

2017

Project Report – IA4SIM

Real-Time Simulation and Unit-Based Artificial Intelligence



Quentin AUBERTOT (CI2017)

Tithnara SUN (CI2017)

16/03/2017

Table of contents

Abstract	3
Introduction.....	4
I - Status report.....	5
A - Languages.....	6
B – Editors.....	7
C - Decision Support Systems	9
II - Ontologies	11
III - Our battleground ontology	14
IV – Rules	18
V – Jena.....	21
VI - Adapted algorithm of Ford-Fulkerson.....	22
Table of figures.....	26
Index	27
Bibliography.....	28
Appendix.....	29

Abstract

Keywords: Inference engine, ontology, OWL, SPARQL

Military officers must train themselves in taking real-time decisions in a battlefield. To do so, the military uses computer simulation tools, as it would be too complicated and not very cost-effective to make a mock battlefield using real soldiers every time an officer must train.

To make a realistic simulation environment, the simulation tool has to recreate the behavior of real soldiers, this is why it is crucial to design a proper artificial intelligence system. One of the ways of making an artificial intelligence system is to design an ontology and a reasoner based on such ontology. This is the goal of the project IA4SIM.

After some intensive research, we reached the conclusion to use OpenSource ways to reach our goal as it was the most cost-effective ways, the ontology was written using Protégé and the reasoner was written using the Jena plugin in the Eclipse environment. Also we decided to use Operations Research means to complete our reasoner because it seemed like the natural evolution for our project.

Ultimately, we were able to design an ontology including many military concepts such as Unit, Mission, Target, Environment & Equipment. We could create a reasoner that supports 35 rules and we completed it with an implemented adapted Ford-Fulkerson algorithm to assign missions to units.

The project is currently fully functional. It can be further developed to complexify the ontology or optimize the algorithm. Also rules can be added for a sharper artificial intelligence.

Introduction

First of all, we would like to thank Mr. Olivier VERRON, our project technical and professional supervisor, for his advices.

This document presents the project IA4SIM following different steps of research and conception.

First we will present the status report about ontologies and inference engines because it is our support in this project. Then we will explain the different parts of this project sorted by creation date. Finally, we will conclude with the possible evolution for this project.

I - Status report

The IA4SIM Project (short for Artificial Intelligence, or IA in French, and SIM for Simulation) is a Computer Modeling project. The goal is to create a model for the real-time simulation of a military battlefield and to create an automated decision making process on the simulation. To do so, we need to define an ontology and write an artificial intelligence based on this ontology.

An ontology in computer science is a formal definition of entities and links between said entities. It can be seen as a way to organize data and to limit the complexity of a system. As such, ontologies are used for problem solving and they are usually dedicated to a given domain of application. For example, in our case, we need to create an ontology dedicated to military operations in order to formulate decisions given the current state of the known battlefield. The entities we will have to create revolve around (but are not limited to) weapons, units, buildings and missions.

The Decision Support System (DSS) is a Knowledge-Based System (KBS) as expected of an artificial intelligence using an ontology. It will use an inference engine which is an artificial intelligence tool that can analyze a given database and make decisions. For example, in our case, we will be able to know which units have the firepower to destroy a given objective.

In this part, we will describe the different existing solutions available for our project and explain our final choice. Furthermore, we will make a preview advancement schedule.

In order to build an ontology architecture, we obviously need a language to describe our model. With the limits of classical languages, news languages were created to meet the needs of the semantic web. Among these languages, we find RDF and OWL with their own set of weakness and advantages. [1]

RDF

RDF language is a basic language to introduce the web semantic. It allows to define the properties of an object, like a title, subject, size... It was one of the first languages recommended by W3C (The World Wide Web Consortium) 2003 to uniform data online. Its capabilities are quite limited in comparison with others languages, but its strength is its portability because every languages using RDF as a basis.

OWL

OWL is an ontology language developed by W3C Web Ontology. OWL include three sub languages called: OWL-Lite, OWL DL and OWL Full. OWL became a W3C Recommendation in 2004. OWL incorporates functionalities like RDF, like hierarchies, bind between object... OWL also includes an abstract syntax.

OWL seems better for our project, since the relations between objects will be one of our focus. So RDF seems limited for our project. Furthermore, OWL is a popular language for the semantic web and efforts from many people are mobilized to make OWL “the” Semantic Web language. So we can easily find documentation on the internet and find more tools usable with OWL like for example Jena. There are two mains frameworks at the moment: Sesame and Jena. They are really close from each other. Sesame is a little easier to use and Jena a little more complete but they are both java frameworks. However, Jena incorporates an API (Application Programming Interface) for OWL. That is why we chose Jena to work with OWL.

SPARQL

SPARQL, for SPARQL Protocol and RDF Query Language, is a semantic query language for RDF and OWL. It is used for triple-pattern-queries. It will be compulsory if we use OWL or RDF.

After the choice of the language, we must find an ontology editor related to our previous choice. Indeed, like a text editor, there are editors for ontology. They are useful to generate a graphic representation of the project and easily bind classes with properties. After this, we can generate the code automatically. With an appropriate choice of editor, we can save a huge amount of time afterwards. It is why we must not underestimate this choice. [2]

So, in this part, we will compare some ontology editors to choose the best for our project. Given the sheer number of editors, we chose after a fast research five editors that seems relevant. Indeed, these editors are the most used in the world.

To compare, we defined some criteria:

- Price
- User-friendliness
- Functionalities
- Documentation

Apollo

Apollo was developed by Knowledge Media Institute. Their objective was to create a user-friendly editor. To reach this objective, the ontology model uses basic relationships between classes and instances for example. Apollo does not support graph view or the collaborative process and it only supports the Apollo Meta language. We can find a documentation about the editor on the website, where we can also download it for free.

<http://apollo.open.ac.uk/>

OntoStudio

OntoStudio was developed by Semafora. The model is based on a client/server architecture. It has several functions like graphic tools, web-service, collaborative development, supports several languages and additional plug-ins. On the website, we can find a tutorial video and an Online Help System. However, we can only download a free trial version lasting a month.

<http://www.semafora-systems.com/en/products/ontostudio/>

Protégé

Protégé was developed by Stanford University School. There are many different plug-ins to improve the user-interface. In particular, an Application Programming Interface (API) to ease the work with java and a plug-in for OWL. One of the main advantages of Protégé is its active community that

contributes for documentation and plug-ins. We can download Protégé for free online and start with an easy and complete tutorial to try it.

<http://protege.stanford.edu/>

Swoop

Swoop was developed by University of Maryland. Swoop can be integrated in your browser and is based on OWL. We can load multiple ontologies at the same time and easily navigate between each views. We can download the source code on Git-hub but the editor seems less documented than previously discussed editors.

<https://github.com/ronwalf/swoop>

TopBraid

TopBraid was developed by TopQuadrant. There are three editions: The Free Edition with main functions, the Standard Edition with all functions and the Maestro Edition with all functions and a live support. It is based on the Eclipse Platform and the Jena API. Several languages are available like OWL for example. There is a good documentation for TopBraid. However, for a Standard Edition Academic License, the price costs 595 dollars.

