# Flexible Task Resourcing for Intelligent Agents *

Murat Şensoy
Computing Science
University of Aberdeen
AB24 3UE, United Kingdom
m.sensoy@abdn.ac.uk

Wamberto W. Vasconcelos
Computing Science
University of Aberdeen
AB24 3UE, United Kingdom
wvasconcelos@acm.org

Timothy J. Norman
Computing Science
University of Aberdeen
AB24 3UE, United Kingdom
t.j.norman@abdn.ac.uk

## ABSTRACT

In many applications, tasks can be delegated to intelligent agents. In order to carry out a task, an agent should reason about what types of resources the task requires. However, determining the right resource types requires extensive expertise and domain knowledge. In this paper, we propose means to automate the selection of resource types that are required to fulfil tasks. Our approach combines ontological reasoning and logic programming for a fl xible matchmaking of resources to tasks. Using the proposed approach, intelligent agents can autonomously reason about the resources and tasks in various real-life settings. Using a case-study, we describe and evaluate how agents can use the proposed approach to promote resource sharing. Our evaluations show that the proposed approach is efficien and very useful for multi-agent systems.

## Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent Systems

## General Terms

Design, Experimentation

## Keywords

Knowledge Representation, Semantic Matchmaking, Agents

## 1. INTRODUCTION

Tasks are crucial activities for organizations. The success of many organizations may depend on the appropriate execution of their tasks. In many scenarios, tasks are monitored, managed, or executed by intelligent agents (i.e., task agents) [7]. Tasks can be simple (i.e., atomic) or composite (i.e., 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:

1. **Availability of expertise:** Some organizations may not have human experts to determine the required types of resources for their tasks.
2. **Scalability:** As the number and complexity of tasks increase, the process of manually determining types of resources becomes infeasible and highly error-prone.
3. **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, as in Example 1.
4. **Intolerance to delays:** As demonstrated in Example 1, 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 failures in the tasks. However, fully automated determination of the required resource types enables task agents to act autonomously without human intervention.

EXAMPLE 1. *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 the possible threats. For this purpose, an agent responsible for the "surveillance" task decides to deploy five Unmanned Aerial Vehicles (UAVs) with optical cameras, i.e., five Global Hawks with EOCameras. Half-way through the "surveillance" task, part of the road between mountains has been covered by fog. Two UAVs 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, new UAVs with the necessary equipment should be allocated immediately. Using the new constraints, the task agent decides to deploy Global Hawks 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 Hawks with SAR sensors to resume the task in the foggy area.*

In summary, without enabling agents to reason about resources for tasks, the human experts should always be "in the loop" during the specificatio 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

*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 officia 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.

**Cite as:** Flexible Task Resourcing for Intelligent Agents, Murat Şensoy, Wamberto W. Vasconcelos and Timothy J. Norman, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. 465-472

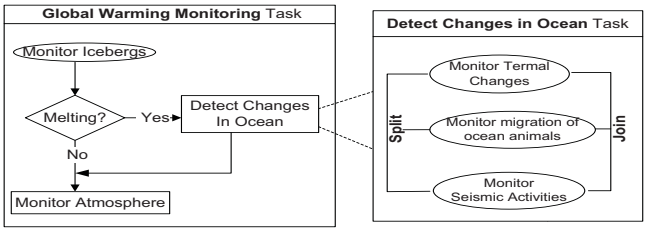


**Figure 1: A composite task example for monitoring effects of global warming (composite tasks are represented using rectangles while atomic tasks are represented using ellipses).**

in delays, which may lead to severe failures. An appropriate approach for reasoning about tasks and resources should address the following issues.

1. **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 semantic rules, because some crucial knowledge can only be captured using rules (i.e., if an area is foggy, sensors with fog penetration capability should be used to sense activities within the area).
2. **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 fl xible enough to accommodate different matchmaking mechanisms for different tasks.
3. **Expressiveness:** If a matchmaking mechanism is semantically represented, it can be interpreted by various agents to fl xibly reason about tasks and resources. However, representation of matchmaking mechanisms may require complex data types/structures, which cannot be expressed by current languages for Semantic Web.

