Reasoning about Resources and Hierarchical Tasks Using OWL and SWRL

Daniel Elenius, David Martin, Reginald Ford, and Grit Denker

SRI International, Menlo Park, California, USA
firstname.lastname@sri.com

Abstract. Military training and testing events are highly complex affairs, potentially involving dozens of legacy systems that need to interoperate in a meaningful way. There are superficial interoperability concerns (such as two systems not sharing the same messaging formats), but also substantive problems such as different systems not sharing the same understanding of the terrain, positions of entities, and so forth. We describe our approach to facilitating such events: describe the systems and requirements in great detail using ontologies, and use automated reasoning to automatically find and help resolve problems. The complexity of our problem took us to the limits of what one can do with OWL, and we needed to introduce some innovative techniques of using and extending it. We describe our novel ways of using SWRL and discuss its limitations as well as extensions to it that we found necessary or desirable. Another innovation is our representation of hierarchical tasks in OWL, and an engine that reasons about them. Our task ontology has proved to be a very flexible and expressive framework to describe requirements on resources and their capabilities in order to achieve some purpose.

1 Introduction

In military training and testing events, many heterogeneous systems and resources, such as simulation programs, virtual trainers, and live training instrumentation, are used. Often, the systems were not designed to be used together. Therefore, many interoperability problems arise. These problems range from the superficial – such as two systems not sharing the same messaging formats – to more substantive problems such as different systems not sharing the same understanding of the terrain, positions of entities, and so forth.

In [1], we described an approach for automated analysis of military training events. The approach used OWL ontologies describing the systems and the purpose for which they were to interoperate. A custom Prolog program was used to produce warnings concerning potential interoperability problems, as well as "configuration artifacts" such as a chain of mediation components that could be used to connect two systems that otherwise would not be able to communicate.

A. Bernstein et al. (Eds.): ISWC 2009, LNCS 5823, pp. 795–810, 2009.
© Springer-Verlag Berlin Heidelberg 2009

This paper describes our further work in this area[1], which overcomes many of the limitations of our previous work. We provide two main contributions. First, we have extended OWL with a representation of hierarchical *tasks* in OWL. Tasks describe the structure of events and the requirements of resources that perform them. This has proved to be a flexible and expressive framework to describe military training and testing events in a way that allows automated reasoning in order to find problems and their solutions. Second, where we previously used hard-coded Prolog rules to detect interoperability problems, we are now using SWRL[2] rules and constraints. This approach is more declarative and extensible. We discuss shortcomings of SWRL as applied to our problem domain, and propose solutions to overcoming some of these. We have implemented general-purpose task processing tools to manage and reason about resources and hierarchical tasks. While our work has been driven by military training and testing domain, most of the problems and solutions discussed in this paper are applicable to other domains.

The paper is organized as follows. Section 2 describes some of the core ontologies underlying our approach. Section 3 discusses our task and task plan concepts. Section 4 discusses some specific uses of SWRL, and the benefits and limitations derived from its use. Section 5 describes our implementation of task processing tools. Section 6 discusses related work. Section 7 summarizes the lessons that we have learned and our conclusions.

2 Ontologies

Our approach to interoperability analysis depends on good-quality, authoritative ontologies. Since our intent is to find very subtle problems, many details must be ontologized regarding simulators, training instrumentation, vehicles, communication architectures, terrain, training and testing events, and so forth. It is also important that the task processing software does not depend on specific domain ontologies. Most of these will not be under the control of ONISTT developers, and we also want the software to be reusable for other domains. To that end, we have defined a very small set of core ontologies. Different organizations can create and maintain their own domain-specific ontologies that import the core ontologies and create subclasses of classes therein.

We investigate system interoperability in the context of a specific *purpose*. The key concepts to capture purpose are as follows. A `Task` is an intended action that requires some resource(s), and potentially has some additional constraints associated with it. A `Role` is a "slot" of a task that needs to be filled with some resource. A `TaskPlan` is a plan for how to perform a task, including assignment of resources to roles. The task and task plan ontologies are described in more detail in Section 3.

---

[1] This work was performed under the ONISTT project, sponsored by DUSD/R-RTPP (Training Transformation) and the ANSC project, sponsored by USD/AT&L-TRMC (S&T Portfolio).

[2] `http://www.w3.org/Submission/swrl/swrl.owl`

The key concepts to capture details about resources are as follows. A `Resource` is a thing that can *do* something, through the use of capabilities. Examples of resources are a tank, a human, a simulator, or training instrumentation. We also allow for intangible resources such as software systems. A resource can have subresources. A `Capability` is a discrete piece of functionality that belongs to a resource. Examples include the capability to send a "weapon fired" message using some message format, the capability to physically move (at some maximum speed), or the capability to detect things in the infrared spectrum. A `Confederation` is a set of resources.

