FOI-R--0498--SE
May 2002
ISSN 1650-1942
Scientific report

Tobias Horney

Design of an ontological knowledge structure for a query language for multiple data sources

Command and Control Systems
SE-581 11 Linköping

SWEDISH DEFENCE RESEARCH AGENCY

Command and Control Systems
P.O. Box 1165
SE-581 11 Linköping

FOI-R--0498--SE

May 2002

ISSN 1650-1942

Scientific report

Tobias Horney

Design of an ontological knowledge structure for a query language for multiple data sources

Issuing organization	Report number, ISRN	Report type
FOI – Swedish Defence Research Agency Command and Control Systems P.O. Box 1165 SE-581 11 Linköping	FOI-R--0498--SE	Scientific report
	Research area code 4. C4ISR	
	Month year May 2002	**Project no.** E7030
	Customers code 5. Commissioned Research	
	Sub area code 42 Surveillance Sensors	

Author/s (editor/s)	Project manager
Tobias Horney	Erland Jungert
	Approved by Lennart Nyström
	Sponsoring agency The Swedish Armed Forces
	Scientifically and technically responsible Erland Jungert

Report title

Design of an ontological knowledge structure for a query language for multiple data sources

Abstract (not more than 200 words)

In a command and control system which uses different kinds of sensors attached to various platforms, it is important to know which sensor data to use and which recognition algorithm to apply to that data when searching for a certain object type in a certain area at a certain point or interval in time.

In this work an ontological knowledge structure, implemented as a knowledge base, is used to model apriori knowledge about the objects of interest, the sensors, the sensor platforms, the recognition algorithms used for searching in the sensor data, as well as external conditions. All these aspects have an impact on which sensor and algorithm is the most appropriate to use.

A rule manager has been implemented. It is used for editing the rules describing how the sensors and algorithms are affected by the different conditions. Also implemented is an algorithm that uses the ontological knowledge base and the rules created in the rule manager for making actual decisions on which sensors and algorithms are the most appropriate given a complete set of input parameters (e.g. object to search for, area of interest, timestamp, existing conditions, etc).

Keywords

ontology, data fusion, information fusion, query language

Further bibliographic information	Language English
ISSN 1650-1942	**Pages** 64 p.
	Price acc. to pricelist

Utgivare Totalförsvarets Forskningsinstitut - FOI Ledningssystem Box 1165 581 11 Linköping	**Rapportnummer, ISRN** FOI-R--0498--SE	**Klassificering** Vetenskaplig rapport
	Forskningsområde 4. Spaning och ledning	
	Månad, år Maj 2002	**Projektnummer** E7030
	Verksamhetsgren 5. Uppdragsfinansierad verksamhet	
	Delområde 42 Spaningssensorer	
Författare/redaktör Tobias Horney	**Projektledare** Erland Jungert	
	Godkänd av Lennart Nyström	
	Uppdragsgivare/kundbeteckning Försvarsmakten	
	Tekniskt och/eller vetenskapligt ansvarig Erland Jungert	

Rapportens titel (i översättning)

Design av ontologisk kunskapsstruktur för frågespråk för multipla datakällor

Sammanfattning (högst 200 ord)

I ett ledningssystem som använder olika typer av sensorer fastsatta på plattformar är det viktigt att veta vilken sensordata man ska använda och vilka igenkänningsalgoritmer man ska applicera på den datan när man söker efter en viss objekttyp i ett visst område vid ett visst tillfälle eller tidsintervall.

I detta arbete har en ontologisk kunskapsbas tagits fram. Denna används för att modellera apriori-kunskap om objekten man vill söka efter, sensorerna, sensorplattformarna, igenkänningsalgoritmerna och yttre omständigheter. Alla dessa faktorer påverkar vilka sensorer och igenkänningsalgoritmer som är bäst lämpade att använda.

En regelhanterare har implementerats. Denna används för att editera de regler som beskriver hur sensorerna och igenkänningsalgoritmerna påverkas av de olika yttre omständigheterna. En algoritm som använder den ontologiska kunskapsbasen och reglerna som skapats i regelhanteraren har också tagits fram. Denna algoritm fattar beslut om vilka sensorer och igenkänningsalgoritmer som är bäst lämpade att använda givet en komplett uppsättning inparametrar (eftersökt objekt, område av intresse, tidpunkt, rådande yttre omständigheter, etc).

Nyckelord

ontologi, datafusion, informationsfusion, frågespråk

Övriga bibliografiska uppgifter	**Språk** Engelska
ISSN 1650-1942	**Antal sidor:** 64 s.
Distribution enligt missiv	**Pris: Enligt prislista**

Contents

Chapter 1 - Introduction

This report is a master's thesis of the Department of Computer and Information Science at Linköping University, Sweden. The work was done at the Swedish Defence Research Agency (FOI) in Linköping, Sweden.

1.1 Objectives

The objective of this work is to develop an ontological knowledge base system that can help the ΣQL query engine (see 2.2) in the "Information system for target recognition (ISM)" project (see 2.1) to detect which sensor data should be used and which recognition algorithm(s) should be applied to that data given a specific ΣQL query. In a real-world application, a system like this needs to be populated with information from domain experts. The focus in this work, however, is on the overall design and the basic structure of the ontological knowledge base system.

1.2 Problem description

It is necessary to select appropriate sensor data and appropriate recognition algorithms when searching for objects because the amount of data captured by the sensors during a sensor platform flight is often enormous. There is too much data to apply all recognition algorithms to all the captured sensor data for every query posed. Also, certain recognition algorithms only works well with data captured by certain sensor types.

Achieving sensor data independence is important. This means that the user should not know anything about which sensor data type(s) or recognition algorithm(s) that are used to answer the query. Sensor data independence makes it unnecessary for the end-user to know anything about the sensor data.

The complexity of the problem is evident from the following list of issues that have to be considered when designing an ontological knowledge base system required to support a query system of the type in focus for this work.

- Thing to be sensed (TTBS)
- Area of interest (AOI)
- Time interval of interest (TIOI)
- Sensor characteristics
- Recognition algorithm characteristics
- Meta data such as data availability and data quality
- Weather conditions

- Light condition
- Terrain background

All of the above, except for sensor and recognition algorithm characteristics, are needed to complete the ΣQL query; directly (thing to be sensed, area of interest, time interval of interest) or indirectly (meta data, weather conditions, light condition, terrain background).

What complicates things further is that many of the factors change over time. For example light and weather conditions can change drastically during the time interval of interest. When those change, also the appropriateness of using a specific sensor and/or algorithm may very well change. For example, during a time interval of interest spanning from day to night a video camera sensor becomes much less appropriate as it gets darker.

Data availability and data quality may also change rapidly. The sensors can collect data only from the area where they are located. When searching in data already collected (TIOI ends before present time), there is no way of getting data from spots where no sensor platform has flown during TIOI. When searching in present or future time, sensor platform flights can be planned so that high quality data from the AOI can be captured. Thus, sometimes we have data from the AOI at one or more points in time during TIOI, sometimes we have data from the AOI during the entire TIOI and sometimes we might not have any data at all from the AOI at the interesting time. Moreover, the data might be of poor quality, depending on the type of sensor platform the sensor was mounted on, how far away from the AOI the platform was located, what angle (measured to ground level) the sensor had when it captured the data, etc. The object being searched for might be partly hidden by a tree or other terrain object.

1.3 Related work

A case study in building and (re)using an ontology in a specific application domain – in this case air campaign planning is described in [1].

In [2] a generic and extensible prototype platform for intelligence fusion is presented. The fusion process is designed as an “algebra” of fusion operations and the data and knowledge semantics are given explicitly by an input ontology together with descriptive models of the ontology concepts.

Ontology-based programming using the NUT language is presented in [3]. A specification method and problem-solving techniques are demonstrated by an example. Also, structural synthesis of programs – a technique essential for domain knowledge handling is briefly discussed.

1.4 Definitions

In addition to the above-mentioned issues it is also necessary to consider the following aspects and their definitions.

- View – provided by the multi-level view manager. The global view is primarily obtained from a GIS (geographic information system). The local and object views are more detailed descriptions about local areas and objects (see 2.2.3).
- Type of sensor platform – a kind of meta data describing the characteristics of the sensor platform (see 3.3.2).
- Local geographic coordinate system (LGCS). A local coordinate system with its origin in a geographic object. The LGCS is used to get a higher precision coordinate system in the area of interest (see 3.5).

Chapter 2 - Background

2.1 Information system for target recognition (ISM)

This work was done as part of the FOI project “Information system for target recognition (ISM)” [4]. The main goal of the project is to show how information from sensors attached to various airborne platforms can be used by an information system that is to be integrated into a command and control system.

Methods for classifying ground targets using different kinds of sensors are developed. The sensor types primarily used are CCD (digital camera), IR (infra red) and LR (laser radar). The CCD camera captures video images with a resolution of 3056 x 2032 pixels at an image frequency of 0,7 Hz. The IR camera captures images with a resolution of 320 x 240 pixels at an image frequency of 60 Hz from the infra red spectrum (wavelengths 8,0-9,2 µm). The LR is a scanning device sending 7 000 laser pulses per second capturing the reflections of those pulses. The time for the pulses to return and the intensity of the returned pulses are measured. Other sensor types might be used in the future.

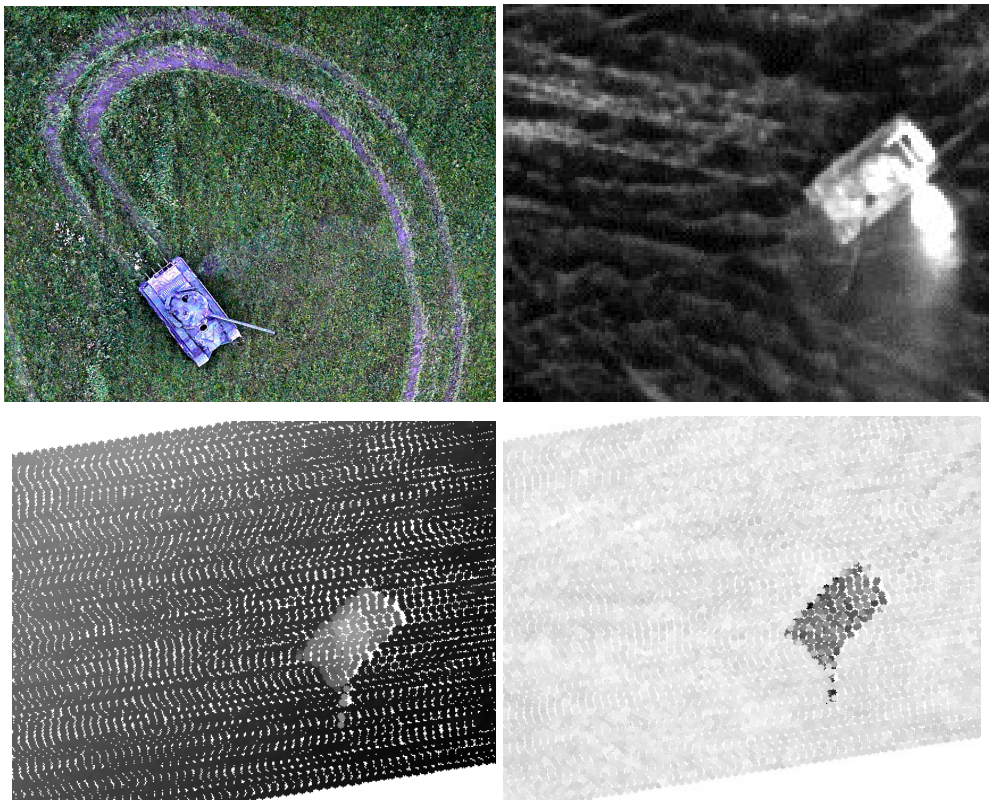

Figure 1 *T72 tank images. CCD camera (top-left), IR camera (top-right), LR height (bottom-left) and LR intensity (bottom-right).*

Example images captured by the three sensors mentioned above are shown in Figure 1.

An important part of the information system is a query language for multiple sensor data sources called ΣQL (see 2.2). The query language will be able to perform data fusion on data from the different sensors.

A high-resolution terrain model based on laser radar data is also developed and will be used to visualize and determine terrain structures, for example hills and ditches [5].

2.2 Environment (ΣQL)

The discussion of ΣQL as presented here is taken from [6]. More detailed information on ΣQL can be found in [7, 8].

To support the retrieval and fusion of multimedia information from multiple sources and databases in sensor data fusion applications, a spatial/temporal query language called ΣQL was proposed [7, 8]. Queries in ΣQL are expressible in an SQL-like syntax. The natural extension of SQL to ΣQL allows a user to specify powerful spatial/temporal queries for multimedia data sources and multimedia databases.

2.2.1 Sensor dependency tree

The sensor dependency tree is defined in [9]. The purpose of the sensor dependency tree is to keep track of the order in which various subqueries can be applied. Eventually, the returned result of these subqueries will be fused, i.e. the last operation determined by the dependency tree.

The sensor dependency tree is a tree in which each node P_i has the following parameters:

$object_i$	is the object type to be recognized
$source_i$	is the information source
$recog_i$	is the object recognition algorithm to be applied
sqo_i	is the spatial coordinates of the query originator
tqo_i	is the temporal coordinates of the query originator
aoi_i	is the spatial area of interest for object recognition
ioi_i	is the temporal interval of interest for object recognition
$time_i$	is the estimated computation time e.g. in seconds
$range_i$	is the confidence range in applying the recognition algorithm represented by two numbers min and max from the closed interval [0,1]

If a node P_2 requires input from the computation results of a node P_1, there is a directed arc from P_1 to P_2. The directed arcs originate from the leave nodes and terminate at the root node.

The information sources of the leave nodes are the sensors, i.e. laser radar, infrared camera, CCD camera, etc. The intermediate nodes of the tree return the objects to be

recognized. The result of the root node of the tree is the result of the sensor data fusion process.

2.2.2 Query refinement

The discussion of query refinement as presented here is taken from [9].

Data from a single sensor yields poor results in object recognition. This may be due to the target object being partly hidden by an occluding object such as a tree. To overcome this problem multiple sensor data fusion will be used as an efficient tool for improving query results.

Object recognition (classification and identification) can be improved if a refined query is generated, allowing the target being partly hidden. The main idea is that one or more sensors may serve as a guide to the other sensors by providing status information such as position, time and accuracy, which can be incorporated in multiple views (see 2.2.3 and 3.3.1) and formulated as constraints in the refined query.

A flowchart describing a proposed algorithm for sensor data fusion and query refinement [9] in ΣQL is shown in Figure 2.

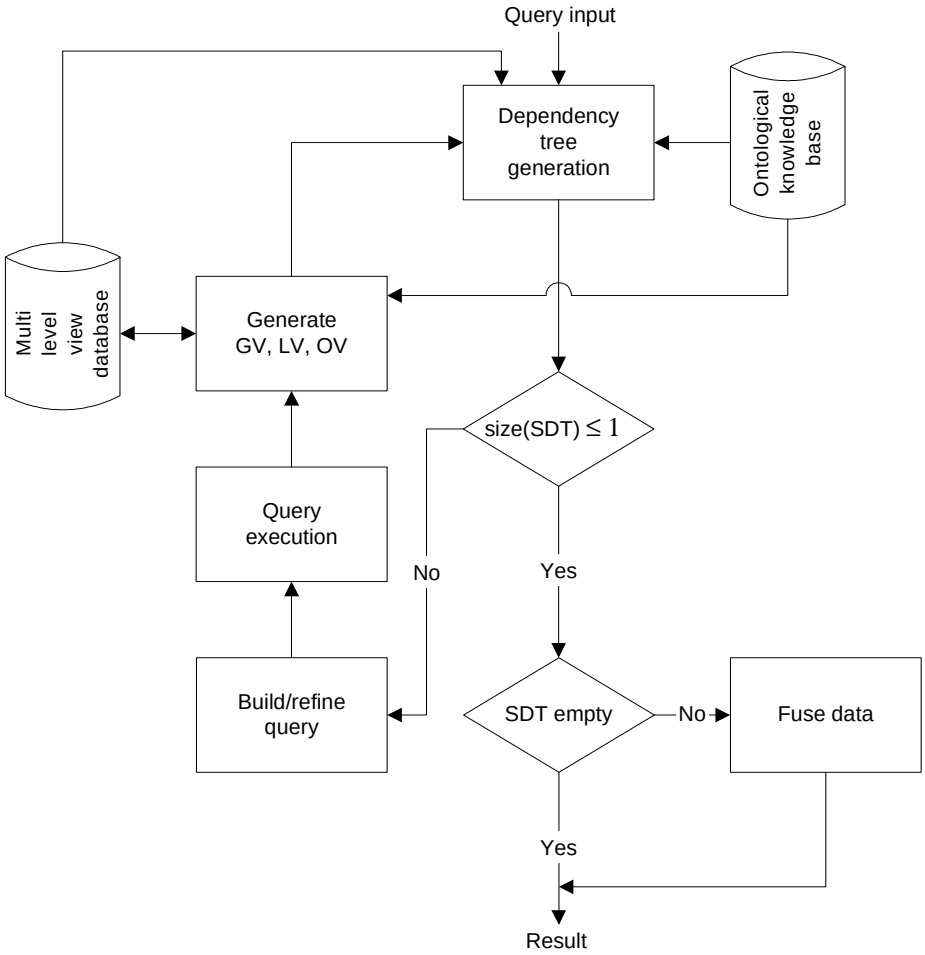

Figure 2 *Flowchart for a query refinement algorithm [9].*

2.2.3 Multi-level view manager

The discussion on the multi-level view manager as presented here is based on the discussion in [9].

From the sensors the status information is obtained, i.e. object type, position, orientation, time, accuracy, etc. This is processed by the view manager to be stored in the multi-level view database. Whenever the query processor needs some information, it asks the view manager. The view manager also shields the rest of the system from the details of managing sensory data, thus achieving sensor data independence.

The three views are organized in a hierarchical structure: the global view, the local view and the object view. The global view describes where the target object is situated in relation to some other objects, e.g. a road. This will help the sensor analysis program to more exactly locate the target object and thereby decreasing the search time since the search area decreases. The local view provides such information as the target object is partly hidden. The local view can be described, for example, in terms of Symbolic Projection [10]. Consequently, there is a need for a symbolic object description. The views can also be used later on in other tasks such as in situation analysis.