Considering these issues, in this paper, we propose a fl xible Semantic Web approach for intelligent agents to reason about the requirements of tasks and capabilities of resources. As a result of this reasoning process, the agents can determine what types of resources they should use to achieve a specifi task.

The rest of the paper is organized as follows. In Section 2, using an example, we overview a multi-agent system (MAS), in which tasks are delegated to agents that need to reason about the resources to achieve their tasks. Section 3 describes how tasks and resources are described semantically using ontologies. Section 4 proposes our approach for fl xibly determining resource types for tasks using ontological reasoning and logic programming. Section 5 evaluates the proposed approach and illustrates, with a case-study, how the proposed approach fit in agent-based solutions to real-life problems. Lastly, Section 6 discusses the proposed approach with references to the literature.

## 2. DELEGATION OF TASKS TO AGENTS

We envisage a multi-agent system where each task is represented by a software agent, named "task agent". The task agent is responsible for the task. If a task is composed of subtasks, then that task's agent delegates those subtasks to other task agents. If a task agent represents an atomic task, then the agent is only responsible for the reasoning about the required types of resources for the 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, some 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 shareable 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 of Figure 1, Figure 2 illustrates how tasks are delegated to agents and how these agents interact in our framework. We can summarize the scenario in Figure 2 as follows:

1. The semantic description of Global Warming Monitoring (GWM) task is fed into the system.
2. The GWM task is delegated to a task agent (GWM agent), which initiates and coordinates subtasks of the GWM task using the task description.
3. 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.
4. After determining the required types of resources, the MI agent allocates resources required to achieve the task.
5. During the execution of MI task, a melting iceberg is detected and the GWM agent is informed by the MI agent.
6. As define 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 Figure 2 reveals two important challenges. The firs 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 fl w (e.g., which subtasks should be activated) and determination of required types of resources. Semantic Web technologies provide knowledge representation languages and tools to support task-level reasoning. The second challenge is the allocation of specifi 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 firs challenge. The second challenge has been extensively studied in the MAS community [3] and is not within the scope of this paper. The next two sections give details about our approach for the representation of tasks and determination of required resource types for a specifi task.

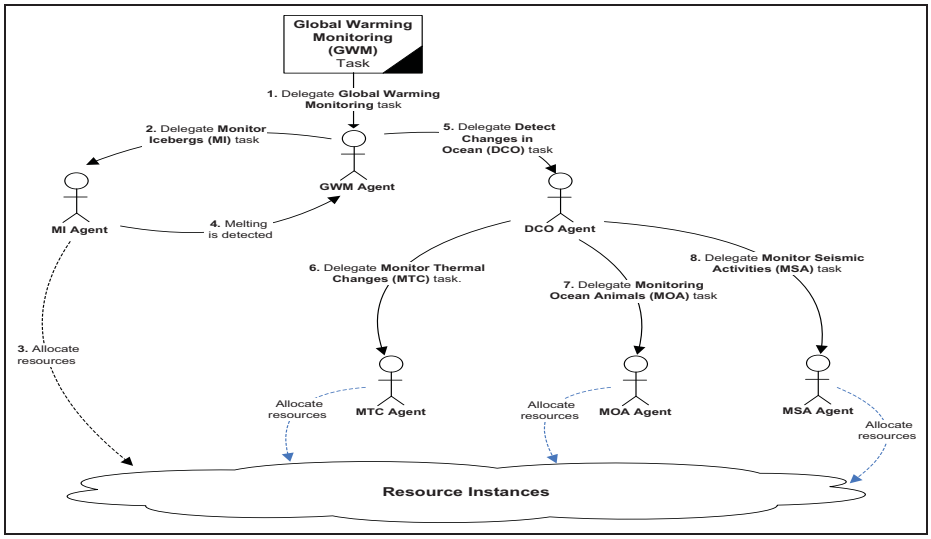


**Figure 2: A partial scenario for Global Warming Monitoring task of Figure 1.**

# 3. KNOWLEDGE REPRESENTATION

In this section, firs we propose means to semantically represent tasks using OWL-DL [11] and domain ontologies. Then, we formally describe the relationships between tasks and resources.