<http://www.topquadrant.com/tools/ide-topbraid-composer-maestro-edition/>

	Apollo	OntoStudio	Protégé	Swoop	TopBraid
Availability	Open Source	Software License	Open Source	Open Source	Software License
Portability	Poor	Medium (OWL...)	Strong (OWL, Jena...)	Poor	Strong (OWL, Jena...)
Graphical Views	No	Yes	Yes	No	Yes
Documentation	Documentation on website	Video + Online Help System	Tutorials + Active Community	Poor	Documentation on website

Figure 1 : Comparative Table

Finally, we chose Protégé for our project for many reasons. In the first place, it is Open Source and we do not really have the budget to buy an ontology editor especially if we do not really need it. Furthermore, the active community around Protégé is really suited if we need any help. About the technology now, Protégé seems complete, in particular with the plug-in for OWL and the API Jena without becoming too complex for us to use it.

Decision Support Systems (DSS) are IT-based technologies designed to help strategic managers. As their name would imply, they are not supposed to replace said managers but they can manage tasks that humans can not such as huge data stream collection and real-time analysis. They can be applied to many different domains including but not limited to forest management and marketing strategy. DSS can be split into five non-exclusive taxonomies: communication-driven DSS, data-driven DSS, document-driven DSS, knowledge-driven DSS and model-driven DSS. Given our project, we will mostly create a knowledge-driven DSS, also known as Knowledge-Based System (KBS). Among the KBS, we will show two examples: MYCIN and DIPMETER ADVISOR. [3]

MYCIN

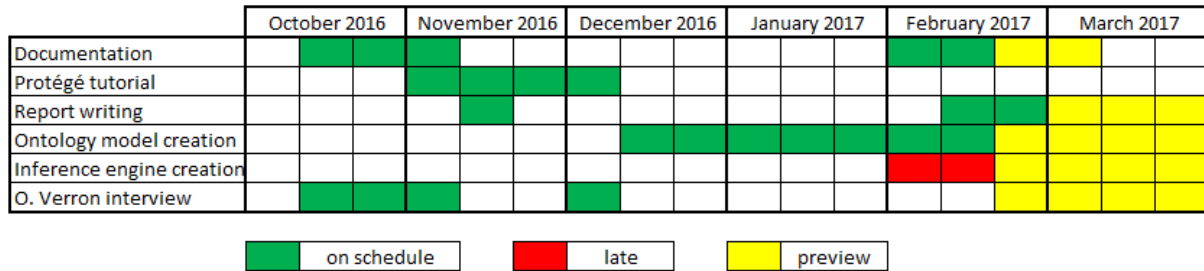
First of all, MYCIN is a KBS that was never actually used in practice but that was statistically proven more effective than infectious disease experts in identifying bacteria and recommending antibiotics with the dosage adjusted for patient's body weight. It was written in LISP by Edward Shortliffe as a doctoral dissertation. It uses probabilistic assumptions and evidence combination based on about 600 rules. MYCIN was ultimately proven to suggest an acceptable therapy in 69% of cases which was better than human judgement. [4]

DIPMETER ADVISOR

DIPMETER ADVISOR is a KBS developed by Schlumberger, the world's largest oilfield services company, in the 1980s. It was written in LISP to analyze data gathered during oil exploration. While DIPMETER was not the most complex inference engine (only about 90 rules), its user interface was very ergonomic and the Advisor was ultimately proven particularly effective for the company, as exhibited by their leading position in the oilfield today.

As we have seen with these two examples, designing the appropriate KBS is crucial. Given our time constraints, we will have to focus on building the inference engine (mostly the set of rules). The key qualities our KBS will have to have are simplicity, clarity, flexibility and adaptability.

We uncovered several technical solutions for our project. Thanks to our research, we decided to adopt OWL as our ontology language and Jena as our inference framework. And that is why we chose Protégé for our ontology editor, as it is well-documented and OpenSource. We believe that we made an appropriate choice but there are many viable options. This is the schedule that we agreed upon:



The ontology is based on a triple-structure with a subject, a link and an object. For example, with a family: in the triple “A is the mother of B”. “A” is the subject, “is the mother of” is the link and “B” is the object. We can use this example to see the differences between a database and an ontology.

If we used a simple database, we would get an array (or table) for each type of elements like this:

Mother	Daughter
A	B
C	D
E	F
B	G

Figure 3 : Simple Data Base

We can easily use a query to find the mother of A or the daughter of F. But can we find grandmothers? Indeed, we can see that A is the mother of B and B is the mother of G. So, we can easily say, A is the grandmother of G. But with a simple database, it's hard to do this or at least, it's not optimized because it is both time & resource consuming. In our example, if we need to check if A is a grandmother, first we must find A in the mother table, then deduce its daughter, and finally search for the daughter in the mother table. We can build a second table with grandmother and granddaughter, but in this case, we have a redundant database which is poor in resource consumption, since we can deduct everything with our current two tables.

To take the same example but with an ontology, we must define a model first. So, in our model we have « Daughters », « Mothers » and a link « is the mother of » as concepts.

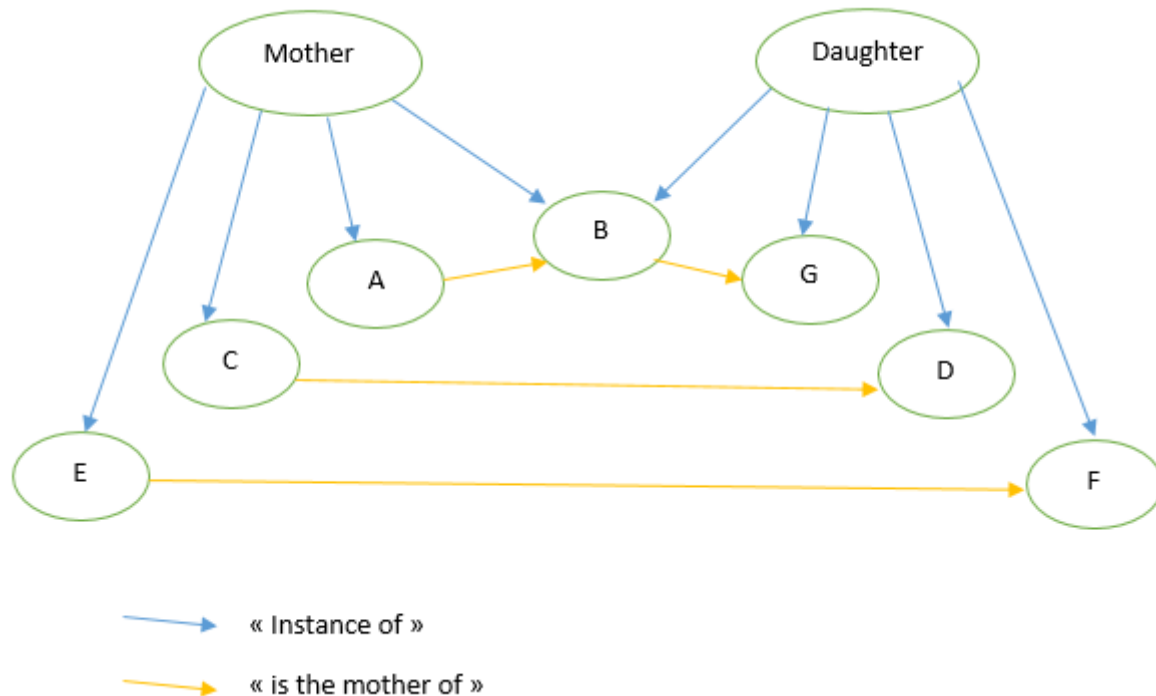


Figure 4 : Simple Ontology

In our case, « Daughters » and « Mothers » are classes and they aren't disjoint, in other words, a mother can be a daughter too and a daughter can be a mother too. A, B, C are individuals. Now we create an Object Property « is the mother of ». It represents a link between a domain and a range. The domain of this Object Property is « Mother » and the range is « Daughter ».