A `Deployment` connects the purpose and resource descriptions. It describes an event (such as a training or testing event) in terms of one `TaskPlan` and one `Confederation`.

As an example of extending these core ontologies, we have an ontology of capability types for military training and testing resources containing subclasses of the `Capability` class, such as `MovementCapability`, `DetectabilityCapability` and `DirectFireCapability`, and an ontology of military `Task` types such as `TST` (time-sensitive targeting) and `JCAS` (joint close air support).

3 Tasks and Task Plans

A task is a description of the structure and requirements of some set of intended actions. The structural aspects of tasks are based on two fundamental ideas: *composition* and *refinement*. It is useful to express tasks in a compositional way, grouping tasks into composite tasks. This allows reuse of larger units, and makes it easier to understand a complex task at a high level. Furthermore, there are often different ways to perform a task, imposing different requirements on the participating resources. We model this using abstract tasks that can be refined into several different more concrete tasks.

In the following, we define a task ontology. One could view this ontology as an encoding of the *syntax* of a *task language*. This is analogous to the OWL encoding of SWRL and OWL-S[3]. In all three cases, an OWL encoding is useful in order to achieve a tight integration with OWL. However, as in the case of OWL-S process models and SWRL rules, OWL cannot express the intended semantics of our task concepts. Therefore, we provide a dedicated semantics for the new concepts. We have in effect extended the OWL language. We have also defined new reasoning problems (task analysis and task synthesis) that could not practically be reduced to the standard OWL reasoning problems (such as class subsumption checking).

3.1 Task Ontology

The task ontology is shown in Figure 1. Tasks are structured in a hierarchical way. There are three kinds of tasks: abstract, primitive, and composite. All tasks have *formal arguments*. The arguments are instances of the `Role` class.

[3] `http://www.daml.org/services/owl-s/`

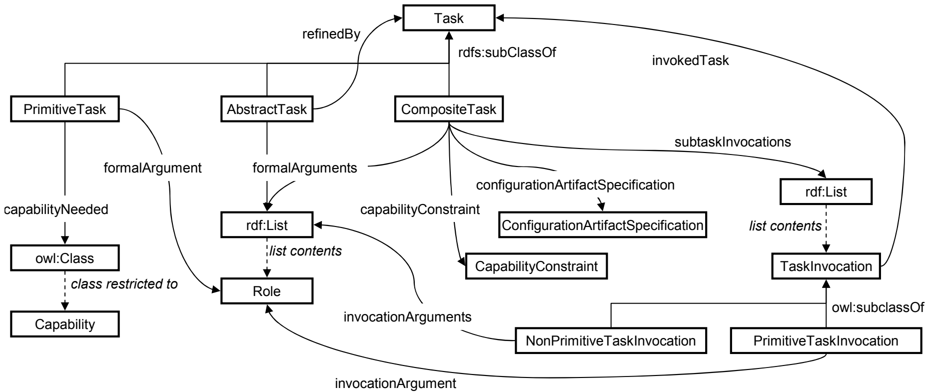

Fig. 1. Task ontology

Arguments are variables that can be assigned to resources in order to say what resources are involved in performing a task. Role-resource assignments are discussed below.

Abstract tasks can be performed, or *refined*, in several alternative ways. Each refinement is itself a task. Composite tasks have a list of *subtasks*, all of which must be performed for the composite task to succeed. *Task invocations* are used to bind the arguments of the composite task to those of its subtasks. Composite tasks can also have *constraints* on their subtasks, and *configuration artifacts* derived from their subtasks. Constraints and configuration artifacts are explained below. Primitive tasks have only one formal argument and thus are performed by one resource. Primitive tasks have an associated *capability needed*. The capability needed is a subclass of the `Capability` class[4]. A resource can be assigned to a primitive task only if it has a capability individual that is an instance of the capability needed class.

3.2 Semantics of Tasks

The following semantics is intended to facilitate the understanding of the meaning of tasks, and to provide an exact criterion for whether or not a task can be performed.

The intended meaning of a task is a set of Horn clauses. The *task atom* of a task T with formal arguments $\bar{\varphi}$ is defined as the atomic formula $\alpha(T) = T(\bar{\varphi})$. Similarly, a *task invocation atom* of a subtask invocation for task S with invocation arguments $\bar{\varphi'}$ is defined as the atomic formula $\iota(S) = S(\bar{\varphi'})$[5]. We define the function $\mathcal{H}$, denoting the Horn clauses of a task, in the following way (keeping the universal quantification of the variables in the clauses implicit). For an abstract task

[4] This is a use of classes-as-instances, and puts the ontology in OWL Full. This does not cause us problems, because we do not perform DL reasoning on task structures.

