



Multi-Reasoner Inference

Software Architecture Document

Guillaume Morin-Brassard
Fujitsu Consulting (Canada) Inc.

Vincent Giroux
Fujitsu Consulting (Canada) Inc.

Prepared By:
Fujitsu Consulting (Canada) Inc.
2000 Boulevard Lebourgneuf
Bureau 300
Québec (Québec)
G2K 0E8

Defence and Public Security
Contractor's Document Number: MRI-242-0449
Contract Project Manager: Gilles Clairoux, (514) 393-8822 x 318
PWGSC Contract Number: W7701-10-4064
CSA: Étienne Martineau, Defense Scientist, 418-844-4000 x 4501

The scientific or technical validity of this Contract Report is entirely the responsibility of the Contractor and the contents do not necessarily have the approval or endorsement of Defence R&D Canada.

Defence R&D Canada – Valcartier

Contract Report
DRDC Valcartier CR 2012-004
January 2012

Canada 

Multi-Reasoner Inference

Software Architecture Document

Guillaume Morin-Brassard
Fujitsu Consulting (Canada) Inc.

Vincent Giroux
Fujitsu Consulting (Canada) Inc.

Prepared By:
Fujitsu Consulting (Canada) Inc.
2000 Boulevard Lebourgneuf
Bureau 300
Québec (Québec)
G2K 0E8

Defence and Public Security
Contractor's Document Number: MRI-242-0449
Contract Project Manager: Gilles Clairoux, (514) 393-8822 x 318
PWGSC Contract Number: W7701-10-4064
CSA: Étienne Martineau, Defense Scientist, 418-844-4000 x 4501

The scientific or technical validity of this Contract Report is entirely the responsibility of the Contractor and the contents do not necessarily have the approval or endorsement of Defence R&D Canada.

Defence R&D Canada – Valcartier

Contract Report
DRDC Valcartier CR 2012-004
January 2012

This page intentionally left blank.

Abstract

To support its research activities in the intelligence domain, the Intelligence and Information (I&I) Section at DRDC Valcartier is developing the Intelligence Science & Technology Platform (ISTIP) as a major component of its R&D infrastructures. To improve the reasoning capabilities of the platform, the mandate of this contract is to produce a Multi-Reasoner Inference (MRI) capability based on the Multi-Intelligence Tool Suite (MITS) and the ISTIP software components previously developed by the I&I Section. Five main different services have been developed containing four individual reasoners and one multi-reasoner orchestrator. The reasoners that have been created are a Case-Based Reasoner (CBR), a Rule-Based Reasoner (RBR), a Descriptive-Logic Reasoner (DLR) and a KInematics and Geospatial Analysis Reasoner (KIGAR) which is based on the KIGAM module of the Inference of Situational Facts through Automated Reasoning (ISFAR) tool. Through the use of a common reasoning framework, these reasoners can now leverage their reasoning capabilities by sharing their strength to other reasoners and achieve an amazing synergy. This document describes the Software Architecture of the MRI.

Résumé

Afin de supporter ces activités de recherche dans le domaine du renseignement, la Section du Renseignement et Information de RDDC Valcartier développe la Plate-forme de Science et Technologie du Renseignement (ISTIP) comme un composant majeur de ses infrastructures de R&D. Afin d'améliorer les aptitudes de raisonnement de la plate-forme, le mandat de ce contrat est de créer un outil d'inférence Multi-Raisonneur (MRI) basé sur la « Multi-Intelligence Tool Suite » (MITS) et sur les composants logiciels déjà implémentés par la section I&I. Cinq différents services ont été développés comprenant quatre raisonneurs individuels et un orchestrateur multi-raisonneur. Les raisonneurs qui ont été créés sont un raisonneur par cas (CBR), un raisonneur par règles (RBR), un raisonneur ontologique (DLR) et un raisonneur d'analyse cinématique et géo-spatiale (KIGAR) basé sur le module KIGAM de l'outil d'Inférence Automatisée de Faits Situationnels (ISFAR). Grâce à l'utilisation d'un cadre de raisonnement commun, ces raisonneurs peuvent désormais exploiter leurs capacités de raisonnement en partageant leurs forces à d'autres raisonneurs et parvenir à une synergie épatante. Ce document décrit l'Architecture Logicielle du MRI.

