Expert Systems with Applications 39 (2012) 1998–2010

Contents lists available at SciVerse ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

Reasoning support for flexible task resourcing

Murat Şensoy [a,*], Wamberto W. Vasconcelos [a], Timothy J. Norman [a], Katia Sycara [a,b]

[a] Department of Computing Science, University of Aberdeen, AB24 3UE Aberdeen, UK
[b] Carnegie Mellon University, Robotics Institute, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA

ARTICLE INFO

Keywords:
Knowledge representation
Semantic Web
Ontological reasoning
Logic Programming
Multi-agent systems

ABSTRACT

In many settings, fully automated reasoning about tasks and resources is crucial. This is particularly important in multi-agent systems where tasks are monitored, managed and performed by intelligent agents. For these agents, it is critical to autonomously reason about the types of resources a task may require. However, determining appropriate resource types requires extensive expertise and domain knowledge. In this paper, we propose a means to automate the selection of resource types that are required to fulfil tasks. Our approach combines ontological reasoning and Logic Programming in a novel way for flexible matchmaking of resources to tasks. Using the proposed approach, intelligent agents can autonomously reason about the resources and tasks in various real-life settings and we demonstrate this here through case-studies. Our evaluation shows that the proposed approach equips intelligent agents with flexible reasoning support for task resourcing.

1. Introduction

In many scenarios, tasks are monitored, managed, or executed by intelligent agents (task agents) (He & Ioerger, 2005). Tasks can be simple (atomic) or composite (composed of various subtasks). Their success depends on various factors; one of them is the appropriate selection of resources. Determining *what types of resources should be used to accomplish a specific task* requires extensive domain knowledge and expertise. However, tools for fully automated determination of necessary resource types are important for the following reasons:

(i) **Availability of expertise:** Some organizations may not have human experts to determine the required types of resources for their tasks.
(ii) **Scalability:** As the number and complexity of tasks increase, the process of manually determining types of resources becomes infeasible and highly error-prone.
(iii) **Dynamicity:** The types of resources required by a task may change depending on outputs of other tasks and unpredictable environmental factors. Therefore, it may not be possible to precisely determine what types of resources should be used for a task, in advance; instead, at run-time, the agent responsible for the task should consider the existing conditions and reason about the necessary resource types.
(iv) **Intolerance to delays:** In some settings, it is critical to determine necessary resources for tasks in a timely fashion. In these settings, keeping human experts always "in the loop" may lead to unacceptable delays and may lead to task failure. Fully automated determination of the required resource types enables task agents to act autonomously without human intervention.

Example 1. Consider a composite "fire detection and extinguishing" task in a large forest area. This task is composed of two subtasks: "detect fire" and "extinguish fire". For the "detect fire" task, a set of Unmanned Aerial Vehicles (AUVs) with the necessary sensors (e.g., optical cameras) is used. If a fire is detected in a specific location, the agent responsible for extinguishing fires is informed of the fire coordinates. The types of resources that should be allocated strictly depend on the location, terrain properties, and weather conditions. For example, if there is an appropriate water resource (e.g., a lake) nearby, fire extinguishing aircraft and helicopters should be used. If the location is mountainous, the types of these resources should be chosen so that they can fly over the terrain safely. If the weather conditions are not good, aerial vehicles may not be appropriate; in this case, only ground-based fire extinguishing vehicles should be allocated.

Example 2. Consider a poor country where there is an ongoing civil war. The civilians that have lost their homes are sheltered in a civil camp. The Red Cross aims to send medicines, medical equipment and doctors to the camp. However, there is a possibility of being attacked by armed groups on the road. In this context, it is critical to conduct a surveillance task that monitors the road to the camp and informs the authorities about any possible threats.

* Corresponding author. Tel.: +44 1224 27 4174.
E-mail addresses: m.sensoy@abdn.ac.uk (M. Şensoy), wvasconcelos@acm.org (W.W. Vasconcelos), t.j.norman@abdn.ac.uk (T.J. Norman), katia@cs.cmu.edu (K. Sycara).

0957-4174/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2011.08.041

For this purpose, an agent responsible for the "surveillance" task decides to deploy three Unmanned Aerial Vehicles (UAVs) with optical cameras, e.g., three Global Hawks with EOCameras. Halfway through the "surveillance" task, part of the road between mountains has been covered by fog. The UAV responsible for monitoring this part immediately become useless, because the attached optical sensors cannot be used to sense activities under fog. In order to successfully complete the task, a new UAV with the necessary equipment should be allocated immediately. Using the new constraints, the task agent decides to deploy a Global Hawk equipped with SAR, where SAR is a radar sensor that can be used to detect activities under fog. The agent immediately allocates the nearest available Global Hawk with SAR sensors to resume the task in the foggy area.

These examples illustrate that without enabling agents to reason about resources for tasks, the human experts should always be "in the loop" during the specification and execution of tasks. Especially in dynamic environments, task agents should be intensively supported by human experts. This hinders autonomy of the agents and may result in delays, which may lead to severe failures. An appropriate approach for reasoning about tasks and resources should address the following issues:

(i) **Representation of domain knowledge:** The domain knowledge should be represented semantically using standard languages, so the knowledge created by a party can be clearly interpreted and reused by others. This representation should support rules, because some crucial knowledge can only be captured using rules (e.g., if an area is foggy, sensors with fog penetration capability should be used).
(ii) **Flexibility:** Different tasks may require different mechanisms to match resource types to their needs. Therefore, the approach for reasoning about tasks and resources should be flexible enough to accommodate different matchmaking mechanisms for different types of tasks.
(iii) **Expressiveness:** If a matchmaking mechanism is semantically represented, it can be interpreted by various agents to flexibly reason about tasks and resources. However, representation of matchmaking mechanisms may require complex data types/structures, which cannot be expressed using current languages for the Semantic Web.

Considering these issues, this paper significantly extends (Sensoy, Le, Vasconcelos, Norman, & Preece, 2010; Şensoy et al., 2010) and proposes a flexible approach for intelligent agents to reason about the requirements of tasks and capabilities of resources. As a result of this reasoning process, agents determine what types of resources they should use to achieve a specific task.

The rest of the paper is organized as follows. In Section 2, using an example, we outline a multi-agent system (MAS) in which tasks are delegated to agents that need to reason about the resources to achieve them. Section 3 describes how tasks and resources are described semantically using ontologies. Section 4 proposes our approach for flexible determination of resource types for tasks using ontological reasoning and Logic Programming. Section 5 evaluates the proposed approach and illustrates, through case-studies, how the approach provides effective solutions to real-life problems. Section 6 discusses the approach with references to the literature. Finally, we present our conclusions in Section 7.

2. Task delegation

We envisage a multi-agent system where each task is represented by a software agent, referred to as "task agent". The task agent is responsible for the task. If a task is composed of subtasks, then the task agent concerned delegates those subtasks to others. If a task agent represents an atomic task, then the agent is only responsible for the reasoning about the required types of resources for that task. Once the required types of resources are determined by the agent, instances of these resources are allocated to execute the task.

Resource determination and allocation for each atomic task is managed by the agent of that task. Hence, for a composite task, overall resource determination and allocation is achieved in a decentralized manner by the agents representing atomic tasks within the composite task. Dependencies between tasks are managed by the task agents. While some dependencies (e.g., input–output relationships) are explicit, others are not. For example, a task T_X needs resources of type either R_A or R_B; on the other hand, another task T_Y can only use resources of type R_A. Unfortunately, R_A has only one instance, which is not sharable between tasks. In this case, if T_X allocates instances of R_A instead of those of R_B, T_Y cannot be executed because of the lack of resources. These interdependencies should also be considered by agents during the allocation of resource instances.

Using the composite task example presented in Figs. 1, 2 illustrates how tasks are delegated to agents and how these agents interact in our framework. We can summarize the scenario in Fig. 2 as follows:

(i) The semantic description of the Global Warming Monitoring (GWM) task is fed into the system.
(ii) The GWM task is delegated to a task agent (GWM agent), which initiates and coordinates subtasks of the GWM task using the task description.
(iii) The GWM agent delegates the atomic task *Monitoring Icebergs* (*MI*) to a task agent (MI agent). The MI agent reasons about the required resource types using the description of the MI task.
(iv) After determining the required types of resources, the MI agent allocates resources required to achieve the task.
(v) During the execution of the MI task, a significantly melting iceberg is detected and the GWM agent is informed by the MI agent.
(vi) As defined in the description of GWM task, the GWM agent initiates *Detect Changes in Ocean* (*DCO*) task by delegating it to a new task agent (DCO agent). The DCO task is composed of three parallel atomic tasks. Therefore, the DCO agent delegates those tasks to three task agents, which are responsible of the atomic tasks: *Monitoring Thermal Changes*, *Monitoring Ocean Animals* and *Monitoring Seismic Activities* respectively. These agents autonomously reason about the atomic tasks they represent and determine the most useful resource types. Lastly, they allocate instances of the determined resource types.