[5] From the perspective of the task ontology, the arguments are `Roles`. Logically, they are variables ranging over `Resources`.

A with refining tasks $R_1 \ldots R_n$, $\mathcal{H}(A) = \{\alpha(A) \Leftarrow \alpha(R_1), \ldots, \alpha(A) \Leftarrow \alpha(R_n)\}$. For a composite task C with subtask invocations $S_1 \ldots S_n$, $\mathcal{H}(C) = \{\alpha(C) \Leftarrow \iota(S_1) \wedge \ldots \wedge \iota(S_n)\}$. For a primitive task P with $\alpha(P) = P(x)$ and capability needed `Cap`, $\mathcal{H}(P) = \{\alpha(P) \Leftarrow \texttt{Resource}(x) \wedge \texttt{capability}(x, y) \wedge \texttt{Cap}(y)\}$. A *task library* L is a set of tasks $T_1 \ldots T_n$, and we extend the translation so that $\mathcal{H}(L) = \mathcal{H}(T_1) \cup \ldots \cup \mathcal{H}(T_n)$. Let KB be an OWL knowledge base, defined in the usual way, possibly containing SWRL rules, and $\mathcal{T}$ a function that translates such knowledge bases to their first-order logic equivalent [2]. A task T in a task library L is *performable*, given KB, iff $\alpha(T)$ is satisfiable by $\mathcal{T}(KB) \cup \mathcal{H}(L)$.

The mathematical descriptions account for the satisfiability of tasks. However, tasks have other characteristics that do not affect satisfiability but can be used by a task processing engine to produce useful information for the user, or to guide its processing. These features include constraints and configuration artifacts, both of which are described below.

It may reasonably be asked why we do not use SWRL rules (being a representation of Horn clauses) directly to represent the tasks. One reason was mentioned above: the "additional features" of tasks that are not formalized in the Horn clause semantics. The other reason is that we want to constrain the users somewhat, and not allow arbitrary rules as task descriptions.

3.3 Task Plans

The task plan ontology (see Figure 2) provides a layer on top of the task ontology, whereby one can describe *how* to perform a task. Task plans add five types of information to task descriptions:

- Role-resource assignments. A determination of which resource is to fill a particular role of the task.
- Capability assignments. A determination of which capability of the resource assigned to a primitive task is to be used for the task.
- Choices of which refining tasks to use for abstract tasks. For each abstract task, at most *one* refining task is chosen, by creating a refining task plan.
- Information about which constraints failed.
- Values for configuration artifacts.

Task plans can be partial, i.e. they do not have to have resource or capability assignments, refining plans, or subtask invocation plans.

Figure 3 shows an example of how the task and task plan ontologies can be instantiated. The task instances are in the left half of the figure, and the task plan instances in the right half. A `RoleAssignment` is used on the primitive task plan `SendPlan_1` to assign the resource `UAV1` to the `Sender` role. A capability of `UAV1` is also assigned to the task plan via the `assignedCapability` property. Note that the task instances are reusable, whereas the task plan instances are not reusable to the same extent – they show a particular way of performing the task.

Fig. 2. Task plan ontology

Fig. 3. Example of task and task plan instances

3.4 Semantics of Task Plans

The following is a sketch of the semantics of task plans. We leave out the precise details because of space considerations.

The meaning of a task plan P is a set of Horn clauses $\mathcal{H}_p(P)$. A role assignment is a substitution of a resource for a variable (role) that can be applied to a formula. Similarly, a capability assignment is a substitution of a capability for the

variable corresponding to the capability needed in a primitive task. Intuitively, the Horn clauses for a task plan are the Horn clauses for the corresponding task, with the role and capability assignments applied as variable substitutions where appropriate, and the choices of refining tasks for abstract tasks narrowed down to what is selected in the task plan. We define the *task plan atom* α_p of a task plan P for task T and with role assignments R as the result of applying R to $\alpha(T)$. A task plan P is *valid*, given an OWL knowledge base KB, iff $\alpha_p(P)$ is satisfiable by $\mathcal{T}(KB) \cup \mathcal{H}_p(P)$. Note that if a task plan is valid, the corresponding task is performable, but the opposite is not true in general. We call the determination of task plan validity the *Task Plan Analysis problem.*

A task plan is *complete* if it is fully specified to the primitive level, and assigns all roles and capabilities. Note that a complete task plan is a set of *ground* Horn clauses. A task plan P_c is a *completion* of a task plan P if $\mathcal{H}_p(P_c)$ can be produced by instantiating all the variables (roles), and removing alternative clauses for abstract tasks, in $\mathcal{H}_p(P)$. A completion is always complete. Note that there are no valid completions of an invalid task plan. The *Task Plan Synthesis problem* is to generate all *valid completions* from a task plan.

