasks.

In order to be able to contribute proactively to the tasks, smart products must be able to execute these activities. A task control procedure, thus, has to be deployed on each product as a part of the Reasoner component. To support the current version of the task model, a version of the task control mechanism was implemented[7].

An interesting special case involves the varying granularity level of ambiances: an ambiance as a whole can participate in another ambiance. For example, in the WP9 use case a car represents an ambiance of several products (Blue&me and smart vehicle components). However, in some scenarios the car can be seen by other products as a single smart product: (e.g., in a workshop). In this case, one product in the ambiance serves as a gateway to another ambiance and is responsible for external communication: e.g., it can receive tasks from an external ambiance, advertise them within the internal one and pass the results back.

3.7 Implementation plan: integration into the SmartProducts architecture

The aim of the task model is to provide basic data structures which should support communication between smart products. Within the SmartProducts architecture, semantically structured data are stored by the Proactive Knowledge Base component and processed using the Reasoner module (Figure 8). The communication mechanisms themselves, however, are implemented using other components: most importantly, Communication Middleware, which implements low-level data exchange and remote procedure calling, and Context Manager which is responsible for distributing up-to-date context data to relevant smart products.
Interchange of task-related information also has to be implemented on top of these components.

[7] The current version is implemented in the OCML language (http:// kmi.open.ac.uk/projects/ocml/).

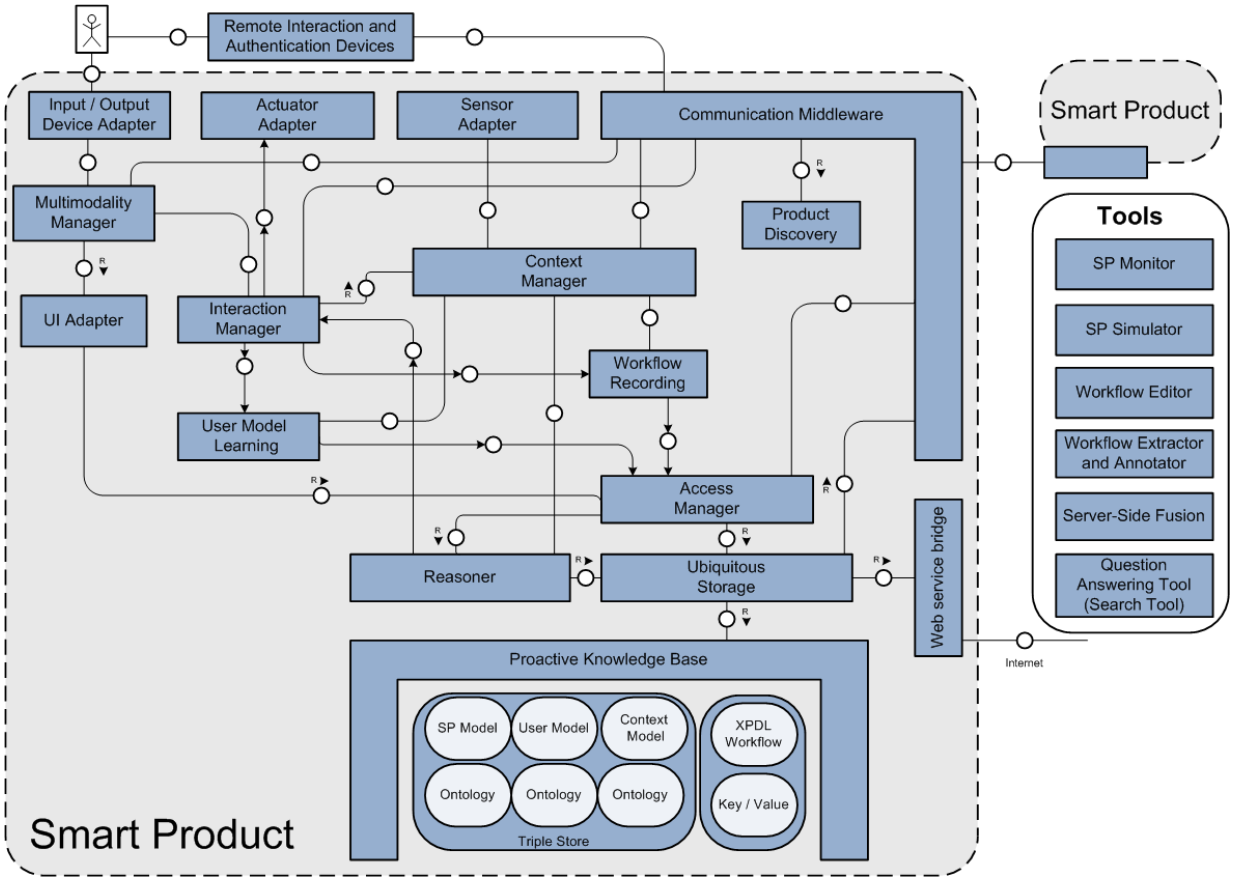

Figure 8: SmartProducts architecture (from [D6.2.2])

Since information about on-going tasks constitutes a part of the product's context, actual exchange of this information is going to be implemented similarly to other types of context processed using Context Manager. The roles of different components in the interchange of task-model structures are provided in

Table 1: Architecture components involved in the task-solving procedure

Component	Description
Proactive Knowledge Base	Proactive Knowledge Base stores the SmartProducts ontologies populated with semantic data. In particular, the product's capabilities are stored permanently.
Reasoner	Reasoner processes both semantic information permanently stored in the Proactive Knowledge Base and runtime information acquired via the Context Manager. The Reasoner component contains implementation of the task control mechanism and problem-solving methods. The Reasoner component is responsible for:

	- Matching the descriptions of on-going tasks with available capabilities. - Making decisions about contributing to an on-going task. - Initiating the distribution of updated task status information (posting the task results and initiating subtasks of an accepted complex task).
Context Manager	Sharing task data between relevant products. RDF descriptions instantiating the SmartProducts ontology structures are serialized using the Context Manager API.
Communication Middleware	Data interchange between products in the ambiance.

3.8 Example: smart kitchen scenario

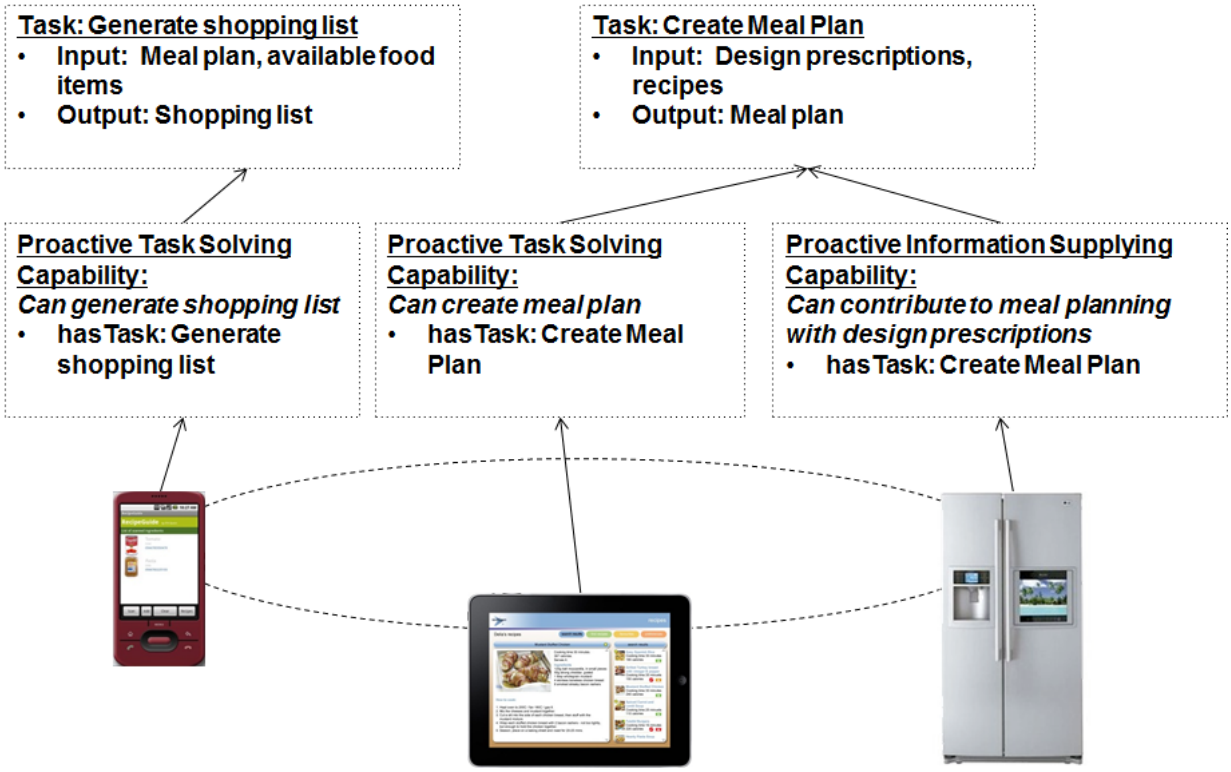

Figure 9: Example kitchen ambiance

An example ambiance related to the smart kitchen use case scenario is shown in Figure 9. The ambiance contains three smart products: the meal planner (e.g., implemented on a tablet PC), the smart fridge, and the shopping assistant (a mobile application). The meal planner allows the user to input some preferences for the planned meal (such as the number of guests, preferred types of dishes, etc.) and to generate a meal plan based on these preferences: in other words, to

propose recipes for each meal course. The task *CreateMealPlan* takes a set of design prescriptions (preferences and constraints) and known recipes as its input and generates a meal plan as its output.