The scenario in Fig. 2 reveals two important challenges. The first one is the representation of knowledge about the tasks and resources so that agents can reason about them. The reasoning at a task level involves the interpretation of a task's flow (e.g., which subtasks should be activated) and determination of required types of resources. Semantic Web technologies provide knowledge representation languages and mechanisms to support task-level reasoning. The second challenge is the allocation of specific resources for an atomic task, once the types of necessary resources are determined by its agent through reasoning. In this paper, we integrate Semantic Web technologies with multi-agent systems in a novel way to handle the first challenge. The second challenge has been extensively studied in the MAS community (Chevaleyre et al., 2005) and is not within the scope of this paper. The next two sections give details about our approach for the representation

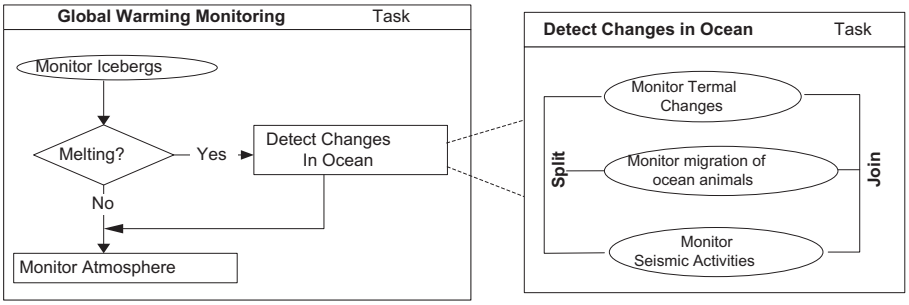

Fig. 1. A composite task example for monitoring effects of global warming (composite tasks are represented using rectangles while atomic tasks are represented using ellipses).

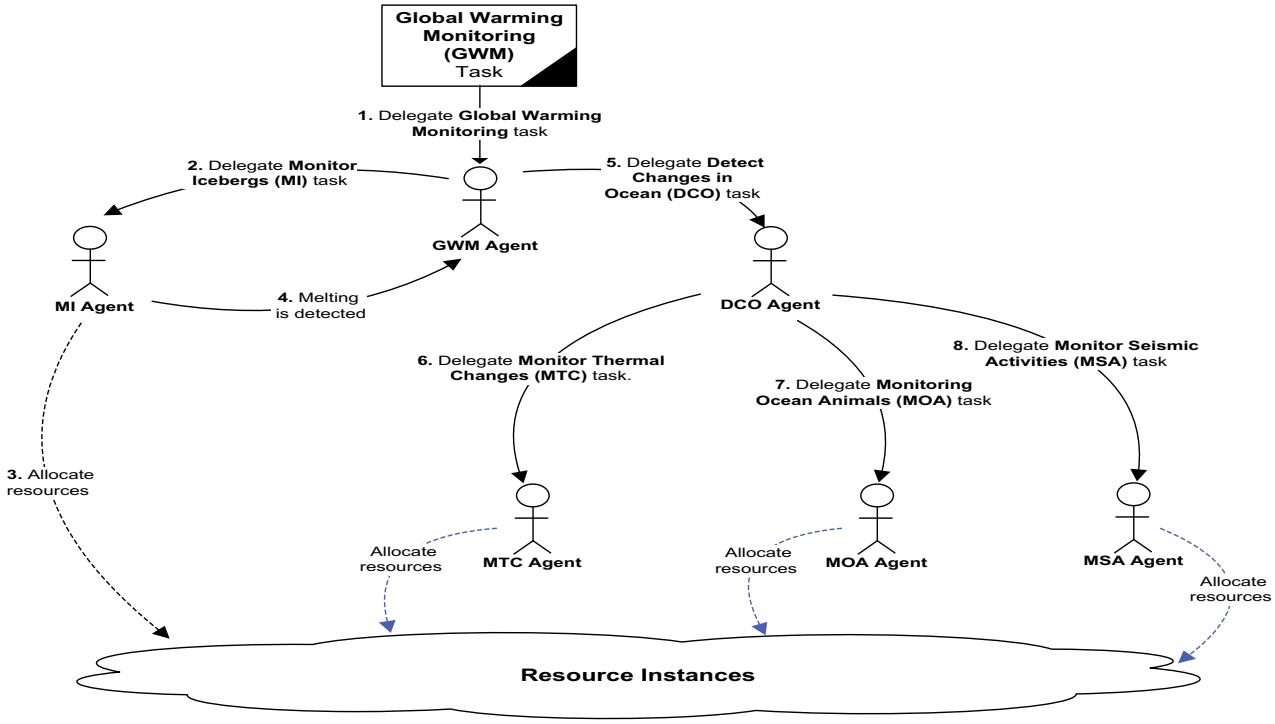

Fig. 2. A partial scenario for **Global Warming Monitoring** task of Fig. 1.

of tasks and determination of required resource types for a specific task.

3. Knowledge representation

In this section, first we propose a means to semantically represent tasks using OWL-DL (Smith et al., 2004) ontologies. Then, we formally describe the relationships among tasks and resources.

3.1. Semantic representation of tasks

We represent a composite task as a workflow composed of a set of subtasks with temporal and logical relationships between them (e.g., sequence, if-then-else and so on). In order to provide a formal grounding to our model of semantic task workflows, we build upon the OWL-S process ontology (Martin et al., 2004) combined with an OWL-DL (Smith et al., 2004) ontology describing task resourcing. The task resourcing ontology defines the relationship between tasks and resources. That is, it asserts the following statements:

(a) A task can be atomic or composite.
(b) An atomic task is an atomic process defined by the OWL-S process ontology, while a composite task is a composite process,
(c) Each atomic task can be associated with a matchmaking mechanism, which can be used to determine types of resources the task may need,
(d) Each task may directly require capabilities or has requirements that are satisfied by certain capabilities,
(e) An individual resource type or a set of resource types may provide certain capabilities.

In summary, the task resourcing ontology describes the relationships between resources and tasks requiring those resources, while the OWL-S process ontology describes the relationships among tasks. Specifically, this ontology states that tasks have requirements that are satisfied by the capabilities of resources. We note that this ontology is domain independent. In order to represent tasks and resources in a specific domain, we need a ontology that captures these properties for the domain. Software agents can then interpret and reason about task and resource types within the domain.

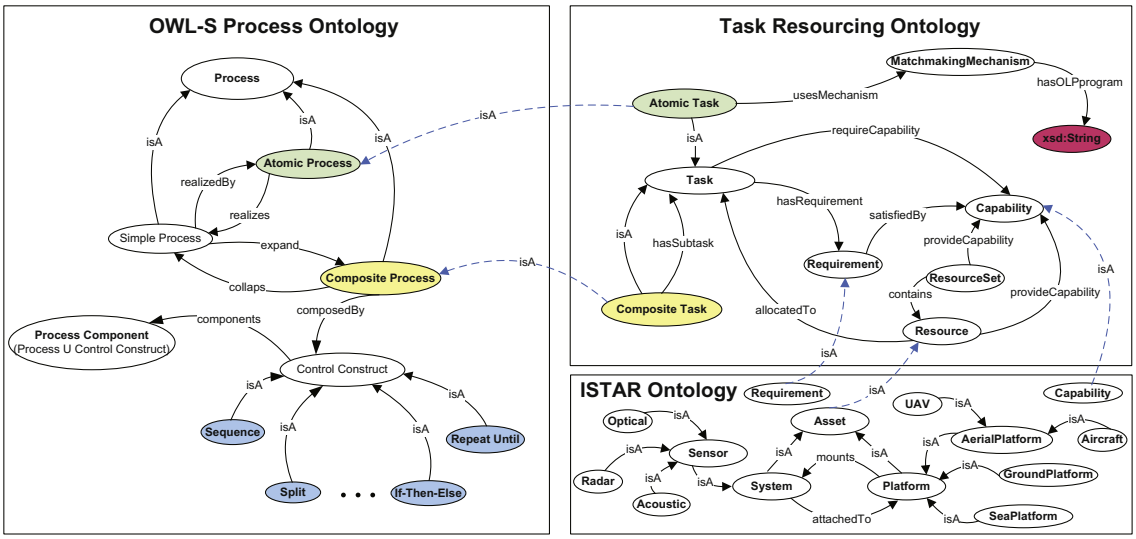

Fig. 3. Combination of OWL-S ontology with the task resourcing and ISTAR ontologies.

The OWL-S process ontology is designed for the description Web Service processes as semantic workflows (Martin et al., 2004). OWL-S defines three categories of process: simple, atomic and composite. Composite processes are described using atomic or other composite processes using temporal and logical relationships (e.g., sequence, split, if-then-else and so on). In our context, atomic tasks are considered as atomic processes. Similarly, composite tasks are considered composite processes that are composed of other processes. Thus, we can combine the OWL-S process ontology with the task ontology and domain specific ontologies in an intuitive manner to describe tasks recursively with formal semantics.

Our approach is flexible enough to work with different domain ontologies. However, to ground our presentation, we address the *Intelligence, Surveillance, Target Acquisition and Reconnaissance* (ISTAR) domain[1] in the rest of the paper. In order to describe tasks in this domain, an ISTAR ontology is combined with the task resourcing and OWL-S process ontologies as shown in Fig. 3. In the ISTAR ontology, the *Asset* concept is defined as a sub-concept of the *Resource* concept in the task ontology. The *Platform* and *System* concepts are both assets, but systems can be attached to platforms. Sensors are regarded in the ontology as a specialization of systems. We note that the ISTAR ontology shown in Fig. 3 contains only core concepts and relationships. However, it is easily extended by adding other OWL ontologies to further elaborate on different concepts.