## 3.1 Semantic Representation of Tasks

We represent a composite task as a workfl w composed of a set of subtasks with temporal and logical relationships between them (e.g., sequence, if-then-else). In order to provide a formal grounding to our model of semantic task workfl ws, we build upon the OWL-S process ontology [8] combined with an OWL-based domain ontology that captures properties of tasks within a specifi domain. Therefore, task agents can easily understand, interpret and reason about the task representations. The OWL-S process ontology is proposed to describe Web Service processes as semantic workfl ws [8]. OWL-S define 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 domain specifi ontologies in an intuitive manner to describe tasks recursively with formal semantics.

Our approach is fl xible 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 OWL-S process ontology as shown in Figure 3. The ISTAR ontology describes the relationships between resources and tasks requiring those resources, while the OWL-S process ontology describes the relationships between tasks. More specificall , the ISTAR ontology states that each task may require capabilities to achieve its objectives and resources may provide various capabilities. The *Platform* and *System* concepts are both resources, 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 Figure 3 contains only core concepts and relationships. However, it is easily extended by adding other OWL ontologies to further elaborate on different concepts.

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

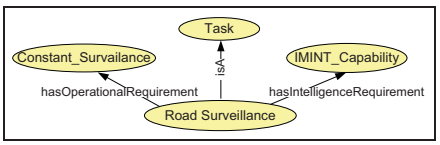


**Figure 4: Abstract task example.**

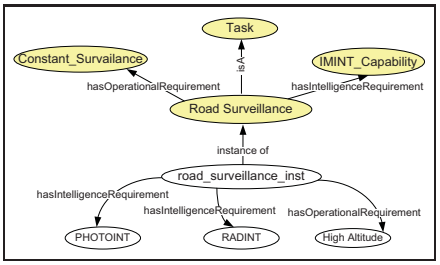


**Figure 5: Task instance example.**

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 define abstractly in an ontology together with its default requirements. Second, new requirements can be explicitly placed onto the task during design time. Third, constraints define within the context of the task may add new requirements to the task, or modify its existing requirements. Figure 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.

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,

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

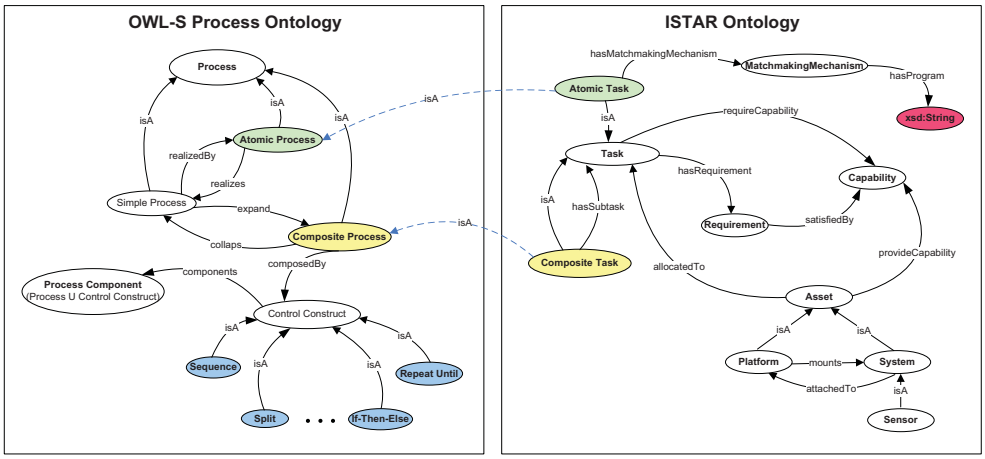


**Figure 3: Combination of OWL-S's process ontology and ISTAR ontology.**

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. Figure 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 [1] 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 crucial 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. This leads to the argument that allocating resources for a task amounts to allocating resources to all of the atomic tasks within the task. Therefore, henceforth we mainly describe our approach using atomic tasks.