So now we have four triples representing the property:

- A is the mother of B
- C is the mother of D
- E is the mother of F
- B is the mother of G

Unlike in a database, we don't need a new table or a new class to manage the grandmothers. Indeed, with an ontology, we can create deductive rules to create new links. But first we must just create a new Object Property: « is the grandmother of ». So, the domain of the previous Object Property is « Mother » and the range « Daughter ». In our example, A will be a mother and a grandmother.

But we don't have to create A twice or create a new class "Grandmother" and a new class "Granddaughter".

So now for the rule, we can write:

$$\begin{cases} ?a \text{ is the mother of } ?b \\ ?b \text{ is the mother of } ?c \end{cases} \\ \rightarrow ?a \text{ is the grandmother of } ?c$$

Which can be read as following: "If ?a is the mother of ?b and ?b is the mother of ?c then ?a is the grandmother of ?c" with ?a, ?b, ?c being individuals from the ontology.

So now, we can use the inference engine and it will find all the grandmothers in our model. (Only A in our example.)

Now that we explained the purposes and advantages of an ontology compared to a database, we will show you ours for our project. Our ontology was created using the open-source software Protégé.



III - Our battleground ontology

As explained before, we must create a model to affect some units to missions. First, we created a class « Mission ». In our ontology, a Mission is defined by a class “Target”, a class “environment” and a needed sight (integer). Indeed, we wanted a user-friendly model for our client. So, the client must only precise that the mission is for example on water, on the ground, in the air or interior. For the target, we defined five levels: Weakest, Weak, Medium, Strong and Strongest because we wanted to make Missions easy to create. And finally, we want know if a unit can do a mission, so we created a link between unit and mission with Object Properties, “can_accomplish” so that our reasoner can create such link later. To test our model, we created six missions.

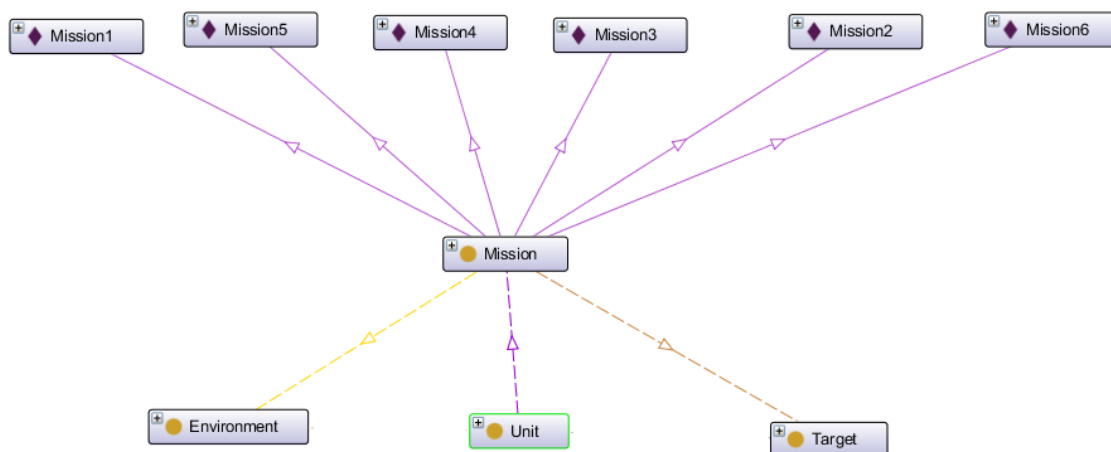


Figure 5 : Class Mission

The Object Properties between mission and target and between mission and environment are functional, in other words, a mission can get only one target and one environment. Actually, an object property can have some properties like:

- Functional (has a unique element in range)
- Inverse functional (has a unique element in domain)
- Transitive (if A is property B and B is property C, then A is property C)
- Symmetric (If A is property B, then B is property A, example: A is married with B)

Then, there is the class « Unit », for the project, we created several units. To test a large panel of situations, we created different types of units like, Infantries, Planes, Ships and Tanks. Each unit can operate on some environment, for example, a tank can only operate on the ground.

Also, we added some subclasses to our different types of units such as Bomber and Fighter for planes because these different types of plane can operate on different battlefields.

Finally, we created instances (or individuals) for each types of units: Tank1, Tank2...

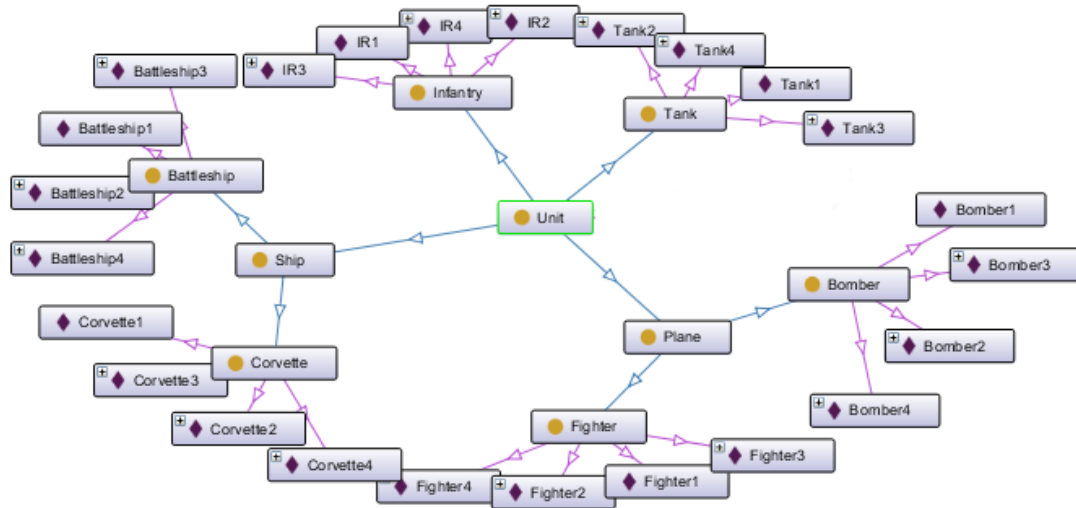


Figure 6 : Class Unit

As of now, if Tank1 can do a mission, so can Tank2 too. The purpose of having multiple instances of the same type of Unit is to assign each unit to a given mission. Also, to complete our model, we created the class “Equipment”. To give an example, an infantry can take a sniper riffle to improve its destructive power and destroy a bigger target. So, we created several equipment for each unit. We did two different types of equipment:

- Equipment to improve destructive power (sniper, bazooka, explosive ammo...)
- Equipment to operate on a different environment (torpedoes for planes, antiaircraft weapons for tanks...)

Like this we can affect some equipment to some units. For example for bombers: there are two types of equipment, « Tactical_nuke » and « Torpedo ».



Figure 7 : Bombers Equipment

For a target on water for example, only the bomber 3 and 4 can do it.

You can find more graphs about the ontology model in the appendix.

For our ontology, these are the individuals and the classes we agreed upon.