Each task may have requirements, which are used to select the most appropriate resources for them. The requirements of a composite task are inherited by its subtasks. Requirements can be associated with the task in at least three ways. First, the task can be defined abstractly in an ontology together with its default requirements. Second, new requirements can be explicitly placed onto the task during design time. Third, constraints defined within the context of the task may add new requirements to the task, or modify its existing requirements. Fig. 4 shows how an atomic task *Road Surveillance* may be defined. This task has one operational requirement[2], namely *Constant Surveillance*, and one intelligence requirement, namely *Imagery Intelligence* (IMINT) capability. If we assume that a road surveillance task is an instance of this task, then it inherits these two requirements.

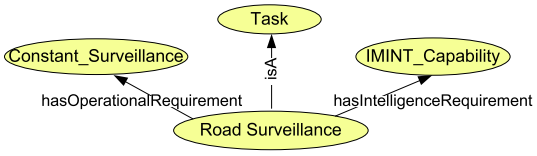

Fig. 4. Abstract task example.

Let us suppose that the road surveillance task to be executed in a mountainous area during the winter. The constraints imposed on the road surveillance task affect its requirements as follows. First, a *high altitude* requirement is added, because this task will be executed in a mountainous area. Second, because the road surveillance will be carried out during the winter (when snow, rain and fog are highly probable and imagery vision can be badly affected), *Radar Intelligence* (RADINT) is added to the requirements of the task. Fig. 5 shows the resulting road surveillance task instance along with its full set of requirements.

As explained above, constraints may affect the requirements of a task. For this purpose, in the domain ontology, we have to use semantic rules (SWRL, 2004) to represent the relationships between constraints (e.g., terrain and weather conditions) and requirements (e.g., high altitude and radar intelligence). These rules capture the domain knowledge critical to the overall tasks.

3.2. Resources required by tasks

Allocating resources to a task corresponds to allocating resources to its subtasks and, ultimately corresponding atomic tasks. Therefore, henceforth we mainly describe our approach using atomic tasks. For this reason, here we typically refer to atomic tasks. Each atomic task may require different types of resources, due to their differing requirements. Moreover, some atomic tasks may have requirements that cannot be met by a single resource type. In those cases, different resource types should be allocated together to meet the requirements of an atomic task. We use the term *Deployable Configuration* in order to refer to the set of resource types that an atomic task needs. Deployable configurations are defined formally in a domain-independent way as follows. First, we make use of three finite and non-empty sets:

– Resource *types* $T = \{t_1, \ldots, t_n\}$

[1] http://www.en.wikipedia.org/wiki/ISTAR

[2] The object properties *hasOperationalRequirement* and *hasIntelligenceRequirement* are defined in the ISTAR ontology as sub-properties of *requireCapability*.

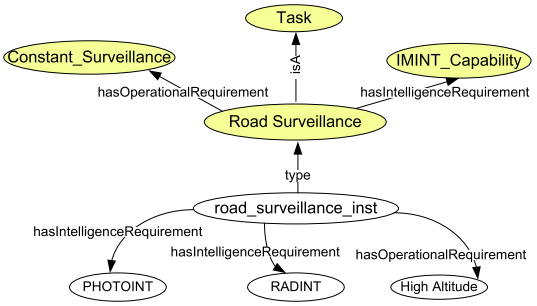

Fig. 5. Task instance example.

- Resource *capabilities* $C = \{c_1, \ldots, c_m\}$
- Task *requirements* $R = \{r_1, \ldots, r_p\}$

Sample sets for the ISTAR domain are.

- T = {*GlobalHawk*, *EOCamera*, *Reaper*, *DayLightTV*}
- C = {*LargeAreaCoverage*, *NightVision*, *HighResImage*}
- R = {*HighAltitude*, *IMINT*}

A set of types $T' \subseteq T$ is formally related via the function κ to a set of capabilities $C' \subseteq C$, that is, $\kappa : 2^T \mapsto 2^C$. This formalization aims at capturing dependencies between resource types while providing certain capabilities – when resource types are put together, they provide *combined* capabilities, as in, for instance, {*GlobalHawk*, *EOCamera*} provides {*LargeAreaCoverage, HighResImage*} but {*GlobalHawk*} only offers an empty set of capabilities. Second, a set of requirements $R' \subseteq R$ is formally related via the function σ to a *set of sets* of capabilities $\{C'_0, \ldots, C'_q\}, C'_i \subseteq C, 0 \leqslant i \leqslant q$, that is, $\sigma : 2^R \mapsto 2^{2^C}$. This formalization aims at capturing another important aspect – a set of requirements can be met by various different capabilities put together. An example of this is how the requirement {*IMINT*} can be met differently by {*NightVision*} or {*HighResImage*}.

Lastly, a *deployable configuration* is a set of resource types providing the necessary (and sufficient) capabilities for a set of requirements. More formally, we have:

Definition 1. Given a set of requirements $R' \subseteq R$ with associated capability sets $\sigma(R') = \left\{C'_0, \ldots, C'_q\right\}$, a deployable configuration $DC \subseteq T$ is a set of types such that, for at least one $C'_i, 0 \leqslant i \leqslant q, C'_i \sqsubseteq \kappa(DC)$ which means that each capability $c' \in C'_i$ is semantically subsumed by a capability $c \in \kappa(DC)$. Moreover, for any proper subset of a deployable configuration $DC_j \subset DC$, $DC_j \neq DC$, there is no $C'_i, 0 \leqslant i \leqslant q$, such that $C'_i \sqsubseteq \kappa(DC_j)$.

The definition above forges the *necessary* (first part of the definition) and *sufficient* conditions (second part of the definition) for deployable configurations. These must be minimal: only those essential types should be in the configuration and nothing else.

As illustrated in Example 3 below, there may be different ways of executing an atomic task using different types of resources. In the definition of deployable configurations, we keep the functions κ and σ as abstract as possible, as they should be defined differently for distinct domains. This means that the mechanism used to determine deployable configurations of an atomic task may change for different tasks. In some settings, each resource type may be independent and provide specific capabilities. In other settings, however, resource types may depend on one another to provide capabilities. Examples 4 and 5 below demonstrate how the determination of deployable platforms may vary in distinct settings.

Example 3. Assume that an atomic task has two deployable configurations: [*GlobalHawk, EOCamera*] and [*Reaper, Daylight TV*], where *GlobalHawk* and *Reaper* are autonomous Unmanned Aerial Vehicles (UAVs) while *EOCamera* and *DaylightTV* are sensor types. This means that there are two different ways of executing this task. The first way is to use only the resource types *GlobalHawk* and *EOCamera*. Alternatively, another way is to use only the resource types *Reaper* and *DaylightTV*. The task may select one of these configurations before allocating resources. The selection may depend on the utility of these configurations for the task.

Example 4. In stationary wireless sensor networks, the capabilities of a sensor type may not depend on other sensor types. For instance, a thermal sensor does not depend on other types of sensors to sense thermal activity within its range. In this setting, capabilities provided by sensor types are additive.

Example 5. In mobile sensor networks, sensors are attached to platforms such as UAVs, Autonomous Underwater Vehicles (AUVs), autonomous robots and so on. Hence, they can move within the region of interest and provide the required sensing information. Platforms have a pre-defined number of slots onto which particular kinds of sensors can be attached. There are domain-specific constraints that determine what platforms can be used with which sensors to provide a specific capability. For instance, a surveillance task may require imagery intelligence. This requirement can be met by sensors that provide the imagery intelligence, however the task may also need platforms to carry and support those sensors. Let us assume we have only two types of platforms that provides constance surveillance capability: *GlobalHawk* and *Reaper*. Additionally, we have only three types of sensors with imagery intelligence capability: *EOCamera*, *IRCamera*, and *DaylightTV*. *GlobalHawk* can only carry and support sensor types *EOCamera* and *IRCamera*, while *Reaper* can only mount *DaylightTV*. As a result, we can only compose three different deployable configurations for the task: [*GlobalHawk, EOCamera*], [*GlobalHawk, IRCamera*], and [*Reaper, DaylightTV*].

Determination of deployable configurations for a specific task requires extensive domain knowledge and expertise as the previous examples illustrate. In the following section, we propose a combination ontological reasoning and Logic Programming to provide a flexible solution for the computation of deployable configurations.

4. Ontological Logic Programming

Determining deployable configurations for a specific task can be modelled as a semantic matchmaking problem where sets of resource types are matched to tasks. During matchmaking, we may use ontological reasoning to determine the best combinations of resources that provide capabilities required by tasks.