This page intentionally left blank.

Executive summary

Multi-Reasoner Inference: Software Architecture Document

**Guillaume Morin-Brassard; Vincent Giroux; DRDC Valcartier CR 2012-004;
Defence R&D Canada – Valcartier; January 2012.**

Introduction or background:

To support its research activities in the intelligence domain, the Intelligence and Information (I&I) Section at DRDC Valcartier is developing the Intelligence Science & Technology Platform (ISTIP) as a major component of its R&D infrastructures. To improve the reasoning capabilities of the platform, the mandate of this contract is to produce a Multi-Reasoner Inference (MRI) capability based on the Multi-Intelligence Tool Suite (MITS) and the ISTIP software components previously developed by the I&I Section.

This document presents the Software Architecture Description (SAD) for the Multi-Reasoner Inference service and related reasoners services, according to the IEEE 12207.

Its purpose is to:

- ♦ To define all of the important system components and the associations between them from the viewpoint of the user.
- ♦ To establish the technical foundations of the system, to partition it into developer subsystems and software components, and to show the associations with the user requirements and technology infrastructure.
- ♦ To describe the users' critical requirements and design principles for the information system.
- ♦ To define global quality criteria against which the users will measure the information system.
- ♦ To identify and provide an initial definition for the manual and automated unit tasks.

Results:

Five main different services have been developed containing four individual reasoners and one multi-reasoner orchestrator. The reasoners that have been created are a Case-Based Reasoner (CBR), a Rule-Based Reasoner (RBR), a Descriptive-Logic Reasoner (DLR) and a Kinematics and Geospatial Analysis Reasoner (KIGAR) which is based on the KIGAM module of the Inference of Situational Facts Through Automated Reasoning (ISFAR) tool.

Sommaire

Multi-Reasoner Inference: Software Architecture Document

Guillaume Morin-Brassard; Vincent Giroux ; DRDC Valcartier CR 2012-004 ; R & D pour la défense Canada – Valcartier; janvier 2012.

Introduction ou contexte :

Afin de supporter ces activités de recherche dans le domaine du renseignement, la Section du Renseignement et Information de RDDC Valcartier développe la Plate-forme de Science et Technologie du Renseignement (ISTIP) comme un composant majeur de ses infrastructures de R&D. Afin d'améliorer les aptitudes de raisonnement de la plate-forme, le mandat de ce contrat est de créer un outil d'inférence Multi-Raisonneur (MRI) basé sur la « Multi-Intelligence Tool Suite » (MITS) et sur les composants logiciels déjà implémentés par la section I&I.

Ce document présente la Description de l'Architecture Logicielle du Raisonneur Multi-Inférence et ses services, en respectant la norme IEEE 12207.

Son but est de:

- ♦ Définir tous les composants importants du système et leurs associations du point de vu de l'utilisateur.
- ♦ Établir les fondations techniques du système, et les partitionner en tant que sous-systèmes et composants pour les développeurs, puis démontrer leurs associations avec les besoins client et l'infrastructure technologique.
- ♦ Décrire les besoins critiques de l'utilisateur et les principes de design pour le système d'information.
- ♦ Définir les critères de qualité globaux avec lesquels l'utilisateur comparera le système.
- ♦ Identifier et fournir une définition initiale des tâches manuelles et automatisées.

Résultats : Cinq différents services ont été développés comprenant quatre raisonneurs individuels et un orchestrateur multi-raisonneur. Les raisonneurs qui ont été créés sont un raisonneur par cas (CBR), un raisonneur par règles (RBR), un raisonneur ontologique (DLR) et un raisonneur d'analyse cinématique et géo-spatiale (KIGAR) basé sur le module KIGAM de l'outil d'Inférence Automatisée de Faits Situationnels (ISFAR).