Each atomic task may require different types of resources, because each atomic task may have different 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 configuration are define formally in a domain-independent way as follows. First, we make use of three finit and non-empty sets:

- Resource *types* $T = \{t_1, \ldots, t_n\}$
- Resource *capabilities* $C = \{c_1, \ldots, c_m\}$
- Task *requirements* $R = \{r_1, \ldots, r_p\}$

Sample sets for 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 $\kappa$ 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 $\sigma$ to a *set of sets* of capabilities $\{C'_0, \ldots, C'_q\}, C'_i \subseteq C, 0 \leq i \leq q$, 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 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 $\sigma(R') = \{C'_0, \ldots, C'_q\}$, a deployable configuration $DC \subseteq T$ is a set of types such that, for at least one $C'_i, 0 \leq i \leq q$, $C'_i \sqsubseteq \kappa(DC)$ 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 $C'_i, 0 \leq i \leq q$, such that $C'_i \sqsubseteq \kappa(DC_j)$.* ∎

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

As illustrated in Example 2 below, there may be different ways of executing an atomic task using different types of resources. In the definitio of deployable configurations we keep the functions $\kappa$ and $\sigma$ as abstract as possible, as they should be define differently for distinct domains. This means that the mechanism used to determine deployable configuration of an atomic task may change for different tasks. In some settings, each resource type may be independent and provide specifi capabilities. In other settings, however, resource types may depend on one another to provide capabilities. Example 3 and 4 below demonstrate how the determination of deployable platforms may vary in distinct settings.

EXAMPLE 2. *Assume that an atomic task has two deployable configurations: [GlobalHawk, EOCamera] and [Reaper, Daylight TV], where GlobalHawk and Reaper are autonomous 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 3. *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 4. *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 compose only three different deployable configurations for the task: [GlobalHawk, EOCamera], [GlobalHawk, IRCamera], and [Reaper, DaylightTV].*

Determination of deployable configuration for a specifi task requires extensive domain knowledge and expertise as the previous examples illustrate. In the following section, we propose to combine ontological reasoning and logic programming to provide a fl xible solution for the computation of deployable configurations

# 4. ONTOLOGICAL LOGIC PROGRAMMING

Determination of deployable configuration for a specifi task can be modeled 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 resource types that provide capabilities required by the tasks.

Examples 3 and 4 show that different tasks may require different mechanisms for computing deployable configurations Hence, we must be able to use different matchmaking mechanisms in order to compute deployable configuration for different tasks. That is, 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 propose Ontological Logic Programming (OLP), which is a novel combination of ontological reasoning and logic programming. Different matchmaking mechanisms are described using OLP and associated with an ontology as an instance of *MatchmakingMechanism* concept. In order to associate different tasks with different matchmaking mechanisms, an object property *hasMatchmakingMechanism* is created, whose domain is the *Task* concept while its range is the *MatchmakingMechanism* concept. In order to fin the most adequate resource types for a specifi task, our matchmaking algorithm simply gets the matchmaking mechanism related to the task and execute the associated OLP program.

Figure 6 shows the OLP stack used for matchmaking. At the base of the stack, we have domain ontologies in OWL [11]. Some rules are associated with these ontologies using SWRL [1]. Above the ontologies and the rules, we have a Description Logic (DL) [2] reasoner such as Pellet [10]. This reasoner is used to infer facts and relationships from the ontologies and rules. Above the reasoner, we have a Logic Programming (LP) interpreter. Our choice

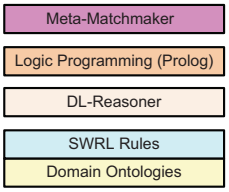


**Figure 6: OLP Stack.**

of LP language is Prolog and in our implementation, we use a pure Java-based Prolog interpreter [9]. At the top of the OLP stack, we have our meta-matchmaker. The meta-matchmaker is responsible for retrieving the right matchmaking mechanism for a task from the ontology, and then it uses an appropriate Prolog meta-interpreter to interpret the related OLP program. The meta-interpreter uses the DL reasoner during an execution and returns the matchmaking results to the meta-matchmaker.