The constraints and artifacts part of task plans does not affect validity, and is not part of the semantics, but task plan processors can use these fields to return useful information. This is discussed in more detail below. In Section 5 we describe our implementation of an engine that performs both task plan analysis and task plan synthesis.

3.5 Constraints

As mentioned above, a composite task can have additional *constraints* on its subtasks. While primitive tasks place constraints on individual resources by forcing them to have a capability of a certain type, the constraints on a composite task are usually used to specify requirements on the interaction between several resources that perform subtasks of the composite task. For example, suppose we have a `TransferVideo` composite task with primitive subtasks `ProvideVideo` and `ConsumeVideo`. A constraint could be used to say that all the `supportedResolutions` of the provider have to be supported by the consumer.

A `CapabilityConstraint` of a composite task has an associated message, a severity, and a constraint atom (see Figure 4). The constraint atom is a SWRL atom. Typically, the predicate of the atom is defined using a SWRL rule. A task processor should try to prove the constraint atom in the context of the surrounding Horn clause. If it fails to do so, the constraint failure is reported in the result of the analysis, using the `failedConstraint` property on the generated task plan. The message associated with a constraint is a natural language description of the problem, which can be shown to the user. The task processor should also generate an overall score for each solution, by adding up the weights of the severities of all the constraints that failed. In other words, a lower score is better, and a score of zero signifies the absence of any known problems. All constraints are *soft*, and as mentioned previously do not affect the performability of the task.

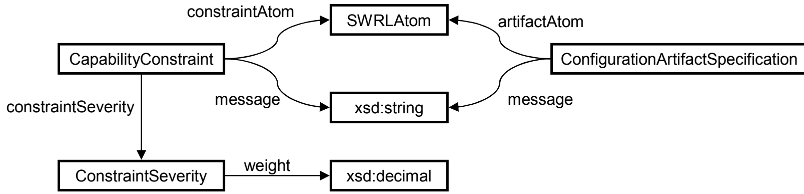

Fig. 4. Ontology elements for constraints and configuration artifacts

3.6 Configuration Artifacts

Configuration artifacts are similar to constraints (see Figure 4), except that they also have a return value. The return value is captured by letting the second argument of the artifact atom be a variable. This variable is bound when the task processor evaluates the artifact atom. The return value could be an `rdf:List` (created using the SWRL list built-ins) or any OWL individual or data value.

A typical use of configuration artifacts is to explain why a constraint failed. Continuing on the example above with supported resolutions, we could have a configuration artifact returning all the resolutions supported by the provider but not by the consumer. If the constraint succeeded, the list would be empty, but if it failed, the configuration artifact would show why it failed.

4 Benefits and Limitations of SWRL

We discuss our various uses of SWRL, the benefits we derived from its use, and the limitations that we have run into. We have put SWRL to use in many areas, such as defining constraints and configuration artifacts, reasoning about units, and ontology mapping. While SWRL has allowed us to go far beyond OWL, we have identified several limitations that appeared in different application areas: the limitation to unary and binary predicates, the lack of negation-as-failure and other nonmonotonic operations, and the inability to produce new individuals as a result of evaluating a rule.

We recognize that SWRL is not a standard, but most of what is said here also applies to other prospective rule languages such as RIF[6], and is of general concern.

4.1 Defining Constraints

SWRL rules provide a rather flexible way to evaluate capabilities of resources and perform various operations on them. Indeed, with the SWRL builtins[7], we can even do limited forms of "programming" with rules. There are some serious limitations to the usefulness of SWRL, however. First, SWRL "predicates" are OWL

[6] `http://www.w3.org/2005/rules/wiki/RIF_Working_Group`

[7] `http://www.w3.org/Submission/SWRL/#8`

classes or properties, and can therefore take only one or two arguments. SWRL builtin atoms solve this by using an `rdf:List` to contain the arguments. The same approach can be used with the other types of SWRL atoms in order to get an arbitrary number of arguments, and we use this approach when necessary. However, it is an awkward solution. First, we have to create a list to put arguments into, in the constraint atom. Then, the SWRL rule has to "unpack" the arguments from the list, using the list operations `swrlb:first` and `swrlb:rest`. Predicates with an arbitrary number of arguments is therefore high on our wish list for a future Semantic Web rule language.