Table of contents

Abstract	i
Résumé	i
Executive summary	iii
Sommaire	iv
Table of contents	v
List of figures	vii
List of tables	viii
1 Situation.....	1
1.1 Reasoners core working sets	1
2 General Solution	2
2.1 Common interface	2
2.2 Facts conversions.....	2
2.2.1 Facts to system triplets.....	3
2.2.2 Facts to spatial features.....	4
2.3 The “Know How”	7
2.4 The parameters	7
2.5 Reasoner outputs	7
2.5.1 Facts.....	7
2.5.2 Facts Justifications.....	8
2.6 Reasoner request context.....	8
2.7 Unit/Functional Testing Strategy.....	8
3 Software Architecture	10
3.1 Technologies	10
3.2 Standards	10
4 Reasoning services.....	11
4.1 Common	11
4.1.1 Description.....	11
4.1.2 Service data.....	11
4.1.2.1 Messages	11
4.1.2.2 Facts	11
4.1.2.3 Reasoning context.....	11
4.1.3 Service dynamics	11
4.1.3.1 General use case.....	11
4.1.3.2 Create reasoning context.....	13
4.1.3.3 Add facts to reasoning context.....	13
4.1.3.4 Execute reasoning context	14
4.1.3.5 Get context execution status	15

4.1.3.6	Get reasoning results.....	16
4.1.3.7	Delete reasoning context.....	16
4.2	Kinematic and Geospatial Analysis Reasoner (KIGAR).....	17
4.2.1	Description.....	17
4.2.2	Service data.....	17
4.2.3	Service dynamics.....	17
4.3	Rule-Based Reasoner (RBR).....	19
4.3.1	Description.....	19
4.3.2	Service data.....	19
4.3.3	Service dynamics.....	20
4.4	Case-Based Reasoner (CBR).....	21
4.4.1	Description.....	21
4.4.2	Service data.....	21
4.4.2.1	Situation templates.....	22
4.4.2.2	Global similarity measure + Local Similarity Measures.....	23
4.4.2.3	Typical cases.....	24
4.4.2.4	Solution generation rule.....	25
4.4.3	Service dynamics.....	25
4.5	Descriptive Logic Reasoner (DLR).....	28
4.5.1	Description.....	28
4.5.2	Service data.....	29
4.5.2.1	URI Mappings.....	29
4.5.2.2	Ontology References.....	29
4.5.3	Service dynamics.....	29
4.6	Multi-Reasoners Orchestrator.....	34
4.6.1	Description.....	34
4.6.2	Service data.....	34
4.6.2.1	Reasoner configuration.....	34
4.6.3	Service dynamics.....	34
5	References.....	36
Annex A	System Attributes.....	37
	List of symbols/abbreviations/acronyms/initialisms.....	41

List of figures

Figure 1: Reasoners common interface	2
Figure 2: Triplet mappings	3
Figure 3: Triplet Mappings Results	4
Figure 4 - General use case	12
Figure 5 - Create a reasoning context activity diagram	13
Figure 6 - Add facts to context activity diagram	14
Figure 7 - Execute reasoning context activity diagram	15
Figure 8 - Get context execution status activity diagram	16
Figure 9 - Get reasoning results activity diagram	16
Figure 10 - Delete context activity diagram	17
Figure 11: KIGAR Workflow	18
Figure 12: Rule-Based Reasoner Workflow	20
Figure 13: Case-Based Reasoner Data	22
Figure 14: Case-Based Reasoner Join Constraint	23
Figure 15: Similarity Measures	24
Figure 16: Cases	24
Figure 17: Case-Based Reasoner Processing Workflow	25
Figure 18: Fact to Situations Templates	26
Figure 19: Situation and Cases Similarity Processing	27
Figure 20: Case-Based Reasoner Output	28
Figure 21: Descriptive Logic Reasoner Workflow	30
Figure 22: URI Mappings	31
Figure 23: Triplets Insertion	33
Figure 24: Pellet categorization	33
Figure 25: MRI Orchestrator Processing Workflow	35

List of tables

Table 1 : System Attributes 37

1 Situation

Currently, each reasoner core requires different inputs and generates outputs that are not necessarily usable by the other reasoners. Moreover, the reasoners are actually all aggregated in a single application and cannot be called separately without invoking the ISFAR main interface.