The global view is obtained primarily from GIS. The local and the object views are more detailed descriptions of local areas and objects, but the results of a query may also lead to the updating of all three views.

2.3 Ontology

Provided here is a brief explanation of the ontology concept and what it can be used for. The explanation is based on [11].

A body of formally represented knowledge is based on a conceptualization: the objects, concepts and other entities assumed to exist in some domain and the relationships that hold between them. A conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose. Every knowledge base, knowledge-based system, or agent is commited to some conceptualization, explicitly or implicitly.

An ontology is an explicit specification of a conceptualization. The term is borrowed from philosophy, where an ontology is a systematic account of existence. For artificial intelligence (AI) systems, what “exists” is that which can be represented. When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is called the universe of discourse (the “world”). This set of objects, and the describable relationships among them, is reflected in the representational vocabulary with which a knowledge-based program represents knowledge.

In [12] it is stated that we can call a specification of the entities of the subject domain and of the properties of interest to us an “ontology” for the subject domain. An

ontology of the domain naturally involves a representation of the composition of the entities of the domain.

The main reasons why someone would want to develop an ontology are [13]:

- To share a common understanding of the structure of information among people or software agents.
- To enable reuse of domain knowledge.
- To make domain assumptions explicit.
- To separate domain knowledge from the operational knowledge.
- To analyze domain knowledge.

2.4 Designing the ontology

This section describes the considerations made regarding the process of designing the ontology.

2.4.1 The ontology-editing environment

Several environments for editing ontologies are available. Some well known are Ontolingua [14], Chimaera [15] and Protégé-2000 [16]. Because developing an ontology is an iterative process it is very important that the ontology-editing environment enables easy visualization of the structure of the ontology (the "ontology tree") and provides support for the user to easily extend and alter the ontology.

To be able to use the information in the ontological knowledge base created using the ontology-editor, it must be easy to access the information from the system or systems that are designed to use the ontological knowledge base.

Protégé-2000 was chosen as the ontology-editing environment because it best suits the needs of the present study. The primary reason is that it is an easy-to-use product that works well. Furthermore it is developed in Java and has a Java API by which it is easy to integrate the ontological knowledge base constructed using the Protégé-2000 application into the Java programs that implement this study. Java is used here because the surrounding system will be implemented in Java.

2.4.2 Purpose of the knowledge system

The overall purpose of the knowledge system is to provide apriori information about the world (the universe of discourse) in a structured and formal manner so that all (sub) systems that need to know something about the world could use the knowledge system to obtain that information. The primary application of the knowledge system is also the objective of this thesis.

2.4.3 Ontology development process

When developing the ontological knowledge structure a process based on guidelines was used. The process is a step-by-step method described in [13]. It is independent of

the ontology-editing environment, but the examples in [13] are developed using Protégé-2000.

Other more advanced ontology development methodologies like the one in [17] which benefits from the increasing amount of available electronic texts and of the maturity of natural language processing tools are proposed. However, those methodologies are considered too immature to be used in this work.

2.4.4 Stand-alone application

An ontological knowledge base system can be built in a few different ways. Protégé-2000 has a graphical user interface with a plugin system that enables users to create their own subsystems to implement their own applications, using the Protégé knowledge base, inside the Protégé environment. Another way is to write a standalone-application for implementing the algorithms to be applied to the information in the knowledge base. Stand-alone applications written in Java can easily access knowledge bases created by using Protégé-2000, because a Java API is provided to access the knowledge base of a Protégé project.

The results of this study are implemented as a stand-alone Java application using the Protégé API to access the knowledge base created by using the Protégé-2000 application. This way the algorithms are separated from the data modeling (the knowledge).

Chapter 3 - Knowledge structure (ontology)

The knowledge represented in the ontological knowledge base is modeled in a hierarchical manner known as the ontology (also called the ontology tree). All concepts in the universe of discourse, the interesting properties of the concepts and the important relations between the concepts are modeled.

The hierarchy has the ultimately general concept called `Thing` at the top. All other concepts inherit directly or indirectly from `Thing`. This means that "Everything is a thing". The concept `Thing` has no properties and no relations; it just acts as the parent of all other concepts. The hierarchy is organized so that more specialized concepts appear further down the inheritance chain.

The concepts of the ontology are divided into three main parts. One part models everything that can be sensed by the sensors and everything that can be recognized by the recognition algorithms (see 3.1). The second part models the characteristics of the sensors and the recognition algorithms (see 3.2). The third and last part models all the conditions that have impact on the appropriateness of the sensors and recognition algorithms (see 3.3).

When the ontological structure, as described above, has been created it is populated with instances. For example in Figure 4 on page 12 the `Tank` concept is shown. Different tank instances, e.g. Leopard, T-72 and others are added to the ontology as instances of that concept. When instances are added to an ontology the ontology becomes a knowledge base. In reality, there is a fine line between where the ontology ends and the knowledge base begins [13]. In this work, the ontology filled with instances will be called the ontological knowledge base, or simply the knowledge base.

3.1 Things to be sensed and recognized

This part of the ontology tree models everything that can be sensed by the sensors and everything that can be recognized by the recognition algorithms. It is represented in the ontology tree (see Figure 3) by the `ThingToBeSensed` concept and subconcepts. Two subbranches exist: `PropertyToBeSensed` and `RecognizableObject`.

The `ThingToBeSensed` concept has the relation `HasAppropriateSRA` describing which combinations of sensors and recognitions algorithms (SRAs, see 3.2.4) are appropriate for finding the `ThingToBeSensed`. This is an extremely important relation for the AFFAS algorithm, which is the algorithm designed to determine which sensors and which recognition algorithms to apply in a certain query (see 4.1).

Note that relations are inherited, so a relation that exists in `ThingToBeSensed` also exists in all concepts that inherit from `ThingToBeSensed`.

3.1.1 Property to be sensed

This models the attributes (e.g. color) and status values (e.g. orientation) that the recognizable objects might have. Attributes and status values are different kinds of object properties where the latter may change more frequently over time than the former. Representing object properties separately enables specific SRAs to be applied to determine specific attributes and status values, not only to determine recognizable objects. This is represented in the ontology tree by the `PropertyToBeSensed` concept.

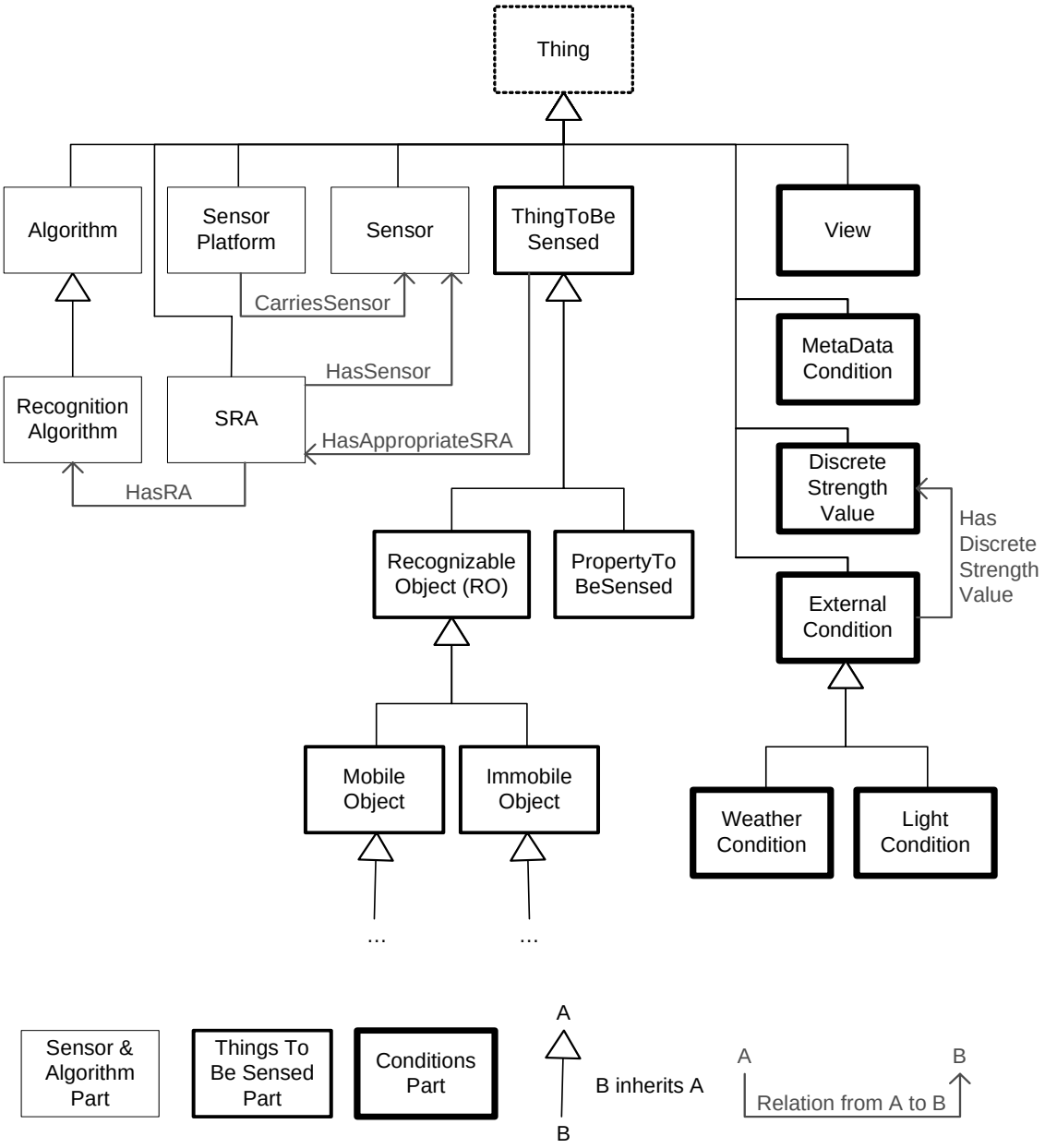

Figure 3 *Ontology overview describing the general knowledge structure.*

3.1.2 Recognizable object

This models the objects that can be recognized by the recognition algorithms. Notice that every recognizable object (RO) has the relation `HasAppropriateSRA` (see 3.4) because every RO inherits from `ThingToBeSensed`. It is represented in the ontology tree by the `RecognizableObject` concept.

Recognizable objects are further divided into mobile and immobile objects.

Mobile object

Mobile objects are objects that can move. Therefore they are not suitable as reference points in an LGCS (see 3.5). The main interest in the ISM project is recognition of ground vehicles; therefore the ontology is populated with vehicles that are taxonomically located under `MobileObject`, see Figure 4.

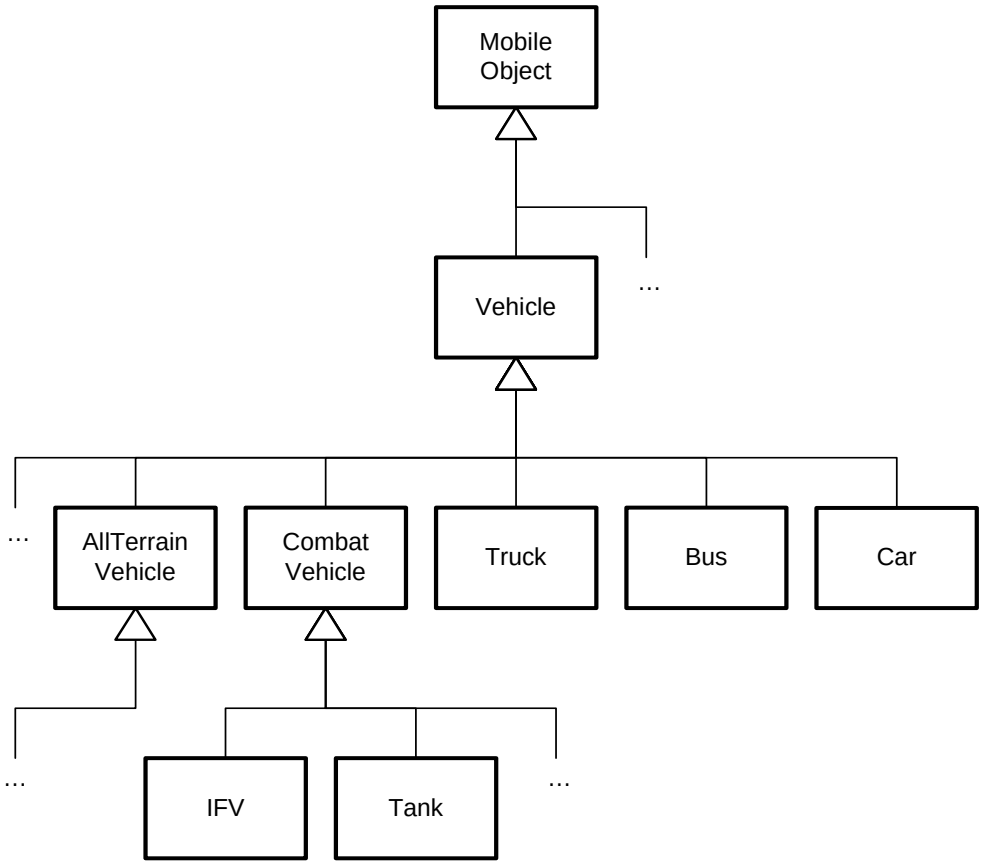

Figure 4 *Knowledge structure for mobile objects.*

Immobile object

Immobile objects are objects that cannot move. Important immobile objects are different kinds of terrain objects and buildings. These are appropriate as reference points in an LGCS as they do not move, see Figure 5.

The `Vegetation` concept is used for modeling the condition terrain background. Therefore `Vegetation` not only fits into the `ThingsToBeSensed` part of the

ontology but also fits into the `Conditions` part because it models a condition that is considered in the AFFAS algorithm (see 4.1).

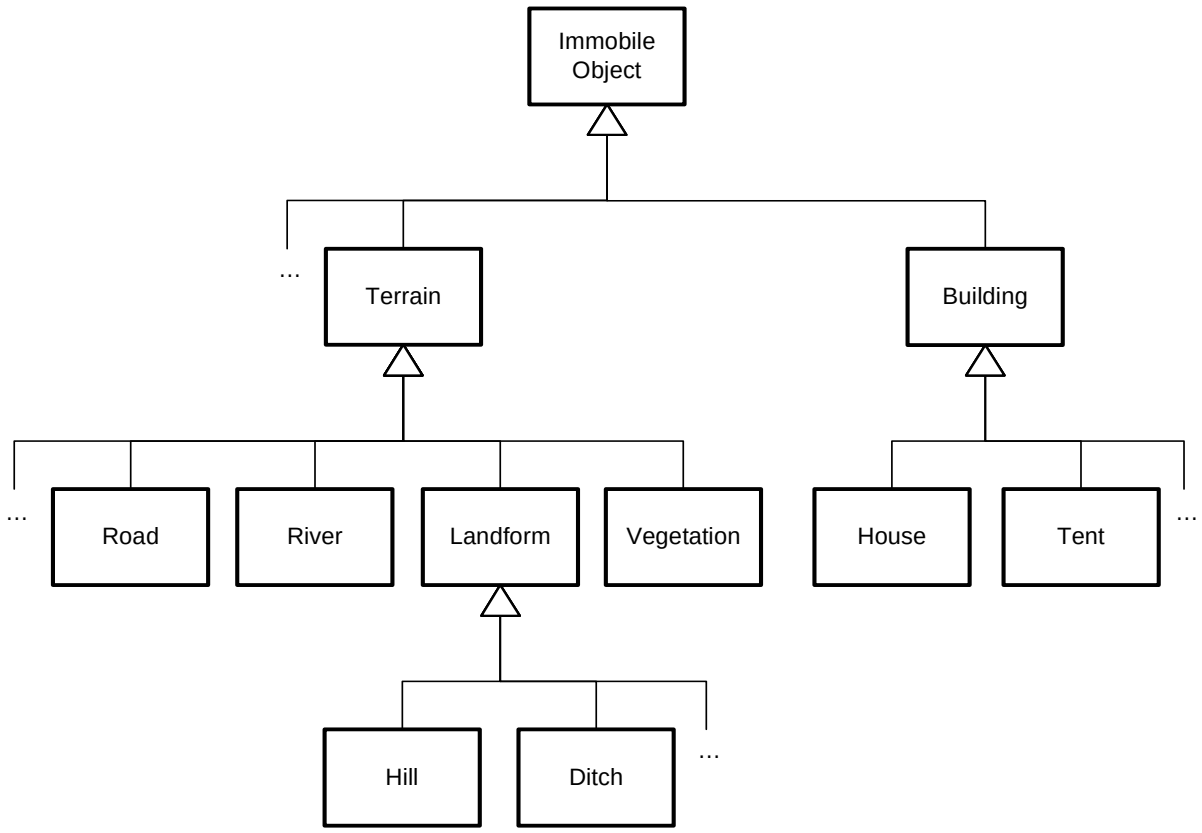

Figure 5 *Knowledge structure for immobile objects.*

3.2 Sensor characteristics and recognition algorithms

This part of the ontology tree models the sensor characteristics and the recognition algorithms, see Figure 3.

3.2.1 Sensor platform

The `SensorPlatform` concept models the available sensor platforms. This concept is also used for modeling the condition sensor platform, where the type of the platform is taken into consideration. Therefore `SensorPlatform` not only fits into the `Sensor & Algorithm` part of the ontology but it also fits into the `Conditions` part because it models a condition that is considered in the AFFAS algorithm, see Figure 3. The `SensorPlatform` concept has the relation `CarriesSensor` describing which sensors the platform carries.

3.2.2 Sensor

The `Sensor` concept is used for modeling sensors. A sensor has a type. The list of types available can be changed in the ontology-editor, but in the test ontology the list includes the following sensor types:

- CCD (video)
- IR (infra red)
- LR (laser radar)
- Acoustic
- Seismic

Only the first three types are used in the test cases, because the other sensor types are not intended to be mounted on flying sensor platforms. They are included in the ontology only because other systems using these sensor types might later be integrated into the system.