Figure 8 : Some Individuals

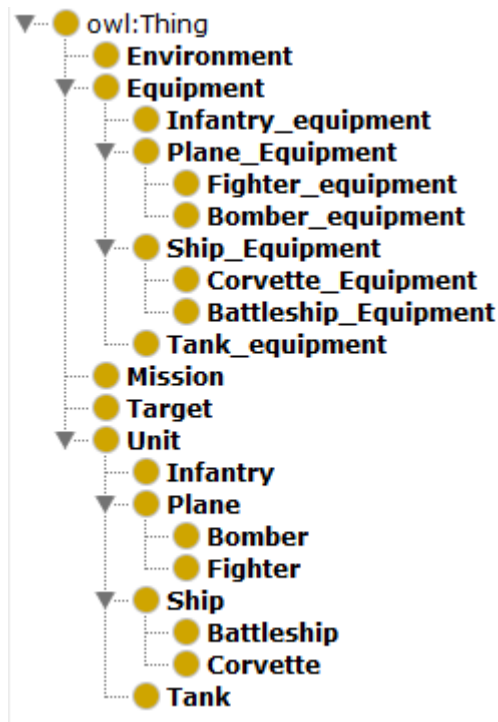


Figure 9 : Complete Classes

One of the many crucial point of the ontology is that an individual can be in more than one class depending on the model. In our case, an Infantry can't be a tank or a plane at the same time. So, we can use the option « disjoint » in Protégé. Like this, if we use the reasoner on the ontology, we can detect type errors. It is useful to check the consistency of our model. If we take the individual « Anti_aircraft_ship », this equipment can be use on a corvette or a battleship, it's why Battleship_Equipment and Corvette_Equipment are not disjoint.

Another point is that object properties accept a class (or classes) for domain and range. But there are also data properties. The difference is that the data properties accept only a class (or classes) for the domain. There range is defined as a basic datatype such as integer, boolean, string...

So, to complete our ontology, we created two data properties of type integer (in meters), « has_sight » and « need_sight ». Indeed, we can imagine a stealth mission where the unit must be far

enough not to be detected by the enemy. The first data property is « has_sight », its domain is Unit and Equipment. Each basic unit like infantry or tank has a basic sight. In our model for example, Infantries have a sight of 50 (meters). Then, some Equipment can have a sight, but not all type of Equipment. For example, an infantry with a sniper riffle can destroy a bigger target and this infantry gets a better sight too because the sniper riffle has a sight of 150 (meters). For the mission, the need_sight is mandatory, so we must write 0 if we don't need a specific sight.

Now, the ontology is complete, but we can add some individuals or more objects properties to adapt to the user's demands. Next, we must use an inference engine to create automatically new links between some individuals following rules that will be detailed next chapter. Typically, on our model, the engine must find which units can_accomplish Mission1, Mission2 etc...

IV – Rules

Our reasoner has 35 rules. Some rules are similar between each other and you can find the rule file in the appendix. The first rule that we created is a rule about the destructive power. We defined the targets with “weakest”, “weak”, “medium”, “strong” and “strongest”. One can wonder how the reasoner can know that `medium_target` is harder to destroy than a `weak_target`.

We created 4 rules to do so:

```
[rule_strongest_then_strong:
(?x rdf:type myOnto:Unit)
(?x myOnto:can_destroy myOnto:Strongest_target)
-> (?x myOnto:can_destroy myOnto:Strong_target)]
```

```
[rule_strong_then_medium:
(?x rdf:type myOnto:Unit)
(?x myOnto:can_destroy myOnto:Strong_target)
-> (?x myOnto:can_destroy myOnto:Medium_target)]
```

```
[rule_medium_then_weak:
(?x rdf:type myOnto:Unit)
(?x myOnto:can_destroy myOnto:Medium_target)
-> (?x myOnto:can_destroy myOnto:Weak_target)]
```

```
[rule_weak_then_weakest:
(?x rdf:type myOnto:Unit)
(?x myOnto:can_destroy myOnto:Weak_target)
-> (?x myOnto:can_destroy myOnto:Weakest_target)]
```

Now with these rules, a unit that can destroy a `strongest_target` can destroy all targets. To give more details about rules, we can explain the first rule:

“`[rule_strongest_then_strong:`”

The name we gave for the rule is not relevant for the reasoner, but it is better to use understandable name.

“`(?x rdf:type myOnto:Unit)`”

We search for an object of type “Unit” in the variable x. (If we can’t find one, the rule is not applied)

`(?x myOnto:can_destroy myOnto:Strongest_target)`

We check if “x” can destroy a `strongest_target`. (If we can not destroy, we restart with another “x”)

-> (?x myOnto:can_destroy myOnto:Strong_target)]

“x” can now destroy a strong target.

With these basic rules, we do not need to specify for each unit all destroyable targets but only the biggest destroyable target. We did similar rules for types of equipment that can destroy a target.

Each unit has defaults characteristics, for example in our ontology, all infantry can destroy a “weakest_target” without equipment, a bomber can destroy a “strong_target” and so on. In order to avoid the repetitions for each units, we created rules to give defaults characteristics for each unit.

```
[rule_tank:
(?x rdf:type myOnto:Tank)
-> (?x myOnto:can_operate myOnto:Ground)
(?x myOnto:can_destroy myOnto:Medium_target)
(?x myOnto:has_sight 100)]
```

The rule_tank (in three parts in the rule file) states that an individual of type tank can operate on the ground, it can destroy a medium target and it has a sight of 100 meters. We did the same for all units.

Also we created rules linking units and their equipment, for example, a plane with a tactical nuke can destroy a “strongest_target”. However, a bomber can destroy a “strong_target” without equipment and it can be equipped with a tactical_nuke to destroy a “strongest_target” and with armour piercing anti-air bullets to destroy a target in the air. But, the bomber can not combine both weapons to destroy a “strongest_target” in the air, whereas an infantry could combine his equipment to better its capabilities. For example, an infantry without equipment can only destroy a “weakest_target” on the water. But if we affect a zodiac and a sniper riffle, we can easily imagine that they can take the sniper riffle on the ship and destroy a “weak_target” on water. That is why we created different rules for infantries.

Rule for infantries:

```
[rule_can_operate_i:
(?x rdf:type myOnto:Infantry)
(?y rdf:type myOnto:Equipment)
(?a rdf:type myOnto:Environment)
(?x myOnto:has_equipment ?y)
(?y myOnto:can_operate ?a)
-> (?x myOnto:can_operate ?a)]
```

If an infantry has an equipment to operate in a new environment, then the infantry can now operate in this environment, we did the same rule for “can_destroy”. This rule is only for infantries.

We also created several rules to check if a unit can accomplish a mission, with or without equipment.

```
[rule_can_do_mission_e1:  
(?x rdf:type myOnto:Unit),  
(?y rdf:type myOnto:Mission),  
(?a rdf:type myOnto:Environment),  
(?t rdf:type myOnto:Target),  
(?e rdf:type myOnto:Equipment),  
(?y myOnto:need_sight ?ns),  
(?x myOnto:has_sight ?hs),  
ge(?hs,?ns),  
(?x myOnto:has_equipment ?e),  
(?y myOnto:has_environment ?a),  
(?e myOnto:can_operate ?a),  
(?y myOnto:has_target ?t),  
(?x myOnto:can_destroy ?t)  
-> (?x myOnto:can_accomplish ?y)]
```

With this rule, we check if the unit has enough sight for the mission (sight is a special property that we decided to bind to the unit as long as it has access to the piece of equipment because a unit equipped with binoculars is able to use its weapons further regardless of the weapon), and we check the other requirements to the mission and if it matches we conclude that the unit can accomplish the mission.