Since the ISTIP platform aims at being an SOA platform, some work needs to be done to expose each reasoners as separate services. Moreover, since we want to improve usability of these services and be able to make them interoperable within the MRI service, more work has to be done to make them work with a common set of inputs and outputs which here is the SFM fact model.

1.1 Reasoners core working sets

The following section describes each reasoner core working set – which type of information is required by each of them and the data format suited to each of them.

Rule-Based Reasoner (RBR) – JBoss Drools

- ♦ Facts (can natively use facts from SF fact model)
- ♦ Rules (DRL)

Descriptive Logic Reasoner (DLR) – Pellet OWL reasoner

- ♦ Axioms – Triplets
- ♦ Ontology

Case-Based Reasoner (CBR) – jCollibri

- ♦ Case base (which can be anything supported by the similarity measures);
- ♦ Similarity measures and decision thresholds;

KIGAR

- ♦ KIGAM tracks (which could easily be adapted to spatial features)
- ♦ Spatial features
- ♦ Analysis parameters and spatial features filters to apply specific parameters to specific spatial features;

2 General Solution

2.1 Common interface

To improve the usability of the system, a common interface has to be implemented by each reasoner. This interface makes sure each reasoner implements a set of methods normalizing the interaction with the reasoner.

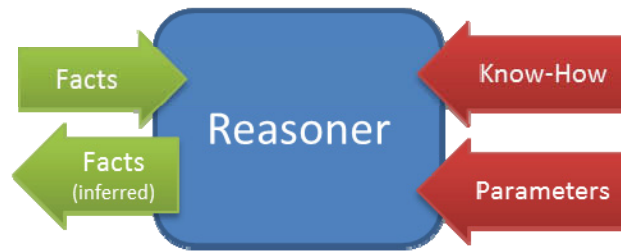


Figure 1: Reasoners common interface

This interface exposes:

- ♦ methods to create a new inference context and have a reference to that context;
- ♦ methods to add facts to the context (facts are the common input and output of each reasoner);
- ♦ methods to set the “Know How” of each reasoner context (The definition of what is the “Know How” will be given below);
- ♦ methods to set the execution parameters of each reasoner context (The definition of what are the parameters will be given below);
- ♦ methods to retrieve the status and other exploitation data of the reasoner for a given context;
- ♦ methods to retrieve the reasoning results for a given context;

Moreover, the MRI orchestrator has the exact same interface than each individual reasoner and internally dispatches facts, proper “Know How” and parameters to each reasoner underneath.

2.2 Facts conversions

The facts are the common input and output of each reasoner. Even if facts are passed from a reasoner to another, it is not all reasoners that work directly with facts as shown in the previous section. Some of the reasoners need to convert these facts in a form that is more suitable for them. For example, the Domain Logic Reasoner does not know how to infer new knowledge from complex facts, it is only able to work with triplets. The facts must then be converted to triplets to be usable by the DLR.

Here is a list of the required fact conversion for each reasoner:

- ♦ DLR: Facts → System triplets → Triplets with ontology specific attribute names

- ♦ KIGAR: Facts \rightarrow System triplets \rightarrow SpatialFeatures
- ♦ RBR: Facts (No conversion required)
- ♦ CBR: Facts \rightarrow Situations (logically aggregated facts)

2.2.1 Facts to system triplets

A Fact to subject-attribute-value mapping has to be specified with each atom definition so that facts can be expressed as subject-attribute-values triplets. A mapping is defined this way:

For a Fact “F” with arguments “A₁, A₂” like F(A₁, A₂) and V(A_n) being the value of the argument A_n, a triplet could be specified as such:

- ♦ Subject \rightarrow V(A₁)
- ♦ Attribute \rightarrow “attribute X” or V(A_x)
- ♦ Value \rightarrow “value Y” or V(A₂)

Where the value Y and attribute name X are any String.