The meal planner itself possesses a method solving this task, which implements a generic parametric design procedure, so it can solve the task autonomously without involving other products (*ProactiveTaskSolvingCapability* for the task *CreateMealPlan*). However, other products in the ambiance can contribute to solving the task by providing additional information. In addition to explicit user preferences, the meal planner can use other relevant information as a source of design prescriptions: e.g., user health profile and information about available ingredients. While some of this information is likely to be stored internally (e.g., health profile), other information can only be available from other products (e.g., available food products). This information can be obtained by sending an explicit request to other products. However, this reactive way of handling a task requires that the meal planner is aware of all relevant information to be queried. Instead, a smart fridge can proactively decide on whether it can contribute to the task and how it can be done. For instance, a "less smart" fridge can generate a list of additional preferences based on the food items it stores, so that the recipes which contain available food items are given preference. A "smarter" fridge can in addition use information about the expiry dates of products: a food item which is soon about to expire should be consumed first. In order to support the proactive way of contributing to the task, the smart fridge needs to have a *ProactiveInformationSupplyingCapability* with respect to the task *CreateMealPlan*.

The shopping assistant can use the generated plan to create and store a shopping list. When the user goes to the supermarket, the shopping assistant can join the ambiance of the supermarket and advertise the task *SuggestShoppingOptions* using the shopping list as an input role. Based on this information, the supermarket server can suggest articles to buy using information about available discounts.

4 Context model

The context model covers several aspects of the environment which are considered the relevant context of smart products. Structures described by this model are shared between products by means of the Context Manager component [D6.2.2] and then are integrated with other parts of the knowledge base and processed by the Reasoner module [D2.2.1]. These relevant aspects of the context include:

- *Task context*, which includes information about the current activities in which the product is involved and the perceived goals of the user of the product.
- *Sensor context*, which includes "low-level" context information which is directly perceived by sensors.
- *Situational context,* which represents an abstract description of the situation in which the task is performed[8].
- *Location context*, which describes the position of products, users, and other objects in space.

Modelling of the task context has been addressed by the task model (see section 3). Situational context and sensor context models underwent several minor revisions in the second version of the ontology. The primary motivation was the alignment of the SmartProducts sensor model with the W3C Semantic Sensor Networks ontology[9], which was developed with the participation of SmartProducts partners (OU). The initial location model defined in [D2.1.2], however, has been substantially revised for the second version.

Two main reasons for this revision were:

- The need to integrate different models into a unified ontological framework while following common ontology design patterns.
- The need to simplify the location description structures, in particular, imprecise relative locations (such as "object X is located on top of the object Y").

[8] Unlike the task context, which concerns the actual goal of the activity, situational context refers to the circumstances surrounding the activity. For instance, when the meal planning task aims at "choosing a meal plan for the children's party", then "choosing a meal plan" relates to the task and "children's party" constitutes the situational context.

[9] http://www.w3.org/2005/Incubator/ssn/wiki/Main_Page

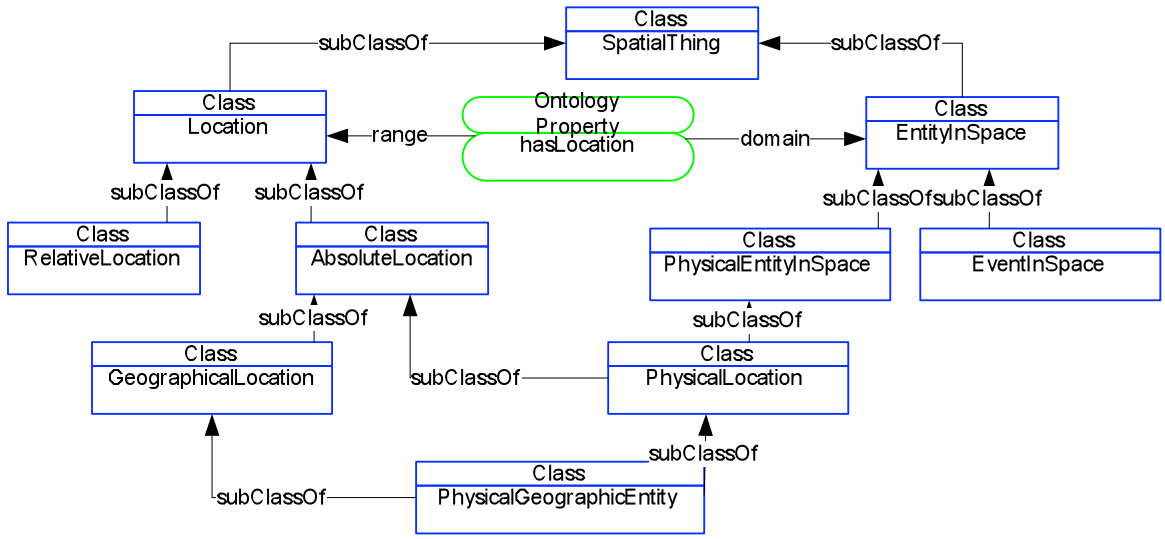

Figure 10: Hierarchy of location-related concepts (fragment)

As the root of all classes used to describe the spatial information, the concept *SpatialThing* is defined (Figure 10). This concept does not in itself define any common properties for its instances. Two subclasses of this class describe the entities which can be spatially positioned (*EntityInSpace*) and the entities which represent locations themselves (*Location*). In the same way as in the initial version, the property *hasLocation* is used to specify the position of an object. Location can reflect a physical position of an object in space as well as more abstract sense of belonging: e.g., that a smart product is located in an ambiance, where ambiance represents a network of devices.

The subclass *PhysicalEntityInSpace* of the class *EntityInSpace* serves as a common superclass for all objects which conceptually can have a location. These also include virtual objects, like copies of software located on some media.

The class *Location*, in turn, has two subclasses reflecting two ways of positioning objects: using an absolute pointer in space (*AbsoluteLocation*) or in relation to another location (*RelativeLocation*). Relative locations are described using the following properties:

- *relativeTo*. This property represents a pointer to another *Location* instance, in relation to which the location is specified.
- *relativeLocationModifier*. This property points to an instance of the class *RelativeLocationModifier* which in turn specifies the type of spatial relation and can be primitive (e.g., “on top”, “inside”, “outside”) or composite (e.g., “3 meters to the west”).

Three special cases of absolute locations are distinguished in the ontology (these classes are not mutually disjoint):

- *AbstractAbsoluteLocationDescription*. This class represents different abstract descriptions which are used to point to a location. The subclasses of this class include, for example, *Absolute3DLocation* which point to a region in a 3D system of coordinates and *GeographicalCoordinateLocation* which specify a location expressed in terms of latitude and longitude.
- *PhysicalLocation*. This class describes a special case of physical objects which themselves play the role of locations (e.g., "the table" if something is located "on the table"). Thus, *PhysicalLocation* is defined as a subclass of both *PhysicalEntityInSpace* and *AbsoluteLocation*.
- *GeographicalLocation*. This class serves as a common superclass for locations of geographical features.

Geographical features such as countries, cities, and provinces represent a special case of such locations. The class *PhysicalGeographicEntity* serves as a common superclass for these features.

5 Management model

Common tools such as triple stores dedicated to the storage of semantic data (i.e., data for the Semantic Web) do not include any mechanism to define access rights and access policies. This is especially surprising considering that, on the one hand, more and more scenarios are being considered where semantic data would be used where a combination of public, open data with more private data is necessary. On the other hand, it appears clear that access control models employed in other environments such as file systems and relational databases would need to be adapted to fit the more flexible and complex environment of semantic data, where no rigid schema is imposed, openness is a basic assumption and new data can be inferred from existing data through reasoning. This is especially the case of the SmartProducts' scenarios where semantic data and knowledge is distributed amongst different devices, managed by different users, as highlighted in particular in requirements related to security and privacy in [D4.1.1].
In this section, we investigate an initial model for access control over semantic data. This model, while seemingly simple, rely on concepts for access control identified as relevant to Smart products in [D4.2.2] and makes use of complex features of ontology languages such as OWL. We discuss it from a conceptual point of view, but also give directions towards its concrete operationalisation.

5.1 A Semantic Model for Access Control over Semantic Data

The flexibility and modelling capabilities offered by semantic technologies such as RDF and OWL make that some of the assumptions common access control models are relying on cannot be straightforwardly transferred in a semantic environment. On the other hand, new opportunities are created through the use of such technologies at the basis of an access control model. We identify some of these challenges and requirements on which a novel, semantic access control model can be built, as well as elements of the solutions in an ontological model.

To come up with an ontological approach to access control, we rely on a rather simplistic view of 'access rights', consistent with the recommendations described in [D4.2.2] regarding approaches to access control in Smart Products. We illustrate this view on a common application, inspired from social networking websites. We consider here the notion of 'user', which is in general terms an 'agent' (referred to as an "Entity" in [D4.2.2]). A user can belong to one or several 'groups'. Groups can be related to 'roles' in Role-Based Access Control approaches (see [D4.2.2.]), but should be more generally see as significant sets of agents. We use the generic word 'dataset' to refer to a sub-part of the data on which particular access rights might be applied. Access rights are defined at the level of groups, which means that if a group

has access to a dataset, individuals belonging to the group have access to the dataset (this is not a strong restriction, as there can be groups of 1 individual).