In order to manipulate our ontology and create a reasoner based on it, we decided to use the plugin Jena for the open-source Integrated Development Environment (IDE) Eclipse.



Jena is an Eclipse Java library containing functions & data structures dedicated to OWL Ontologies. Our main Java class is called Battleground.java.











<<Java Class>>	
 Battleground (default package)	
	<u>urlModel: String</u>
	<u>urlRules: String</u>
	<u>Battleground()</u>
	<u>main(String[]): void</u>
	<u>myOnto(String): String</u>
	<u>assignEquipment(FordFulkerson, Model): void</u>
	<u>findEquipment(Unit, String, Model): void</u>
	<u>printStatements(Model, Resource, Property, Resource): void</u>
	<u>processRules(String, InfModel): Model</u>

Figure 10 : Battleground Class

From an ontology model written in the .owl format and from a rule file written in .txt, this main class create a new ontology model with an inference engine with the previously described rules. To do so, the Jena library contains a parser that can create rule structures from a .txt files.

The main method contains several steps of the process:

- The FileManager structure in the Jena library handles the given ontology at urlModel.
- The rules are applied on the ontology and the resulting model is saved in the variable postRuleModel.

The new ontology can be explored via SPARQL queries to get specifically which missions can be accomplished by a given unit or which units can accomplish a given mission.

VI - Adapted algorithm of Ford-Fulkerson

Up to now, our program can just tell if a unit can do a mission. To improve the user experience with our project, we wanted to assign a unit for each mission (if possible) and if we can't, find one of the best solution (the fewest missions without units). So, we adapted the algorithm of Ford-Fulkerson for our problem. It is a standard algorithm to solve maximum flow problem, our case can be viewed as a simple maximum flow problem. To explain the algorithm, we can take an example with four missions and five units.

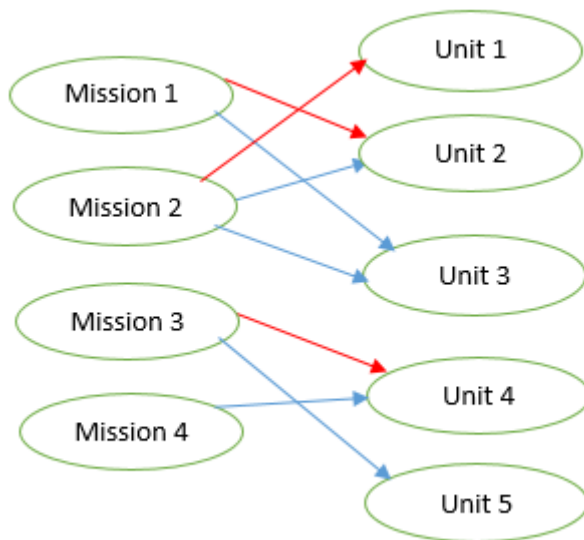


Figure 11 : Algorithm, Step 1

The Mission 1 can be accomplished by the Unit 2, the Mission 2 can be accomplished by the Unit 1, 2 and 3 and so on. Every arrow represents a possibility.

Step by step, the algorithm tries to assign a Unit to a Mission. First Mission 1 is assigned to the first Unit available, Unit 2. (The arrow becomes red.)

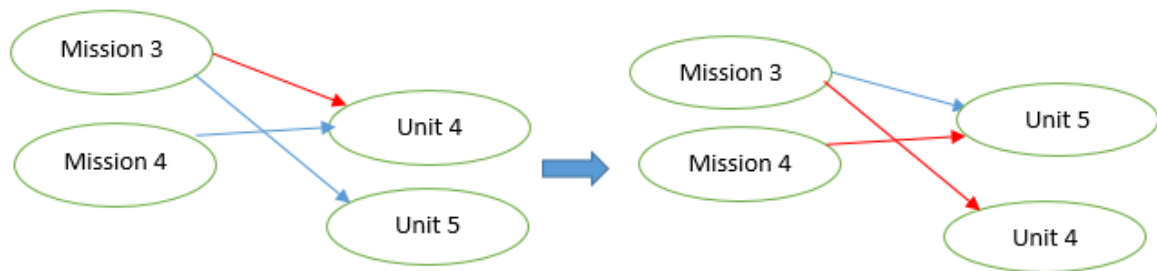
Then Mission 2 is assigned to Unit 1.

Then Mission 3 is assigned to Unit 4.

Then the algorithm tries to assign Mission 4 but it can't.

If the algorithm can not find an assignment, it tries to find a path of arrow of alternative colors. (Blue then Red then Blue and so on) Once the path is found, it flips the colors of the arrow, creating new assignments which are strictly more numerous than the previous configuration. It uses a recursive function that search for an increasingly long path that is limited by the number of missions. (If all missions have been checked and there is no path, there is no point in searching further) If this algorithm can't find a Unit for a Mission, it guarantees the fact that it is impossible to find without sacrificing another Mission.

In our example, here is the path that our algorithm would have found:



Path: Mission 4 \Rightarrow Unit 4 \Rightarrow Mission 3 \Rightarrow Unit 5

Figure 12 : Algorithm, Step 2

However, our algorithm finds an optimal solution in a possible set of optimal solutions. For example, in our case, the mission 1 could have been assigned to Unit 2 or Unit 3.

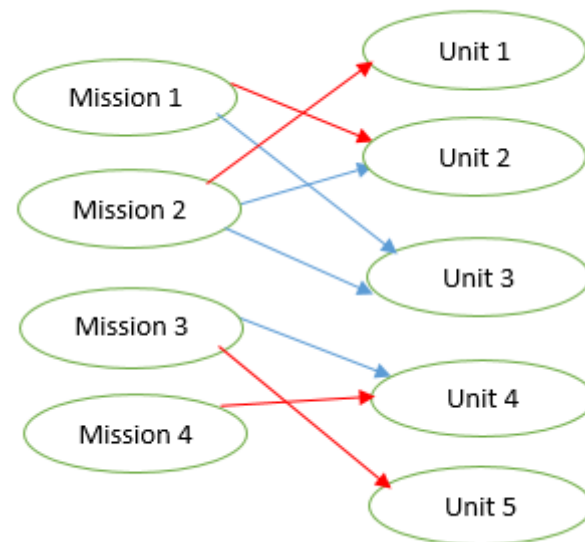


Figure 13 : Algorithm, Step 3

To implement this algorithm in our Eclipse project in Java, we decided to create a new package, "Algorithm" from which you can see the class diagram in the appendix.

The entities Mission & Unit are represented as Node in our structure. The links between them are called MissionAssignment & UnitAssignment inheriting from the class Assignment. The global structure is manipulated from the class FordFulkerson that has the variables missions & units containing respectively the list of missions and the list of units. The assignments are created through the method *addCandidate(Unit u)* of the Mission class.

The FordFulkerson class also has a method *saveGraph(String filename)* that prints the results of our algorithm in the wanted filename.

```
Report
mission [
name Mission1
is assigned to Battleship4.]

mission [
name Mission2
is assigned to Bomber2.]

mission [
name Mission3
is assigned to Fighter3.
needs Armour_piercing.]

mission [
name Mission4
was not assigned.]

mission [
name Mission5
is assigned to IR2.
needs Sniper.]

mission [
name Mission6
is assigned to Battleship3.]
```

Figure 14 : Report

We improved our main class Battleground.java in order to use our new Algorithm package. When the postRuleModel has been obtained, we use two SPARQL queries to find all the missions and all the available units. Then we create an instance of the FordFulkerson class and we fill it with our data sample. Our algorithm processes the data and finds a solution which is printed after assigning each unit to its required equipment via SPARQL queries. Finally, we can print the results via *saveGraph("Report.txt")*.