Figure 7 shows a simplifie version of the Prolog meta-interpreter used to interpret OLP programs through the *eval* predicate, while Figure 8 shows a simple matchmaking mechanism written in OLP. The code in Figure 8 is a Prolog program, where concept and properties from the underlying ontologies are referenced directly. In order to differentiate between ontological and other predicates, we use name-space prefi es separated from the predicate name by a colon. For example, we can directly use ontological predicate *istar:requireCapability* in a OLP program without definin its semantics in Prolog, where *istar* is a prefi that refers to *http://www.csd.abdn.ac.uk/ita/istar*. The Prolog knowledge-base does not have any information about ontological predicates, since these predicates are not define in Prolog, but described separately in an ontology. In order to interpret an OLP program, the Prolog meta-interpreter needs the ontological reasoning provided by the DL reasoner. The meta-interpreter accesses the DL reasoner through the *loadFromOntology* predicate as shown in Figure 7. This predicate is a reference to a Java method, which queries the reasoner and loads the necessary facts into the Prolog knowledge-base.

During the interpretation of an OLP program, if a predicate in *prefix:name* format is encountered, the DL reasoner below the interpreter in the OLP stack is queried to get direct or inferred facts about the predicate in the underlying ontologies. For example, if the meta-interpreter encounters the ontological predicate *istar:requireCapability* during its interpretation of an OLP program, it queries the DL reasoner. The *requireCapability* predicate is define in ISTAR ontology using SWRL rules, so the reasoner interprets these rules to derive facts. Then the facts returned by the reasoner are loaded into the Prolog knowledge-base and interpretation of the OLP program is resumed. Therefore, we can directly use the concepts and properties from ontologies while writing logic programs and the facts are imported from the ontology through a reasoner when necessary. Caching mechanisms are used to improve performance. OLP enables us to combine the advantages of logic programming (e.g., complex data types/structures, negation by failure and so on) and ontological reasoning. Moreover, logic programming aspect enables automated creation of explanations, which provide transparency and rationales for utility measures.

The OLP program in Figure 8 is a simple matchmaking mechanism, where the *getConfigurations* predicate computes deployable configuration for a specifi 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

```
eval(not(G))   :- not(eval(G)).
eval((G1,G2))  :- eval(G1),eval(G2).
eval((G1;G2))  :- eval(G1);eval(G2).
eval((O:G))    :- ontology(O,G).
eval(G)        :- not(complex(G)), (clause(G,B), eval(B);
                  not(clause(G,_)),call(G)).
complex(G)     :- G=not(_) ; G=(_;_) ; G=(_,_) ; G=(_:_).
ontology(O,G)  :- loadFromOntology(O,G),call(O:G).
```

**Figure 7: Simplified Prolog meta-interpretter for OLP.**

```
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),
  istar:'provideCapability'(X,C).
requireCapability(T,S,C):-
  istar:'requireCapability'(T,C),
  not(provideCapability(S,C)).
provideCapability([Y|Tail],C):-
  istar:'provideCapability'(Y,C),!;
  provideCapability(Tail,C).
```

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

types are independent and capabilities are additive. Hence, this algorithm can be used to determine deployable configuration for the case in Example 3. In an 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 *hasMatchmakingMechanism* object property.

Although our example in Figure 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 Figure 9 computes deployable configuration for settings such as those in Example 4. 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 configuration with that sensor type. Using this knowledge, a tailored and efficien matchmaker can be employed. This matchmaker firs identifie 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. EVALUATION

Our approach is implemented using Java and Pellet is used as an OWL-DL reasoner. This section is composed of two parts. In the firs part, we empirically compare the proposed approach with an exhaustive search approach from the literature using a 2.16 GHz Intel Core Duo PC with a 2GB RAM. Note that the proposed approach is designed as a tool for intelligent agents to reason about tasks and resources. In the second part, we show a case-study where task agents use the proposed reasoning methods to cooperatively enhance their performance by sharing their resources.

```
getConfigurations(T,[P|S]):-
  deployablePlatform(T,P),
  extendSolution(T,P,[],S).
deployablePlatform(T,P):-
  istar:'Platform'(P),
  not((istar:'requireOperationalCapability'(T,C),
       not(istar:'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),
  istar:'Sensor'(X),
  istar:'provideCapability'(X,C).
requireCapability(T,P,S,C):-
  istar:'requireCapability'(T,C),
  not(provideCapability(S,C)),
  not(provideCapability([P],C)).
provideCapability([Y|Tail],C):-
  istar:'provideCapability'(Y,C),!;
  provideCapability(Tail,C).
```