5.1.1 The access control model should be defined homogeneously to the data

Semantic technologies such as RDF and OWL correspond to flexible data models for information especially regarding relationships between various types of objects. In the simplest form, an access right model is a relationship between a group of users (or agents) and a dataset (or a sub-dataset). It is a desirable property that objects such as users and datasets are represented in a way homogeneous to the data which is being accessed. Figure 11 illustrates an OWL ontology for a simple access model based on this idea. The class *AgentGroup* is used to represent groups of users, which might have particular roles, giving them access to datasets, which are instances of the class *Dataset*. The relation *hasAccessTo* represents the link between a group and the datasets they can access, and a particular *Agent*, representing a user, belongs to a group. Groups and datasets can have sub-groups and sub-datasets respectively.

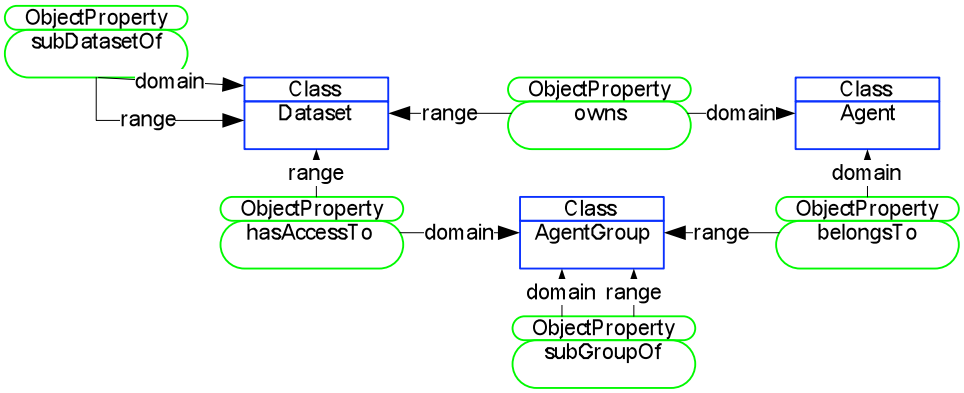

Figure 11: A simplistic ontology for access control.

While this first model is rather simplistic, it shows how access related information can be represented, manipulated and queried, using the same mechanisms that are used for the data itself. For example, imagine a user *Bert* who wants to give access to her contact information to her family. *Bert* is represented in the system as an instance of the class *Person*, which is a subclass of *Agent*: *Person(Bert)*, *Person* $\subseteq$ *Agent*. *Bert* also owns the dataset *BertContactInfo* which is an RDF/OWL based representation of her contact information: *Dataset(BertContactInfo)*, *owns(Bert, BertContactInfo)*. The group *BertFamily* can then be defined, with two members *Jeff* and *Caroline*, and as a subgroup of the group *BertFriends*: *AgentGroup(BertFamily)*, *Person(Jeff)*, *Person(Caroline)*, *subGroupOf(BertFamily, BertFriend)*, *belongsTo(Jeff, BertFamily)*, *belongsTo(Caroline, BertFamily)*. Finally, the group

BertFamily can be given access to the dataset BertContactInfo: *hasAccessTo(BertFamily, BertContactInfo)*.

5.1.2 The access control model and the corresponding mechanisms are independent from specific domains and storage systems

Another advantage of semantic technologies is that ontological models can be defined that are independent from any application domain, but can easily integrate with a specific domain model. Indeed, the model depicted in Figure 11 does not rely on any element of the domain (whether we are talking about friends or colleagues or administrators, or whether the data is about contact information, information about pictures or financial information does not impact on the general model). As shown in the example, the connection to the domain is realised when instantiating the model, and defining the groups and datasets.

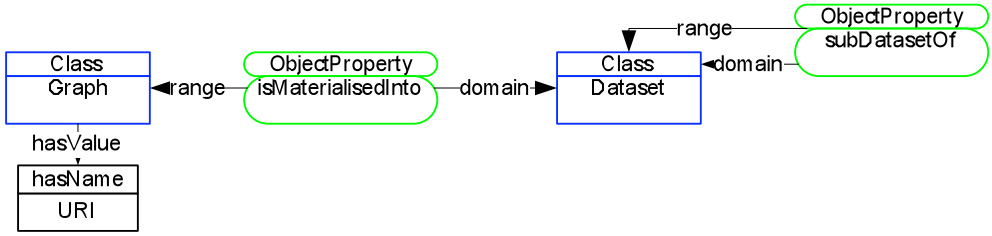

Figure 12: **A dataset can be associated with a named graph, to provide a standard way to restrict to specific data**.

In addition, an ontology is also an abstract model which relies on semantic technologies, but is independent from any specific implementation of these technologies. Ideally, we would indeed want to be able to implement the corresponding access control mechanisms as a common 'layer on top of' existing RDF triple stores. This suggests the ability to rely on a standard query mechanism which can be restricted to particular subsets of the data. The SPARQL query language and the named graph mechanism provide such standards. Indeed, as depicted in Figure 12, the idea is that each dataset, as represented in the access control model, is associated (materialised into) a named graph. Therefore, whenever a SPARQL query is being put to the system from an authenticated agent (e.g., *Jeff*), the access control model can check the groups this agent belongs to (*BertFamily*), the datasets these groups have access to (*BertContactInfo*), and transform the query by restricting it to the corresponding named graphs. This therefore provides an elegant access control mechanism relying entirely on standards RDF data models and SPARQL queries.

5.1.3 Groups can be dynamic and defined intentionally

A more sophisticated requirement relates to the way groups are defined. Indeed, we could distinguish between defining them extensionally, by listing their members, or intensionally, by defining the criteria required to be a member (this corresponds to the different between Role-Based Models and Attribute-Based Models in classical access right approaches, see [D4.2.2]). In the example above, we used an extensional approach, by declaring explicitly that *Jeff* and *Mary* were part of *BertFamily*. However, in many scenarios, such an approach becomes unrealistic and groups need to be defined based on the characteristics they have in the data. Indeed, we can introduce the group of Bert's friends (*BertFriend*). In the context of a social networking website, this group could be defined as anybody with a connection to *Bert*. In a similar way, one would not want to have to list all the members of the group of the friends of the friends of *Bert*, but rather to define it as any user with a connection to at least one of the friends of *Bert*.

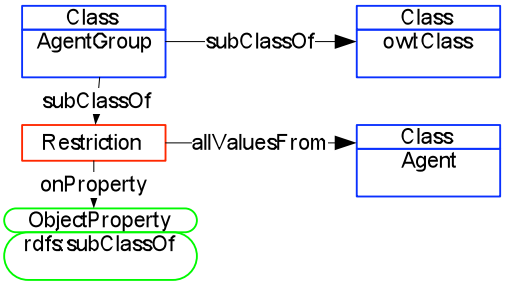

Figure 13: Representing groups of agents as classes of agents.

Through class definitions, OWL provides a convenient way to realise such an intentional definition of a group of agents. It has to be noticed however that this requires the use of meta-modelling, which is a very sophisticated and usually not recommended feature of RDF/OWL. Indeed, as depicted in Figure 13, the idea is to modify the definition of groups, so that any group now becomes a class. This is realised through declaring *AgentGroup* as a subclass of *owl:Class*, so that it represents the class of all "classes of agents" (i.e., the complete definition of *AgentGroup* becomes *AgentGroup = owl: Class AND ∀subClassOf.{Agent}*).

As a consequence, a particular group such as *BertFriend* can now be declared as a class, with restrictions indicating the criteria for belonging to the group. In the case of *BertFriend*, a sensible definition is *BertFriend = Person AND ∀knows.{Bert}*. Such a definition can also be reused in other group definitions, such as the one of the friends of friends of Bert: *BertFriendOfFriend = Person AND ∀knows.BertFriend*. In this model, members of a group become instances of the group, which can either be declared (e.g., *BertFriend(Joe)*), or inferred from the definition of the group (e.g., *BertFriend = Person AND ∀knows.{Bert}, knows(Joe,*

Bert) ⇒ *BertFriend(Joe)*). In addition, being a subgroup is now represented through the standard subclass relation in RDFS/OWL (e.g., BertFamily ⊆ BertFriend), so that the mechanism that makes a member of a group to be also a member of its super-group becomes naturally integrated (e.g., *BertFamily(Jeff), BertFamily* ⊆ *BertFriend* ⇒ *BertFriend(Jeff)*).

5.1.4 Datasets can be dynamic and defined intentionally

The same remark made above regarding groups can be made equally for datasets. Indeed, in a social network scenario for example, it would be reasonable to expect to be able to express access information such as "the friends of the friends of Bert have access to the list of friends of Bert and to the photos uploaded by Bert". As shown above, we might want to explicitly list the friends of Bert, or to be able to define intentionally the criteria for data to be recognised as part of this dataset.

The approach we want to investigate here is to use, in a similar way as for groups, meta-modelling, but in a way even more sophisticated than before: using RDF statement reification. Indeed, statements or triples are the basic, elementary constituents of data in RDF. In other terms, a dataset can be defined as a set of triples. We can therefore envisage to represent datasets in the access control model as 'classes of statements' (see Figure 14). In such a model, the dataset of the friends of Bert can be defined as a class in the following way: *BertFriendDataset = rdf:Statement AND ∀object.{Bert} AND ∀predicate.{knows}*. As a consequence, a reified statement making a connection between *Bert* and one of her friends (e.g., *<Joe, knows, Bert>*) becomes an instance of the dataset, and sub-dataset relations can be expressed, and inferred, using the subclass relation.