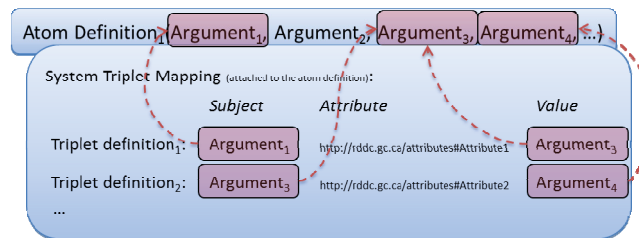


Figure 2: Triplet mappings

This would allow to automatically and easily convert any facts into triplets, no matter how the facts have been modeled.

Here is an example:

The following triplet mapping could be attached to the atom definition Vessel(VesselId, VesselName, CargoType, Flag, Owner):

- ♦ V(VesselId) – “name” – V(VesselName)
- ♦ V(VesselId) – “hasCargoType” – V(CargoType)
- ♦ V(VesselId) – “hasFlag” – V(Flag)
- ♦ V(VesselId) – “hasOwner” – V(Owner)

Afterward, any fact based on this atom definition could be automatically translated into triplets:

- ♦ Vessel(“MMSI123411”, “Great Catch”, “FSH”, “CAN”, “Bob”)
 - ♦ Generated triplets:
 - “MMSI123411” – “name” – “Great Catch”
 - “MMSI123411” – “hasCargoType” – “FSH”
 - “MMSI123411” – “hasFlag” – “CAN”

- ♦ “MMSI123411” – “hasOwner” – “Bob”
- ♦ Vessel(“MMSI999862”, “Big Bertha”, “OIL”, “USA”, “Oil co.”)
 - ♦ Generated triplets:
 - “MMSI999862” – “name” – “Big Bertha”
 - “MMSI999862” – “hasCargoType” – “OIL”
 - “MMSI999862” – “hasFlag” – “USA”
 - “MMSI999862” – “hasOwner” – “Oil co.”

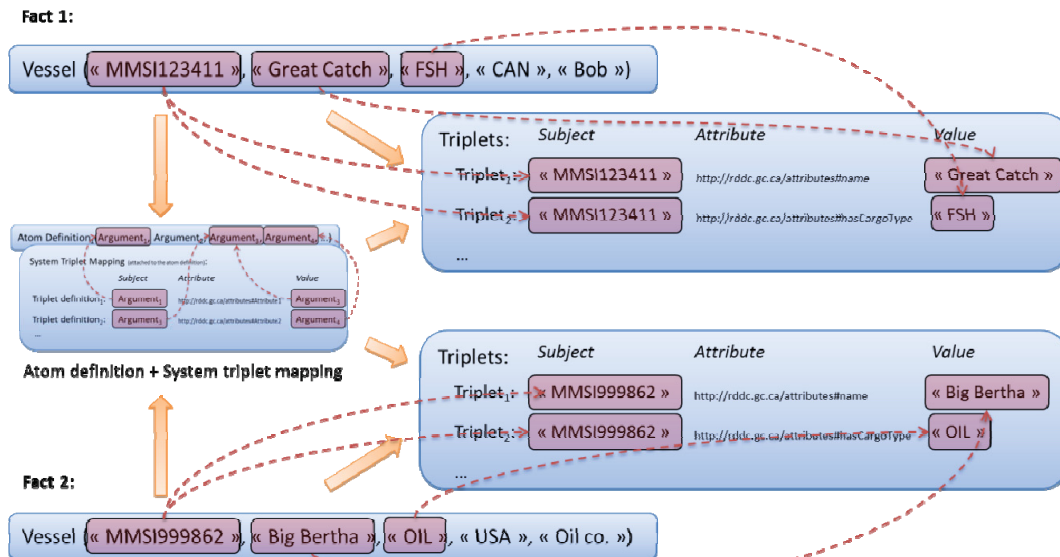


Figure 3: Triplet Mappings Results

Note: Please note that the example above was exclusively using text argument types for simplification purpose. However, since fact arguments are strongly typed (their values are of a specific type that is defined in the atom definition – Ex: text, numerical, ontology entity reference, etc.), the generated triplets subjects and values will be strongly typed as well since their value are based on fact argument values.