The lack of negation in SWRL rules may be an even more serious limitation. In principle, SWRL allows any class expression, which means that one can use complement classes, but this allows us to negate only class expressions, not arbitrary SWRL formulas. In addition, this is classical negation, whereas we often need negation-as-failure. Going back to an example from Section 3, we may want to check that a video consumer can handle all the resolutions that a video provider can provide. Some capability of the provider and some capability of the consumer have a property `supportedResolution`. We need to check that all the provider's property values are also property values of the consumer. However, without negation-as-failure or a "closed world assumption" we cannot express this in OWL or SWRL. OWL's open world assumption means that there could always be more values of a property than what has been asserted. One could introduce several additional axioms to express that the property values are exactly the asserted property values:

```
Individual(ProvideVideoCapability
  type(restriction(supportedResolution cardinality(3)))
  value(supportedResolution 640x480)
  value(supportedResolution 800x600)
  value(supportedResolution 1024x768))

DifferentIndividuals(640x480 800x600 1024x768)
```

However, this quickly becomes unwieldy if one has to do this on all capabilities and properties in order to evaluate constraints on them like the one discussed here. Furthermore, we do not *want* to limit the capability to specific values in general – we want to retain the ability to define a capability across different ontologies in an open-ended manner. A better solution would be to make a "local closed world assumption" inside the rule. Of course, SWRL offers no such capability. For the time being, we decided on the following solution. We introduce a new SWRL builtin, called `allKnown`. This works similarly to the `setof` predicate in Prolog – it returns a list of all the "known" values of some property, for some individual. Once we have lists of property values, we can use SWRL's list built-ins to check whether one list is contained in another and so forth. The `allKnown` operation is nonmonotonic, and does not fit neatly into OWL's semantic framework. A more principled approach would be desirable, and is something we would like to see in a future Semantic Web rule language. A promising starting

point is presented in [3], where a subset of the description logic underlying OWL is augmented with an auto-epistemic operator **K** that can be used with class and role expressions. More general approaches of combining description logic with "logic programming" are presented in [4].

4.2 Defining Configuration Artifacts

The representation of configuration artifacts using SWRL shares all the issues discussed above, and introduces an additional problem. As we mentioned in Section 3, configuration artifacts are defined using a predicate (object property) where the second argument is a "return value". The value can be any OWL individual. However, there is no way to produce a *new* individual using a SWRL rule (with the exception that some SWRL built-ins for lists can produce new lists). We will see in the following sections how the same problem reappears in different contexts.

4.3 Ontology Mapping

Addressing military training and testing problems on a large scale using our ontology-based approach requires many detailed domain ontologies spanning a wide range of subjects. The development of such ontologies will be distributed among different stakeholders. However, not everyone will adhere to the same ways of describing resources and capabilities. Therefore, ontology mapping will be necessary. Ontology mapping is a wide topic, studied by many researchers using different approaches [5]. For our purposes, what is needed is to bring in knowledge from outside ontologies under our own upper ontologies (of resources, capabilities and so forth).

A mapping from one ontology to another can be defined in some special format, but a more flexible approach is to use a well-known logical formalism to describe the relationships between the two ontologies. When there is a relatively simple correspondence between entities in the two ontologies, one can define class and property equivalence or subsumption between the existing classes and properties, using OWL axioms. With SWRL rules, we can define more complex mappings.

As an example, suppose we have a "normative" `video` ontology to describe video provider resources, and an "external" `camera` ontology that we want to map to the `video` ontology (see Figure 5).

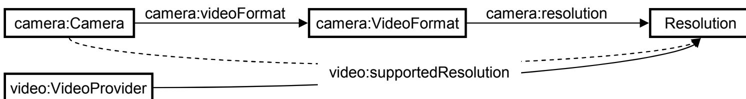

Fig. 5. Different representations of supported resolutions of a video provider resource. Dashed line shows relationship that needs to be generated.

We can define `camera:Camera` to be a subclass of `video:VideoProvider`, but the `camera` ontology describes the supported resolutions of the video provider in a different way than the `video` ontology. This difference in representation can be mapped using the SWRL rule.

$$\texttt{camera:Camera}(?x) \wedge \texttt{camera:videoFormat}(?x, ?y) \wedge$$
$$\texttt{camera:resolution}(?y, ?z) \Rightarrow \texttt{video:supportedResolution}(?x, ?z)$$

However, consider what happens if the "intermediate" instance is in the target ontology. This is a typical case in our real ontologies: Resources have *capabilities* (the intermediate individuals), which in turn have properties. For example, a resource might have a `video:ProvideVideoCapability` that has the `video:supportedResolution` property. Figure 6 shows the desired mapping.