Examples 4 and 5 highlight the fact that different tasks may require different mechanisms for computing deployable configurations. Assuming that a generic matchmaking mechanism is able to handle all tasks is not realistic and can be very restrictive in many settings. Instead, in this section, we propose to enable different tasks to use different matchmaking mechanisms. For this purpose, we have developed Ontological Logic Programming (OLP), which is a novel combination of ontological reasoning and Logic Programming. In order to represent matchmaking mechanisms, we have included the *MatchmakingMechanism* concept in the task resourcing ontology. Instances of this concept are associated with OLP programs that, given a task, compute appropriate deployable configurations. In this way, different matchmaking algorithms can be implemented using OLP and accommodated in an ontology. Different tasks are associated with different matchmaking mechanisms through the object property *hasMechanism*, whose domain is

the *Task* concept while its range is the *MatchmakingMechanism* concept. In order to find the most adequate resource types for a specific task, our matchmaking algorithm simply retrieves the matchmaking mechanism related to the task and executes the associated OLP program. In the following sections, we describe OLP in detail.

4.1. Architecture

Fig. 6 shows the stack of technologies and components used to interpret OLP programs. At the top of the stack, we have the OLP interpreter, which sits on a Logic Programming (LP) layer. The LP layer is handled by a Prolog engine. The Prolog engine uses two different knowledge bases; one is a standard Prolog knowledge base of facts and clauses while the other is a semantic knowledge base composed of OWL-DL ontologies and SWRL rules. Pellet (Sirin, Parsia, Grau, Kalyanpur, & Katz, 2007) has been used as the DL reasoner to interface between the Prolog engine and the semantic knowledge base.

Our choice of LP language is Prolog and in this work, we use a pure Java implementation, *tuProlog* (Piancastelli, Benini, & Omicini, 2008). The OLP interpreter is a Prolog meta-interpreter with a set of OLP-specific predicates, described in Section 4.2. Fig. 7 shows a simplified version of the OLP interpreter used to evaluate OLP programs through the *eval*/1 predicate. While interpreting OLP programs, the system behaves as if it is evaluating a standard Prolog program until it encounters an ontological predicate. In order to differentiate ontological and conventional predicates, we use name-space prefixes separated from the predicate name by a colon, i.e., ":". For example, we can directly use ontological predicate *task:requireCapability* in an OLP program without the need to define its semantics, where *task* is a name-space prefix that refers to http://www.csd.abdn.ac.uk/task_resourcing#. The Prolog knowledge base does not have any knowledge about ontological predicates, since these predicates are not defined in Prolog; they are described separately in an ontology, using DL (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2003). In order to interpret ontological predicates, the OLP interpreter needs ontological reasoning services provided by a DL reasoner. Hence, we have a DL reasoning layer below the LP layer. The interpreter accesses the DL reasoner through the *dl_reasoner*/1 predicate as shown in Fig. 7. This predicate is a reference to a Java method, which queries the reasoner and evaluates the ontological predicates based on ontological reasoning. For example, when the meta-interpreter encounters *task:requireCapability (D, R)* during its interpretation of an OLP program, it queries the DL reasoner to get direct or inferred facts about the predicate, because *task:requireCapability* is an ontological predicate. The *requireCapability* predicate is defined in the task resourcing ontology, so the reasoner interprets its semantics to infer direct and derived facts about it. Using this inferred knowledge, the variables *D* and *R* are unified with the appropriate terms from the ontology. Then, using these unifications, the interpretation of the OLP program is resumed. Therefore, we can directly use the concepts and properties

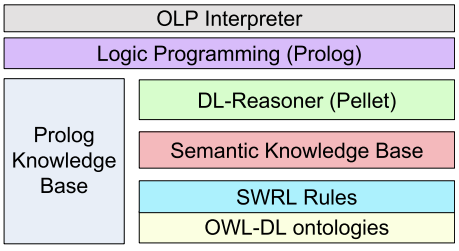

Fig. 6. OLP Stack.

```
:- op(550,xfy,':').
eval((O:G)) :- dl_reasoner((O:G)).
eval(assert((O:G))) :- assert_into_ontology((O:G)).
eval(retract((O:G))):- retract_from_ontology((O:G)).
eval(not(G))  :- not(eval(G)).
eval((G1,G2)) :- eval(G1),eval(G2).
eval((G1;G2)) :- eval(G1);eval(G2).
eval(G) :- not(complex(G)),(clause(G,B),eval(B);
           not(clause(G,_)),call(G)).
complex(G) :- G=not(_);G=(_,_);G=(_;_);G=(_:_);
              G=assert(_:_);G=retract(_:_).
```

Fig. 7. Simplified Prolog meta-interpreter for OLP interpreter.

from ontologies while writing logic programs and the direct and derived facts are imported from the ontology through a reasoner when necessary. In this way, OLP enables us to combine the advantages of Logic Programming (e.g., complex data types/structures, negation-as-failure and so on) and ontological reasoning. Moreover, the use of Logic Programming enables us to easily extend the OLP interpreter to provide, together with answers, explanations of the reasoning which took place (Sterling & Yalcinalp, 1989).

4.2. Semantic knowledge and OLP

OLP not only uses the semantic knowledge within ontologies, but it may also modify this knowledge by importing new ontologies, and adding or removing concepts, roles, individuals and facts (i.e., RDF statements Smith et al., 2004). Here, we outline how OLP may be used to modify the semantic knowledge base:

(a) **Importing ontologies.** In a classical Prolog program, domain knowledge is encoded as a part of the Prolog knowledge base. To facilitate the reuse of standardized domain ontologies, OLP enables Prolog programs to directly refer to predicates defined in ontologies. An OLP program may import a number of ontologies to access the domain knowledge encoded within them. We provide two mechanisms to do this. First, at the beginning of an OLP program, lines starting with *%import* are interpreted as an instruction to import an ontology located at a specific URI (note that these lines start with *%*, so they are regarded as comments by the Prolog engine). Second, the *import_ontology* predicate can be directly used within an OLP program to dynamically import new ontologies.

(b) **Addition and removal of statements.** As shown in Fig. 7, the OLP interpreter evaluates *assert* and *retract* predicates differently depending on whether these are ontological and non-ontological facts. If *assert* is used with an ontological statement as in *assert (istar:'Sensor'(olp:x))*, the *assert_into_ontology* predicate is used by the interpreter to add this statement to the semantic knowledge base. That is, the semantic knowledge base is modified by declaring *olp:x* as an instance of the *Sensor* concept. On the other hand, if *assert* is used with non-ontological predicates as in *assert* (*deployable* (*istar:'GlobalHawk'*)), a new fact is added to the underlying Prolog knowledge base. It should be noted that the addition of a new statement to the semantic knowledge base may make it inconsistent. For example, the addition of the statement *rdf:subConceptOf (istar:'Platform', istar:'Sensor')* results in an inconsistent semantic knowledge base, because the *Sensor* and *Platform* concepts in the ISTAR ontology are defined as disjoint concepts. Therefore, before adding the statement, *assert_into_ontology* checks whether the addition would result in an inconsistency. If the addition would

result in an inconsistency, *assert_into_ontology* returns *false* without adding the statement. Otherwise, it modifies the knowledge base and returns *true*. The *retract* predicate works in a similar way: ontological facts are removed from the underlying semantic knowledge base using the *retract_from_ontology* predicate, while others are removed directly from the Prolog knowledge base.

(c) **Addition and removal of individuals.** New individuals can be created using the *create_individual* predicate. For example, *create_individual* (*istar:mini*) creates the individual *mini* within the name-space *istar* as an instance of *owl:Thing*. Then using *assert (istar:'UAV'(istar:mini))*, we can declare that *istar:mini* is an Unmanned Aerial Vehicle. Moreover, using the *remove_individual* predicate, we can remove an individual and all statements about that individual from the semantic knowledge base (e.g., *remove_individual (istar:mini)*).

(d) **Addition and removal of concepts.** Through the *create_concept* predicate, a new OWL-DL concept can be created based on a DL class description. If the described concept is not satisfiable, the predicate returns *false* without creating the concept; otherwise it returns *true* after creating the concept. A concept description is an OWL-DL class expression (Smith et al., 2004), which can be a single concept name, a restriction on properties, or created using the intersection or the union of two class expressions or the complement of a class expression. A restriction on a property can be specified through *someValuesFrom*, *allValuesFrom*, *minCardinality*, *maxCardinality* and exact *cardinality* restrictions (Smith et al., 2004). The inverse of a property can also be used in a concept description. Moreover, a concept can be described by enumerating all of its instances; i.e., an enumerated class (Smith et al., 2004). Table 1 shows examples of concept descriptions. OLP also allows the removal of concepts from the semantic knowledge base using *remove_concept* predicate. When this predicate is used, not only the concept but also all statements about the concept are removed from the semantic knowledge base.

In the next sub-section, we briefly describe how OLP can be used to implement different matchmaking mechanisms for different tasks.

4.3. Resource-task matchmaking with OLP

Fig. 8 shows a simple matchmaking mechanism in OLP. The OLP program in the figure is a Prolog program, where concepts and properties from the underlying ontologies are referenced directly. This OLP program defines a simple matchmaking mechanism, where the *getConfigurations* predicate computes deployable configurations for a specific task. The program implements an incremental algorithm, which starts with an empty set and iteratively attempts to add new resource types to this set if the resource type provides a required capability, which is not provided by the resource types in the current set. The algorithm assumes that resource types are independent and capabilities are additive. Hence, this algorithm can be used to determine deployable configurations for Example 4. Recall that, in our ontology, OLP programs are associated with instances of the *MatchmakingMechanism* concept using a data type property and tasks are associated with different matchmaking mechanisms using *hasMechanism* object property.