3.2.3 Algorithm

This models algorithms used for extracting information from the sensor data collected by the sensors. The most important type of algorithm is `RecognitionAlgorithm`. Later, other types of algorithms, like detection algorithms, might very well be modeled. That is why the general `Algorithm` concept is included.

The `RecognitionAlgorithm` concept models the recognition algorithms used for recognizing different types of things, primarily ground vehicles but also all the other things we want to recognize.

3.2.4 SRA

`SRA` (`Sensor-RecognitionAlgorithm`) is a very important concept since it models the combination of a sensor and a recognition algorithm. The combination is modeled by the two relations `HasSensor` and `HasRA`. The `HasSensor` relation connects the `SRA` to a `Sensor` and the `HasRA` relation connects it to a `RecognitionAlgorithm`.

The `SRA` concept is introduced because some recognition algorithms work only with one sensor type, whereas others work with several sensor types. To model this, an SRA is created in the ontological knowledge base for every sensor and recognition algorithm combination that works together.

Having all appropriate combinations of sensors and recognition algorithms modeled in the ontology, we can work with SRAs instead of sensors and recognition algorithms separately when trying to decide which sensors and recognition algorithms are the most appropriate in a specific situation (see 4.1).

3.3 Conditions

This part of the ontology tree models the conditions that have an impact on the appropriateness of the sensors and the recognition algorithms. The conditions are state conditions describing the state of something, for example how rainy it is. State conditions in an ontology the way it is done in this work seem to be a rather unique knowledge structure as far as the author knows.

3.3.1 View

The view concept is modeled in the ontology tree by `View`. See 1.4 for the definition of view.

3.3.2 Meta data condition

The meta data conditions are represented in the ontology tree by the `MetaDataCondition` concept. Meta data means data about data. In this case we are interested in the quality of the data and also what sensor platform that captured the data.

Data quality

The data quality is represented by a number in the interval [0..1] where 0 means completely useless or nonexistent data and 1 means that the data quality is perfect. Data quality is not modeled in the ontology; instead it is controlled by the meta data manager. However, the `MetaDataCondition` concept has an instance `DataQuality` that indicates that data quality is a (meta data) condition that should be considered by the AFFAS algorithm.

Sensor platform

Sensor platform is a condition that is modeled much like the terrain background condition (see 3.3.5). It uses the instances of the `SensorPlatform` concept to make up its range of permitted values. The `MetaDataCondition` concept has an instance `SensorPlatform` that indicates that the sensor platform is a (meta data) condition that should be considered in the AFFAS algorithm.

3.3.3 External condition

External conditions are represented in the ontology tree by the `ExternalCondition` concept. External conditions are weather conditions (`WeatherCondition` concept) and light condition (`LightCondition` concept). Seasonal conditions like “There is snow on the ground” or “There is ice on the lakes” or “There are no leaves on the trees in deciduous forests” are just special cases of the more general concept `WeatherCondition`. However, no special subconcept to weather condition (for example `SeasonCondition`) has been created because the semantic difference is not important in this work. Thus, season conditions will be instances of the `WeatherCondition` concept.

The `ExternalCondition` concept has a relation `HasDiscreteStrengthValue` that provides the connection between external conditions and the corresponding discrete strength values.

3.3.4 Discrete strength value

Discrete strength value is represented in the ontology tree by the `DiscreteStrengthValue` concept.

A condition (as described above) has a specific value at a given time and place. A discrete range of permitted values has been defined for each condition. For example `Rain` might have the following discrete range of permitted values: { `Dry`, `Gentle`, `Heavy` }.

The `DiscreteStrengthValue` concept models all discrete strength values for all external conditions. The `HasDiscreteStrengthValue` relation provides the connection between external conditions and the corresponding discrete strength values.

3.3.5 Terrain background (vegetation)

Terrain background is a condition that is somewhat unique in the way it is modeled. This condition is modeled using the `Vegetation` concept in the `ThingsToBeSensed` part of the ontology tree. This is because all types of vegetation known by the system are modeled using that concept, and that is exactly what the terrain background condition wants to take into consideration. Therefore this condition uses the instances of the `Vegetation` concept to make up its range of permitted values. Examples of terrain backgrounds are sand, water and grass.

3.4 Relations

Relations are used to model how the concepts in the ontology are related to each other. Five relations are defined in this study. It is important to note that relations are inherited, meaning that if concept B inherits concept A and concept A has a relation to C, then concept B automatically has that relation to C as well. An example of relation inheritance is that the `Tank` concept has the `HasAppropriateSRA` relation because `it is` inherited from the `ThingToBeSensed` concept where that relation is defined.

3.4.1 Inverse relations

In the context of ontologies a relation is always defined to be *from* a certain concept *to* another concept, e.g. the `HasSensor` relation is defined to be *from* the `SRA` concept *to* the `Sensor` concept. Of course there is an inverse relation saying that a `Sensor` is part of an `SRA`. However, this is not explicitly defined, because the inverse relation is not required here. In all relations defined in this study it is only necessary for one of the concepts of the relation to know that the relation exists. Therefore inverse relations are not explicitly defined.

3.4.2 Descriptions of defined relations

`HasRA` and `HasSensor` in the `SRA` concept model the fact that an `SRA` is built up of a sensor and a recognition algorithm. The relations are both “1 to 1”, meaning that every single SRA has exactly one sensor (`HasSensor`) and exactly one recognition algorithm (`HasRA`).

`CarriesSensor` in the `SensorPlatform` concept models the fact that a sensor platform carries sensors. The relation is “1 to 1..*”, meaning that every single sensor

platform carries at least one sensor, but with no upper limit to the number of sensors it may carry.

`HasAppropriateSRA` in the `ThingToBeSensed` concept is an extremely important relation. It is used to model which of the SRAs that are appropriate when searching for a specific `ThingToBeSensed`. Because relations are inherited, all subconcepts to `ThingToBeSensed` also have this relation. The relation has a strength, an appropriateness value (Ap), in the interval [0..1] describing how appropriate the SRA is. This Ap value is an apriori value; no external factors are considered. When looking at the list of issues that must be considered (see 1.2), this relation models one and only one of those things: "Thing to be sensed". All the other factors that have to be taken into consideration are modeled in other ways. The relation is "1 to 0..*", meaning that for every single `ThingToBeSensed` there may be anything from none to an infinite number of (apriori) appropriate SRAs.

`HasDiscreteStrengthValue` in the `ExternalCondition` concept models the fact that an external condition has a discrete range of permitted values. Every permitted value is a discrete strength value, and this relation indicates which discrete strength values are in the range of permitted values for a specific external condition. The relation is "1 to 2..*", meaning that for every single `ExternalCondition` the range has to include at least two strength values (otherwise it would be completely meaningless) and that there is no upper limit to the number of discrete strength values that may be defined to be in the range.

3.5 Local geographic coordinate system (LGCS)

It is possible to create a local coordinate system with higher precision than the one normally obtained from the positioning system in the sensor platforms. This local coordinate system will be used to make it easier to perform sensor data fusion on data from different sensors (possibly mounted on different sensor platforms). When fusing data from different sources, it is very important to know which data in the different sources that represents the same object. This becomes easier with a more precise coordinate system.

A suitable object in the terrain is chosen as the reference point (origo) for the local coordinate system. An object can be more or less appropriate for being used as a reference point. The system is designed so that any immobile object can be used as reference point for a local coordinate system. The appropriateness of a specific immobile object to be used as a reference point is modeled in the respective concept in the ontology. The `ImmobileObject` concept in the ontology has a property called `GeoPositioningAppropriateness` to model this. The value of the `GeoPositioningAppropriateness` property is in the interval [0..1], where 0 means completely inappropriate and 1 means perfectly appropriate. The term geopositioning is used throughout this work to describe the appropriateness for objects being used as reference points in an LGCS.

3.5.1 Recognition algorithm object positioning appropriateness

Recognition algorithms are more or less appropriate for finding objects suitable as reference points in an LGCS because the algorithms are more or less suitable for estimating the position of the object they recognize. This fact is modeled in the `RecognitionAlgorithm` concept using a property called `ObjectPositioningAppropriateness`. The value of this property is in the interval [0..1], where 0 means completely inappropriate and 1 means perfectly appropriate.

3.6 Ontology implementation

The ontology is implemented as a Protégé-2000 project. A description of the terms used by Protégé-2000 and how those terms connect to the general ontology terms is provided here to give a better understanding of how Protégé-2000 handles ontologies.

3.6.1 Classes

Classes in Protégé are used to model concepts. To create a new concept, a new class is created. The class is created as a subclass (=subconcept) of some existing class. The parent of all concepts is `:THING` which is predefined in every Protégé project.

3.6.2 Slots

Slots are used to model properties of concepts. The following is a description of slots as described in [13].

Several types of properties can become slots in an ontology:

- "Intrinsic" properties such as the flavor of a wine
- "Extrinsic" properties such as a wine's name, and the area it comes from
- Parts, if the object is structured; these can be both physical and abstract "parts" (e.g. the courses of a meal)
- Relationships to other individuals; there are relationships between individual members of the class and other items (e.g. the maker of a wine, representing a relationship between a wine and a winery)

Slots are inherited, so all properties (including relations) will automatically be available in all subclasses to the class where they are defined.

Slots can have different facets (restrictions) describing the value type, allowed values, the number of values (cardinality), and other features.

Cardinality

The cardinality facet of a slot describes how many values a slot can have. In Protégé it is possible to define a minimum and a maximum cardinality. The maximum cardinality may be set to infinity. This is accomplished in Protégé by defining the cardinality as "multiple". Instead of saying the minimum cardinality is 1, it is possible to say it is "required".

Value type

A value type facet describes what types of values are allowed in the slot. In Protégé there are several value types for slots; String, Number, Boolean, Enumerated and Instance. Enumerated specifies a list of specific allowed values. The Instance value type is used for modeling relations. Slots with value type Instance must also define a list of allowed classes from which the instances can come.

3.6.3 Instances

When all classes and slots are defined, it is time to create individual instances of the classes in the hierarchy. This is a matter of first choosing for which class to create an instance, then creating an individual instance and finally filling in the slot values of the newly created instance.

3.7 Test model

The ontology presented in this thesis is populated with a small set of sample data. The final ontology may or may not include the concepts of the sample ontology which is used for demonstration and testing purposes. However, the main structure (upper levels in the ontology tree) is general enough to be left very much the way it is. The relations defined between those concepts are also meant to be useful in the future, no matter what concepts might be added or removed from the lower levels of the ontology tree.

3.7.1 Concepts that may change or be added

Concepts modeling other algorithm types (for example a concept for detection algorithms) may be added as children to the `Algorithm` concept (see 3.2.3). Moreover, other conditions may be added under the `ExternalCondition` concept (see 3.3.3).

Little effort has been put into modeling object properties, which will be subconcepts to `PropertyToBeSensed`. For example, modeling the concept of color is a matter of adding a subconcept called `Color` to `PropertyToBeSensed`, adding instances to `Color` that represent the different colors to be modeled and finally creating a relation called `HasColor` from the most general concept that may have a color to `Color`.

Most object properties are simple and have either a number or a string as the value type. To model such properties it is not necessary to add a new concept. Instead, a slot with the required value type is added to the Protégé class representing the most general concept that has the property.

Finally, a lot may happen below the `MobileObject` and `ImmobileObject` concepts (see Figure 4 and Figure 5 on pages 12 and 13). All concepts defined there should be looked upon as sample concepts, even though some effort has been put into making those sample concepts as realistic as possible. Domain experts must probably remodel this part of the ontology before the system can be used for its real purpose.

3.8 Working with the ontology and the knowledge base rules

Documentation describing the details of how things are modeled in Protégé, how to add new information to the ontology, how to edit the rules and how to use all that information from the application implementing the intelligence of the system is provided in appendix A.

Chapter 4 - Appropriate sensors and algorithms

As stated in the objectives of this study, the ontological knowledge base has been designed to help answer such questions as which sensor data to use under certain circumstances. Also important is which recognition algorithm(s) that should be applied. An algorithm that does just this has been developed. That is, it uses the knowledge in the ontological knowledge base and the rules that are defined using the Knowledge Base (KB) rule manager application to discover which sensors and recognition algorithms are the most appropriate under the given circumstances, i.e. the actual ΣQL query, the meta data conditions, the external conditions and the terrain background.

4.1 Algorithm For Finding Appropriate SRAs (AFFAS)

This is the algorithm developed to find the appropriate combinations of sensors and recognition algorithms (SRAs).

4.1.1 Description of terminology

This section presents a description of the terminology used when talking about AFFAS.

Discrete strength value

The purpose of a discrete strength value is to keep information about a value that an external condition may have. All the values that the conditions may have are represented in the ontology as discrete strength values.

In addition to what is stated in 3.3.4, some conditions (the conditions that are not external) have discrete ranges of permitted values that are not defined by the `DiscreteStrengthValue` concept in the ontology. These conditions are “Type of sensor platform”, “Terrain background” and “View”. However, those ranges consist of all the instances of the ontology concepts `SensorPlatform`, `Vegetation` and `View`, respectively. The advantage of this design is that when a new sensor platform, type of vegetation or view is added to the ontology, automatically and immediately the entire system will know that the corresponding discrete range of permitted values has been expanded.

Impact factor

The purpose of an impact factor is to keep information about one of all the things that impact the appropriateness of an SRA.

An impact factor impacts the result in step 3 of AFFAS (see Figure 6) and corresponds to one of the conditions that are considered. The impact factor describes

how much the corresponding condition impacts a certain sensor or a certain recognition algorithm given the current strength value of the condition. Thus, for every condition, there is a corresponding impact factor. The impact factor depends not only on the current strength of the condition, but also on the sensor/RA. Therefore, a certain impact factor can have different values (impact strength values, see below) for different sensors and recognition algorithms.

Impact strength value

The purpose of an impact strength value is to keep information about a value that an impact factor can have.

Each impact factor has an impact strength value that describes how strong the impact is. There is a discrete set of impact strength values. This set may be subject to change in the future, but at the moment it corresponds to the following ordered value set:

{ None, Little, Medium, Strong, Complete }

None means that the impact factor does not impact the performance of the sensor/RA at all. Complete means that the sensor/RA is completely useless under that condition. Little, Medium and Strong make up the range between the two extremes. In other words, this value set is qualitative.

Which impact strength value a certain impact factor has is determined by the knowledge base rules.

4.1.2 AFFAS methods for weighting together subresults

Step 4 of the AFFAS algorithm weights together all the impact factors from step 3 and applies those to the SRAs from step 2. The impact factors can be weighted together in numerous ways. Doing so in an optimally, or even in a good way is difficult, mainly because the impact factors depend on each other.

Two different methods for weighting together impact factors have been chosen; one multiplicative and one additive. By using two different kinds of weighting methods it is possible to evaluate which one performs better in certain situations. Also, using two weighting methods in parallel increases the belief in the result when both methods return similar results.

Simple multiplicative method

This method assigns a rational number in the interval [0..1] to each impact strength value as shown in Table 1. It then uses those values when multiplying the impact factors together to obtain a factor describing the total impact of all impact factors. This total impact factor is then multiplied with the Ap value of the corresponding SRA.

The multiplicative method can be described by the following formula:

$$WeightedSRAAp = SRA_{Ap} \cdot \prod_{j} impfact_{j}$$

WeightedSRAAp = Weighted appropriateness value of the SRA
SRA_{Ap} = Ap of the SRA before weighting in the impact factors
impfact = impact factor using numbers assigned to impact strength values as described in Table 1.

From the formula it can be concluded that higher *WeightedSRAAp* means more appropriate SRA.

Table 1 *Impact strength value numbers for the multiplicative method.*

Impact strength value	*Assigned number*
None	1.0
Little	0.75
Medium	0.5
Strong	0.25
Complete	0.0

Weighted sum method

This method assigns an integer number to each impact strength value as shown in Table 2. It then uses those integer values when adding the impact factors together to obtain a sum describing the total impact of all impact factors. This total impact sum is then added to the weighted sum in the corresponding SRA.

The weighted sum method can be described by the following formula:

$$WeightedSRASum = SRA_{WSum} + \sum_{j} impfact_{j}$$

WeightedSRASum = Weighted appropriateness sum of the SRA
SRA_{Wsum} = Appropriateness sum of the SRA before weighting in the impact factors
impfact = impact factor using numbers assigned to impact strength values as described in Table 2.

From the formula it can be concluded that higher *WeightedSRASum* means more appropriate SRA.

Table 2 *Impact strength value numbers for the weighted sum method.*

Impact strength value	*Assigned number*
None	5
Little	4
Medium	3
Strong	2
Complete	1

It might be more intuitive to give *Complete* the value 0 (zero) instead of 1, because the sensor/RA is supposed to be useless under that condition. However, the numbers assigned here are not intended to be perfect. More research is needed to be able to assign optimal numbers. This thesis does not focus on choosing the correct numbers for either the multiplicative or the weighted sum methods. The assignments of numbers made here are assumed to be good enough for testing purposes.

Multiplication factors

The things that impact the choice of SRA that do not have discrete strength values are thing to be sensed, geopositioning (appropriateness for finding objects suitable as reference points in an LGCS), recognition algorithm object positioning and data quality. Those do not have impact strength values with weight factors as defined above; instead they have multiplication factors. Those multiplication factors are used in the weighted sum method to multiply the value in the interval [0..1] obtained for each of those four factors. Those rational numbers are designed to be directly used in the multiplicative method, but must be multiplied by a multiplication factor to get a reasonable weight before they are added to the weighted sum.

Example:

The data quality is 0.85. This is multiplied by the “Ap” of the SRA in the multiplicative method. To let data quality impact the weighted sum of the SRA in a reasonable way, 0.85 is multiplied by a multiplication factor of 10.0 before it is added to the weighted sum.

The exact values of the example above are of course not important. It works the same way for thing to be sensed, geopositioning and recognition algorithm object positioning. The multiplication factors need not be the same for all factors, thus data quality may have a multiplication factor that differs from the multiplication factor for geopositioning.