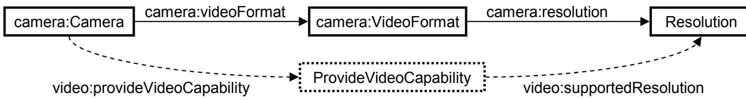

Fig. 6. Ontology mapping example. New property values (dashed lines) as well as a new instance (dashed box) need to be generated.

The mapping can be expressed by the rule

$$\texttt{camera:Camera}(?x) \wedge \texttt{camera:videoFormat}(?x, ?y) \wedge$$
$$\texttt{camera:resolution}(?y, ?z) \Rightarrow \exists ?c\texttt{: capability}(?x, ?c) \wedge$$
$$\texttt{video:ProvideVideoCapability}(?c) \wedge \texttt{video:supportedResolution}(?c, ?z)$$

Here we have an existentially quantified variable in the rule head, representing a "new instance" that needs to be created, viz. the `ProvideVideoCapability` of the resource. This cannot be expressed in Horn logic or SWRL rules. A limited form of existential quantification is possible by using `someValuesFrom` class expressions in a rule. However, the example above can still not be encoded in this way. The pattern discussed here is very common in OWL, since all structured data is described using property-value chains. Thus, SWRL can be used only for relatively simple types of mapping, where there is little structure in the target ontology.

4.4 Reasoning about Units

A common problem in ontologies is how to represent and reason about units of measure. Units are ubiquitous in our military training and testing domain, for example, in describing formats representing time and space positions, or the speed of vehicles. OWL by itself can be used to represent information about units, but is not adequate for making the right inferences from the information. However, this is an area where SWRL can be used to great advantage.

Figure 7 shows the essence of our “quantity” ontology. A quantity is an entity that has a unit and a magnitude, for example 5 kg or 10 lbs. A unit has a primary unit and a conversion factor to its primary unit. For example, the unit lbs has primary unit kg and conversion factor 0.4535924. Based on this quantity ontology, we have developed ontologies for engineering values (e.g., accelerations, areas, frequencies) and computation values (e.g., bits per second, megabytes, mebibytes).

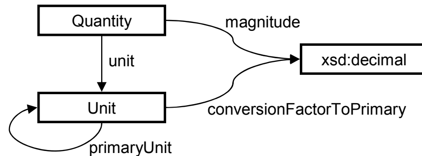

Fig. 7. Quantity ontology

Being able to describe quantities is only the first step, however. We also want to do things with them. For example, we want to compare quantities in different units. The quantity ontology defines a number of operations on quantities, using SWRL rules. First, we define a “helper” predicate `primaryMagnitude`. This is the magnitude of a quantity in its primary unit. `swrlb:multiply` is a SWRL built-in, where the first argument is the result of multiplying the rest of the arguments.

$$\texttt{magnitude}(?q, ?mag) \wedge \texttt{unit}(?q, ?u) \wedge \texttt{conversionFactorToPrimary}(?u, ?convf) \wedge \\ \texttt{swrlb:multiply}(?pmag, ?mag, ?convf) \Rightarrow \texttt{primaryMagnitude}(?q, ?pmag)$$

Next, we can define equals, less than, and so on, using this helper predicate:

$$\texttt{primaryMagnitude}(?q1, ?pmag1) \wedge \texttt{primaryMagnitude}(?q2, ?pmag2) \wedge \\ \texttt{swrlb:lessThan}(?pmag1, ?pmag2) \Rightarrow \texttt{qLessThan}(?q1, ?q2)$$

These operations can be used to determine for example that 10 lbs is less than 5 kg. This shows a common use of SWRL in our domain: We define some “abstract data type” [6] along with some operations on it. Another example (not shown here), which builds on the previously defined operations (e.g., less than) is quantity intervals, with operations such as checking whether two quantity intervals overlap.

One limitation is that we cannot use SWRL to define operations that *produce* new entities. This cannot be expressed using SWRL rules because of the limitation regarding existential variables illustrated in the discussion about ontology mapping above. For example, adding two quantities produces a new quantity that is the sum of the two. For some operations, like division, the result could even have a different unit than the inputs.

5 Implementation

We have implemented tools that realize the task planning framework described in Section 3. The two main components are a Protégé plug-in and a task engine, described below. The overall architecture is shown in Figure 8.