Table 1
Simple concept description examples.

Concept Description	Satisfiable
istar:'Sensor'	Yes
(*istar:'Sensor'; istar:'Platform'*)	Yes
(*istar:'Sensor', istar:'Platform'*)	No
Enum (*istar:'GlobalHawk', istar:'EOCamera'*)	Yes
(*istar:'Sensor', not* (*enum* (*istar:'EOCamera'*))	Yes
Value (*istar:'mounts', istar:'EOCamera'*)	Yes
All (*inverse* (*istar:'mounts'*), *istar:'UAV'*)	Yes
Some (*istar:'attachedTo', istar:'UAV'*)	Yes

```
%import http://www.csd.abdn.ac.uk/~murat/ita/istar.owl
getConfigurations(Task,Sensors):-
   extendSolution(Task,[], Sensors).
extendSolution(T,Prev,Next):-
   requireSensor(T,Prev,X),
   A=[X|Prev],
   extendSolution(T,A,Next).
extendSolution(T,S,S):-
   not(requireCapability(T,S,_)).
requireSensor(T,S,X):-
   requireCapability(T,S,C),
   istar:'Sensor'(X),
   task:'provideCapability'(X,C).
requireCapability(T,S,C):-
   task:'requireCapability'(T,C),
   not(provideCapability(S,C)).
provideCapability([Y|Tail],C):-
   task:'provideCapability'(Y,C),!;
   provideCapability(Tail,C).
```

Fig. 8. A matchmaking mechanism example, where a deployable configuration is composed of sensor types whose capabilities are additive. The function *getConfigurations* computes deployable configurations for a specific task.

Although our example in Fig. 8 is simple, it is straightforward to create sophisticated matchmaking mechanisms for the cases where resources are co-dependent and their capabilities are not additive. For example, the OLP program in Fig. 9 computes deployable configurations for settings such as those in Example 5. In such settings, each sensor must be carried by an available platform that provides all of the operational requirements of the task (e.g., constant surveillance). If a sensor cannot be carried by an available platform, there is no point in considering deployable configurations with that sensor type. Using this knowledge, a tailored and efficient matchmaker can be employed. This matchmaker first identifies the deployable platforms that meet the requirements of the task. Once many possibilities are narrowed down by determining deployable platforms, the sensor types that provide the intelligence capabilities required by the task are determined so that those sensors can be mounted on the deployable platforms.

5. Case-studies

Our approach is implemented using Java; tuProlog is used as a Prolog engine, Jena is used as a Semantic Web framework (McBride, 2002), and Pellet is used as an OWL-DL reasoner (Sirin et al., 2007). We ran our experiments using a 2.16 GHz Intel Core Duo PC with 2 GB RAM. In this section, we demonstrate four case-studies to show the usefulness of the proposed approach. In the first case-study, we empirically compare the proposed approach with an exhaustive search approach used in a real-life application. Note that the proposed approach is designed as a tool for intelligent agents to reason about tasks and resources. Therefore, in the second case-study, we demonstrate how intelligent agents equipped with the proposed techniques can support planning activities of human experts. In the third case-study, we demonstrate how task agents may use the proposed reasoning mechanisms to cooperatively enhance their performance by sharing their resources. In the last case-study, we show how our approach can be used in other application domains.

```
%import http://www.csd.abdn.ac.uk/~murat/ita/istar.owl
getConfigurations(T,[P|S]):-
   deployablePlatform(T,P),
   extendSolution(T,P,[],S).
deployablePlatform(T,P):-
   istar:'Platform'(P),
   not((istar:'requireOperationalCapability'(T,C),
       not(task:'provideCapability'(P,C)))).
extendSolution(T,P,Prev,Next):-
   requireSensor(T,P,Prev,X),
   istar:'mounts'(P,X),
   A=[X|Prev],
   extendSolution(T,P,A,Next).
extendSolution(T,P,S,S):-
   not(requireCapability(T,P,S,_)).
requireSensor(T,P,S,X):-
   requireCapability(T,P,S,C),
   task:'provideCapability'(X,C).
requireCapability(T,P,S,C):-
   task:'requireCapability'(T,C),
   not(provideCapability(S,C)),
   not(provideCapability([P],C)).
provideCapability([Y|Tail],C):-
   task:'provideCapability'(Y,C),!;
   provideCapability(Tail,C).
```

Fig. 9. A matchmaking mechanism example, where a deployable configuration is composed of platform and sensor types whose capabilities are not additive; instead sensors and platforms are interdependent.

5.1. Sensor assignment to missions

The International Technology Alliance[3] (ITA) is a research program initiated by the UK Ministry of Defence and the US Army Research Laboratory. ITA focuses on the research problems related to wireless and sensor networks. One of these research problems is the selection of appropriate sensing resources for ISTAR tasks. In order to solve this problem, we have previously implemented a system called *Sensor assignment to missions* (SAM). The first version of SAM (Gomez et al., 2008) uses a minimal set covering algorithm to compute deployable configurations for an ISTAR task. That algorithm enumerates all possible sets of asset types so that each set has at most n members. Then, a set is regarded as a deployable configuration of the task if it satisfies its requirements. This, rather standard, approach is based on an exhaustive search algorithm and implemented in Java using Pellet for DL reasoning. Here, we demonstrate how SAM has been significantly improved using OLP.

In SAM version 2.0, we employ the more flexible approach described in this paper, where the OLP program shown in Fig. 9 to compute deployable configurations. We have compared the OLP-based matchmaker with the exhaustive search approach (SAM v1.0) in terms of time complexity. For this purpose, we randomly created 908 tasks using the ISTAR ontology. Fig. 10 shows our results, where the *x-axis* is the maximum number of items in deployable configurations and the *y-axis* is the average time consumed by each approach to find all of the deployable configurations of a task. When the maximum size of deployable configurations increases, the OLP-based approach outperforms the exhaustive search approach significantly; time complexity of the exhaustive search increases exponentially while that of the proposed approach remains almost linear within the range of maximum deployable configuration sizes explored (it should be noted that for this domain, this is a reasonable range for most realistic problems). These results are intuitive because the OLP program of Fig. 9 is based on the idea

[3] http://www.en.wikipedia.org/wiki/International_Technology_Alliance.

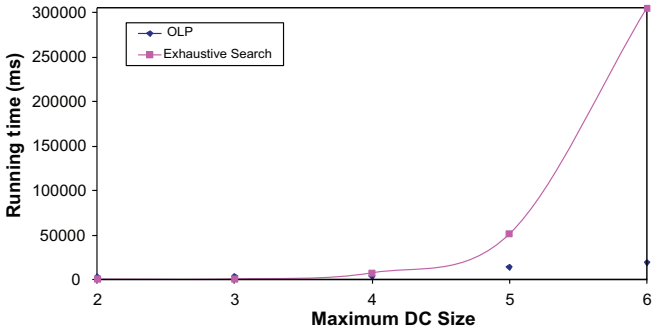

Fig. 10. Comparison between OLP program of Fig. 9 and exhaustive search algorithm to find deployable configurations.

that the search space can be significantly reduced using domain knowledge; i.e., dependencies between sensors and platforms: not every type of sensors can be used with a specific type platform. Using this principle, at each iteration, it rules out many combinations and significantly reduces the time required to compute deployable configurations.

5.2. Supporting planning activities

Planning is the process of producing a (partially ordered) sequence of actions to achieve a set of goals. A given set of goals may be achieved in various ways. A plan can be considered as a composite task, which is composed of smaller tasks with temporal and logical relationships among them. A task may fail if some of its atomic sub-tasks fail. An atomic task may fail because of various reasons, one of which is the unavailability of the resources meeting the task's requirements. Some possible reasons for the unavailability and their resolution using the proposed approach are shortly explained below:

(i) **Overloaded requirements.** if an atomic task has too many requires, then there may not be any deployable configuration meeting these requirements. Consider the mobile sensor networks described in Example 5. Assume an atomic task has *Constant_Surveillance* and *High_Altitude* as operational requirements and *PHOTOINT* and *RADINT* as intelligence requirements. Unfortunately, there is no platform that meets all of the operational capability requirements as well as being able to mount the sensors providing the required intelligence capabilities. This means that the requirements of this task cannot be met by any deployable configuration. Once this is detected by an agent, the planner can be alerted and the situation can be resolved by either relaxing the requirements of the task or converting the task into a composite task. The composite task should consist of atomic tasks, each of which only has a subset of the original task's requirements.

(ii) **Policies.** Some policies may prohibit the use of specific types of resources. For example, let us assume that a policy prohibits the use of *Global_Hawk* in military operations. Then, this policy makes deployable configurations containing *Global_Hawk* useless if they belong to military-related tasks. Our approach can be used to eliminate deployable configurations containing prohibited resource types. If all of a task's deployable configurations are eliminated, then this task cannot be achieved. In this case, as explained above, the agent may ask the planner to change the requirements of the task.