**Figure 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 Computational Complexity

Gomez *et al.* propose a similar approach for automatically determining deployable configuration for tasks using the ISTAR ontology and Pellet as reasoner [5]. Their work depends on a minimal set covering algorithm. In order to determine deployable configura tions for an atomic task, this algorithm enumerates all possible sets of resource types so that each set has at most $n$ members. Then, a set is considered as a deployable configuratio of the task if it is composed of a platform type $P$ and set of sensor types $S$ so that $P$ can mount all of the sensor types in $S$. This approach is implemented in Java using ontological reasoning and an exhaustive search algorithm with a limit define by $n$.

Given a specifi task, the output of the OLP program in Figure 9 and that of the approach proposed by Gomez *et al.* should be the same because they are using the same definitio of deployable configurations However, the proposed algorithm of Figure 9 is based on the idea that the search space can be significantl reduced using domain knowledge (i.e, dependencies between sensors and platforms; not every types of sensors can be used with a specifi type platform). Using this principle, at each iteration, it rules out many combinations and significantl reduces the time required to compute deployable configurations

To confir our assertions, we have implemented the exhaustive search algorithm in Java and empirically compared it and our OLP-based algorithm in Figure 9 in terms of time consumption. Our results are demonstrated in Figure 10, where the x-axis is the maximum number of items in deployable configuration and y-axis is the average time consumed by each approach to fin all of the deployable configuration of an atomic task. When the maximum size of deployable configuration is lower than four, the exhaustive search algorithm is faster then our approach. This performance difference is originated from the overhead of using OLP stack to interpret the logic program in Figure 9. However, when the maximum size of deployable configuration is greater than three, the proposed approach outperforms the exhaustive search algorithm significantly time consumption of the exhaustive search increases exponentially while that of the proposed approach looks mostly linear.

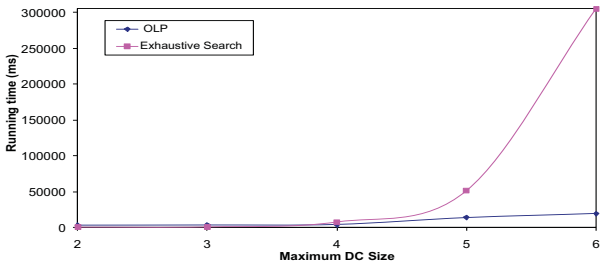


**Figure 10: Comparison between OLP program of Figure 9 and exhaustive search algorithm to find deployable configurations.**

```
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]
```

**Figure 11: Some voting examples.**

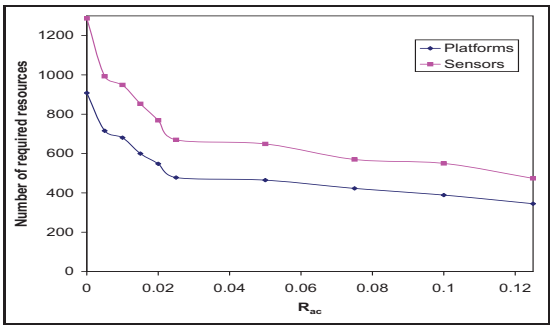


**Figure 12: Required resources vs. $R_{ac}$.**

## 5.2 Case-Study

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

### *5.2.1 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 configuratio cooperatively as follows. First, on a message board, the task agent publishes its desired deployable configuration (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 specifi deployable configuratio is based on the utility of sharing the assets of the deployable configuration Figure 11 shows votes of a task agent for three different deployable configurations where each deployable configuratio 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 configuratio that will be used during its executions. That is, task agents select deployable configuration 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 of the resources they need, because some of these resources are shared with them. The sharing depends on many constraints like policies, date/time, location, properties of resources and so on. Details of this approach are explained in [4].