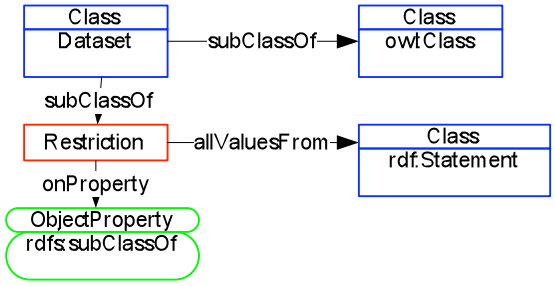

Figure 14: Representing datasets as classes of RDF statements.

It should be noticed however that this approach suffers from a number of disadvantages, discussed in the next section. It is presented here for the sake of discussion, but is likely not to be actually used in a practical implementation.

5.1.5 The access control model and mechanisms can take benefit from the inference capabilities of OWL

When dealing with complex configurations, with many datasets, sub-datasets and groups, making sense of an access control model can be a difficult task. Datasets and groups relate with

each other as well as with other groups and datasets. As shown earlier, a meta-modelling approach to the representation of groups allows for some level of automation through inference, recognising for example that a particular agent is a member of a particular group. In a similar way, it is possible to apply inferences to classify a particular model and derive possibly implicit and unexpected consequences.

Indeed, let us now consider the example of the groups *BertFriend = Person AND ∀knows.{Bert}* and *BertFriendOfFriend = Person AND ∀knows.BertFriend*. Each of these groups can be associated to certain datasets they have access to. We can for example indicate that *BertFriend* $\subseteq$ *∀hasAccessTo.{BertContactInfo}* to mean that any member of the group *BertFriend* has access to the dataset *BertContactInfo*, and *BertFriendOfFriend* $\subseteq$ *∀hasAccessTo.{BertFriendDataset}* to mean that any member of the group *BertFriendOfFriend* has access to the list of *Bert*'s friends.

As we could expect, the ontology we use describes the property *knows* as reflective and symmetric (but not transitive or functional). While the situation in this case appears quite straightforward, we could derive inferences which might not have been intended, including that BertFriend $\subseteq$ BertFriendOfFriend and so that any member of *BertFriend* will have access to the same datasets as members of *BertFriendOfFriend*, or that *Bert* herself is a member of both *BertFriend* and *BertFriendOfFriend*. We believe that making such inferences implicit and exposing them to the user could allow the access control model in place to be validated by making explicit potentially undesirable consequences.

Figure 15 summarises the access control model discussed here, which, while simple in principle, employs sophisticated ontological mechanisms to represent access control over semantic data. It could be argued that this model is limited to simple read access. However, we believe that it can be easily extended to include any other kind of access. In the next section, we discuss some of the issues related to the operationalisation of this model.

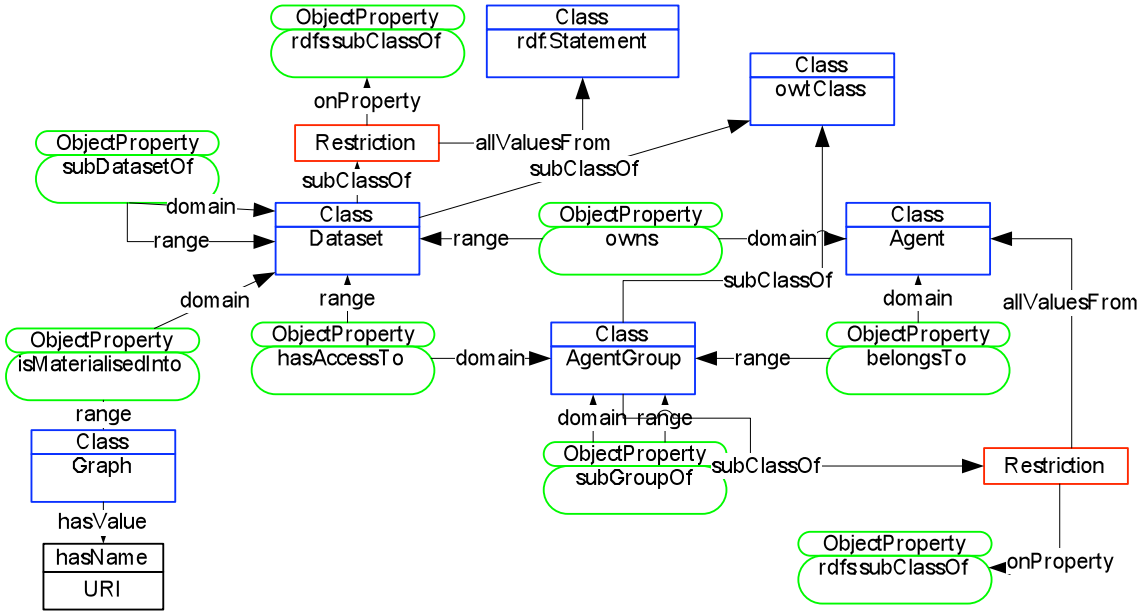

Figure 15: Overview of the complete semantic access control model for semantic data.

5.2 Towards an operational model for access control over semantic data

It appears obvious that, by following an approach that makes an advanced and sophisticated use of the features provided by semantic technologies, the model above represents an idealistic view that might not be operationable in practice. We identify here some of the challenges related to the concrete use of the proposed access model, and discuss possible solutions which, while deriving slightly from the original approach, represent concrete methods to realise a semantic access control model over semantic data.

5.2.1 Complexity and Performance

One of the most common criticism of semantic technologies, especially ontologies and ontological inferences, concerns their complexity and scalability. With respect to this, it is worth looking at the different operations one can expect to be available over such an access control model as described previously. The first one is, of course, querying. In this task, an authenticated agent sends a SPARQL query to the triple store. As explained before, the task here is to identify the groups to which this agent belongs, and the datasets these groups have access to, so as to transform the SPARQL query and restrict it to the corresponding named graphs. This is realised through a mechanism called instantiation: the instance that represents the agent is being classified as belonging to certain classes of the models, to find out in which classes of agents (i.e., group) it belongs, so that it can 'inherit' from these classes the information about accessible datasets. Instantiation is a common task in ontological reasoning which can be realised in models containing thousands of classes (i.e., groups). In addition,

inference can be applied once for each agent in the base, and reused at the time of querying, as long as the definitions of groups do not change. This gives an indication of the kind and scale of the scenarios in which such an approach could be applied.

More complicated is the case of loading data into the store. Indeed, every time new data is being added to the store, each constituting triple has to be reified into a RDF representation, instantiated as a member of one or several classes of statements (i.e., datasets), to be finally added to the corresponding graph. Moreover, a complete re-instantiation of each triple in each dataset is necessary every time the definition or structure of the datasets is changed. This appears to be unfeasible in any scenario of realistic scale.

One possible solution to this problem is to replace the intentional definition of a dataset by a SPARQL construct query. Indeed, a construct query concretely represents a filter for statements that can identify the boundary of a dataset within a triple store. For example, the dataset previously defined as *BertFriendDataset = Statement AND ∀object.\{Bert\} AND ∀predicate.{knows}* could equivalently be defined as the construct query:

```
CONSTRUCT {?x knows Bert}
WHERE {?x knows Bert}
```

The advantage of this approach is that loading new data now only requires execution of the construct queries from each defined dataset, to check whether some of the statements need to be added to the corresponding graphs. In the same way, modifying the definition of a dataset only requires the execution of the new construct query onto the data in the store. However, because the kind of inferences applicable to class definitions in OWL do not apply to construct queries, the hierarchy of datasets and sub-datasets in this approach cannot be automatically inferred, and cannot be guaranteed to match the definitions of the datasets (i.e., we cannot check whether a construct query is 'more general' or is 'included' in another one).

5.2.2 Usability

Usability has been identified as an important requirement in [D4.1.1]. As discussed in [D4.2.2], usability in access control depends on the amount of manual work required by the user to define the access control model, the complexity of this model, and the ability of the user to comprehend the impact of the model. In that sense, the discussed model improves usability by allowing some of the implicit consequences of the access control definitions in a particular scenario to be inferred. However, defining the model according to an ontology like this one is a difficult task, which requires a good understanding of sophisticated knowledge representation mechanisms, and can easily lead to possibly catastrophic errors. The definition of groups can for example be too inclusive, because of some elements of the model triggering inferences which would not be natural to the user. Many of these situations could be detected by materialising these inferences, but good end-user tools would be needed to provide, on the one

hand, appropriate editing mechanisms for the access right model that would hide the complexity of the underlying ontology, and on the other hand, understandable explanations for the inferences realised upon the model.

5.2.3 Representation of Negative Information

One of the elements making Semantic Web representation formats such as RDF and OWL different from most other data models is the so-called *open world assumption*. Indeed, contrary to databases or other logic-based systems such as Prolog, in OWL, the fact that something cannot be inferred (i.e., proved) to be true does not mean that it is false.