4.1.3 Handling of time interval of interest (TIOI)

The ΣQL queries span across a time interval. During this time interval of interest (TIOI), much can happen. It may get dark, it may start raining, sensor platforms may not have captured data from the area of interest during the entire TIOI, etc. Therefore,

the AFFAS algorithm must know the timestamp inside TIOI that is the currently interesting timestamp.

The graphical overview of AFFAS (see Figure 6) shows that the system performing the actual search in the sensor data must ask the ontology manager to run the AFFAS algorithm for every interesting timestamp in TIOI, because which sensors and recognition algorithms that are appropriate may very well change during TIOI.

This should not be too much of a problem, as the system performing the search must choose some (or all) timestamps in TIOI and search in data with those timestamps. It might not be necessary to rerun AFFAS for every new timestamp being searched if the time interval between the searched timestamps is short. The problem of choosing timestamps in TIOI intelligently and running AFFAS only when it is necessary is not the focus in this work. Therefore it is assumed that AFFAS will be run for every important timestamp.

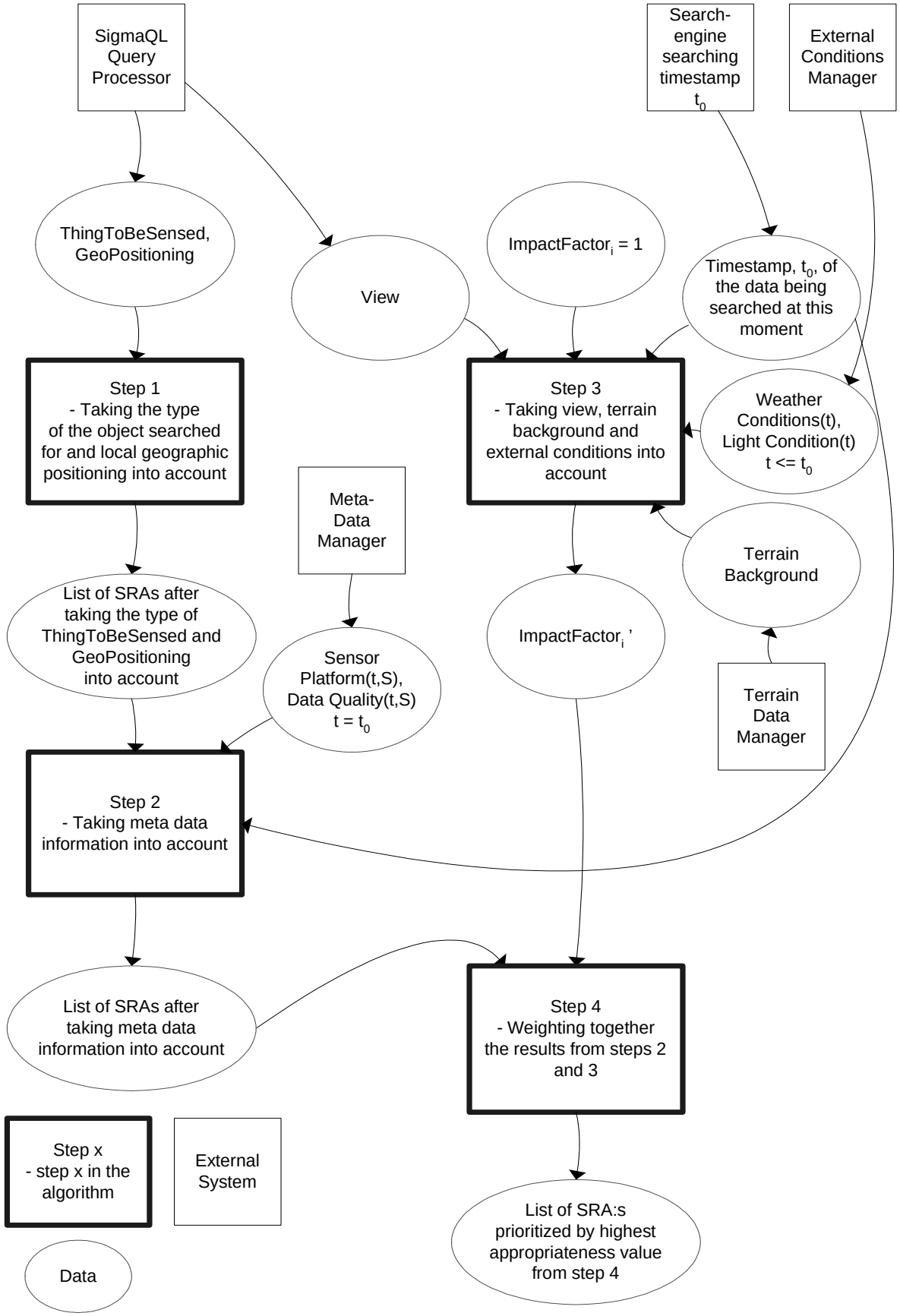

Figure 6 *Graphical overview of AFFAS.*

4.1.4 Overview of AFFAS

A graphical overview of the AFFAS algorithm is presented in Figure 6. A short description of the four steps the algorithm consists of is provided below.

Step 1

The following things are considered in step 1.

- Thing to be sensed
- Geopositioning (including RA object positioning appropriateness)

This step creates a list of appropriate SRAs with respect only to the type of thing to be sensed and geopositioned. Each SRA has an appropriateness value (“Ap”) and a weighted sum describing its appropriateness connected to it.

Step 2

The following things are considered in step 2.

- Meta data conditions
  - Type of sensor platform
  - Data quality

This step alters the appropriateness values and adds to the weighted sums of the SRAs from step 1 with respect to the information in the meta data.

Step 3

The following things are considered in step 3.

- External conditions
  - Weather conditions
  - Light condition
- View
- Terrain background

This step creates impact factors describing how strong the impacts are from the current external conditions, the view and the terrain background on each sensor and each recognition algorithm that exists in the SRAs in the result from step 2.

Step 4

In step 4 the results from step 2 and step 3 are weighted together.

This step alters the “Ap” values and adds to the weighted sums of the SRAs in the result from step 2 according to the result from step 3. In the list created in this step, the SRAs are prioritized according to the weighted appropriateness values. The most appropriate SRA is sorted as the first element on the list, the second most appropriate follows as the second element, and so on.

A detailed description of the AFFAS algorithm is presented in appendix B.

4.2 Knowledge base rules

In the process of deciding upon appropriate sensors and algorithms it is necessary to have rules describing under which conditions certain sensors and algorithms are appropriate. These rules are constructed using the KB rule manager application and are saved in a special file.

The rules that are used to decide how the impact factors impact the sensors and recognition algorithms in the SRAs can be written in the following form:

"*If* an impact factor **x** has the discrete strength value **y** *then* the impact on the sensor/RA **z** is impact strength value **v**"

Example 1:

"*If* the impact factor **Rain** has the discrete strength value **Gentle** *then* the impact on the RA **BuildingAlgorithm** is impact strength value **Little**"

Example 2:

"*If* the impact factor **View** has the discrete strength value **Local** *then* the impact on the sensor **Standard CCD Sensor** is impact strength value **None**"

4.3 KB rule manager application

The knowledge base rule manager application, sometimes abbreviated as the KB rule manager, is a tool for defining and editing the rules that the AFFAS algorithm uses to do its job.

For AFFAS to work, a complete set of rules must be defined, i.e. for every sensor and for every recognition algorithm there must be a rule defined for every discrete strength value of every condition.

In the manager the user can choose to edit rules for either sensors or recognition algorithms. This selection is done using the combo box at the top (see Figure 7). Depending on the choice of editing sensor or recognition algorithm rules, the user can choose which specific sensor/RA to edit the rules for by using the Previous and Next buttons at the bottom (where the name of the sensor/RA currently being edited is displayed). When the sensor/RA has been selected it is time to take a look at the main panel of the application. The leftmost column lists all conditions. For every row (=a specific condition), the different columns indicate the names of the discrete strength values in the range of permitted values for that condition.

Once the condition (a row) and the discrete strength value (a column) to edit the rule for have been found, there is a combo box that can be clicked and from which the impact strength value for that rule can be chosen.

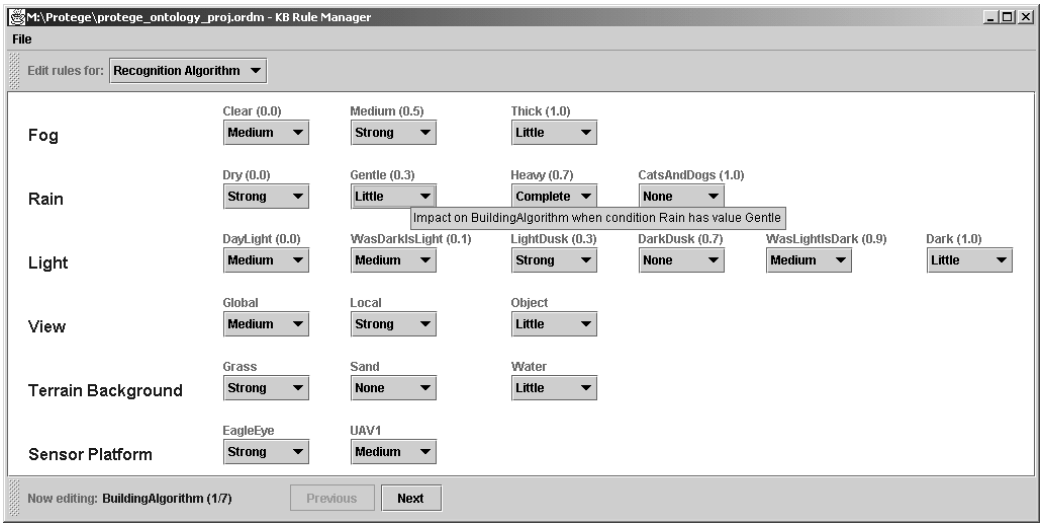

Figure 7 *Graphical user interface of the KB rule manager application.*

The tool tip in the screenshot above from the KB rule manager application indicates the first sample rule from 4.2. The screenshot shows the user editing the rules for recognition algorithm `BuildingAlgorithm`, which is recognition algorithm 1 of 7 (see bottom of screenshot).

Each cell in the table made up by the rows of conditions and the columns of discrete strength values corresponds to a rule for the selected sensor/RA. When another sensor/RA is selected, the combo boxes change to reflect the defined rules for the newly selected sensor/RA. Each combo box indicates the current value of a rule.

4.3.1 Rule Data Model

The rule data model, sometimes called the ontological rule data model, is used for storing all the rules. All the rules defined are stored in the rule data model and eventually used by the AFFAS implementation. The rule data model file format is described in appendix C.

Chapter 5 - AFFAS evaluation

5.1 Test introduction

The AFFAS algorithm uses the knowledge base rules collected in a rule data model to decide which sensors and algorithms to recommend as appropriate for a certain task. Three different scenarios will be used to show how the rules affect the result of the AFFAS algorithm.

The external managers must take the area of interest and timestamp into consideration so those systems can provide correct information about the respective conditions at the time of interest in the area of interest. The external managers are test implementations that are fed from the AFFAS test application with the information they will provide to the AFFAS algorithm. Therefore, area of interest and timestamp are not considered in the tests.

Thing to be sensed and geopositioning are not handled by the rules. Information for handling such things is directly entered into the ontology using the ontology-editing environment. Therefore, several test objects to search for (thing to be sensed) have been created and different appropriateness values for geopositioning for those objects are used. Also, varying object positioning appropriateness values for the recognition algorithms to be applied are used. The impact of those factors will be evaluated.

5.2 AFFAS test application

To test the implementation of the ontology manager and the AFFAS algorithm, a test application with which it is possible to set up a scenario using a graphical user interface has been developed. All parameters that affect the AFFAS algorithm can be set up independently to form a test scenario. When all parameters are set, the user can push the “Find appropriate SRAs” button at the top (see Figure 8) to feed the data into the external managers and the AFFAS algorithm. The input fields in the application are grouped according to which system the data entered into the fields is to be fed into.

Figure 8 *Graphical user interface of the AFFAS test application.*

5.3 Test setup

In this section the information used in the test scenarios is described.

In Figure 9 - Figure 14 (part of) the ontology tree is displayed in the leftmost panel (classes). In the middle panel (direct instances) the instances of the class selected in the leftmost panel are displayed. The rightmost panel displays a form where the properties of the direct instance selected in the middle panel can be viewed and altered.

5.3.1 Ontology test sensors and test recognition algorithms

The sensors used in the tests are:

- Standard CCD Sensor (type CCD)
- Standard IR Sensor (type IR, see Figure 9)
- Standard LR Sensor (type LR)

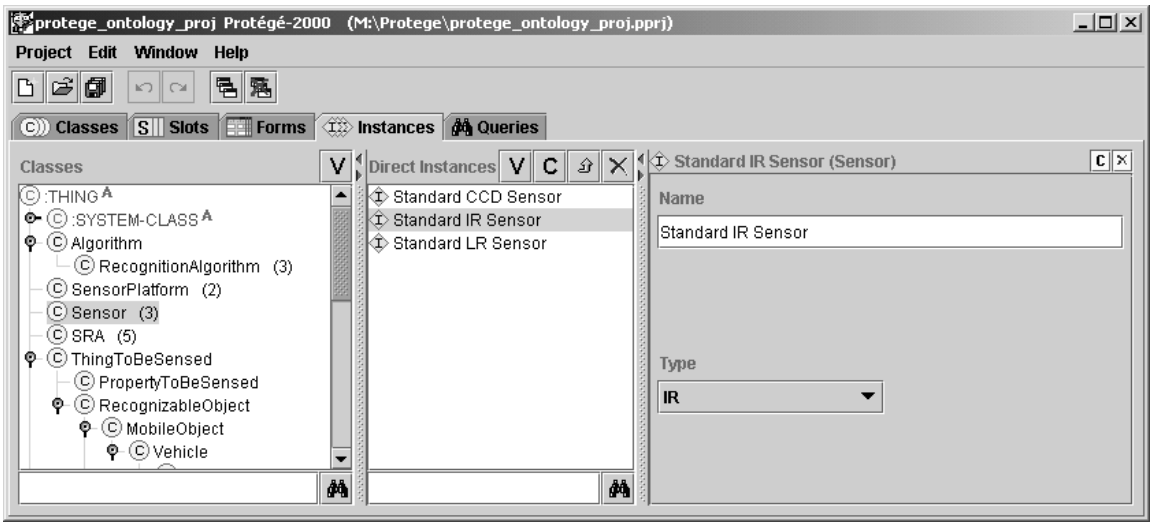

Figure 9 *Example of a sensor.*

The recognition algorithms used in the tests are:

- RA1 (ObjectPositioningAppropriateness set to 0.1)
- BuildingAlgorithm (ObjectPositioningAppropriateness set to 0.8, see Figure 10)
- TankAlgorithm (ObjectPositioningAppropriateness set to 0.0)

The ObjectPositioningAppropriateness values are chosen to differ significantly to make the influence of ObjectPositioningAppropriateness substantial.

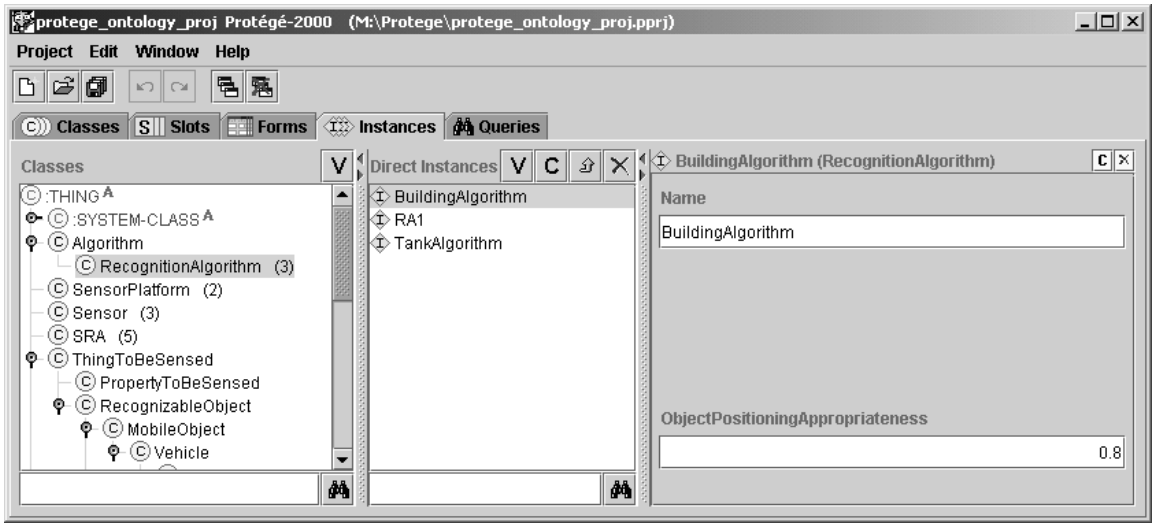

Figure 10 *Example of a recognition algorithm.*

The SRAs created for the tests are the following (the SRA names indicate which test sensors and test recognition algorithms they are using).