### *5.2.2 Experiments*

We have randomly prepared a set of composite tasks as described in Section 3. This set includes 908 atomic tasks in total. Requirements and constraints of each task is set so that it 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 adequate to cooperate, which means that there should not be any reason preventing these tasks from cooperating. Policies or conflictin constraints may lead to inadequacy to cooperate. 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 conflictin constraints (e.g., location and date of tasks etc.) do not allow a task to cooperate or share 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 cooperate or share its resources with other tasks in the system. Note that $R_{ac}$ define 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 are adequate to 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 the cooperation or resource sharing between the 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. Figure 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 are not adequate to cooperate, 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. Figure 13 demonstrates the ratio of executable

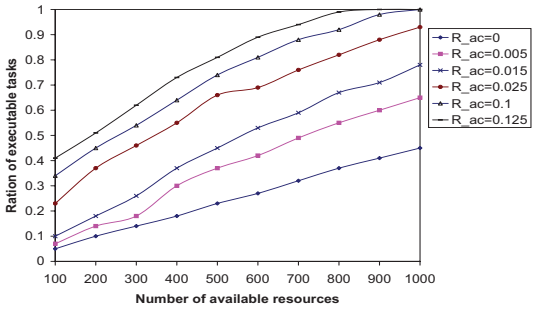


**Figure 13: Changing ratios of executable tasks.**

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 enable tasks to discover their opportunities to share resources, so more tasks are executed with fewer resources. When $R_{ac} = 0.125$, the number of platform and sensor instances required to execute all of the tasks is only 819.

## 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. 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 define and placed in an ontology, a task agent can use it to determine deployable configuration for its tasks. This fl xibility enables intelligent agents to reason about resources differently for different tasks and contexts. 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 fl xible but also enables efficien determination of deployable config urations. In order to show how the proposed reasoning mechanisms fi into multi-agent problems, we present a case-study where task agents use the proposed approach to promote resource sharing among tasks and carry out the tasks with fewer resources.

Tasks introduced in this paper can be considered as workfl ws, whose components are described semantically using an ontology. In the literature, there are approaches that describe workfl ws semantically using an ontology [12]. 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.

Grosof *et al.* propose Description Logic Programs (DLP), a combination of logic programs with description logic [6]. DLP relies at the intersection of Description Logic, Horn Logic and Logic Programs. 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.

Approaches like [5] consider the dependencies between resource types. In these approaches, instead of individual resource types, set of resource types are matched against tasks, which may exponentially increase the complexity of the matchmaking. Our approach does not depend on any assumption on the matchmaking mechanism, instead it enables different matchmaking algorithms to be define fl xibly using the terms from domain ontologies. Hence, unlike other approaches from the literature, our approach provides intelligent software agents to reason about tasks and resources without having any specifi matchmaking mechanism embedded in their architecture. In this paper, we choose the ISTAR domain to explain the proposed approach. As a future work, we want to evaluate the proposed approach in other domains using different scenarios. Furthermore, we want to use the proposed approach to improve the performance of resource allocation approaches in dynamic and uncertain environments.

## 7. REFERENCES

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

[2] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[3] Y. Chevaleyre, P. E. Dunne, U. Endriss, J. Lang, N. Maudet, and J. A. Rodríguez-aguilar. Multiagent resource allocation. *Knowl. Eng. Rev.*, 20(2):143–149, 2005.

[4] M. Şensoy, W. Vasconcelos, G. de Mel, and T. Norman. 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)*, pages 73–84, 2009.

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

[6] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logic. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 48–57, New York, NY, USA, 2003. ACM.

[7] L. He and T. R. Ioerger. 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*, pages 84–91, New York, NY, USA, 2005. ACM.

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

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

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

[11] M. K. Smith, C. Welty, and D. L. McGuinness. OWL: Web ontology language guide, February 2004.

[12] Y. Wang, J. Cao, and M. Li. Goal-driven semantic description and query for grid workfl w. In *Proceedings of the Third International Conference on Semantics, Knowledge and Grid*, pages 598–599, 2007.