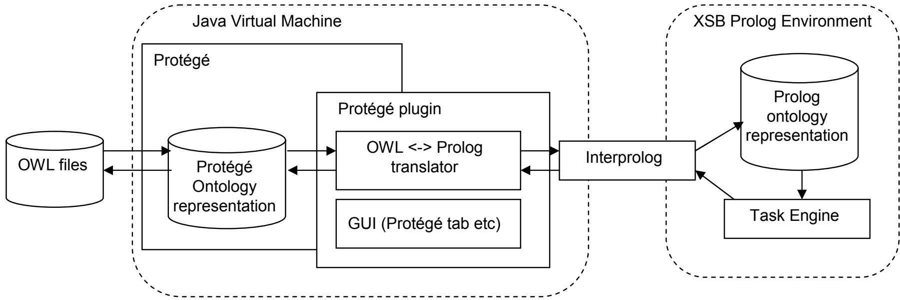

Fig. 8. Implementation architecture. The new components are the Protégé plug-in and task engine. As shown by arrows, results from the task engine can be translated all the way back to OWL files.

5.1 Task Engine

Our task engine is a program that solves the task plan analysis and synthesis problems discussed in Section 3.3. The engine is implemented in XSB Prolog[8]. Prolog was a natural choice because it provides built-in backtracking, which we use to generate all solutions during task plan synthesis.

Given that the meaning of a task or task plan is a set of Horn clauses, for task plan *analysis* we could just translate task plans directly into Prolog rules, query the task plan atom, and see if it succeeded or not. However, for the more interesting task plan *synthesis* problem, we also want to know *how* the task succeeded. This means that the engine has to "interpret" the task descriptions and construct structured result terms. This is similar to evaluating rules and returning the call tree and all variable assignments generated along the way.

As mentioned in Section 3, the engine depends on the entire OWL KB (i.e., knowledge beyond the task structure) for two purposes:

- When a primitive task is evaluated, the engine must perform a KB query of the form $\texttt{Resource}(x) \wedge \texttt{capability}(x, y) \wedge \texttt{Cap}(y)$.
- When a constraint atom or artifact atom is evaluated, the engine must perform a KB query given by atom.

To perform these queries, the OWL KB is translated to Prolog (by the OWL $\leftrightarrow$ Prolog translator component in Figure 8). The translation uses the well-known correspondence of a large subset of OWL, called DLP [7], to Horn clauses

[8] `http://xsb.sourceforge.net/`

(the translation is the same as in our previous work [1]). This means that not all of OWL's semantics is covered (i.e., the query answering is not complete), but in practice we have not found this to be a limitation for the ontologies that we work with, as we do not tend to rely on the more complex OWL axioms and the inferences that they would enable.

5.2 Protégé Plug-In

We developed a plug-in for Protégé [8] to help in creating tasks and task plans, invoking the task engine, and navigating results from the task engine.

Figure 9 shows one of several views of the results of running task synthesis for a given deployment. Results can be fairly complex. Our plug-in helps the user explore the result in terms of which resources were used, what warnings were generated, and the structure of the generated task plan.

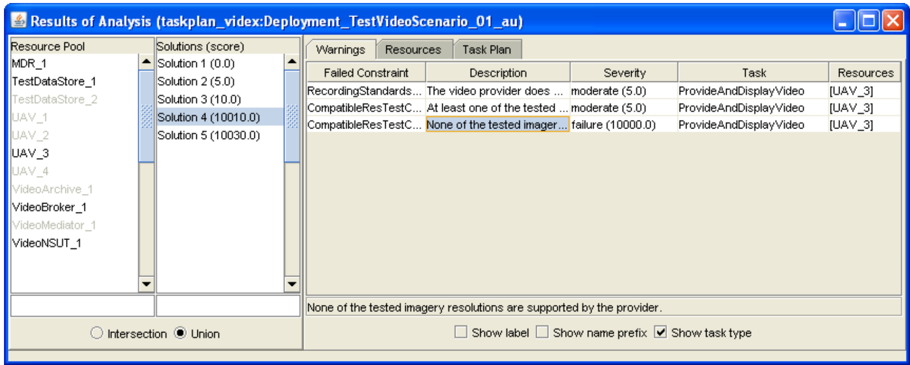

Fig. 9. Results of running the task engine from the Protégé plug-in. This view shows which resources were used to generate the selected solution. In the view shown, the user can explore warnings due to failed constraints.

6 Related Work

A related paradigm for task planning is *Hierarchical Task Network (HTN) Planning* [9]. HTNs have *tasks* (corresponding to our abstract tasks), *methods* (corresponding to our composite tasks), and *primitive tasks*. The goal is to decompose tasks down to primitive tasks, and assign *operations* to the primitive tasks. This is constrained by preconditions and effects on the tasks. Thus, the HTN planning engine has to keep track of the state of the world, and changes to it that tasks achieve. This makes HTN planning a harder problem than ours, since we do not care about the *order* in which tasks are performed. On the other hand, HTN planning is also *easier* than our problem, because the arguments to tasks are known in advance, whereas our task planning paradigm allows us to run the planning with variable "roles", which then need to be assigned by the planning engine.