- SRA_CCD_Building
- SRA_CCD_RA1

- SRA_IR_Tank (see Figure 11)
- SRA_LR_Building
- SRA_LR_Tank

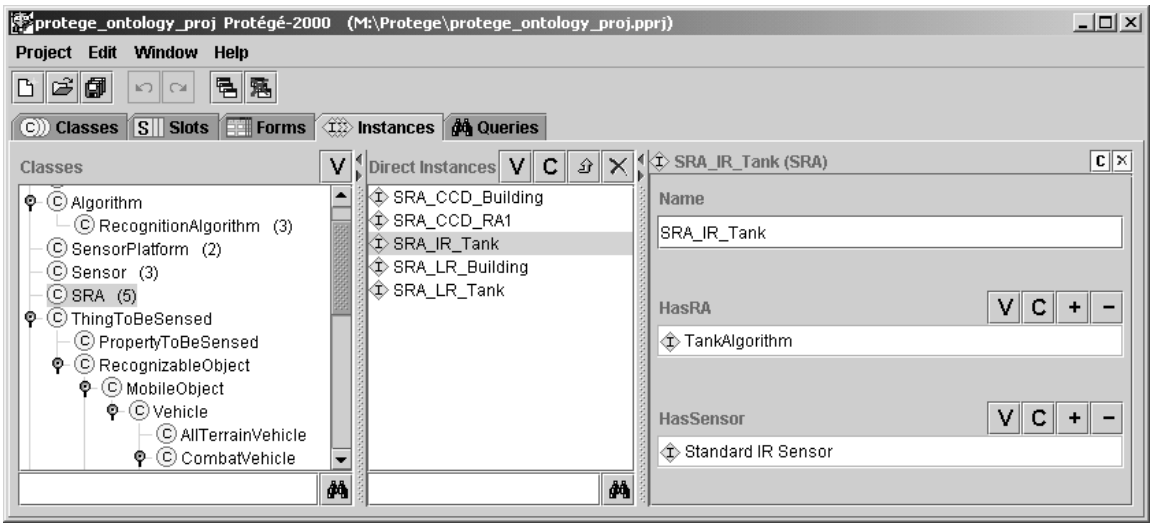

Figure 11 *Example of an SRA.*

5.3.2 Ontology test objects

The objects used in the tests are:

- Leopard (instance of Tank, see Figure 12)
- House (default instance of House, see Figure 13)
- Tent (default instance of Tent, see Figure 14)

The values of the different slots of the objects used in the tests are presented in the following screenshots; see Figure 12 - Figure 14.

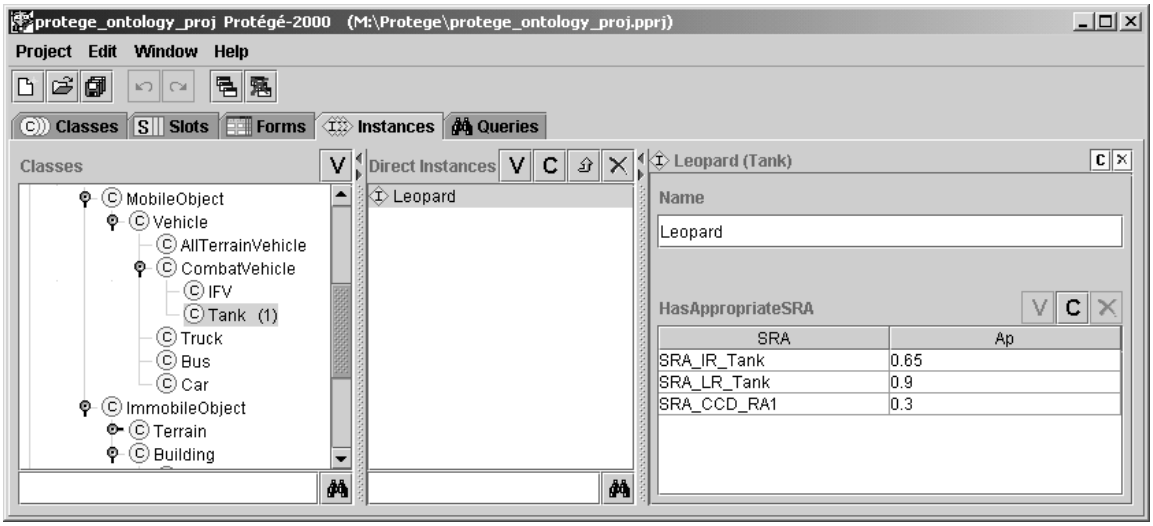

Figure 12 *Leopard tank ontology instance.*

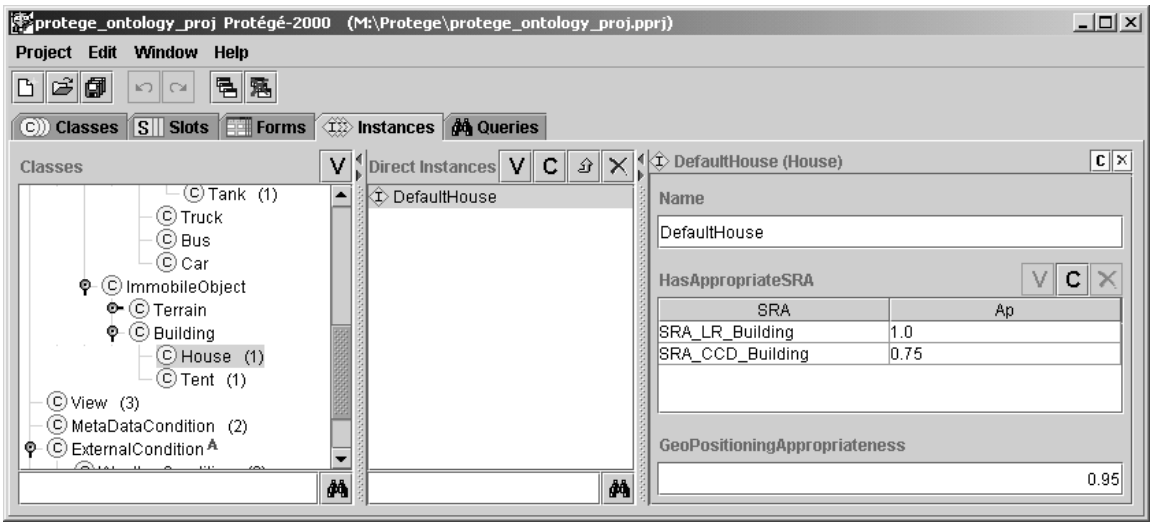

Figure 13 *Default house ontology instance.*

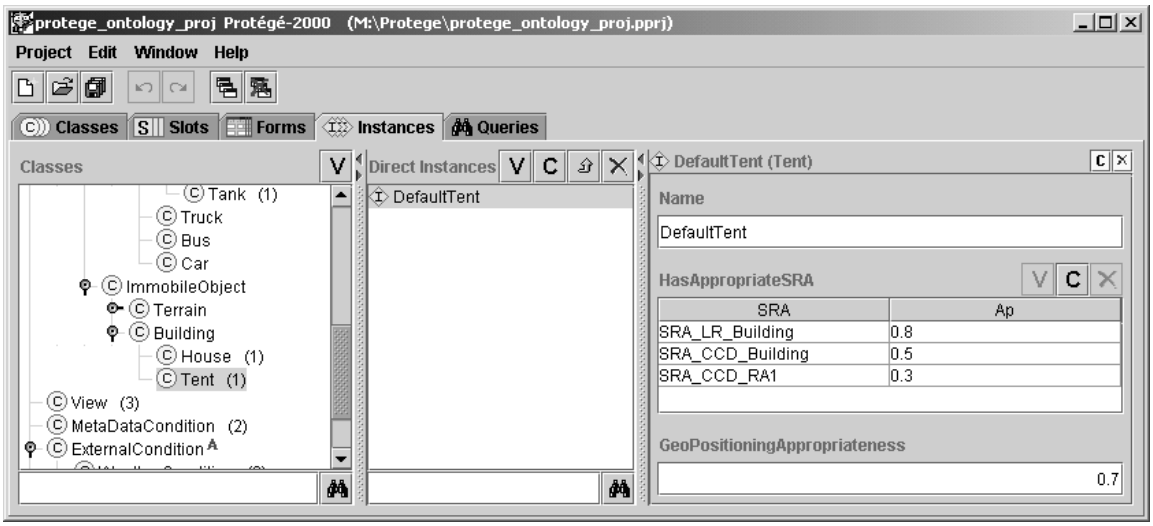

Figure 14 *Default tent ontology instance.*

Some test instances of SensorPlatform, Vegetation, ExternalCondition and LightCondition have also been created. To be able to use those, test discrete strength values for all the conditions are provided. The test conditions and their strength values can be seen in the screenshots from the KB rule manager application, see Figure 15.

5.3.3 Test rules

The rules for the sensors and recognition algorithms that are used in the tests are presented in the following screenshots from the KB rule manager application, see Figure 15 - Figure 20.

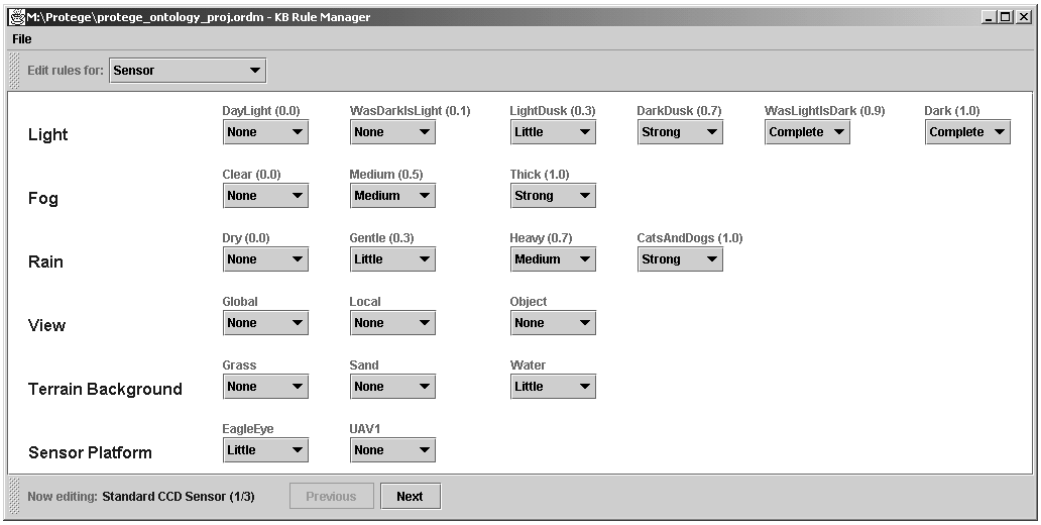

Figure 15 *Rules for the standard CCD sensor.*

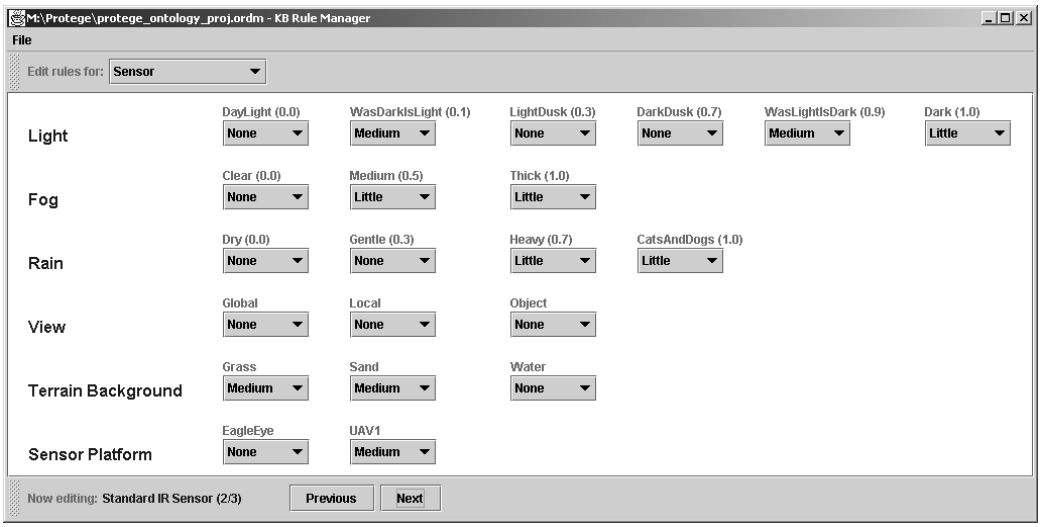

Figure 16 *Rules for the standard IR sensor.*

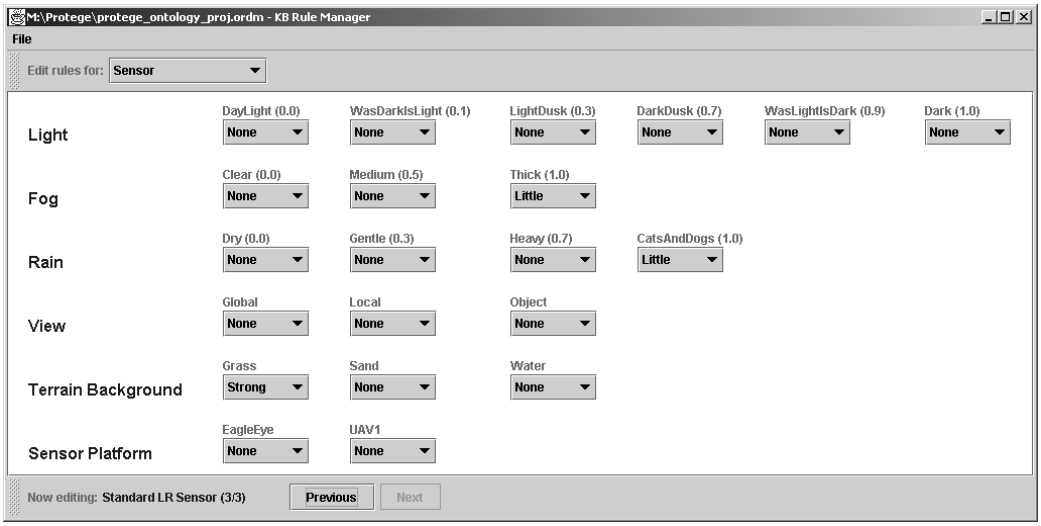

Figure 17 *Rules for the standard LR sensor.*

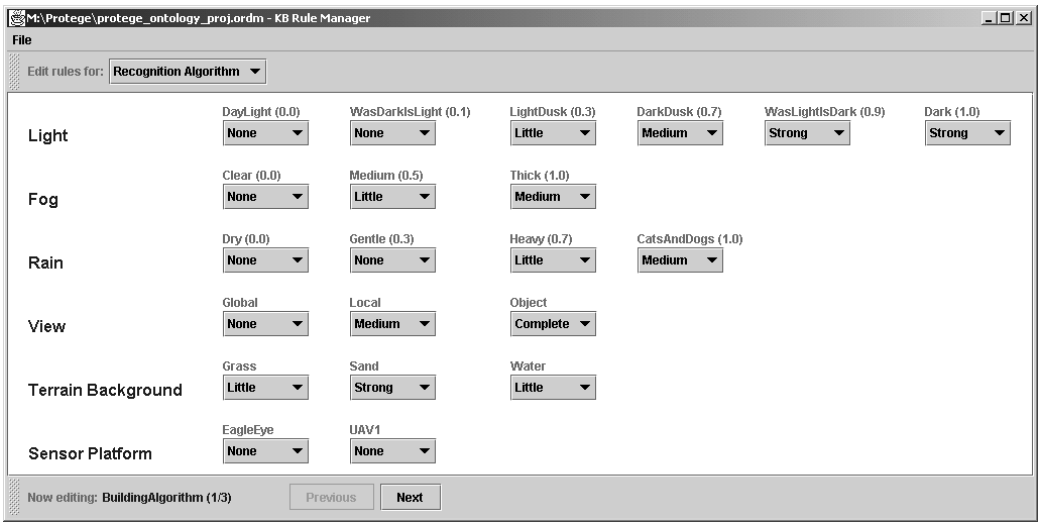

Figure 18 *Rules for the building algorithm.*

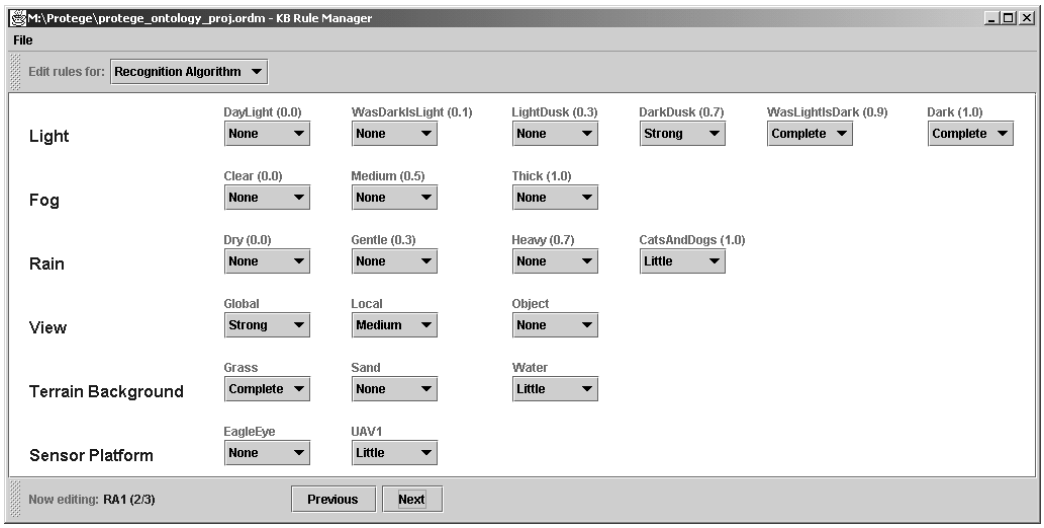

Figure 19 *Rules for RA1.*

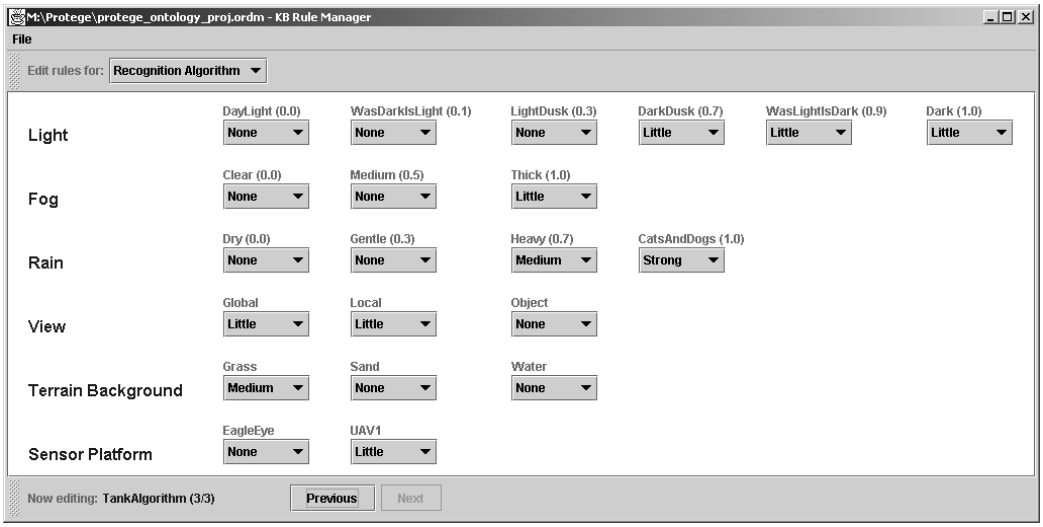

Figure 20 *Rules for the tank algorithm.*

5.4 Test scenarios

In this section the test scenarios are presented and the results are analyzed.