This is an interesting issue, as it means that it becomes very difficult to represent negative information, such as access restrictions. In our model, we avoid this problem by expressing only positive information (i.e., a group has access to a dataset), but it is not hard to imagine scenarios where expressing similar, negative access restrictions would be more convenient (i.e., a group does not have access to a dataset). Such an ability would provide interesting possibilities even in our model, as we could indicate for example that a dataset is accessible only to users who are not the friends of *Bert*. Expressing such a group as *NotFriendOfBert* = ¬*BertFriend* does not work in practice, as it defines this group as the one of people who could be proved not to be the friends of *Bert* (which is very hard to do) and not as we would naturally expect as the people who are not members of the group *BertFriend*.

6 User model

The original model of the user profile was mainly extended based on implementation work and initial evaluation. In particular, revisions have been made as a result of collaboration between WP2 and WP5 in order to model the concepts needed to support interaction with the user.

6.1 Overview of interaction types

Smart products should be able to proactively engage in natural, multimodal interaction with the user. The interaction can be implicit and explicit.

The most natural interaction is interaction that is not perceived as such by the user, i.e., implicit interaction. To support this, smart products require an ontology for describing context events, relevant to the application. Such events are expressed using the context model. These context events can be used as triggers for interaction: for example, next step of instructions can be shown if completion of the previous step is detected, or the user can be notified that some operation (e.g., fastening a screw) was performed incorrectly. [D5.1.3] presents examples of using context events in interaction in different domains.

Explicit interaction is interaction via a dedicated interface, [D5.1.3] presents also examples of explicit interaction in different domains. The content and domain of discourse of the interaction between the user and the product is different for each product. For example, a coffee machine in a kitchen requires different information compared to a smart wrench in an aircraft manufacturing scenario. Thus, any attempts to provide models to describe the content of interaction between the user and the product semantically as part of the general SmartProducts platform must fall short. The SmartProducts platform only aims at providing a model for the representation of interaction, which is the well-known HTML / XForms standard for describing interfaces. The developer of a smart product is expected to provide interface templates and custom HTML / XForm instances that can be used for interaction in different circumstances. For example, one instance may be used for information delivery via audio, another instance – via images, third one – via GUI text etc. We call each such instance an *interaction option*. This allows the product developer to exercise fine-grained control over the look and feel of the interface, e.g., if they want to use a certain colour scheme in a GUI, use the voice of a popular actor in a VUI etc.

6.2 Support for interaction options

As the developer cannot know the exact situation in which interaction will take place during development time, she must provide information alongside the interaction options that enable the system to choose the correct interaction options at runtime. To this end, the SmartProducts platform provides an ontology to describe these interaction options. Each interaction option is

described with a set of properties that describe the intended usage scenario, for example, whether this option is delivered via the Text-To-Speech engine.

At runtime, most appropriate interaction option(s) are selected based on their descriptions, the user model and device descriptions: first, device description (interaction resources) is used for excluding interaction options not supported by the user device (for example, video instructions are excluded if the user device cannot show videos).

The model of interaction resources in the smart products ontology is based on the well-established CC/PP model[10]. Interaction resources are described using the following properties:

- *hasScreenWidthChar*: screen width in characters.
- *bitsPerPixel*: output resolution measured in bits per pixel.
- *hasPixelAspectRatio*: pixel aspect ratio.
- *hasNumberOfSoftKeys*: number of soft keys the device requires.
- *hasScreenHeightChar*: screen height in characters.
- *hasScreenWidth*: screen width in pixels.
- *hasModel*: device model identifier.
- *hasVendor*: device vendor name.
- *hasOutputCharset*: output charset.
- *hasInputCharset*: input charset.
- *acceptsSoftwareType*: type of software modules which can the device accepts (e.g., application/vnd.wap.wbxml)
- *acceptsEncoding*: acceptable encoding scheme, e.g., base64.
- *hasKeyboard*: defines whether or not the device has a keyboard.
- *isTextInputCapable*: specifies whether the device accepts text as input.
- *isSoundOutputCapable*: specifies whether the device can produce sound as output.
- *isImageCapable*: specifies whether the device can output images.
- *isVoiceInputCapable*: specifies whether the device accepts voice input.
- *isColorCapable*: specifies whether the device is available to show colours in the visual output.
- *acceptsDownloadableSoftware*: specifies whether the device accepts software modules downloaded from the Web.

After excluding interaction options not supported by the current device, the remaining options are ranked, based on their descriptions and user model, as described in D5.2.1 [D5.2.1]. Then top-ranked interaction options for each modality are used in interaction, provided that their ranks exceed acceptance threshold, set by smart products designer.

[10] http://www.w3.org/TR/CCPP-struct-vocab/

As interaction options are matched against interaction resources and parameters of user profile, their descriptors should be related to each other, interaction options have to be linked to the required parameters of devices (e.g., output modality) and parameters of the user profile (e.g., language and level of visibility).

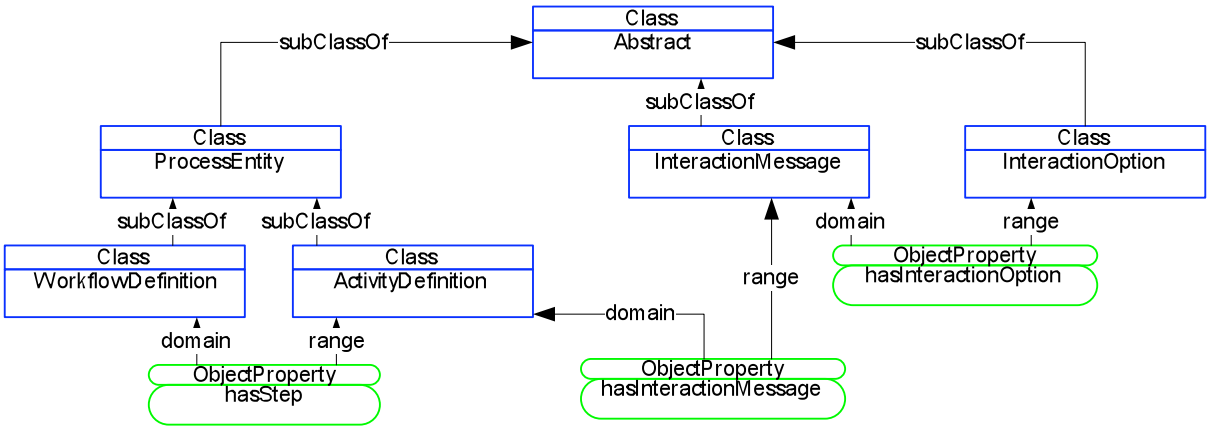

Figure 16: Modelling user interaction options

The skeleton of the model representing user interaction options is shown in Figure 16. The user interaction process is represented as a workflow. Steps of the workflow (instances of *ActivityDefinition*) are linked to interaction messages (instances of *InteractionMessage*). Each instance of *InteractionMessage* can be linked to several instances of the class *InteractionOption*. An appropriate interaction option is selected at the execution time.

Descriptions of interaction options include the following properties:

- type of information: primary, complimentary, notification, control etc.;
- level of visibility: e.g., audio message is more likely to attract user attention than a message box in a GUI;
- modality: GUI, audio etc.;
- presentation medium: text, image, beep etc.;
- level of details;
- language;
- anti-option: this descriptor is used only for interaction options which are not configuration controls. For example, if a smart product asks the user whether she already finished some action, anti-option to this could be buttons or icons “ask me later” and “never ask me again”. Anti-option for concise instructions would be a button “more details” (if this button is not already present in the GUI).

6.3 User model extensions

Initial user model was mainly concerned with interaction between a smart product and an individual user, and it was mainly aiming at supporting automatic adaptation of smart products behaviour to users and contexts. The user model was updated to support the following features:

- *Customisation* (user-controlled adaptation of smart products behaviour). Customisation functionality was added because humans generally like to stay in control over computers [Cheverst 2005]. During our initial user study, described in [D5.2.1.Annex], test subjects highly appreciated customisation features of the first prototype of the Cooking Guide, and suggested to add several other customisation features.
- Automatic and user-controlled *adaptation* to multiple users. This functionality was added because situations when several users use same smart products are not uncommon, for example, several friends or family members can cook together. During our initial user study, described in D5.2.1.Annex [D5.2.1.Annex], test subjects appeared to be very interested in multi-user adaptation [Vildjiounaite 2011].
- *Learning of user preferences*. During our second user study, described in D5.5.1 [D5.5.1], test subjects approved existing learning functionality.
- *Use of domain information*: first, information supporting primary user task (for example, recipes database) and complimentary information – information which is related to the current user task, but is not really necessary (for example, information how to adapt recipes to different kinds of diets).

Additionally, the model of user preferences was updated to include preferences regarding input to smart products, especially sensor-based tracking of user activities and audio analysis, because not all test subjects approved this functionality in all contexts [Vildjiounaite 2011].

Parts of the user model, related to automatic adaptation of smart products behaviour to individual users did not change because user studies, described in [D5.2.1.Annex] and [D5.5.1], did not show the need to for the change. For details, how initial user model is used in interaction, please refer to [D5.2.1]. [D5.1.3] presents interaction examples in the form of screenshots of application mock-ups, developed for testing user modelling methodology.