(iii) **Scarcity of resources.** In many settings, critical resources are scarce. If all instances of a specific resource type are allocated by other tasks, a task cannot use deployable configurations

containing this resource type. If all of the deployable configurations of a task contain this resource type, then the planner is alerted to relax some requirements of the task. An alternative to relaxing requirements can be the sharing of scarce resources among tasks.

5.3. Promoting resource sharing

In this paper, we combine ontological reasoning and Logic Programming to enable flexible matchmaking of resource types to tasks. In this section, we illustrate how the proposed methods fit in (and possibly improve) classical solutions for coordination, cooperation, and competition among intelligent software agents. For this purpose, we select a specific case-study in which intelligent software agents use the proposed mechanisms to cooperatively determine the best deployable configurations to promote resource sharing among tasks, which leads to an improvement in the number of executable tasks when available resources are limited.

5.3.1. Method

Each task is represented by an agent as introduced in Section 2. When an atomic task is delegated to a task agent, the agent uses the proposed approach to compute deployable configurations. Before allocating resources, the agent has to select one deployable configuration. Instead of selecting it individually, the agent selects this deployable configuration cooperatively as follows. First, on a message board, the task agent publishes its desired deployable configurations (DCs) together with the task information (e.g., owner of the task, date, location, duration and so on). Then, the agent lets other task agents vote for DCs. An agent's vote for a specific deployable configuration is based on the utility of sharing the assets of the deployable configuration. Fig. 11 shows votes of a task agent for three different deployable configurations, where each deployable configuration is composed of a platform and a set of sensors that can be mounted by the platform. Lastly, depending on the voting results, each task agent decides on a deployable configuration that will be used during its executions. That is, task agents select deployable configurations that enable them to share as many resources as possible with others. After selecting a deployable configuration, the agent allocates resources accordingly and shares these resources with the ones that vote for the deployable configuration. As a result, some voting agents do not have to allocate all the resources they need, because some of these resources are shared. The sharing depends on many constraints such as policies, date/time, location, properties of resources and so on. Details of this approach are explained in Şensoy, Vasconcelos, and de Mel (2009).

5.3.2. Experiments

We have randomly created a set of composite tasks as described in Section 3. This set includes 908 atomic tasks in total. The requirements and constraints of each task are set so that they will have at least 4 deployable configurations. Hence, tasks can choose between different alternatives using the votes for those configurations. In this section, we show how the proposed approach enables tasks to achieve their goals with fewer resources by promoting resource sharing.

For two atomic tasks to cooperate (that is, to share their resources), they should be able to cooperate, which means that there should not be any reason preventing these tasks from cooperating. Policies or conflicting constraints may prevent cooperation. This leads us to determine a parameter in our experiments, namely the *ratio of tasks adequate for cooperation* (R_{ac}). R_{ac} = 0.0 means that policies or conflicting constraints (e.g., location and date of tasks etc.) do not allow a task to cooperate by sharing its resources with other tasks in the system. On the other hand, R_{ac} = 1.0 means that policies and constraints are created so that each task is adequate to share its resources with any other task in the system. Note that R_{ac} defines only the adequacy to share resources, but not the actual degree of resource sharing. That is, if two tasks require different resources, they cannot share their resources even though they can cooperate in terms of their policies or constraints.

For comparison reasons, we also implement a naive approach to select resources for the tasks. In this approach, for each task, we individually select the best resources according to the requirements and the constraints of the task. This approach does not consider cooperation (or resource sharing) between tasks. Hence, each resource is allocated to one task. Our experiments show that this leads to 2195 different resources (908 platforms and 1287 sensors) being allocated. Fig. 12 shows the total number of required resources for different values of R_{ac}, when our approach is used. When R_{ac} = 0.0, task agents may not cooperate, and so the performance of our approach is the same as that of the naive approach. However, when we increase R_{ac}, our approach enables task agents to search for possible ways of sharing their resources. This leads to a dramatic decrease in the required number of resources to carry out the tasks. For example, when R_{ac} = 0.125, many resources could be shared among tasks, so the number of required resources decreases to 819 (345 platform instances and 474 sensor instances).

If there are not enough resources, a task cannot be executed. In the next step of our evaluations, we measure how our approach improves the ratio of executable tasks when the number of available resources is limited. Fig. 13 demonstrates the ratio of executable tasks for different R_{ac} values, while the number of resources ranges between 100 and 1000. When tasks are not adequate for cooperation, only 45% of the tasks can be executed with all of the 1000 available resources (R_{ac} = 0.0). For higher values of R_{ac}, our approach enables all of the tasks to be executed with fewer resources. This is expected, since our approach enables tasks to discover opportunities for sharing resources. When R_{ac} = 0.125, the number

```
TaskAgent_1 votes 1.0 for the DC
        {Platform: Nimrod_MR2;    Sensors: [IRCamera, EOCamera]}
         Because it meets all operational requirements and
         provides [IRINTCapability, ELECTRO-OPTINTCapability]

TaskAgent_1  votes 0.66 for the DC
        {Platform: I_GNAT;            Sensors: [SAR, EOCamera]}
         Because it meets all operational requirements and
         provides [ELECTRO-OPTINTCapability], but
         cannot provide [IRINTCapability]

TaskAgent_1 votes 0.33 for the DC
        {Platform: Predator_A; Sensors: [SAR, TVCamera]}
         Because it meets all operational requirements, but
         cannot provide [IRINTCapability, ELECTRO-OPTINTCapability]
```

Fig. 11. Some voting examples.

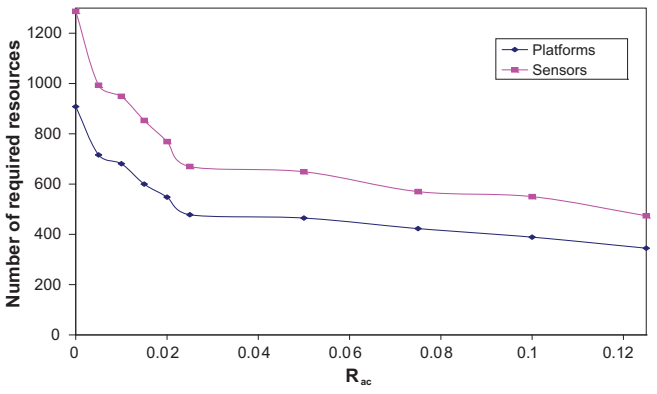

Fig. 12. Required resources vs. R_{ac}.

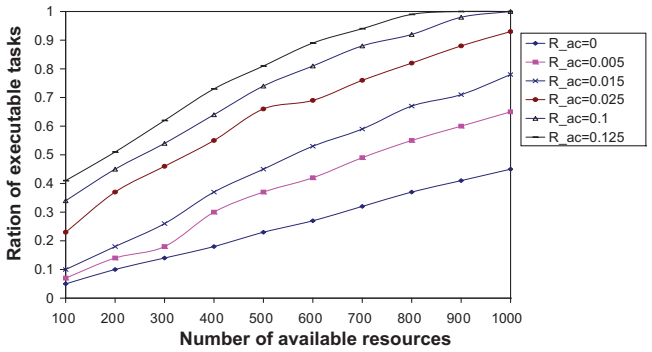

Fig. 13. Changing ratios of executable tasks.

of platform and sensor instances required to execute all of the tasks is only 819.

5.4. Team formation based on stereotypes

The proposed methods can also be used in different domains with little effort. In this section, we present a case-study to show how the proposed techniques can be used to compose medical teams to operate in emergency settings. For this purpose, we first need to define a domain ontology to describe emergency tasks, their capability requirements, and resources providing these capabilities. Fig. 14 shows an example ontology describing a *Kidney-Transplantation* task with its requirements.

In order to transplant a kidney to a patient with renal failure, we have to compose a surgery team immediately. This team should have expertise in surgery, anesthetics, and nephrology in addition to providing scrub assistance. These requirements are met by the capabilities of resources, which are doctors and nurses in this domain. For the sake of simplicity, we assume capabilities of resources are additive and do not depend on the relationships between them. Therefore, we need a matchmaking mechanism similar to the one in Fig. 8. However, in this case-study, we also consider some stereotypes that specify whether a surgery team is considered reliable (i.e., not untrustworthy) according to hospital policy.

Stereotypes are beliefs about specific social groups or types of individuals. They are an integral part of human social decision making (Meyerson, Weick, & Kramer, 1996; MacRae & Bodenhausen, 2001; Jarvenpaa & Leidner, 1998). Burnett et al. have proposed methods for agents to automatically learn stereotypes based on past experiences (Burnett et al., 2010). We can use DL to describe *untrustworthy* surgery teams based on stereotypes about medical staff. Table 2 shows some examples of stereotypes and the untrustworthy team descriptions they imply. Based on these descriptions, we can create sub-concepts of the *UntrustworthyTeam* concept. The third column in the table shows the names of these concepts.

The OLP program in Fig. 15 is a matchmaking mechanism designed for this case-study. It is basically the same program in Fig. 8, but it also considers stereotypes while composing deployable configurations, or medical teams in this case. Deployable configurations for a task are computed by the *getMedicalTeams* predicate, which gets the name of the task as input and returns a set of medical staff (the surgery team) that is suitable for the task. The computation starts with an empty set of staff, then at each iteration a new member of staff is added to the set if this member provides a capability required by the task but not provided by the other members in the set. The addition of a new member to the set may make the corresponding team untrustworthy, because of the stereotypes. Therefore, the algorithm avoids adding a specific member of staff to the set if this addition makes the corresponding team *untrustworthy*. This is tested using the *untrustworthy* predicate. Given a set of medical staff, this predicate creates a temporary team instance, whose members are the members of the set. Then, it checks whether the resulting team is also an instance of the *UntrustworthyTeam* concept or not. In this way, untrustworthy teams are detected and eliminated.