2.2.2 Facts to spatial features

Spatial features can be regenerated from facts since every spatial feature is ultimately referring to a subject (an ontology instance extending the subject class of the subject ontology) having geospatial attributes attached to it. However, a fact does not explicitly define which argument is the subject and since we need this information to rebuild the spatial features and attach correctly its attributes back together, we need another formalism better aligned to a subject-attribute-value formalism. Therefore, the facts are first converted into system triplets to ease that reconstruction. This also allows modeling the spatial feature data into any fact structure as long as the triplet

mapping defines the proper attributes required for spatial feature conversion. This process is explained in the previous section ([Facts to system triplets](#)).

Afterward, based on the subject referred in the facts, the corresponding spatial features could be converted automatically based on system attributes such as:

- ♦ *hasMotionTrajectory* (<http://ca.gc.rddc/ontology/attributes.owl#hasMotionTrajectory>)
- ♦ *hasContact* (<http://ca.gc.rddc/ontology/attributes.owl#hasContact>)
- ♦ *hasGeometry* (<http://ca.gc.rddc/ontology/attributes.owl#hasGeometry>)
- ♦ *IsA* (<http://ca.gc.rddc/ontology/attributes.owl#isA>)
- ♦ *hasAltitude* (<http://ca.gc.rddc/ontology/attributes.owl#hasAltitude>)
- ♦ *hasLatitude* (<http://ca.gc.rddc/ontology/attributes.owl#hasLatitude>)
- ♦ *hasLongitude* (<http://ca.gc.rddc/ontology/attributes.owl#hasLongitude>)
- ♦ *hasSpeed* (<http://ca.gc.rddc/ontology/attributes.owl#hasSpeed>)
- ♦ *hasOrientation* (<http://ca.gc.rddc/ontology/attributes.owl#hasOrientation>)
- ♦ *hasTimestamp* (<http://ca.gc.rddc/ontology/attributes.owl#hasTimestamp>)
- ♦ *hasDestination* (<http://ca.gc.rddc/ontology/attributes.owl#hasDestination>)
- ♦ *hasEstimatedTimeOfArrival* (<http://ca.gc.rddc/ontology/attributes.owl#hasEstimatedTimeOfArrival>)
- ♦ *hasWidth* (<http://ca.gc.rddc/ontology/attributes.owl#hasWidth>)
- ♦ *hasMinimumSpeed* (<http://ca.gc.rddc/ontology/attributes.owl#hasMinimumSpeed>)
- ♦ *hasMaximumSpeed* (<http://ca.gc.rddc/ontology/attributes.owl#hasMaximumSpeed>)

More precisely, the triplets to spatial feature converter perform the following steps to build a spatial feature:

1. An empty spatial feature is created based on the subject URI to convert (the subject URIs to convert are extracted by extracting all subjects from triplets with an *IS-A* attribute (<http://ca.gc.rddc/ontology/attributes.owl#isA>) where the value is equal to <http://rddc.gc.ca/ISTIP/ontologies/Subjects.owl#SpatialFeature>;
2. If a subject's triplet containing the *hasGeometry* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasGeometry>) is found, its value is set as the spatial feature geometry;
3. If a subject's triplet containing the *hasMinimumSpeed* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasMinimumSpeed>) is found, its value is set as the spatial feature minimal speed;
4. If a subject's triplet containing the *hasMaximumSpeed* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasMaximumSpeed>) is found, its value is set as the spatial feature maximal speed;
5. If a subject's triplet containing the *hasWidth* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasWidth>) is found, its value is set as the spatial feature width;
6. If a subject's triplet containing the *hasMotionTrajectory* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasMotionTrajectory>) is found, it means that the spatial feature is a spatiotemporal feature since it has motion trajectories. Therefore, the triplet value is considered as a motion trajectory and the motion

trajectory is then converted using the following sub-steps and added then to the spatiotemporal feature. Please note that the triplet subject for these sub-steps is the motion trajectory identifier found in the triplet value :