Currently the user model for interaction includes the following parts:

1. Personal information, such as name, age and gender etc. These data can be used for activating interaction stereotypes when user preferences are not known.
2. Personal capabilities, such as knowledge of languages and health problems. These data can be used as hard constraints, for example, it does not make sense to interact with a person in an unknown language.
3. Personal preferences:

- Preferences regarding smart products output:
    - Preferences regarding modalities (audio, GUI text, GUI images/ videos) and levels of details of presenting primary task-related information, such as guiding step instructions.
    - Preferences regarding task overview presentation: whether to provide only step names or also a brief overview of each step.
    - Preferences regarding presentation of optional task-related information, such as tips on adjusting a recipe to a personal diet: whether to present this information at all and if so, via which modalities (e.g., audio message or beep or GUI or everything).
    - Preferences regarding system-initiated notifications, such as that baking time has expired: whether a smart product is allowed to initiate interaction and if so, via which modalities.
    - Preferences regarding explanations of smart products logic, such as why notifications were disabled and which risks their disabling may cause: whether to deliver explanations and if so, via which modalities.
    - Preferences regarding learning, such as whether smart products shall learn different types of user preferences or use system default ones.
    - Preferences regarding system-initiated requests, related to learning, such as requests for explicit user feedback on automatic adaptation.
- Preferences regarding smart products input:
    - Whether smart products are allowed to track users' activities via environmental sensors to enable implicit interaction or not (if smart products are allowed to track user actions, which context events are allowed to trigger which interface changes: for example, during user tests some subjects approved product-initiated notifications, triggered by context events, but did not like event-triggered transition to the next step instructions, while some other users had the opposite preferences).
    - Whether audio analysis should be enabled or disabled.
    - Preferences regarding configuration controls for customisation of smart products input and output: which controls (e.g., "audio on/off" button) are frequently used and thus should be immediately visible, and which ones may be moved to less accessible menu part.

Preferences regarding smart products input and output can be context-dependent, for example, audio output may be undesirable at night, and speech recognition may be undesirable in social settings (during the user study in WP5 [Vildjiounaite 2011] several test subjects stated that it is

impolite to talk to a computer while being in a group of other people). These preferences can be device-dependent too, for example many test subjects said that showing images in a small screen is needed only when it is really difficult to explain something verbally. Although need in controls for quick customisation of smart products input and output can depend on context too, it is feasible to consider only device-dependency of preferences regarding customisation controls because interface should not look inconsistently. Personal preferences can be acquired via questionnaires and/ or via observations of users' choices, and can be used for ranking possible options to interact with the users.

Another aspect arising if the ambiance is used by several users concerns the preferred way to combine preferences of multiple users, interacting with the same application. For example, if several persons cook together and some of them consider audio presentation of cooking instructions annoying, should it be enabled or disabled? Possible ways to combine preferences of multiple users include:

- *dictator*: use preferences of the device owner;
- *democratic*: try to satisfy the majority of users;
- *shy*: activate some functionality only if all users would activate it when using the smart product alone;
- *aggressive*: activate some functionality if at least one of the users would activate it when using the smart product alone;
- *default*: use default settings of a smart product;
- *last used*: use the settings used by this group of persons when they were using the application previous time);

Each user can have one preferred way of combining preferences for the case when the same group of persons did not use the application earlier, it can be any of the first five options. For the case when the same group of persons already used the application earlier, it is possible to specify that "Last Used" way is preferred. It is also possible to have just one preferred way, for example, to be always a dictator.

7 Domain model: kitchen appliances

During the first development stage, the WP2 effort primarily concentrated on the smart kitchen scenario as the one which better illustrates the knowledge modelling and reasoning aspects which need to be tackled for generic smart products scenarios (see [D8.2.1]).

When working on the smart kitchen scenario, the representation of food-related concepts in the ontology has been changed in order to support the problem-solving reasoning better.

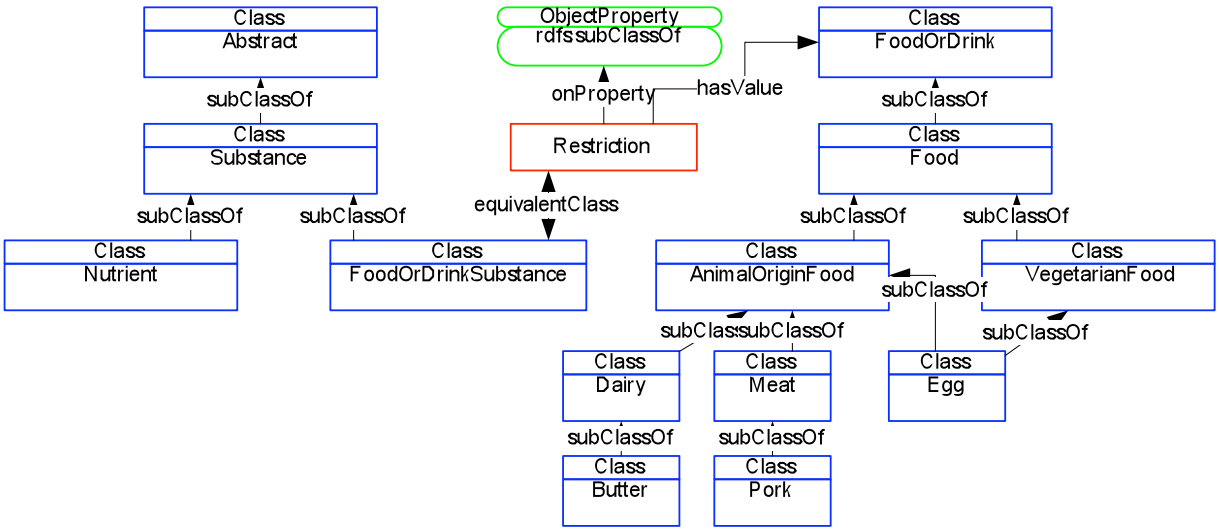

Figure 17: Modelling food substances as both classes (subclasses of FoodOrDrink) and instances (members of FoodOrDrinkSubstance)

In particular, one of the requirements arising during the implementation of the meal planning problem-solving method was the need to categorise food items into many overlapping categories in order to be able to match them against various user preferences in a flexible way.

On the other hand, there is a need to represent substances, of which actual food items are made. These substances generally correspond to food categories. The granularity of descriptions in different recipes can be different: e.g., one recipe can require "0.5 cup of onion", while another one "0.5 cup of red onion". When matching the description against the list of available food items, red onion should be suitable for both recipes. In order to support this, it was decided to model food substances as both classes (to make use of class-subclass hierarchy) and instances at the same time (Figure 17). The class *FoodOrDrink* serves as a top-level class for different categories of food. These categories can be overlapping (e.g., Egg is both a subclass of *AnimalOriginFood* and *VegetarianFood*), and different branches can have arbitrary granularity (e.g., one can define *ChickenEgg* as a subclass of *Egg* and *SmallChickenEgg* as a subclass of *ChickenEgg*). On the other hand, the class *FoodOrDrinkSubstance* is defined as equivalent to the restriction that all its instances are subclasses of the class *FoodOrDrink*. This allows defining an ingredient portion with the value *ChickenEgg* for the property *madeOfSubstance*.

Population of the class hierarchy of food substances was performed in a semi-automatic way as described in [D2.3.1].

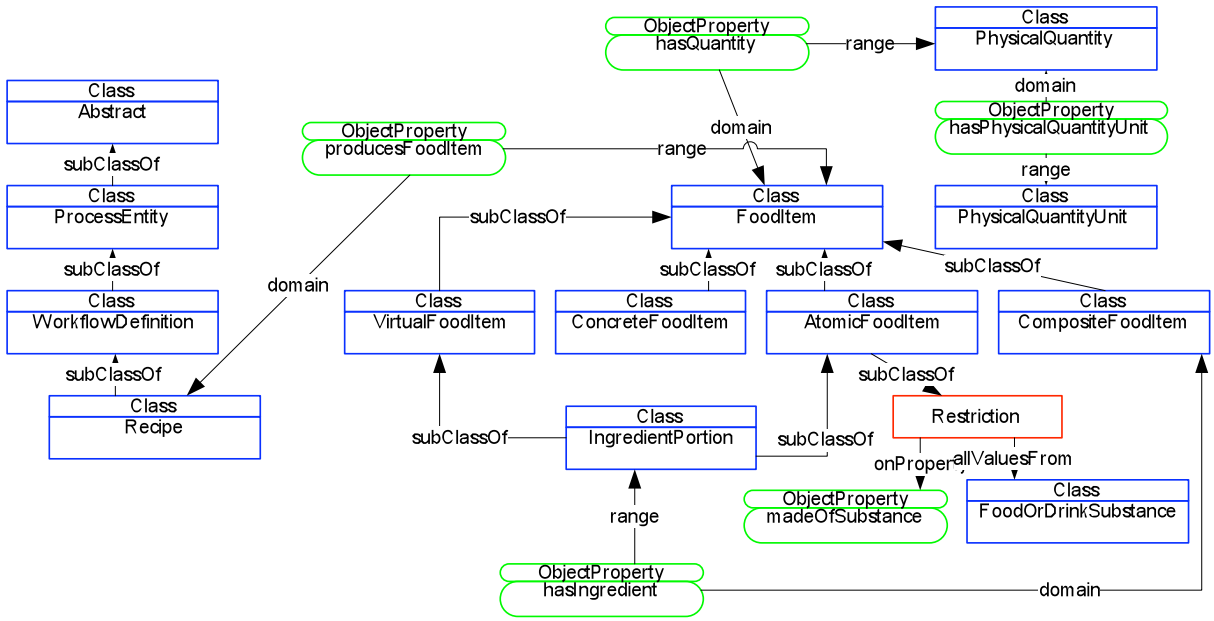

Figure 18: Modelling recipes and ingredients

In the representation of recipes, two components are distinguished: the description of the cooking process and the description of the resulting food item (Figure 18). These two components are modelled by two classes: *Recipe* for modelling the process-related aspects and *FoodItem* for modelling the dish which can be obtained by the process. The class *Recipe* is defined as a subclass of *WorkflowDefinition* and contains the corresponding properties of this class such as:

- *hasDuration*, which describes the estimated time duration of the workflow;
- *hasSteps*, which links the workflow with the activities it involves. Since the OWL ontology only models the aspects of the workflows related to workflow selection [D2.1.2], the steps are not organised in a sequence.