5.4.1 Scenario 1 – searching for a Leopard tank

In this scenario the user wants to search for a Leopard tank. The view has been set to global. Data quality for the different sensor types are similar to each other in this scenario. It is dayligh, thick fog but no rain. The terrain background is sand.

The parameters fed into the external managers and the AFFAS algorithm can be seen in the screenshot from the AFFAS test application presented in Figure 21.

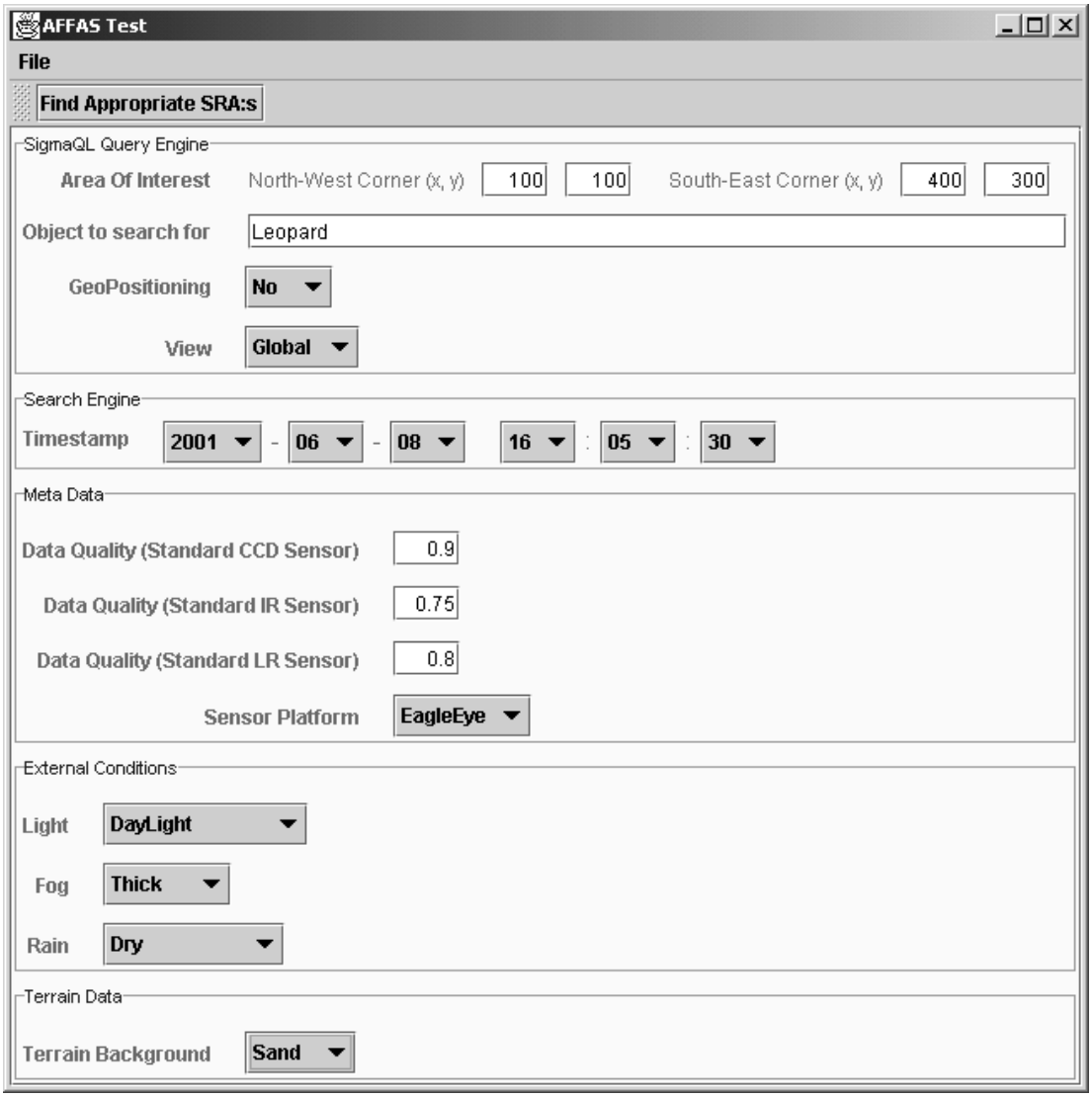

Figure 21 *Test scenario 1 setup.*

The result of pressing the "Find Appropriate SRAs" button in test scenario 1 is presented in Figure 22.

The reasons why SRA_LR_Tank is more appropriate than SRA_IR_Tank are the following:

- It has a higher value in the HasAppropriateSRA relation (see Figure 12) of the Leopard object instance.
- The IR sensor has a higher impact strength value for terrain background sand (see Figure 16 and Figure 17).

The third SRA, SRA_CCD_RA1 is far less appropriate for the following reasons:

- It has a low value in the HasAppropriateSRA relation (see Figure 12) of the Leopard object instance.
- The view is global. That has a strong impact on RA1 (see Figure 19).

Data quality is set to varying but similar values for the different sensors. As a result of the similarity of the data quality values this is not an important factor in the presented test scenarios.

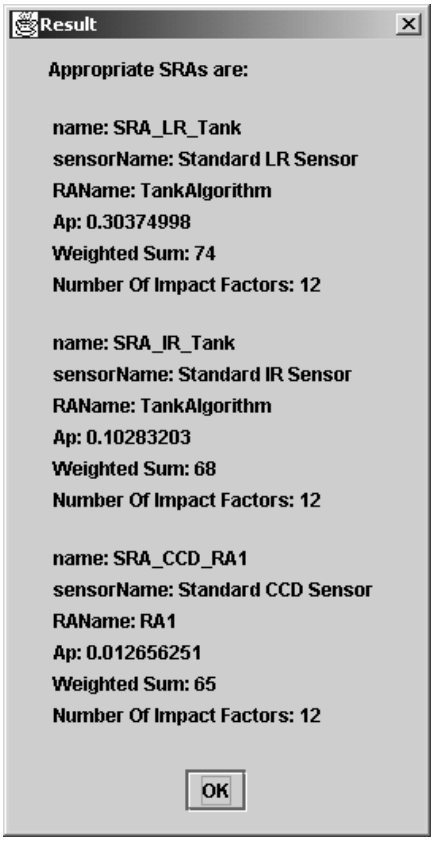

Figure 22 *Result of test scenario 1.*

5.4.2 Scenario 2 – searching for a house

In this scenario the user wants to search for a house. The view has been set to local. Data quality for the different sensor types are similar to each other in this scenario. It is dark, no fog and no rain. The terrain background is sand.

The parameters fed into the external managers and the AFFAS algorithm can be seen in the screenshot from the AFFAS test application presented in Figure 23.

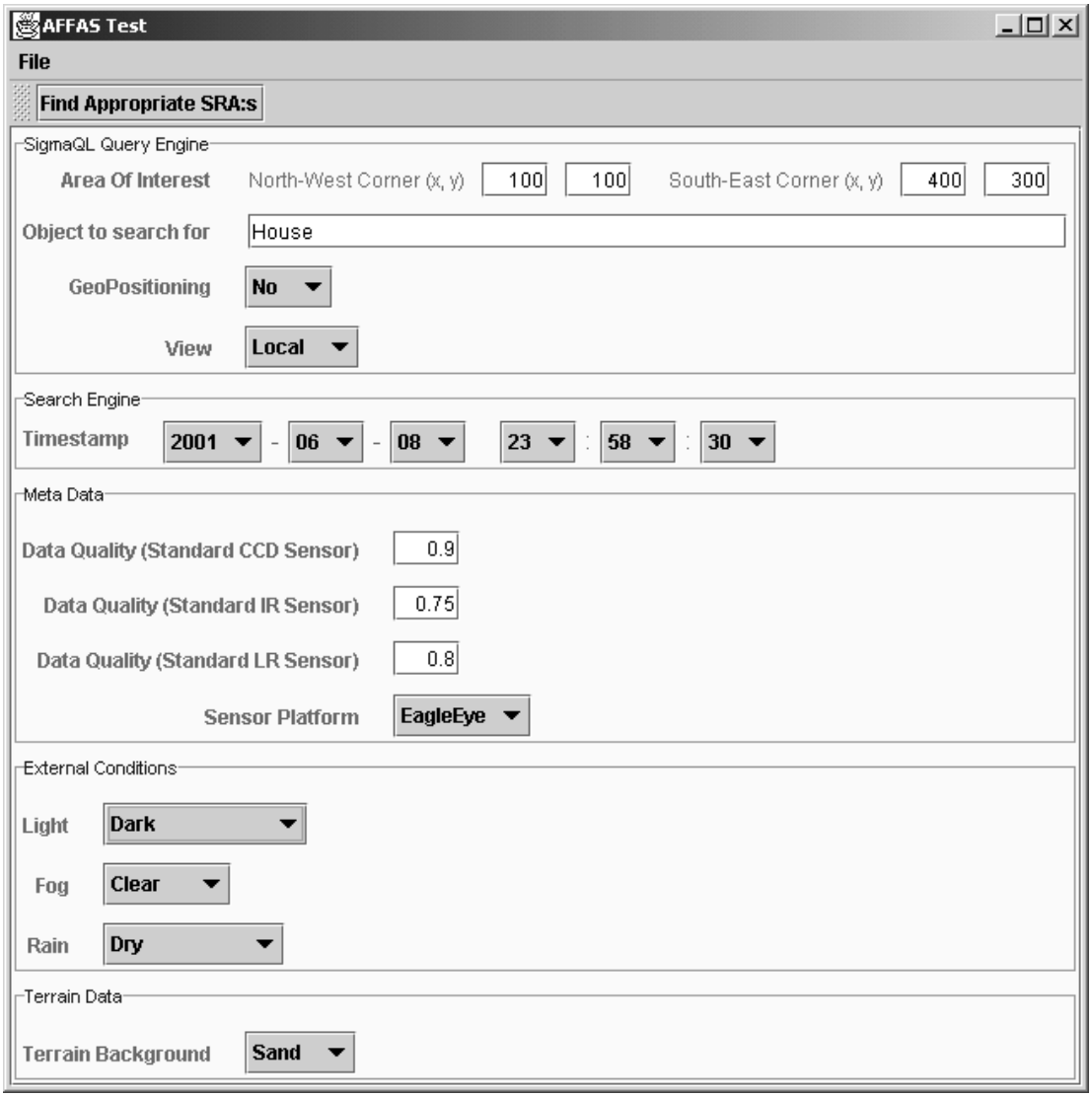

Figure 23 *Test scenario 2 setup.*

The result of pressing the "Find Appropriate SRAs" button in test scenario 2 is presented in Figure 24.

In this scenario, SRA_CCD_Building has an Ap of 0.0, because the CCD sensor has Complete impact when condition Light is Dark (see Figure 15). Impact factors with an impact strength value Complete always make the final Ap zero because the impact strength values are multiplied. In the weighted sum, impact strength value Complete only makes the sum a little smaller than it would have been for another impact strength value.

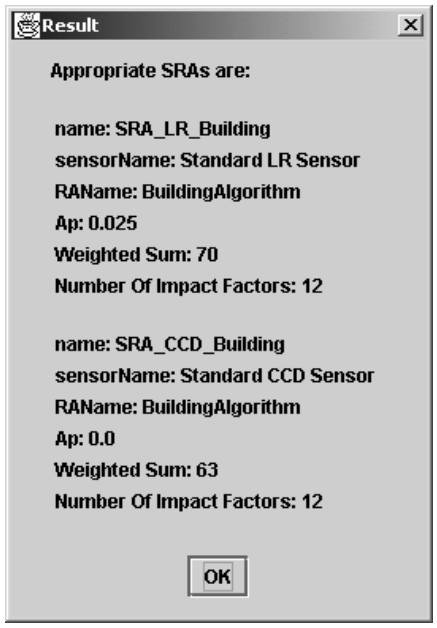

Figure 24 *Result of test scenario 2.*

5.4.3 Scenario 3 – searching for a tent to be used for geopositioning

In this scenario the user wants to search for a tent. The view has been set to global and we are interested in using the tent as a reference point for a local geographic coordinate system, so geopositioning is set to yes. Data quality for the different sensor types are similar to each other in this scenario. It is daylight, no fog and no rain. The terrain background is grass.

The parameters fed into the external managers and the AFFAS algorithm can be seen in the screenshot from the AFFAS test application presented in Figure 25.

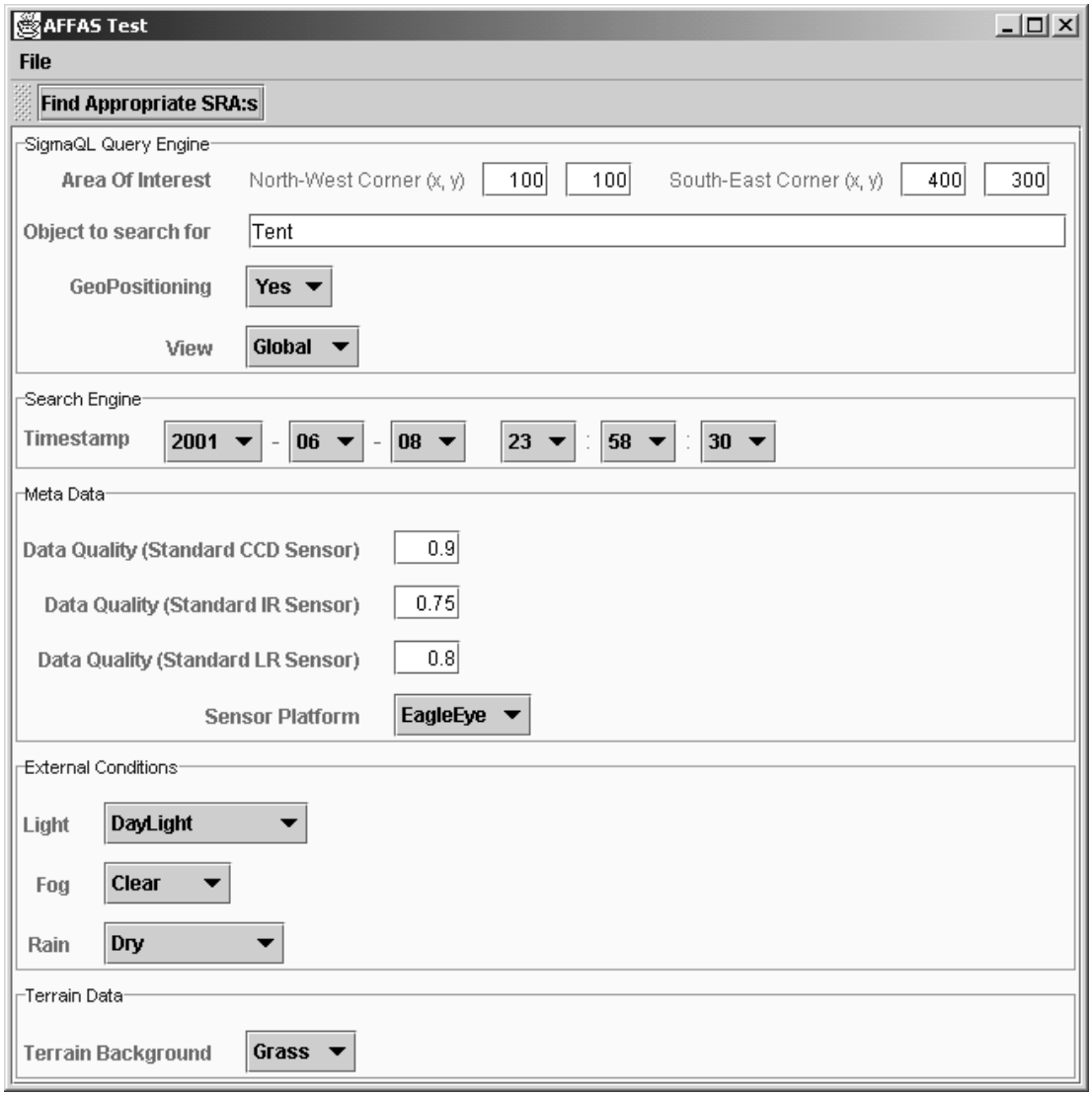

Figure 25 *Test scenario 3 setup.*

The result of pressing the “Find Appropriate SRAs” button in test scenario 3 is presented in Figure 26.

In this scenario SRA_CCD_RA1 is hardly hit for the following two reasons:

- RA1 has a very low ObjectPositioningAppropriateness value (the value is set to 0.1, see 5.3.1).
- RA1 has Complete impact for terrain background grass (see Figure 19).

It can also be noticed that in this case the two different methods for weighting together the results from step 2 and step 3 produce different results regarding SRA_CCD_Building and SRA_LR_Building. The Ap value (multiplicative method) is higher for SRA_CCD_Building, but the weighted sum is the same for SRA_CCD_Building and SRA_LR_Building. This might happen and shows that the way of weighting the impact factors together needs more consideration.