Another related paradigm is Semantic Web Services, and OWL-S in particular. OWL-S is used to describe *services* based primarily on their inputs, outputs, preconditions, and effects (IOPEs). This is useful in order to infer which combination of services can be used to achieve a particular goal. Services can be thought of as tasks, and indeed HTN planning has been used with OWL-S service descriptions [10]. However, OWL-S is an open-ended representation scheme without any particular mandated computational paradigm. Our task ontology focuses on requirements on resources, whereas OWL-S focuses on IOPEs and sequencing of services, using control constructs such as *sequence* and *split-join*.

Finally, there is a wealth of work generally referred to as “scheduling” or “planning with resources” [11]. This focuses on deciding which resources can be used for multiple tasks at the same time and how tasks should be scheduled onto resources in an optimal way. In contrast, we assume that all resources can be used for any number of tasks.

7 Conclusions

Military training and testing events are highly complex affairs, potentially involving dozens of legacy systems that need to interoperate in a meaningful way. Our approach to facilitating such events is ambitious: describe the systems and requirements in great detail using ontologies, and use automated reasoning to automatically find and fix problems.

Our approach relies on ontologies, and it will be infeasible for us (the authors) to create all the domain ontologies. Therefore, a standard ontology language on which everyone can agree is critical, and OWL is the de facto standard. However, the complexity of our problem took us to the limits of what one can do with OWL, and we needed to introduce some innovative techniques of using and extending it.

One of our main contributions is our task and task plan concepts, which can be viewed as extensions to the OWL language. These concepts allow us to represent events and their requirements in a structured way, and break down an overwhelming amount of detail into manageable and reusable chunks. The second main contribution is a discussion of the benefits and limitations of SWRL. We put SWRL to use in many areas, such as defining constraints and configuration artifacts, reasoning about units, and ontology mapping. Among the limitations are the restriction to unary and binary predicates, the lack of negation-as-failure and other nonmonotonic operations, and the inability to produce new individuals as a result of evaluating a rule.

Our next task will be to define and evaluate a large real-world event using the techniques described in this paper. The long-term goal is to provide a complete system that is usable by military training and testing experts who are not necessarily knowledgeable in Semantic Web technologies. For such a transition to be successful, several different Semantic Web technologies and research areas need to progress further. The scale and distributed nature of the necessary ontology development will require significant improvement in ontology engineering approaches and tools.

References

1. Elenius, D., Ford, R., Denker, G., Martin, D., Johnson, M.: Purpose-aware reasoning about interoperability of heterogeneous training systems. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 750–763. Springer, Heidelberg (2007)
2. Tsarkov, D., Riazanov, A., Bechhofer, S., Horrocks, I.: Using Vampire to reason with OWL. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 471–485. Springer, Heidelberg (2004)
3. Grimm, S., Motik, B.: Closed world reasoning in the Semantic Web through epistemic operators. In: Grau, B.C., Horrocks, I., Parsia, B., Patel-Schneider, P. (eds.) Second International Workshop on OWL: Experiences and Directions (OWLED 2006), Galway, Ireland (2005)
4. Motik, B., Rosati, R.: A faithful integration of description logics with logic programming. In: Veloso, M.M. (ed.) Proc. 20th Int. Joint Conference on Artificial Intelligence (IJCAI 2007), Hyderabad, India, pp. 477–482. Morgan Kaufmann Publishers, San Francisco (2007)
5. Kalfoglou, Y., Schorlemmer, M.: Ontology mapping: The state of the art. In: Semantic Interoperability and Integration. Number 04391 in Dagstuhl Seminar Proc., Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2005)
6. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1. Springer, Heidelberg (1985)
7. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logic. In: Proc. of the 2nd International Semantic Web Conference, ISWC 2003 (2003)
8. Knublauch, H., Fergerson, R.W., Noy, N.F., Musen, M.A.: The Protégé OWL plugin: An open development environment for Semantic Web applications. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 229–243. Springer, Heidelberg (2004)
9. Ghallab, M., Nau, D., Traverso, P.: Hierarchical task network planning. In: Automated Planning: Theory and Practice, ch.11. Morgan Kaufmann Publishers, San Francisco (2004)
10. Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D.: HTN planning for web service composition using SHOP2. Web Semantics: Science, Services and Agents on the World Wide Web 1, 377–396 (2004)
11. Ghallab, M., Nau, D., Traverso, P.: Planning and resource scheduling. In: Automated Planning: Theory and Practice, ch.15. Morgan Kaufmann Publishers, San Francisco (2004)