All food items are expressed using instances of the class *FoodItem*. Instances of the class *Recipe* are connected to instances of the class *FoodItem* by the property *producesFoodItem*. The subclasses of the class *FoodItem* shown in Figure 18 represent two pairs of disjoint subcategories: *VirtualFoodItem* (representing descriptions of food items such as ingredient portions or dishes produced by recipes) vs *ConcreteFoodItem* (physical pieces of food, e.g., stored in the fridge) and *AtomicFoodItem* (items made of a single substance) vs *CompositeFoodItem* (items consisting of several ingredients). *IngredientPortion* is defined as both an *AtomicFoodItem* and *VirtualFoodItem*.

Together with the food hierarchy organised as in Figure 17, descriptions of ingredient portions can be used to define classification rules for composite food items: e.g., a food item which contains at least one ingredient portion made of a non-vegetarian food substance is a non-vegetarian food item. Such simple rules can be expressed using the means of OWL language. More complex rules are defined using the custom rule definition syntax of a chosen rule engine (such as BaseVISor [D2.2.1]).

8 Domain model: cars

In deliverable [D9.2.1] the description and the initial design of the 3 FIAT scenarios is reported. These scenarios will be partially or completely implemented in SmartProducts.
Each scenario implies several general and specific concepts that have to be modelled in SP ontologies. With the refinement of scenarios, the initial model proposed in [D2.1.2] had to be extended. Some of the required concepts needed to be revised or created in order to consider all aspects involved in the expected demonstrators.
In the section below each scenario has been analysed from the ontology modelling point of view, and the list of required ontological concepts is presented. Some of the required concepts are present in the generic model and do not require domain-specific extensions. For the others, the list provides an explanation of the reason for the extension and the description of related competency questions that will be used for the final evaluation.

8.1 Scenario 1: Adaptive eLUM for snow chain mounting

- The list of involved smart products: IPAD, Blue & ME, etc.
  To implement the Adaptive eLUM demonstrator, it is necessary to know which smart products are available at a specific time (if the iPAD is available and the snow chains are on board the system can visualize the snow chain mounting procedure on the iPAD gui, etc). This aspect can be covered with the generic model (classes *Ambiance* and *SmartProduct*) and do not require domain-specific extensions.
- Distinction between vehicle components and vehicle compatible components:
  some vehicle components are part of all vehicles belonging to a specific model (*production components*), other can be included in a specific vehicle configuration (*optional components*); a third category include all components that usually are not sold with the vehicle but that are bought by the driver after the vehicle purchase (i.e. snow chains, child seat, etc). These components are usually produced by a different manufacturer that can be also the owner of the related proactive workflow procedure. In this scenario with the goal to adapt the eLUM to a car, beyond the knowledge of which specific components belong to the vehicle, it can be useful to know:
  - which components were added to the vehicle after the purchase? (when?) This information is needed to activate the procedure aimed to update the eLUM with a new proactive workflow procedure. A specific datatype property *installedOn* for the class *VehicleComponent* had to be added.
  - Is the added component compatible with my vehicle? When the snow chains (or in a wider context other external component) is added to the car, the system

should proactively advice the driver if the purchased model is or not suitable for the specific car model.

- List of sensors and related info: Which sensors are available on my car? What is the value of a specific sensor? This information can be expressed with the generic sensor model (part of the context model).
- Car occupants: Which are the occupants of the vehicle? To know which occupants are in the car, it can be useful to adapt the system interaction and the proactive procedure to their specific needs and preferences. This required expressing two different user roles for the car users: *Driver* and *Passenger*.
- Trip recognition: What is the current trip? As already described in [D.9.2.1], the type of trip in progress is one of the information pieces that contributes to the knowledge of the context and can impact in the interaction (knowing that the car is doing a mountain trip, in winter, the system can activate the snow chains mounting procedure). In order to model this, a subclass TripContext was defined as a subclass of the generic type *SituationalContext*.

8.2 Scenario 2: Deprecation alerts

- Involved smart products: as in previous scenarios
- Sensors and sensors data: as in previous scenarios
- Drivers and driving style: In the deprecation alert scenario values collected from several sensors are used to calculate the wear out state of several component what is the state of a specific car component in terms of wear out? and consequently the driving style of the driver: what is the driving style of a specific driver? To represent the state of car components, the generic class *DeviceState* was extended. The *DrivingStyle* concept has to be defined in the domain ontology.
- Actions to be carried out by drivers to improve their driving style and vehicle value: In the deprecation alert scenario suggestions on how improve their driving style are sent to the drivers' mobile devices according with their driving behavior. The system should know the list of actions to be carried out by drivers reported in these persuasive messages. Which actions a driver with a given driving style have to carry out to improve it? A new class *DrivingAction* needed to be defined in the domain ontology to model the driving action categories.

8.3 Scenario 3: Extension to maintenance processes for Repair procedures management Specific tools to mount/dismount specific vehicle components

- Involved smart products: as in previous scenarios
- Specific tools: to properly cover this scenario it is necessary to model a new category of product that are the specific tools needed to mount/dismount components inside a given car model. Which are the specific tools for a given car model/component to manage? An extension of the class *ProductizedDevice* was added to the domain model to represent tools.
- Procedures for mounting/unmounting specific vehicle components: What is the procedure for mounting/unmounting a specific component? These procedures are expressed using standard workflow definition languages.
- Location of specific tools inside workshop: where is a specific tool inside the workshop?
- Location of specific tools inside car: where is a specific tool inside the vehicle?
- Location of vehicle components inside car: where is a specific component inside the vehicle? All these location-related aspects can be expressed using the generic location model (see section 4).

9 Domain model: aircraft manufacturing

The final scenario described in [D10.2.1] is based on three original use cases:

- *Task authoring*, which concerns assisting a planner in order to author a Work Order.
- *A/C context enrichment*, where the aim is to optimise the efficiency of an operator by selecting for presentation only the data relevant for the task and taking into account the user's skill level and preferences.
- *Smart tool usage*, which aims at improving the efficiency of an operator by the usage of smart tools.

Below we provide the list of the concepts which need to be represented in the model in order to support this scenario and specify, whether these concepts were already represented in the generic model or needed to be extended or defined in the domain model.

- *Devices*, in particular, the nomadic device (mobile user assistant) and smart tools. These can be represented by reusing the existing product model. Properties describing specific configuration parameters of smart tools (such as torque for a smart wrench), however, had to be added to the ontology.
- *Work order*. Work order represents a sequence of tasks which has to be executed by the user with the help of smart tools. The work order can be described using the existing workflow model. Since the work orders are only intended for human use, there is no need to involve the task model described in section 3.
- *Aircraft components*. The standard model for devices and assembly components can be reused to describe aircraft components, in a similar way as it is used in the WP9 scenario. Relative position of different components (e.g., "X is attached to Y") is expressed using the location model.
- *Anomaly dossier*. In order to support the anomaly dossier, the history of previous tasks performed on the aircraft needs to be stored. To represent this history, the following properties were added to the class *AircraftComponent*:
    - *installedOn* specifying the time when a particular component was installed.
    - *lastAccessedOn* specifying the time when the component was accessed last time by an operator.
    - *activityPerformed* linking the component with the element of the corresponding work order, in which an operator performed some task on the component.
- *Skill level*. The class *SkillLevel* was added as an element of the user profile.

10 Outlook and Future Work

The set of SmartProducts ontologies is aimed at supporting the reasoning capabilities of smart products and their proactive behaviour. At the next stages of the project, the work on knowledge models is going to focus on two main directions.

The first direction of on-going work is finalising the integration of knowledge processing components with the SmartProducts platform (see section 3.7). In particular, this includes mapping of the interaction mechanism based on advertising and handling tasks within an ambiance with the common communication middleware. The integrated set of software components together with the supporting set of ontologies must provide a reusable asset for developers of smart products in the domains not related to the original SmartProducts use cases.

In order to achieve that, the second direction of work involves extending the knowledge representation and processing components in order to exploit other promising aspects of smart products. These aspects, for example, involve supporting the functionalities of smart products having access to multiple ambiances. Improving the reusability of the set of SmartProducts ontologies, thus, constitutes the major goal for the final stage of the project.