Conclusion

The project is fully functional. We used Protégé to create an ontology based on a battleground. Using Jena, we wrote some inference rules to create new relations in our ontology, typically “can_accomplish” in our case. Then, we use SPARQL to query the ontology to know which units can accomplish which missions. We input this data sample in our algorithm and we obtain an optimal solution. Finally, we give each chosen unit the appropriate equipment with SPARQL and we print the result in a text file.

We identified two ways of improving our project:

The first way is to complete the ontology even more. Indeed, we did an “simple” ontology but complete enough to see the possibilities of an ontology. We used different classes, objects properties, data properties. If a customer needs more types of unit or types of equipment, we can easily add them in our ontology. Furthermore, we can design new missions such as defend objectives or escort a payload, but we will have to write new rules.

The second way is to improve our algorithm to assign missions. We can develop priorities to particular units instead of randomly assigning possible units. We can imagine a better algorithm to assign mission depending of some criteria. For example, our algorithm can choose a bomber to destroy a “weakest_target” on the ground. It is valid, but maybe it is better to use only an infantry if possible.

Our work can be found at the following link:

<https://github.com/Quentin81/IA4SIM>

Table of figures

Figure 1 : Comparative Table	8
Figure 2 : Gantt Chart	10
Figure 3 : Simple Data Base	11
Figure 4 : Simple Ontology	12
Figure 5 : Class Mission	14
Figure 6 : Class Unit	15
Figure 7 : Bombers Equipment	15
Figure 8 : Some Individuals.....	16
Figure 9 : Complete Classes.....	16
Figure 10 : Battleground Class.....	21
Figure 11 : Algorithm, Step 1.....	22
Figure 12 : Algorithm, Step 2.....	23
Figure 13 : Algorithm, Step 3.....	23
Figure 14 : Report.....	24

Algorithm, 22

Data Base, 11

Inference Engine, 9

Jena, 21

Ontology, 12

OWL, 6

Protégé, 7, 13

Rules, 18

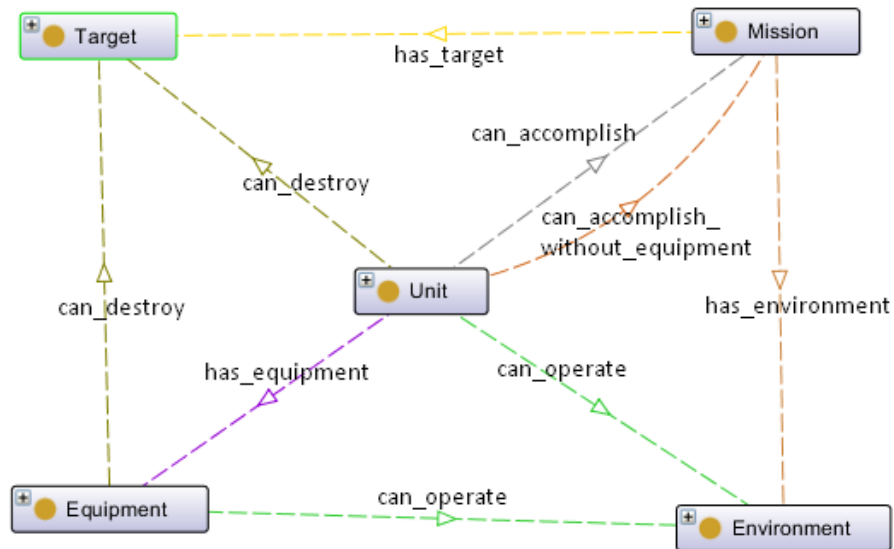
SPARQL, 6

Bibliography

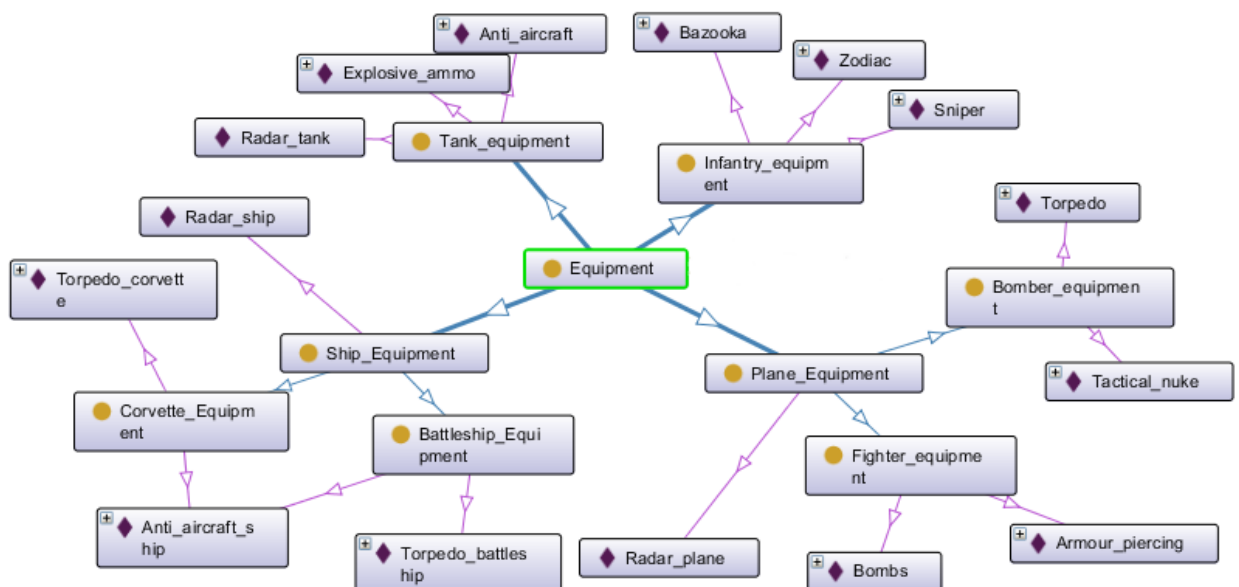
- [1] ALATRISH, Emhmed « Comparison Some of Ontology Editors » [Online] (April 2012)
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.193.6443&rep=rep1&type=pdf>
- [2] ZHDANOVA, Anna V. and KELLER, Uwe « Choosing an Ontology Language » [Online] (2005)
<http://www.ef.uns.ac.rs/mis/archive-pdf/2013%20-%20No2/MIS2013-2-4.pdf>
- [3] DELIC, K.A, DOUILLET, L. and DAYAL, U. « Towards an architecture for real-time decision support systems: challenges and solutions » [Online] (July 2001)
<http://ieeexplore.ieee.org/document/938098/?arnumber=938098>
- [4] SMITH, Reid G. « Knowledge-Based Systems Concepts, Techniques, Examples » [Online] (May 1985)
http://www.reidgsmith.com/Knowledge-Based_Systems_-_Concepts_Techniques_Examples_08-May-1985.pdf