6. Discussion and related work

Fully automated reasoning mechanisms about tasks and resources enable intelligent agents to autonomously determine the most appropriate resources to achieve tasks in dynamic and time-critical settings. In this paper, we formulate a way of semantically representing tasks. This representation enables us to describe the components of a task in terms of their requirements and relationships. Additionally, we combined ontological reasoning and Logic Programming in a simple and practical way to enable different tasks to have different matchmaking mechanisms. Once the matchmaking mechanism is defined and placed in an ontology, a task agent can use it to determine deployable configurations for

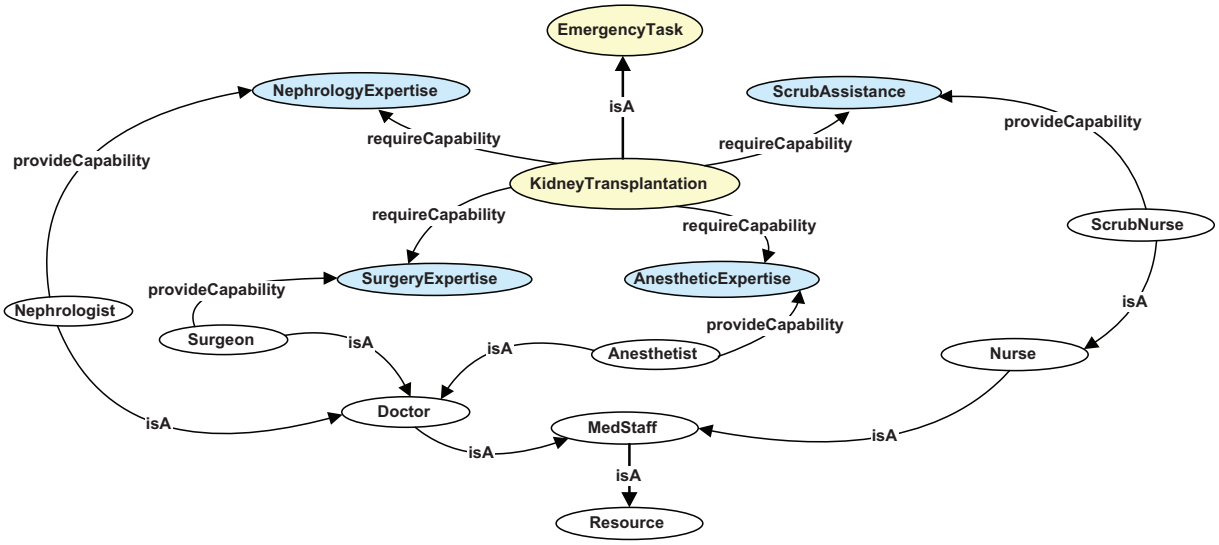

Fig. 14. Kidney transplantation task, its requirements and resources meeting these requirements.

Table 2
Some stereotypes about emergency conditions and the resulting untrustworthy medical team descriptions.

Stereotype	Untrustworthy Team Description	Corresponding Concept
Medical staff become unreliable after 12 shift hours	Team ⊓ ∃has. (MedStaff ⊓ ∃worked.[minExc (12)])	UT_0
Doctors with less than 5 years experience are untrustworthy	Team ⊓ ∃has. (Doctor ⊓ ∃medExp.[maxExc (5)])	UT_1
Doctors younger than 35 years old are untrustworthy	Team ⊓ ∃has. (Doctor ⊓ ∃hasAge.[maxExc (35)])	UT_2
Nurses with less than 3 years experience are untrustworthy	Team ⊓ ∃has. (Nurse ⊓ ∃medExp.[maxExc (3)])	UT_3
Doctors with 10 years or less experience should not work With nurses having less than 5 years experience	Team ⊓ ∃has. (Doctor ⊓ ∃medExp.[maxInc (10)]) ⊓ ∃has. (Nurse ⊓ ∃medExp.[maxExc (5)])	UT_4

```
%import http://www.csd.abdn.ac.uk/~murat/ita/med.owl
getMedicalTeams(Task,StaffSet):-
   extendSolution(Task,[], StaffSet).
extendSolution(T,Prev,Next):-
   requireStaff(T,Prev,X),
   A=[X|Prev],
   not(untrustworthy(A)),
   extendSolution(T,A,Next).
extendSolution(T,S,S):-
   not(requireCapability(T,S,_)).
requireStaff(T,S,X):-
   requireCapability(T,S,C),
   med:'MedStaff'(X),
   task:'provideCapability'(X,C).
requireCapability(T,S,C):-
   task:'requireCapability'(T,C),
   not(provideCapability(S,C)).
provideCapability([Y|Tail],C):-
   task:'provideCapability'(Y,C),!;
   provideCapability(Tail,C).
untrustworthy(StaffSet):-
   create_individual(olp:'tempTeam'),
   assert(med:'Team'(olp:'tempTeam')),
   addMembers(olp:'team',StaffSet),
   med:'UntrustworthyTeam'(olp:'tempTeam'),
   remove_individual(olp:'tempTeam');
   remove_individual(olp:'tempTeam'), fail.
addMembers(_,[]).
addMembers(Team,[H|T]):-
   assert(med:'has'(Team,H)),
   addMembers(Team,T).
```

Fig. 15. A matchmaking mechanism that computes reliable (not untrustworthy) medical teams in emergency settings.

its tasks. This flexibility enables intelligent agents to reason about resources differently for distinct tasks and contexts. In order to show how the proposed reasoning mechanisms can be used in various settings and to evaluate these mechanisms, we presented a number of case-studies. In the first case-study, we evaluate the performance of the proposed approach with respect to an exhaustive search approach. Our experiments show that the proposed approach is not only flexible but also enables efficient determination of deployable configurations. In the second case-study, we demonstrate how intelligent agents can use the proposed approach to assists planning activities of human experts. In the third case-study, we show how agents can use the proposed approach to promote resource sharing among tasks and carry out these tasks with fewer resources. In the fourth case-study, we illustrate how the proposed methods can be used in domains other than ISTAR; i.e., team formation.

Tasks introduced in this paper can be considered as workflows, whose components are described semantically using an ontology. In the literature, there are approaches that describe workflows semantically using an ontology (Wang et al., 2007). However, these approaches do not consider rules that lead to dynamic addition of new constraints and requirements to tasks, depending on the changes in the environment and outputs of other tasks.

Semantic matchmaking of resources to tasks has been studied extensively in the literature. Approaches such as Mahmood, Ahmed, and ur Rasool (2008) and Grimm and Hitzler (2008) assume that the utility of each resource type for a task is independent of others. On the other hand, in many scenarios, different types of resources should be used together to provide some utility for a task. Approaches such as Gomez et al. (2008) consider the dependencies between resource types. In these approaches, instead of individual resource types, sets of resource types are matched against tasks, which may exponentially increase the complexity of the matchmaking. Our approach does not depend on assumptions about the matchmaking mechanism, instead it enables different matchmaking algorithms to be defined flexibly using the terms from domain ontologies. Hence, unlike other approaches from the literature, our approach enables intelligent software agents to reason about tasks and resources without any specific matchmaking mechanism embedded in their architecture.

Grosof et al. propose Description Logic Programs (DLP), a combination of logic programs with Description logic (Grosof, Horrocks, Volz, & Decker, 2003). DLP relies at the intersection of Description Logic (Baader et al., 2003), Horn Logic (Padawitz, 1988) and Logic Programs (Robinson, 1992). Although it is useful to create semantic rules for ontologies, it does not support important Logic Programming features like negation-as-failures and procedural attachments. The proposed OLP stack can accommodate DLP in the ontology and rule levels.

There are other approaches that combine rules and Logic Programs with ontologies. Jess (Hill, 2003) is a Java-based expert system shell that uses a RETE algorithm (Forgy, 1990) for its forward-chaining rule reasoning engine. JessTab (Eriksson, 2003) is a bridge between Jess and Protégé (Noy, Fergerson, & Musen, 2000), which is an ontology editor and knowledge acquisition system. It enables Jess programs to use and manipulate the knowledge from Protégé knowledge bases. This is achieved by mapping Protégé knowledge bases to Jess assertions. Originally, JessTab was developed to support Protégé-Frames. That is why, JessTab includes only a limited support for handling OWL ontologies. Hence, it does not support OWL restrictions and class expressions such as *someValuesFrom* restrictions while mapping OWL ontologies to Jess assertions.