Annex

A Glossary

Context	*Context* characterizes the actual circumstances in which th application is used. It comprises all information that distinguishe the actual usage from others, in particular characteristics of th user (her location, task at hand, etc) and interfering physical o virtual objects (noise level, nearby resources, etc). We thereb only refer to information as context that can actually be processe by an application (relevant information), but that is not mandator for its normal functionality (auxiliary information).
Environment	An *environment* is an identifiable container with a clear borde that may contain smart products and other, non-smart produ entities. Entities inside the container can influence each other bu they are not influenced by anything outside the container.
Event	Any phenomenon in the real world or any kind of state chang inside an information system can be an event. However, it must b observable and some component in the information system mu observe it in order to notify parties interested in the event.
Lifecycle	The *lifecycle* considered in the SmartProducts project consists o the following four stages: Design, manufacturing, usage an maintenance.
Proactive Knowledge	The *Proactive Knowledge* of a smart product is defined as the ensemble of data and formal knowledge representations, whicl directly or indirectly facilitate its proactive behaviour. Proactive behaviour in turn denotes mixed-initiativ communication, interaction, and action where the actual situatic and goals affect the turn-taking between a smart product and i environment i.e. users and other smart products. In particula proactive knowledge may trigger human-product interaction an product-environment communication based on perceived neec (interaction needs may be 'computed' by the product as part of i smartness, e.g., based on context changes).
Proactivity	*Proactivity* is defined as a capability to initiate actions and exhib goal-driven behaviour without an explicit request or pre-define schedule
Situation	Situations are interpretations of context data. Thus, they can als

	refer to the states of relevant entities.
Smart Products	A *smart product* is an autonomous object designed for self-organized embedding into different environments in the course of its lifecycle, supporting natural and purposeful product-to-human interaction. Smart products proactively approach the user, leveraging sensing, input, and output capabilities of the environment: they are self-aware and context-aware. The related knowledge and functionality is shared by and distributed among multiple smart products and emerges over time.
User	A *user* of a smart product is a person who uses the functionality and/or the supporting tools of smart products. Thereby we distinguish between smart products developers (end-users of the SmartProducts platform, technically skilled), support service workers (end-users of the SmartProducts platform, some technical skills required) and smart products end-users (end-users of the functionality provided by smart products, no technical skills required) which differ in their level of expertise.

B List of Acronyms

OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
XPDL	XML Process Definition Language
WP	Work Package

References

[D2.1.2] D.2.1.2 Initial Version of Conceptual Framework. SmartProducts, 2010.
[D2.2.1] D2.2.1 Initial version of concrete task and PSM libraries, SmartProducts, 2010.
[D2.3.1] D2.3.1 Knowledge translators, SmartProducts, 2011.
[D2.5.1] D2.5.1 Evaluation of active components, SmartProducts, 2010.
[D4.1.1] D4.1.1 Requirements Analysis for Storing, Maintaining, and Distributing Proactive Knowledge Securely. SmartProducts, 2009.
[D4.2.2] D4.2.2 Final Concept for Security and Privacy of the Proactive Knowledge. SmartProducts, 2011.
[D5.1.3] D.5.1.3 Final Description of Interaction Strategies and Mock-Up UIs for Smart Products. SmartProducts, 2011.
[D5.2.1] D.5.2.1 Initial Methodology for Smart Products Usage Modelling and Personalisation. SmartProducts, 2010.
[D5.2.1.Annex] D5.1.2.Annex. Description of User Tests on Methodology for Smart Products Usage Modelling and Personalisation. SmartProducts, 2010.
[D5.3.1] D5.3.1 Initial Version of Methodology and Tools for Multimodal UIs Based on Proactive Knowledge. SmartProducts, 2010.
[D5.5.1] D5.5.1 Evaluation Report for Initial Implementation. SmartProducts, 2010.
[D6.2.2] D6.2.2 Final Architecture and Specification of Platform Core Services. SmartProducts, 2011.
[D8.2.1] D8.2.1 System Design for Smart Consumer Appliances. SmartProducts, 2010.
[D9.2.1] D9.2.1 System Design for Vehicle Product Lifecycle Management Application. SmartProducts, 2010.
[D10.2.1] D10.2.1 System Design for Virtual Product Manufacturing. SmartProducts, 2010.
[WN1.5] WN 1.5. Definitions. SmartProducts, 2010.

[Benjamins-1996] Benjamins, R., Pierret-Golbreich, C. (1996), Assumptions of problem-solving methods, In 9th European Knowledge Acquisition Workshop (EKAW-96), volume 1076 of Lecture Notes in Artificial Intelligence, pp. 1-16. Springer-Verlag.
[Bergenti-2004] Bergenti, F., Gleizes, M. P., Zambonelli F. (2004), *Methodologies and Software Engineering for Agent Systems*, Kluwer.
[Botti-1999] Botti, V., Carrascosa, C., Julian, V., Soler, J. (1999), Modelling Agents in Hard Real-Time Environments, In Proc., 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'99, pp. 63-76.
[Bratman-1988] Bratman, M. E., Israel, D. J., Pollack, M. E. (1988), Plans and resource-bounded practical reasoning, *Computational Intelligence* 4, pp. 349-355.

[Brazier-1995] Brazier, F. M. T., Dunin-Keplicz, B., Jennings, N. R., Treur, J. (1995), Formal Specification of Multi-Agent Systems: A Real-World Case, ICMAS 1995, pp. 25-32.

[Brazier-2002] Brazier, F. M. T., Jonker, C. M., Treur, J. (2002), Principles of component-based design of intelligent agents, *Data Knowl. Eng.* 41(1), pp. 1-27.

[Cadenas-2009] Cadenas A. et al. (2009), Context management is mobile environments: a semantic approach, CIAO '09 Proceedings of the 1st Workshop on Context, Information and Ontologies.

[Chandrasekaran-1992] Chandrasekaran, B., Johnson, T. R., Smith, J. W. (1992), Task-structure analysis for knowledge modeling. *Communications of the ACM*, 35(9), pp. 124-137.

[Cheverst-2005] Cheverst, K., et al. (2005), Exploring Issues of User Model Transparency and Proactive Behaviour in an Office Environment Control System. *User Modeling and User-Adapted Interaction* 15, pp. 235-273.

[Cossentino-2004] Cossentino, M., Sabatucci, L. (2004), Agent System Implementation, In: Agent-Based Manufacturing and Control Systems: New Agile Manufacturing Solutions for Achieving Peak Performance, CRC Press.

[Fensel-2001] Fensel. D., Motta, E. (2001), Structured development of problem-solving methods, *IEEE Transactions on Knowledge and Data Engineering*, 13(6), pp. 913-932.

[Fensel-2003] Fensel, D. et al. (2003), The unified problem-solving method development language UPML, *Knowledge and Information Systems*, 5, pp. 83-131.

[Fink-2004] Fink, E. (2004), Automatic evaluation and selection of problem-solving methods: Theory and experiments. *Journal of Experimental and Theoretical Artificial Intelligence*, 16(2), pp. 73-105.

[Jennings-1993] Jennings, N. R. (1993), Specification and implementation of a belief desire joint-intention architecture for collaborative problem solving. *JICIS* 2(3), pp. 289-318.

[Jonker-1997] Jonker, C. M., Treur, J. (1997), Compositional Verification of Multi-Agent Systems: A Formal Analysis of Pro-activeness and Reactiveness, COMPOS 1997, pp. 350-380.

[Kaelbling-1991] Kaelbling, L. P. (1991), A situated automata approach to the design of embedded agents, *SIGART Bulletin* 2 (4), pp. 85-88.

[Kitamura-2006] Kitamura, Y., Koji, Y., Mizoguchi, R. (2006), An ontological model of device function: industrial deployment and lessons learned, *Applied Ontology* 1(3-4), pp. 237-262.

[Maes-1991] Maes, P. (1991), The agent network architecture (ANA), *SIGART Bulletin* 2 (4), pp. 115-120.

[Motta-1999] [Motta-1999] Motta, E. (1999), *Reusable Components for Knowledge Modelling,* IOS Press, Amsterdam.

[Orsvarn-1998] Klas Orsvarn (1998), Some principles for libraries of task decomposition methods, *International Journal of Human-Computer Studies*, 49, pp. 417-435.

[Preuveneers-2004] Preuveneers, D. (2004), Towards an Extensible Context Ontology for Ambient Intelligence, EUSAI 2004, LNCS 3295, pp. 148–159.

[Schreiber-2000] Schreiber, G., Akkermans, H., Anjewierden, A., de Hoog, R., Schadbolt, N., van de Velde, W., Wielinga, B. (2000), *Knowledge Engineering and Management: The CommonKADS Methodology*, MIT Press, Cambridge, Massachusetts, USA, 2000.

[Swartout-1999] Swartout, B., Gil, Y., Valente, A. (1999), Representing capabilities of problem-solving methods, In IJCAI-99 Workshop on Ontologies and Problem-Solving Methods (KRR5), Stockholm, Sweden.

[Vildjiounaite-2011] Vildjiounaite, E., Kantorovitch, J., Kyllönen, V., Niskanen, N., Hillukkala, M., Virtanen, K., Vuorinen, O., Mäkelä, S., Keränen, T., Peltola, J., Mäntyjärvi, J., Tokmakoff, A. (2011), Designing Socially Acceptable Multimodal Interaction in Cooking Assistants, IUI 2011.

[Welty-2001] Welty, C., Guarino, N. (2001), Supporting Ontological Analysis of Taxonomic Relations, *Data & Knowledge Engineering* 39, pp. 51-74. Elsevier.

[Wooldridge-1995] Wooldridge, M., Jennings. N. (1995), Intelligent Agents: Theory and Practice, *The Knowledge Engineering Review* 10 (2), pp. 115-152.

[Wooldridge-2000] Wooldridge, M., Jennings, N.R., Kinny, D. The Gaia methodology for agent-oriented analysis and design, *Journal of Autonomous Agents and Multi-Agent Systems* 3, pp. 285–315.