Appendix

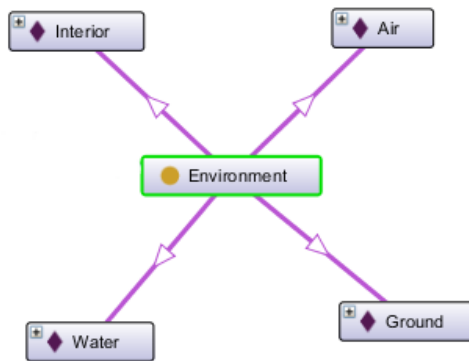
Object properties & main classes



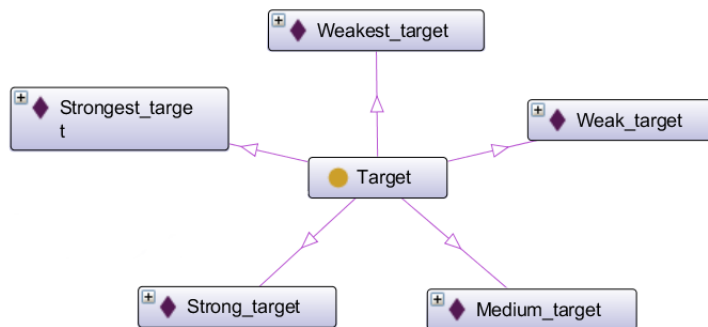
Equipment



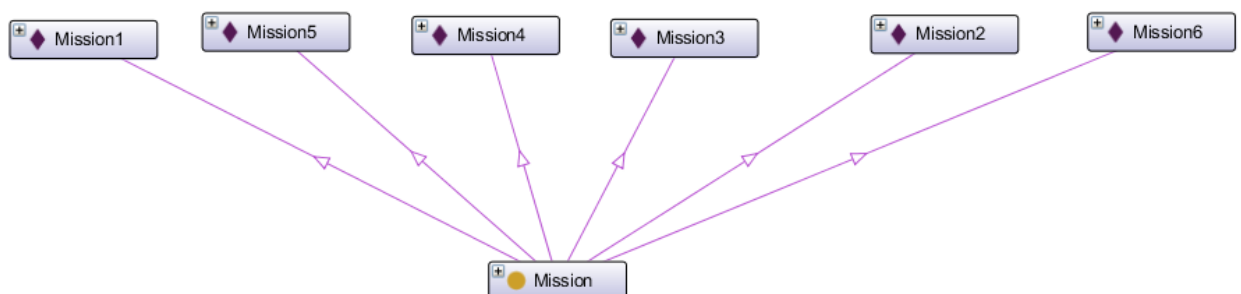
Environment



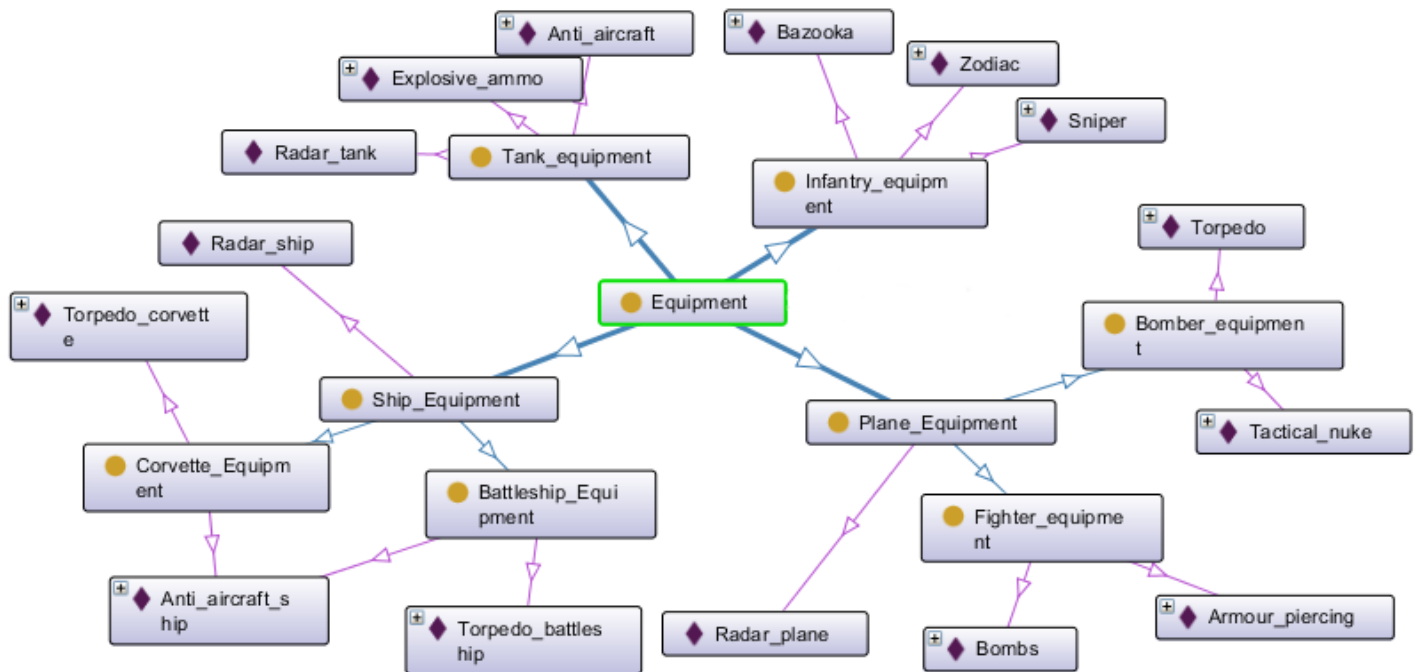
Target



Mission



Unit



Rules

```
@prefix myOnto:
<http://www.semanticweb.org/kentin/ontologies/2017/2/Battleground#>.
@prefix rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs:
<http://www.w3.org/2000/01/rdf-schema#>.
@prefix xsd:
<http://www.w3.org/2001/XMLSchema#>.
```

```
[rule_tank_can_do_mission_without_equipment:
(?x rdf:type myOnto:Tank),
(?y rdf:type myOnto:Mission),
(?a rdf:type myOnto:Environment),
(?t rdf:type myOnto:Target),
(?y myOnto:need_sight ?ns),
(?x myOnto:has_sight ?hs),
ge(?hs,?ns),
(?y myOnto:has_environment ?a),
(?x myOnto:can_operate ?a),
(?y myOnto:has_target ?t),
(?x myOnto:can_destroy ?t)
-> (?x
myOnto:can_accomplish_without_equipment
?y)]
```

```
[rule_plane_can_do_mission_without_equipment:
(?x rdf:type myOnto:Plane),
(?y rdf:type myOnto:Mission),
(?a rdf:type myOnto:Environment),
(?t rdf:type myOnto:Target),
(?y myOnto:need_sight ?ns),
(?x myOnto:has_sight ?hs),
ge(?hs,?ns),
(?y myOnto:has_environment ?a),
(?x myOnto:can_operate ?a),
(?y myOnto:has_target ?t),
(?x myOnto:can_destroy ?t)
-> (?x
myOnto:can_accomplish_without_equipment
?y)]
```

```
[rule_ship_can_do_mission_without_equipment:
(?x rdf:type myOnto:Ship),
(?y rdf:type myOnto:Mission),
```

```
(?a rdf:type myOnto:Environment),
(?t rdf:type myOnto:Target),
(?y myOnto:need_sight ?ns),
(?x myOnto:has_sight ?hs),
ge(?hs,?ns),
(?y myOnto:has_environment ?a),
(?x myOnto:can_operate ?a),
(?y myOnto:has_target ?t),
(?x myOnto:can_destroy ?t)
-> (?x
myOnto:can_accomplish_without_equipment
?y)]
```

```
[rule_can_do_mission_e1:
(?x rdf:type myOnto:Unit),
(?y rdf:type myOnto:Mission),
(?a rdf:type myOnto:Environment),
(?t rdf:type myOnto:Target),
(?e rdf:type myOnto:Equipment),
(?y myOnto:need_sight ?ns),
(?x myOnto:has_sight ?hs),
ge(?hs,?ns),
(?x myOnto:has_equipment ?e),
(?y myOnto:has_environment ?a),
(?e myOnto:can_operate ?a),
(?y myOnto:has_target ?t),
(?x myOnto:can_destroy ?t)
-> (?x myOnto:can_accomplish ?y)]
```

```
[rule_can_do_mission_e2:
(?x rdf:type myOnto:Unit),
(?y rdf:type myOnto:Mission),
(?a rdf:type myOnto:Environment),
(?t rdf:type myOnto:Target),
(?e rdf:type myOnto:Equipment),
(?y myOnto:need_sight ?ns),
(?x myOnto:has_sight ?hs),
ge(?hs,?ns),
(?x myOnto:has_equipment ?e),
(?y myOnto:has_environment ?a),
(?x myOnto:can_operate ?a),
(?y myOnto:has_target ?t),
(?e myOnto:can_destroy ?t)
-> (?x myOnto:can_accomplish ?y)]
```

```
[rule_can_do_mission_e3:
(?x rdf:type myOnto:Unit),
(?y rdf:type myOnto:Mission),
(?a rdf:type myOnto:Environment),
(?t rdf:type myOnto:Target),
```



```
(?e rdf:type myOnto:Equipment),  
(?y myOnto:need_sight ?ns),  
(?x myOnto:has_sight ?hs),  
ge(?hs,?ns),  
(?x myOnto:has_equipment ?e),  
(?e myOnto:has_environment ?a),  
(?e myOnto:can_operate ?a),  
(?y myOnto:has_target ?t),  
(?e myOnto:can_destroy ?t)  
-> (?x myOnto:can_accomplish ?y)]
```

```
[rule_can_destroy_i:  
(?x rdf:type myOnto:Infantry)  
(?y rdf:type myOnto:Equipment)  
(?t rdf:type myOnto:Target)  
(?x myOnto:has_equipment ?y)  
(?y myOnto:can_destroy ?t)  
-> (?x myOnto:can_destroy ?t)]
```

```
[rule_can_operate_i:  
(?x rdf:type myOnto:Infantry)  