- a. If a subject's triplet containing the *hasGeometry* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasGeometry>) is found, its value is set as the motion trajectory moving anchor;
- b. If a subject's triplet containing the *hasEstimatedTimeOfArrival* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasEstimatedTimeOfArrival>) is found, its value is set as the motion trajectory estimated time of arrival;
- c. If a subject's triplet containing the *hasWidth* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasWidth>) is found, its value is set as a motion trajectory custom attribute – This will be used in the case of a *Corridor* spatial feature where the corridor is defined by a series of points with a width;
- d. If a subject's triplet containing the *hasDestination* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasDestination>) is found, its value is set as the motion trajectory destination;
- e. If a subject's triplet containing the *hasContact* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasContact>) is found, it means that the triplet value is actually a motion trajectory contact. Therefore, the contact is then converted using the following sub-steps and then added to the motion trajectory. Please note that the triplet subject for these sub-steps is the contact identifier found in the triplet value:
 - i. If a subject's triplet containing the *hasGeometry* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasGeometry>) is found, its value is set as the contact geometry;
 1. If this attribute is not found, the service will try to find triplet values for the attributes *hasLongitude* (<http://ca.gc.rddc/ontology/attributes.owl#hasLongitude>), *hasLatitude* (<http://ca.gc.rddc/ontology/attributes.owl#hasLatitude>) and optionally *hasAltitude* (<http://ca.gc.rddc/ontology/attributes.owl#hasAltitude>). If these values are found, a geospatial Point is created from these values and added to the contact as the contact geometry;
 - ii. If a subject's triplet containing the *hasSpeed* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasSpeed>) is found, its value is set as the contact reported speed;
 - iii. If a subject's triplet containing the *hasOrientation* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasOrientation>) is found, its value is set as the contact orientation;
 - iv. If a subject's triplet containing the *hasTimestamp* attribute (<http://ca.gc.rddc/ontology/attributes.owl#hasTimestamp>) is found, its value is set as the contact report time;
 - v. Finally, if the contact is still missing its geometry or report time, the contact is discarded since it will not be usable;

7. All *IS-A* attributes (<http://ca.gc.rddc/ontology/attributes.owl#isA>) in the subject's triplets are added as spatial feature custom attributes. The rest of the attributes are not added to the spatial feature since they will not be used by KIGAR analyses.

Therefore, atom definitions for input facts must define triplet mappings using the attributes above in order to convert these facts into spatial features.

2.3 The “Know How”

The “Know How” is the apriori domain expert knowledge required by a reasoner to be able to work properly. It basically indicates specifically to each reasoner how to handle facts received in input to deduce new facts. Here is a list of the “Know How” required by each reasoner:

- ♦ DLR: Ontologies
- ♦ KIGAR: None required since the *knowhow* is static
- ♦ RBR: Rules
- ♦ CBR: Cases (Templates + corresponding solutions) and similarity measures

2.4 The parameters

The parameters allow the “fine-tuning” of each reasoner behavior. Here is a list of high level parameters required by each reasoner:

- ♦ DLR: None
- ♦ KIGAR: A list of parameters defining which analyses will be run, which subjects and objects will be processed and the variable values used within the analyses (ex: proximity thresholds, time extrapolation factor, etc.)
- ♦ RBR: None
- ♦ CBR: Similarity thresholds

2.5 Reasoner outputs

2.5.1 Facts

Each reasoner outputs the facts it inferred but some of the reasoner will output predefined fact types. Here is a list of the types of facts generated by each reasoner:

- ♦ DLR: For the scope of this project, the DLR output facts of type “SubjectType (IsA)” which defines the ontological hierarchy of “subjects” mentioned in facts. It also generates fact of type “Has Property” which defines an object property between to individuals that has been inferred by the DLR.
- ♦ KIGAR: Generates predefined types of facts associated to each analysis. For example, the proximity analysis will generate *InProximity(A1, A2)* facts.
- ♦ RBR: Generates facts defined by the knowhow rules' conclusions.
- ♦ CBR: Generates facts as defined in cases' solutions

Since some facts generated are predefined, we can consider the definition of these facts as being system atom definitions. Defining these definitions as system definitions makes it possible to reuse their result in our rules and in the other reasoners.