There are some other approaches based on Prolog. SweetProlog (Laera, Tamma, & Bench-Capon, 2004) is a Java-based system for translating rules into Prolog. It translates OWL ontologies and semantic rules into a set of facts and rules in Prolog. Then, the reasoning about these facts and rules are made completely in Prolog. This approach uses JIProlog (Laera et al., 2004) as a rule engine. Hence, it translates a OWL subset into simple Prolog predicates. The main limitation of SweetProlog is its expressive power as it uses DLP to enable the integration between ontology and rules. DLP is the intersection of DL and Horn logic programs, so it is less expressive then both DL and Horn logic programs. DR-Prolog (Bikakis & Antoniou, 2005) is a simple rule-based approach to reasoning with incomplete and inconsistent information. It is also based on the translation of ontological knowledge into Prolog.

The system can reason with rules and ontological knowledge written in RDF Schema (RDFS) (Manola & Miller, 2004) or OWL. This is achieved through the transformation of the RDFS constructs and many OWL constructs into rules. Note, however, that a number of OWL constructs cannot be captured. SWORIER (Samuel et al., 2008) is a system that uses Prolog to reason about ontologies and rules in order to answer queries. It translates OWL-DL ontologies with rules in SWRL into Prolog using XSLTs (Extensible Stylesheet Language Transformations). Then, query answering is done in Prolog using this translation. It supports only a subset of OWL-DL constructs.

In the approaches described above, ontological knowledge with rules is translated or mapped to Jess or Prolog assertions. On the other hand, OLP keeps ontological knowledge separated from Prolog programs and transparently delegates ontological reasoning to specialised DL reasoners such as Pellet. Hence, it can use the full power of Prolog and the existing reasoners without any loss in the ontological knowledge and expressiveness.

This paper significantly extends (Sensoy et al., 2010; Şensoy et al., 2010) as follows. First, it separates Prolog knowledge base from semantic knowledge base. While ontological knowledge is imported into Prolog knowlege base in Sensoy et al. (2010) and Şensoy et al. (2010), here we propose to keep semantic knowledge and Prolog knowledge base distinct. This distinction enables us to easily use and modify semantic knowledge using a set of predicates, as described in Section 4.2. Second, in our previous work, the proposed reasoning techniques were strictly coupled with the ISTAR ontology. Hence, it was not trivial to use these techniques within other domains. On the other hand, here we have extended these methods so that they could easily be used in various domains. Third, unlike our previous work, this paper demonstrates the applications of our approach in agent-based planning support and team formation through case-studies. Lastly, we present here a more comprehensive discussion of the proposed approach with respect to the related work.

7. Conclusions and future work

In this paper, we propose mechanisms for agents to autonomously reason about the resources they need to achieve their tasks. Our approach combines Semantic Web technologies and Logic Programming in a novel way for flexible matchmaking of resources to tasks. Using various case-studies, we demonstrated how intelligent agents can use the proposed approach in various real-life settings. Our evaluation shows that the proposed approach equips intelligent agents with useful reasoning capabilities; i.e., these reasoning capabilities lead to a better utilization of existing resources, especially if these resources are scarce. In this paper, we choose the ISTAR domain to explain the proposed approach. As future work, we plan to further explore some of our case-studies such as *team formation* and *supporting planning activities* in dynamic and uncertain environments. We believe that the flexible Semantic Web approach that OLP represents may significantly contribute to the state of the art in these problem domains. Therefore, we made OLP publicly available at http://www.olp-api.sourceforge.net.

Acknowledgements

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., & Patel-Schneider, P. F. (Eds.). (2003). *The description logic handbook: theory, implementation and applications*. Cambridge University Press.

Bikakis, A., & Antoniou, G. (2005). DR-Prolog: a system for reasoning with rules and ontologies on the Semantic Web. In *AAAI'05: Proceedings of the 20th National Conference on Artificial Intelligence* (pp. 1594–1595). AAAI Press.

Burnett, C., Norman, T. J. and Sycara, K. (2010). Bootstrapping trust evaluations through stereotypes. In Proceedings of 9th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS) (pp. 241–248).

Chevaleyre, Y., Dunne, P. E., Endriss, U., Lang, J., Maudet, N., & Rodríguez-aguilar, J. A. (2005). Multiagent resource allocation. *Knowledge Engineering Review, 20*(2), 143–149.

Eriksson, H. (2003). Using JessTab to integrate Protégé and JESS. *IEEE Intelligent Systems, 18*(2), 43–50.

Forgy, C. L. (1990). RETE: a fast algorithm for the many pattern/many object pattern match problem. (pp. 324–341).

Gomez, M., Preece, A., Johnson, M. P., Mel, G., Vasconcelos, W., Gibson, C., Bar-Noy, A., Borowiecki, K., Porta, T., Pizzocaro, D., Rowaihy, H., Pearson, G. and Pham, T. (2008). An ontology-centric approach to sensor-mission assignment. In Proceedings of the 16th international conference on Knowledge Engineering (EKAW'08) (pp. 347–363).

Grimm, S., & Hitzler, P. (2008). Semantic matchmaking of web resources with local closed-world reasoning. *International Journal of Electronic Commerce, 12*(2), 89–126.

Grosof, B. N., Horrocks, I., Volz, R., & Decker, S. (2003). Description logic programs: combining logic programs with description logic. In *Proceedings of the 12th International Conference on World Wide Web (WWW'03)* (pp. 48–57). New York, NY, USA: ACM.

He, L., & Ioerger, T. R. (2005). Forming resource-sharing coalitions: a distributed resource allocation mechanism for self-interested agents in computational grids. In *SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing* (pp. 84–91). New York, NY, USA: ACM.

Hill, E. F. (2003). JESS in Action: Java Rule-Based Systems. Manning Publications Co., Greenwich, CT, USA

Jarvenpaa, S. L., & Leidner, D. E. (1998). Communication and trust in global virtual teams. *Journal of Computer-Mediated Communication, 3*(4).

Laera, L., Tamma, V. A. M., Bench-Capon, T. J. M. and Semeraro, G. (2004). Sweetprolog: A system to integrate ontologies and rules. In Proceedings of the 3rd International Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML'04) (pp. 188–193).

MacRae, C. N., & Bodenhausen, G. V. (2001). Social cognition: categorical person perception. *British Journal of Psychology, 92*(1), 239–255.

Mahmood, S., Ahmed, H. F., ur Rasool, R. and Qadir, K. (2008). Policy-based semantic resource matchmaker for grid environment. In Proceedings of International Conference on Computer and Electrical Engineering (pp. 558–562).

Manola, F. and Miller, E. (2004). RDF primer, February 2004.

Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N. and Sycara, K. (2004). OWL-S: Semantic markup for web services.

McBride, B. (2002). Jena: A Semantic Web toolkit. *IEEE Internet Computing, 6*, 55–59.

Meyerson, D., Weick, K., & Kramer, R. (1996). Swift trust and temporary groups. In R. Kramer & T. Tyler (Eds.), *Trust in Organizations: Frontiers of Theory and Research* (pp. 166–195). Sage Publications.

Noy, N. F., Fergerson, R. W., & Musen, M. A. (2000). The knowledge model of protégé-2000: combining interoperability and flexibility. In *EKAW '00: Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management* (pp. 17–32). London, UK: Springer-Verlag.

Padawitz, P. (1988). *Computing in Horn clause theories*. New York, NY, USA: Springer-Verlag New York, Inc..

Piancastelli, G., Benini, A., Omicini, A. and Ricci, A. (2008). The architecture and design of a malleable object-oriented Prolog engine. In Wainwright, R. L., Haddad, H. M., Menezes, R. and Viroli, M. (Eds.), 23rd ACM Symposium on Applied Computing (SAC 2008) (pp. 191–197).

Robinson, J. A. (1992). Logic and Logic Programming. *Communications of the ACM, 35*(3), 40–65.

Samuel, K., Obrst, L., Stoutenberg, S., Fox, K., Franklin, P., Johnson, A., et al. (2008). Translating OWL and Semantic Web rules into Prolog: Moving toward description logic programs. *Theory and Practice of Log. Program, 8*(3), 301–322.

Şensoy, M., Vasconcelos, W., de Mel, G. and Norman, T. (2009). Selection of resources for missions using semantic-aware cooperative agents. In Proceedings of the International Workshop on Agent-based Technologies and applications for enterprise interoperability (ATOP'09) (pp. 73–84).

Şensoy, M., Vasconcelos, W. W. and Norman, T. J. (2010). Flexible task resourcing for intelligent agents. In Proceedings of 9th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS) (pp. 465–472).

Sensoy, M., Le, T., Vasconcelos, W. W., Norman, T. J., & Preece, A. D. (2010). Resource determination and allocation in sensor networks: a hybrid approach. *Computer Journal*.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., & Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *Web Semant., 5*(2), 51–53.

Smith, M. K., Welty, C. and McGuinness, D. L. (2004). OWL: Webontology language guide.

Sterling, L., and Yalcinalp, L. U. (1989). Explaining Prolog based expert systems using a layered meta-interpreter. In IJCAI'89: Proceedings of the 11th International Joint Conference on Artificial Intelligence (pp. 66–71).

SWRL: A Semantic Web Rule Language Combining OWL and RuleML, 2004. <http://www.w3.org/Submission/SWRL>.

Wang, Y., Cao, J. and Li, M. (2007). Goal-driven semantic description and query for grid workflow. In Proceedings of the Third International Conference on Semantics, Knowledge and Grid (pp. 598–599).