(?y rdf:type myOnto:Equipment)  
(?a rdf:type myOnto:Environment)  
(?x myOnto:has_equipment ?y)  
(?y myOnto:can_operate ?a)  
-> (?x myOnto:can_operate ?a)]
```

```
[rule_sight_e:  
(?x rdf:type myOnto:Unit)  
(?y rdf:type myOnto:Equipment)  
(?x myOnto:has_equipment ?y)  
(?x myOnto:has_sight ?a)  
(?y myOnto:has_sight ?b)  
max(?b,?a,?c)  
-> (?x myOnto:has_sight ?c)]
```

```
[rule_strongest_then_strong:  
(?x rdf:type myOnto:Unit)  
(?x myOnto:can_destroy  
myOnto:Strongest_target)  
-> (?x myOnto:can_destroy  
myOnto:Strong_target)]
```

```
[rule_strong_then_medium:  
(?x rdf:type myOnto:Unit)  
(?x myOnto:can_destroy  
myOnto:Strong_target)  
-> (?x myOnto:can_destroy  
myOnto:Medium_target)]
```

```
[rule_medium_then_weak:  
(?x rdf:type myOnto:Unit)
```

```
(?x myOnto:can_destroy  
myOnto:Medium_target)  
-> (?x myOnto:can_destroy  
myOnto:Weak_target)]
```

```
[rule_weak_then_weakest:  
(?x rdf:type myOnto:Unit)  
(?x myOnto:can_destroy  
myOnto:Weak_target)  
-> (?x myOnto:can_destroy  
myOnto:Weakest_target)]
```

```
[rule_strongest_then_strong_e:  
(?x rdf:type myOnto:Equipment)  
(?x myOnto:can_destroy  
myOnto:Strongest_target)  
-> (?x myOnto:can_destroy  
myOnto:Strong_target)]
```

```
[rule_strong_then_medium_e:  
(?x rdf:type myOnto:Equipment)  
(?x myOnto:can_destroy  
myOnto:Strong_target)  
-> (?x myOnto:can_destroy  
myOnto:Medium_target)]
```

```
[rule_medium_then_weak_e:  
(?x rdf:type myOnto:Equipment)  
(?x myOnto:can_destroy  
myOnto:Medium_target)  
-> (?x myOnto:can_destroy  
myOnto:Weak_target)]
```

```
[rule_weak_then_weakest_e:  
(?x rdf:type myOnto:Equipment)  
(?x myOnto:can_destroy  
myOnto:Weak_target)  
-> (?x myOnto:can_destroy  
myOnto:Weakest_target)]
```

```
[rule_ship_o:  
(?x rdf:type myOnto:Ship)  
-> (?x myOnto:can_operate myOnto:Water)]
```

```
[rule_fighter_o:  
(?x rdf:type myOnto:Fighter)  
-> (?x myOnto:can_operate myOnto:Air)]
```

```
[rule_bomber_o:  
(?x rdf:type myOnto:Bomber)  
-> (?x myOnto:can_operate myOnto:Ground)]
```

```
[rule_infantry_o1:
(?x rdf:type myOnto:Infantry) ->
(?x myOnto:can_operate myOnto:Ground)]

[rule_infantry_o2:
(?x rdf:type myOnto:Infantry)
-> (?x myOnto:can_operate myOnto:Interior)]

[rule_tank_o:
(?x rdf:type myOnto:Tank)
-> (?x myOnto:can_operate myOnto:Ground)]

[rule_corvette_d:
(?x rdf:type myOnto:Corvette)
-> (?x myOnto:can_destroy
myOnto:Weak_target)]

[rule_battleship_d:
(?x rdf:type myOnto:Battleship)
-> (?x myOnto:can_destroy
myOnto:Strong_target)]

[rule_fighter_d:
(?x rdf:type myOnto:Fighter)
-> (?x myOnto:can_destroy
myOnto:Weak_target)]

[rule_bomber_d:
(?x rdf:type myOnto:Bomber)
-> (?x myOnto:can_destroy
myOnto:Strong_target)]
```

```
[rule_infantry_d:
(?x rdf:type myOnto:Infantry)
-> (?x myOnto:can_destroy
myOnto:Weakest_target)]

[rule_tank_d:
(?x rdf:type myOnto:Tank)
-> (?x myOnto:can_destroy
myOnto:Medium_target)]

[rule_corvette_s:
(?x rdf:type myOnto:Corvette)
-> (?x myOnto:has_sight 150)]

[rule_battleship_s:
(?x rdf:type myOnto:Battleship)
-> (?x myOnto:has_sight 200)]

[rule_fighter_s:
(?x rdf:type myOnto:Fighter)
-> (?x myOnto:has_sight 200)]

[rule_bomber_s:
(?x rdf:type myOnto:Bomber)
-> (?x myOnto:has_sight 500)]

[rule_infantry_s:
(?x rdf:type myOnto:Infantry)
-> (?x myOnto:has_sight 50)]

[rule_tank_s:
(?x rdf:type myOnto:Tank)
-> (?x myOnto:has_sight 100)]
```