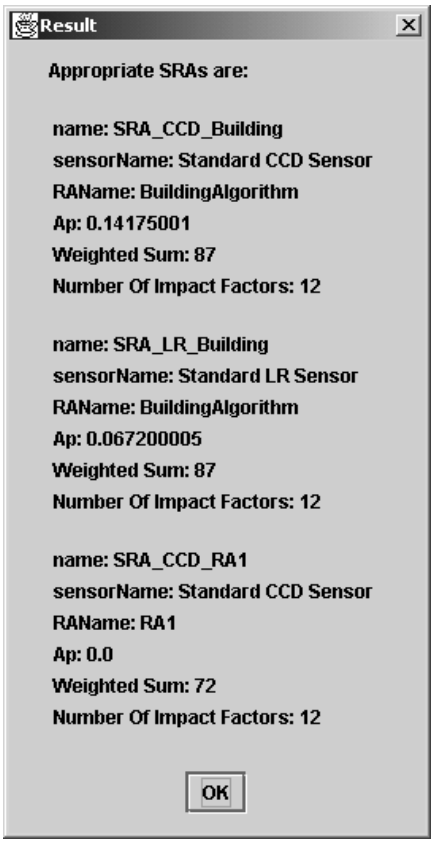

Figure 26 *Result of test scenario 3.*

5.4.4 Exact numerical results of the test scenarios

The exact numbers of the Ap values and weighted sums in the results of the test scenarios are correct according to the way the AFFAS algorithm is constructed. The multiplication factors are all set to 10, and the weights for the impact strength values are set as specified in Table 1 and Table 2 in section 4.1.2. No calculations proving the correctness of the results are presented, but they can be verified quite easily given the detailed description of the AFFAS algorithm and the data provided for the test scenarios.

5.4.5 Comparing Ap values

To compare resulting Ap values and weighted sums it is necessary to consider the number of impact factors. As long as the same number of conditions is used there will be the same number of impact factors. As the methods used in this work are all qualitative, the resulting Ap values and weighted sums cannot be looked upon as exact answers to which SRA is better than some other if their Ap values or weighted sums only differs a little. Instead, also the results must be interpreted in a qualitative fashion, i.e. SRAs with similar Ap values will obtain similar priority.

Chapter 6 - Conclusions and future research

6.1 Conclusions

Creating an ontological knowledge base using the Protégé-2000 application the way it has been done in this thesis is a flexible way of modeling knowledge. It is easy to get a visual overview of the knowledge, and it is easy to edit and change it as needed. It is also easy to access the knowledge from the outside because of the Java API provided by the Protégé-2000 environment.

The knowledge structure designed seems to model all aspects of the world required for the application specified in the objectives of this work. It is also a very extensible structure with few limitations.

The test model contains only a small set of sample knowledge. However, since the purpose of this work is to demonstrate the technical solution required, this should not be a problem as long as the sample data resembles real data in quality. Later, domain experts can easily enter correct information about sensors, algorithms, objects and rules into the system.

Handling rules in a separate application (the KB rule manager) makes the system flexible and the visualization and editing of rules simple.

The algorithm for finding appropriate sensors and recognition algorithms under certain conditions (the AFFAS algorithm) considers things that have an impact on the choice of sensors and recognition algorithms. Considering the conditions given at this time the list of impacting aspects seems to be exhaustive. However, new things may pop up in the future.

Two methods for weighting together the impact factors in the AFFAS algorithm are presented. None of them is perfect; they both have strengths and weaknesses. For example, the multiplicative method has the strength that when a condition that makes the SRA useless is present, the appropriateness value will be zero. The additive method has a weakness in this aspect. If a condition that makes the SRA useless is present, the weighted sum will loose a few points, but if the other conditions have little impact, the weighted sum can still be high. A combination of the two methods might be a good approach. More consideration is needed regarding how to weight together the impact factors.

If a ΣQL query includes several objects and/or several object properties, it is necessary to run the AFFAS algorithm for each object and each object property. One prioritized list of SRAs will be returned for each object and object property, of course. It must then be decided which of the SRAs to use.

The ontological knowledge base system provides sensor data independence from the perspective of the end-user.

6.2 Future Research

This is early research, and much can be done to enhance and further develop the system. In this section several opportunities for future research are presented.

6.2.1 Further ontology usage possibilities

Below, some ways the ontological knowledge base system can be extended are presented.

Modeling and retrieval of generic object properties

It is possible to add any number of properties (modeling different kinds of entities) to the concepts in the ontology. All objects could be modeled in a virtually unlimited detailed way by adding the necessary properties. The ontology could then be used as a knowledge system where general questions about objects can be answered, for example “How many wheels does a Leopard tank have?” or “What is the maximum speed of a truck?”. Questions like these could be answered by implementing a method in the ontology manager that gets the value(s) of a named property in a named object. Of course, the questions would have to be posed in a formal way, not in human language as in the examples above. It will be possible to pose this kind of questions directly to the system, but this functionality is not yet implemented.

Parent concept

Implementing a method in the ontology manager that returns the parent concept of some named concept can easily be accomplished and will soon be done.

Modeling object features estimated by recognition algorithms

Different recognition algorithms recognize different objects and features of objects. Which features a certain algorithm estimates (and possibly how well it estimates the different features) could very well be modeled in the ontology. This could help the sensor data fusion process. This would also make it possible to connect the features a recognition algorithm estimates to the recognition algorithm via a relation “EstimatesFeature” from the concept modeling a recognition algorithm to the concept modeling a feature. Doing this would help solve the feature completeness problem, which is the problem of having as many target object features as possible estimated using a set of recognition algorithms.

Modeling property value uncertainty

It is possible to model whether the value of a specific property is uncertain or not. When measuring or estimating the value of a property (e.g. width, speed, brightness, temperature, etc) there is always an uncertainty in the measurement/estimation. Such an uncertainty value will correspond to the confidence we have in that feature. It should be possible to enter a value of the uncertainty of the property value, not only

specifying whether there is an uncertainty or not. Uncertainty should be subject to further studies.

6.2.2 Machine learning to tune the rules

When a complete test system has been developed and data from tests from the real world become available, it will be possible to use that information to test whether the sensors and algorithms recommended by the system were actually the most appropriate ones. Testing as many combinations of recognition algorithms as possible on as many sensor data types as possible and then evaluating which combinations actually produced the best results could be a challenging way of doing this. Those test results could then be used to tune the rules used for choosing appropriate sensors and recognition algorithms. If a system for machine learning is built, the rule tuning could be done on a semi-automatic basis. As soon as more data becomes available from real test sessions and typical queries are posed on the captured sensor data, the learning system could tune the rules as long as the correct results to the queries are also posted to the system.

6.2.3 Discrete strength values and impact strengths

In this thesis it was decided that strength values and impact strengths should have discrete ranges of permitted values. There are no special limitations that make this necessary, but as the number of sensors and algorithms are and will stay fairly limited there is no reason to use continuous precision in the strength values or impact strengths. Having discrete ranges also makes it a lot easier to handle the rules. It is easy to extend the discrete ranges with more permitted values to get a “more continuous-like” value range if that will be needed.

6.2.4 Area of interest may not have to be rectangular

To simplify things the area of interest (AOI) has been defined as a rectangular area. In the future it might be preferable or even necessary to be able to work with non-rectangular areas of interest.

6.2.5 SRA Cooperation Matrix

During this work there was a discussion about the possibility of considering the fact that SRAs cooperate better or worse with each other. Combining two specific SRAs may provide better feature completeness, for example. To take this into consideration it would be possible to add a fifth step to the AFFAS algorithm. In the new step a matrix would be constructed. Depending on the appropriateness values from step 4 and information about how well the SRAs cooperate, it would be possible to suggest using a combination of SRAs. This matrix could be two-dimensional for a start, but there is no reason why it cannot be generalized to as many dimensions as there are SRAs in the result of step 4.

In the ISM project it has been decided that an SRA cooperation matrix will not be included because the central sensor data fusion module will handle the issue of deciding which SRAs and what information from them will be used. However, it might be interesting as an area of research.

6.2.6 Hidden parts of the terrain and hidden objects

In this section a short discussion about hidden parts of the terrain and hidden objects is presented.

Hidden parts of the terrain

Sometimes, parts of the terrain may be hidden from a sensor by a hill or something else. Which part is hidden at what time depends on the position of the sensor platform and the actual terrain.

It will be necessary to develop a system that can take this into consideration and create a “masked” version of the terrain data where hidden parts are marked or removed.

Hidden objects

Some of the recognition algorithms can perform well on partly hidden objects, especially if the algorithms can get information about which parts of the object are hidden.

No support to partly hidden objects has been given in this work.

6.2.7 Future data prediction in supporting systems

A sketch describing how to make AFFAS transparent to queries about past, present and future time is presented in Figure 27. The external condition manager gets information about the past and present from a weather condition database whereas information about the future is taken from a weather condition prediction system. The external conditions manager handles this transparently to the AFFAS algorithm. In the same way, the meta data manager transparently handles queries about the past by getting information from a database. When handling queries about the future, a sensor platform route planner could be used to plan the sensor platform routes so that the most appropriate sensors can capture data of best possible quality from the most interesting areas. This way the meta data quality is maximized. The route planner must also consider other things like threats, physical limitations of the platforms etc, but that is out of the scope of this work.

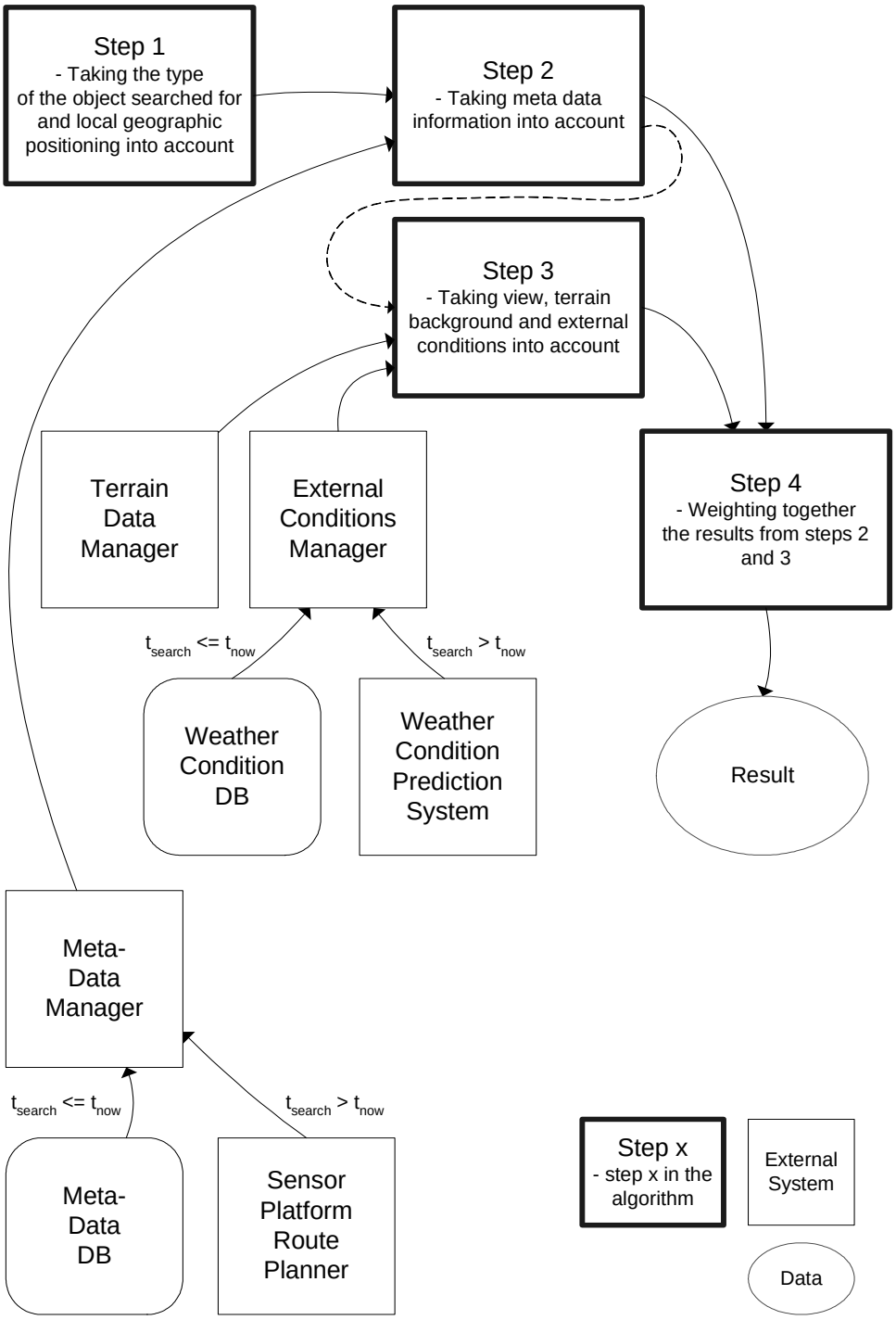

Figure 27 *Proposal about how to incorporate future data prediction.*

6.3 Implementation aspects

In this section some aspects of the implementation are discussed.

6.3.1 Modeling knowledge base rules in the ontology

In this work, knowledge modeling is separated from the algorithms performing the reasoning. This section presents a discussion about the possibility to integrate (some of) the reasoning into the ontological knowledge base.

Jess – the Java Expert System Shell

Jess is a rule engine and scripting environment written entirely in Sun's Java language by Ernest Friedman-Hill at Sandia National Laboratories in Livermore, CA. ***Jess*** was originally inspired by the CLIPS expert system shell, but has grown into a complete, distinct Java-influenced environment of its own. Using ***Jess***, you can build Java applets and applications that have the capacity to "reason" using knowledge you supply in the form of declarative rules. ***Jess*** is surprisingly fast, and for some problems is faster than CLIPS itself (using a good JIT compiler, of course.) [18]

JessTab in Protégé-2000

JessTab is a plug-in for Protégé-2000 that allows you to use Jess and Protégé-2000 together. It gives you the best of both worlds. JessTab provides a Jess console window where you can interact with Jess while running Protégé. Furthermore, JessTab extends Jess with additional functions that allow you to map Protégé knowledge bases to Jess facts. Also, there are functions for manipulating Protégé knowledge bases from Jess. [16]

Modeling rules in the ontology

Using the JessTab in Protégé-2000 and extending the ontology with a new part, a “Rules” part, it would be possible to model the rules in the ontology, instead of having the rules outside the ontology knowledge base as they are at present. Now they are defined using the KB rule manager application, separate from the knowledge base, saved in a separate file.

There are advantages and disadvantages to the approach of modeling the rules in the ontology. The main advantage would be that using Jess it would be possible to define rules using the Protégé-2000 JessTab, which means everything (knowledge and rules) would be kept in the same place, and the rules and knowledge would be automatically synchronized. This could be useful for scientific purposes when one wants to experiment with different rules. In this work, the rules management has been put outside the Protégé system to separate the knowledge modeling from the “intelligence”.

6.3.2 Optimizations

All implementation done as part of this thesis has been done without thoughts of optimizations. Much can be done to make the code run faster. Since the code is well

documented using the Javadoc system, it should be easy to understand it, if the existing code needs to be optimized or rewritten.

6.3.3 AFFAS weights and multiplication factors

In AFFAS, the assignment of numbers to impact strength values, weights and multiplication factors is defined in the Java code that implements the AFFAS algorithm. This is not an optimal solution as the range of impact strength values may be subject to change. It would be an advantage to model the impact strength values and the assignments of numbers to those in the ontology.

On the other hand, the weighting methods themselves are not perfect and might be too simplistic to work sufficiently well in a real system. Weighting together impact factors that depend on each other is a research area in itself, so better methods may very well be needed.

6.4 Acknowledgements

I would like to thank the following people for helping and supporting me during the production of this thesis: Erland Jungert, Morgan Ulvklo, Shi-Kuo Chang, Thomas Kaijser, Mikael Brännström and Henrik Eriksson.

References

[1] Valente, A., Russ, T., MacGregor, R. and Swartout, W. (1999) “Building and Re(Using) an Ontology of Air Campaign Planning”, IEEE Intelligent Systems, vol. 15, no. 1, Jan-Feb 1999, pp 27-36.

[2] Royer, V., Challine, J-F. (2000) “A Platform for Interoperable Fusion Models”, Procs of the 3rd International Conference on Information Fusion (FUSION 2000), France, July 2000, pp ThB1-3 – ThB1-10.

[3] Kotkas, V., Penjam, J., Tyugu, E. (2000) ”Ontology-based design of surveillance systems with NUT”, Procs of the 3rd International Conference on Information Fusion (FUSION 2000), France, July 2000, pp WeB4-3 – WeB4-9.

[4] Jungert, E. and Letalick, D. (in Swedish, 2000), “Informationssystem för måligenkänning – analys av behov och problemställningar”, FOA-R--00-01607-408--SE, FOA, Linköping, Sweden.

[5] Elmqvist, E., Jungert, E., Lantz, F., Persson, Å. and Söderman, U. (2001), ”Terrain Modeling and Analysis using Laser Scanner Data”, Procs of the ISPRS Workshop in Land Surface Mapping and Characterization using Laser Altimery, U.S.A., October 2001, pp 219-226.

[6] Chang, S.-K., Costagliola, G. and Jungert, E. (2000), “Spatial/Temporal Query Processing for Information Fusion Applications”, Procs of Advances in Visual Information Systems (VISUAL 2000), France, November 2000, pp 127-139.

[7] Chang, S.-K. and Jungert, E. (1998), “A Spatial/temporal query language for multiple data sources in a heterogenous information system environment”, The International Journal of Cooperative Information Systems (IJCIS), vol. 7, Nos 2 & 3, 1998, pp 167-186.

[8] Chang, S.-K., Costagliola, G. and Jungert, E. (1999), “Querying Multimedia Data Sources and Databases”, Procs of the 3rd International Conference on Visual Information Systems (VISUAL ’99), The Netherlands, June 1999, pp 19-28.

[9] Chang, S.-K., Costagliola, G. and Jungert, E. (2002), "Multi-Sensor Information Fusion by Query Refinement", Procs of the 5th International Conference on Visual Information Systems (VISUAL 2002), Taiwan, March 2002.

[10] Chang, S.-K., Jungert, E. (1996), "Symbolic Projection for Image Information Retrieval and Spatial Reasoning", Academic Press, London. ISBN 0-12-168030-4

[11] Gruber, T. R. (1993), "Toward Principles for the Design of Ontologies Used for Knowledge Sharing", KSL 93-04, Knowledge Systems Laboratory, Stanford University, 1993.

[12] Burch, R. W. (2000), "Semeiotic Data Fusion", Procs of the 3rd International Conference on Information Fusion (FUSION 2000), France, July 2000, pp WeC4-11 – WeC4-16.

[13] Noy, N. F. and McGuinness, D. L. (2000), "Ontology Development 101: A Guide to Creating Your First Ontology", KSL 01-05, Stanford University, 2001.

[14] Ontolingua, The Ontolingua Project,
http://www.ksl.stanford.edu/software/ontolingua/ (Acc. 2002-01-14)

[15] Chimaera, The Chimaera Ontology Environment,
http://www.ksl.stanford.edu/software/chimaera/ (Acc. 2002-01-14)

[16] Protégé, The Protégé Project,
http://protege.stanford.edu/index.shtml (Acc. 2002-01-14)

[17] Aussenac-Gilles, N., Biébow, B. and Szulman, S. (2000), "Revisiting Ontology Design: A Method Based on Corpus Analysis", Procs of Knowledge Acquisition, Modeling and Management, 12th International Conference (EKAW 2000), France, October 2000, pp 172-188.

[18] Jess, The Java Expert System Shell
http://herzberg.ca.sandia.gov/jess/main.html (Acc. 2002-01-14)

Appendix A - Using the ontology system

This appendix describes how to work with the ontological knowledge base system developed as part of this study.

A.1 System requirements

All software used by implementation of this study can be downloaded from the Internet and used for free.

A.1.1 Java 2 Platform

All the software can be run on any operating system with the *Java 2 Platform, Standard Edition* (J2SE) version 1.3.1 (or later) installed. J2SE is available for download from http://java.sun.com/java2/.

A.1.2 Protégé-2000

Protégé-2000 is the ontology-editing environment used for editing the knowledge in the ontological knowledge base. It is available for download from http://protege.stanford.edu/index.shtml.

A.1.3 XML parser

The rules that the system uses for finding appropriate SRAs are stored in a file format that is XML-based. The implementation of the study therefore needs an XML parser to be able to work with the rule files. The *Java API For XML Processing* (JAXP) has been used in this work and it is available for download from http://java.sun.com/xml/jaxp/index.html.

The Document Type Definition (DTD) file that describes the ontological rule data model (.ordm) format must be available in the same directory as the .ordm-file when an .ordm-file is to be loaded. The DTD-file should be called ordm.dtd and it is included in the distribution of the software implementing the study.

A.2 Extending the ontology

To extend the ontology, use the Protégé-2000 application to open the project file that keeps the knowledge base. The file is called protege_ontology_proj.pprj. The classes and instances are stored in the protege_ontology_proj.pins and protege_ontology_proj.pont files. Those are automatically used by the Protégé-20000 application when the .pprj-file is opened, but one must make sure those files are available in the same directory as the .pprj file.

Using the Protégé-2000 application, it is easy to add new classes, slots and instances to the ontological knowledge base. Documentation on how to do this is available with the distribution of the Protégé-2000 application.

A.3 Working with the rules

If new sensors, recognition algorithms, conditions or discrete strength values have been added it is now time to define rules using the new ontology instances. If a new condition has been added, new rules for each sensor and recognition algorithm must be added dealing with that condition. If a new discrete strength value has been added, a new rule for that strength value for the condition having the new discrete strength value in its range of permitted values must be added to each sensor and recognition algorithm. In other words, there must always be a complete set of rules for all sensors and recognition algorithms for each discrete strength value of each condition. If this is not the case, the AFFAS algorithm will not function correctly.

As soon as any of the changes mentioned above are made to the ontological knowledge base and the changes have been saved to the Protégé project file it is time to open the KB rule manager application and load the Protégé project file into that application.

When the rules have been defined/altered, they must be saved with the same filename as the Protégé project file but with the .ordm suffix instead of .pprj. This enables the ontology manager to find the ontological rule data model (the file with the rules) when it is initialized.

A.4 Using and extending the ontology manager

When the knowledge has been modeled (using Protégé-2000) and the rules are defined (using the KB rule manager) the task is to find appropriate sensors and recognition algorithms under certain conditions. However, there are many other possible usage areas for the ontology system. The design of the entire system has been done with flexibility in mind to allow for other applications.

The ontology manager is the glue between the Java application and the Protégé model. The ontology manager also handles the rule data model.

The functionality that is currently implemented in the ontology manager is an implementation of the AFFAS algorithm and a method for getting all objects that can be used for geopositioning. See the Javadoc documentation for details.

A.4.1 Using the ontology manager

Using the ontology manager is done by calling the getOntologyManager method in the se.foi.ontology.OntologyManager class and then calling the wanted method on the OntologyManager object that is returned. The getOntologyManager method takes the filename to the Protégé project file as its only argument. Make sure the ontological rule data model created using the KB rule manager application is

available in the same directory as the Protégé project file with the same filename but a suffix of .ordm instead of .pprj.

A.4.2 Adding functionality

If additional functionality is needed, a method implementing that functionality in the ontology manager is created. Getting the needed information from the Protégé project is done in the OntologyUtil class. This class works as an abstraction layer between the ontology manager and the ontology implementation (in this case Protégé). Therefore, implementing new functionality is a matter of adding the necessary methods to the OntologyUtil class for retrieving the wanted data using the Protégé API and then using the methods created in the OntologyUtil class from the OntologyManager method implementing the functionality. See the implementation of the AFFAS algorithm for an example of how these things work.

A.5 Documentation in Javadoc format

Complete documentation of the implementation is provided in the Javadoc format. This includes the ontology manager, the KB rule manager, the test implementations of the external managers as well as the AFFAS test application. The Javadoc documentation is available with the distribution of the software implementing the results of this thesis.

Appendix B - Detailed description of AFFAS

The following describes the steps of the AFFAS algorithm in detail.

B.1 AFFAS step 1

- Get the instance (Thing to be sensed) from the knowledge base (KB) corresponding to the object we want to search for. Every concept in the `ThingToBeSensed` part of the ontology must have an instance Default<concept-name> where <concept-name> is the name of the concept. If no instance is found in this step, try prepending Default to the string to see if it is a concept instead of an instance being searched for. This makes it possible to for example search for "Vehicle" without having to specify the actual instance name `DefaultVehicle`. This will work in the same way for all concepts. Assign the thing to be sensed instance to the variable ttbs:

  ttbs = KB.getInstance(concept-name)

- Get all the SRAs connected to this instance (ttbs) via the `HasAppropriateSRA` relation. Also make sure to save the "Ap" value from the relation for each SRA found and initialize the weighted sum (Wsum) of the SRA:

  $S_0 = \{\}$

  For each relation $HasAppropriateSRA_i$ in ttbs:

  S_0.addElement(ttbs.getAppropriateSRA(i))

  For each SRA_i in S_0:

  SRA_i.Wsum = SRA_i.Ap * INITIAL_AP_MULTFACTOR

- If we are *not* interested in geopositioning jump to EndOfStep1.
  - If the object we are searching for (ttbs) is *not* an `ImmobileObject`, then it is not appropriate for geopositioning, so the "Ap" for all SRAs is set to zero:

    For each SRA_i in S_0:

    SRA_i.Ap = 0

    End of For each SRA_i in S_0

    Jump to EndOfStep1

  - If the object we are searching for *is* an `ImmobileObject`:

    For each SRA_i in S_0:

    GeoPosAp = ttbs.getGeoPositioningAppropriateness()

ObjPosAp = SRA_i.getRA().getObjectPositioningAppropriateness()

SRA_i.Ap = SRA_i.Ap * GeoPosAp

SRA_i.Ap = SRA_i.Ap * ObjPosAp

SRA_i.Wsum = SRA_i.Wsum + GeoPosAp * GEOPOS_MULTFACT

SRA_i.Wsum = SRA_i.Wsum + ObjPosAp * OBJPOS_MULTFACT

End of For each SRA_i in S_0

EndOfStep1:

The result from this step is called S_1. It is a list of SRAs where each SRA has an "Ap" and a weighted sum connected to it.

B.2 AFFAS step 2

- For each SRA_i in S_1:
  - Find the connected sensor S by following the `HasSensor` relation:

    S = SRA_i.getSensor()
  - Ask the meta data manager which sensor platform carried the sensor that has captured the best data using a sensor of type S at time t and what the quality of that data is (this also works for queries about the future). Store the name of the sensor platform in sp and the data quality in dq.

    sp = MetaDataManager.getSensorPlatform(S, t)

    dq = MetaDataManager.getDataQuality(S, t)
  - Add impact for data quality.

    SRA_i.Ap = SRA_i.Ap * dq

    SRA_i.Wsum = SRA_i.Wsum + dq * DATAQUALITY_MULTFACT
  - Add sensor platform impact on the sensor and on the RA respectively. Do this by getting the interesting impact strength values from the ontology rule data model (ORDM).

    SensorImp = GetORDMImpact(SRA_i.getSensor(),"Sensor Platform", sp)

    RAImp = GetORDMImpact(SRA_i.getRA(),"Sensor Platform", sp)

    SRA_i.Ap = SRA_i.Ap * GetMultMethodAssignedNum(SensorImp)

    SRA_i.Ap = SRA_i.Ap * GetMultMethodAssignedNum(RAImp)

    SRA_i.Wsum = SRA_i.Wsum + GetSumMethodAssignedNum(SensorImp)

    SRA_i.Wsum = SRA_i.Wsum + GetSumMethodAssignedNum(RAImp)
- End of For each SRA_i in S_1
- Remove all SRAs with an "Ap" value of zero from S_1

The result from this step is called S_2. It is a list of SRAs where each SRA has an "Ap" value and a weighted sum connected to it. It is the same list as S_1, but the "Ap" values and weighted sums have been altered, and SRAs with "Ap" zero have been removed.

B.3 AFFAS step 3

- A is the set of SRAs in the result from step 2 (S_2 that is).
- B is the set of all the Sensors and RAs that are connected to at least one of the SRAs in A via the `HasSensor` or `HasRA` relations.
- C is the set of all the conditions we want to take into consideration. This includes all instances of `WeatherCondition` (e.g. `Rain`, `Fog`, etc.) and the instance of `LightCondition` (which is called `Light`). It also includes `View` and Terrain Background (`Vegetation`).
- In the Ontology Rule Data Model file (.ordm) created using the KB rule manager application, there is information about how each of the conditions in C affects each of the RAs and sensors in B for each of the possible strength values (those are discrete, so it is a finite set of values) of the conditions in C.
- For each instance b in B:
  - Reset the result vector to an empty vector

    Res = []
  - For each instance c in C:
    - Get the current strength of c by asking the appropriate external system, e.g. external conditions manager or terrain data manager, (in the case of View, check instead the input to the algorithm) and assign this value to CurStrength.

      CurStrength = GetCurrentStrength(c)
    - Get the impact strength on b by condition c with strength CurStrength from the ontology rule data model and assign this to CurImpact.

      CurImpact = GetORDMImpact(b, c, CurStrength)
    - Add CurImpact to the result vector

      Res.addElement(CurImpact)
  - End of For each instance c in C
  - Store the information that "the total impact on b is Res".

    TotalImpact(b) = Res
- End of For each instance b in B

The result from this step is the data structure TotalImpact, which for each sensor and RA used in any of the SRAs contains impact strength values describing the impact of the current View, Terrain Background and External conditions.

B.4 AFFAS step 4

- For each SRA_i in S_2:
  - Get the sensor impact by getting the result vector for the sensor of the SRA that is stored in the TotalImpact data structure from step 3.

    $SensorRes_i$ = TotalImpact(SRA_i.getSensor())

    For each $ImpFact_j$ in $SensorRes_i$:

      SRA_i.Ap = SRA_i.Ap * GetMultMethodAssignedNum($ImpFact_j$)

      SRA_i.Wsum = SRA_i.Wsum + GetSumMethodAssignedNum($ImpFact_j$)

    End of For each $ImpFact_j$ in $SensorRes_i$
  - Get the RA impact by getting the result vector for the RA of the SRA that is stored in the TotalImpact data structure from step 3.

    $RARes_i$ = TotalImpact(SRA_i.getRA())

    For each $ImpFact_j$ in $RARes_i$:

      SRA_i.Ap = SRA_i.Ap * GetMultMethodAssignedNum($ImpFact_j$)

      SRA_i.Wsum = SRA_i.Wsum + GetSumMethodAssignedNum($ImpFact_j$)

    End of For each $ImpFact_j$ in $RARes_i$
- End of For each SRA_i in S_2
- Sort the SRAs in descending order according to the “Ap” value of each SRA

The result from this step is the final result. It is a prioritized list of SRAs sorted according to how appropriate the SRAs are under the current circumstances.

Appendix C - Rule Data Model file format

The KB rule manager automatically tries to load an .ordm file with the same name as the Protégé project file being loaded at program startup. Therefore, saving the rule data model with the same name as the Protégé project file in the same directory, but of course with a different extension (.ordm instead of .pprj), will enable the KB rule manager to automatically find and load the rule data model the next time you want to edit the rules for that Protégé project file (.pprj).

The file format is XML-based, which means that the data is saved in a clear text format that can be read by a human being. It also means that it is easy for other applications to import the data. For example, the AFFAS implementation does this.

Structure of the .ordm file format

The ontological rule data model file has the following structure:

```
<ordm>
 <instance
 name="Name of sensor or recognition algorithm">
  <condition
  name="Name of condition">
   <strength
    name="Discrete strength value"
    data="Impact strength value"/>
   ...
  </condition>
  ...
 </instance>
 ...
</ordm>
```

An XML file format uses a document type definition (DTD) file to specifiy the format. The document type definition for the ordm format is kept in a file called ordm.dtd. The ordm.dtd file must be located in the same directory as the .ordm file to ensure it will be found by the XML parser used in the implementation of the present work.

Appendix D - Ontology manager API

To hide the implementation details of the ontological knowledge base, an ontology manager has been created. The ontology manager is a Java class that can be used to perform operations on the knowledge base. For example, the AFFAS algorithm implementation is reachable via the ontology manager. Thus, when a surrounding system wants to access information in the knowledge base, it will do this by calling the appropriate method in the ontology manager.

D.1 Ontology manager

se.foi.ontology.OntologyManager is the Java class that implements the ontology manager. It is used to access the information in the ontology and get questions answered that need information from the ontology. It uses the singleton design pattern to make sure that there can be only one ontology manager.

When the ontology manager initializes itself, it loads the specified Protégé project file (see the getOntologyManager method below). When the Protégé project file is loaded, the ontology manager tries to load an ontological rule data model with rules for the knowledge base in the Protégé project file. The name of the ontological rule data model file must be the same as the Protégé project file but with a different suffix (.ordm instead of .pprj) for the ontology manager to find it.

The most important methods in the OntologyManager class are:

public static OntologyManager getOntologyManager(java.lang.String fileName)

Returns the OntologyManager if it already exists, otherwise creates and then returns it

Parameters:

fileName - The filename of the Protege Project (.pprj)

Returns:

The one and only OntologyManager object

public java.util.Vector getAppropriateSRAs(java.lang.String thingToBeSensed,

boolean geoPositioning,

java.lang.String view,

java.awt.Rectangle AOI,

java.util.Date timestamp)

This is the method that is called by the system using the Ontology Manager when that other system wants to find out which SRAs are the most appropriate under certain circumstances. It is the implementation of the AFFAS algorithm.

Parameters:

thingToBeSensed - The name of the object we are searching for

geoPositioning - True if we are interested in geopositioning, otherwise false

view - The View we are currently working in

AOI - Area Of Interest. Perhaps this should be an Area object rather than a Rectangle.

timestamp - The timestamp of the data we are currently searching (and therefore the timestamp we are interested in finding appropriate SRAs for)

Returns:

Vector of SRAWithAp objects sorted so that SRAs with higher Ap are further to the front of the Vector

public java.util.Vector getObjectsAppropriateForGeoPositioning()

Gets all objects appropriate for geopositioning. Uses the StrengthValue class to store the result in a Vector. The getName() method gives the name of the object; the getStrength() method gives the Ap. The result is a Vector with StrengthValues sorted on descending Ap, so the object with the highest Ap is placed first in the resulting Vector.

Returns:

Vector with StrengthValues sorted on descending Ap

D.2 External managers

The Java package se.foi.ontology.externalmanager includes interfaces and test implementations of the external managers. The external managers are the systems that provide the AFFAS algorithm with information about external conditions, meta data conditions and terrain background.

D.2.1 External conditions manager

The se.foi.ontology.externalmanager.ExternalConditionsManager interface must be implemented by a class if it wants to act as an external conditions manager.

The only method in the ExternalConditionsManager interface is:

public java.lang.String getConditionStrength(java.lang.String condName,

java.awt.Rectangle AOI,

java.util.Date timestamp)

Gets the strength of condition condName in the AOI at the time described by timestamp.

Parameters:

condName - Condition to get strength of

AOI - Area Of Interest

timestamp - Timestamp Of Interest

Returns:

Name of the StrengthValue (e.g. "None", "Little" ...)

D.2.2 Meta data manager

The se.foi.ontology.externalmanager.MetaDataManager interface must be implemented by a class if it wants to act as a meta data manager.

The two methods in the MetaDataManager interface are:

public float getDataQuality(java.lang.String sensorName,

java.awt.Rectangle AOI,

java.util.Date timestamp)

Gets the data quality for sensorName in the AOI at the time described by timestamp.

Parameters:

sensorName - Sensor to get data quality for

AOI - Area Of Interest

timestamp - Timestamp Of Interest

Returns:

float in the interval [0..1] describing the data quality, where 0 means completely useless or nonexistent data and 1 means data of perfect quality

public java.lang.String getSensorPlatform(java.lang.String sensorName,

java.awt.Rectangle AOI,

java.util.Date timestamp)

Gets the sensor platform that carried the sensorName with the best data quality in the AOI at the time described by timestamp.

Parameters:

sensorName - Sensor to get sensor platform for

AOI - Area Of Interest

timestamp - Timestamp Of Interest

Returns:

The name of the sensor platform

D.2.3 Terrain data manager

The se.foi.ontology.externalmanager.TerrainDataManager interface must be implemented by a class if it wants to act as a terrain data manager.

The only method in the TerrainDataManager interface is:

public java.lang.String getVegetation(java.awt.Rectangle AOI,

java.util.Date timestamp)

Gets the vegetation (terrain background) in the AOI at the time described by timestamp.

Parameters:

AOI - Area Of Interest

timestamp - Timestamp Of Interest

Returns:

Name of the vegetation