2.5.2 Facts Justifications

Facts outputted by each of the reasoners have a justification attached to them. These justifications are:

- ♦ The fact pedigree: it specifies which reasoner and/or analysis inferred that fact
- ♦ The fact dependencies: the fact dependencies specifies which facts have been used to deduce the inferred fact (if it was possible to extract that information)
- ♦ Reasoners specific attributes found in the fact attributes:
 - ♦ RBR:
 - adds a “ruleId” fact attribute which specifies the id of the rule which generated the fact.
 - ♦ CBR:
 - adds a “templateId” fact attribute which specifies the id of the template which generated the fact.
 - adds a “similarCaseWithSimilarityValue” fact attribute which specifies the id of the similar cases followed by their similarity factor value.
 - ♦ DLR:
 - adds a “DLR justification” fact attribute which contains the Pellet reasoner justifications that trigger the fact (only when the DLR parameter “justifications” is activated).

2.6 Reasoner request context

Similarly to ISFAR, the client invoking either a specific reasoner or the MRI orchestration will create a reasoning context and will receive a context identifier. This identifier will be used for subsequent requests to modify the context, retrieve the current status of the process and also retrieve inferred facts. This mechanism allows managing multiple inference contexts simultaneously without interfering with each other. Indeed, this context can be seen as a specific sandbox for each inference session.

2.7 Unit/Functional Testing Strategy

For each reasoner and for the orchestrator, a set of unit tests have been developed which runs on the continuous integration periodically to ensure that the code is working properly and that there is no regression occurring. Unit tests should be added each time:

- ♦ a new bug is found
- ♦ new functionalities are implemented

- ♦ etc.

Moreover, for each reasoner and the orchestrator, a set of functional SoapUI tests have been implemented to ensure that the code is working properly and to ensure that clients can use the system without any issues. It also makes sure the interconnections between the systems components are done correctly. These tests are also run on the continuous integration server periodically to make sure there is no regression.

3 Software Architecture

3.1 Technologies

The MRI services are based on the following technologies:

- ♦ JBoss AS 5.1: The application server hosting the web services and handling requests and responses to and from the MRI web services
- ♦ EJB 3.0: The EJB technology is used to provide a simple framework for exposing the reasoning services in a standard manner both through stateless beans (remote and local) and also through SOAP web services. Moreover, each reasoner execution queue is exploited through an EJB 3 message driven bean, which automatically manage the simultaneous process execution by de-queuing messages from the execution queue.
- ♦ EHCACHE: This library is used to cache many data used through the reasoning lifecycle. Some of this information can be resource consuming, so a subset of this data is kept in a memory cache to reuse it whenever possible. EHCACHE automatically handle garbage collection to keep the cache size within a reasonable size.
- ♦ Kryo: This library is used to persist the reasoning context as serialized objects on the file system. This library has been chosen for its high throughput performance.

Moreover, each reasoner is based on an open source reasoning engine:

- ♦ degree API: used for geospatial calculation within KIGAR
- ♦ JBoss Drools: used as the core rule-based reasoning engine within the RBR
- ♦ jColibri: used as the core case-based reasoning engine within the CBR
- ♦ Pellet: Used as the core ontological reasoning engine within the DLR

The MRI services are deployed on a Microsoft Windows Server 2003 virtual machine with 4 CPUs and 10 gigs of RAM allocated.

3.2 Standards

The MRI services are exposed through the following standard endpoints:

- ♦ SOAP Web service
- ♦ Java Stateless session bean exposed remotely through RMI

4 Reasoning services

4.1 Common

4.1.1 Description

This section describes the general workflow and objects used within each individual reasoner.

4.1.2 Service data

Since all reasoner services implements the same interface, base service data objects are provided in a common project.

4.1.2.1 Messages

Messages allow logging feedback messages about the execution of the reasoner. This mainly allows the user to understand what happened during the processing and which errors occurred if any.

4.1.2.2 Facts

Fact objects from the SFM services are used within the different reasoners.

4.1.2.3 Reasoning context

A base reasoning context is provided as the basis working memory for each reasoner. It mainly contains the following aspects:

- ♦ The input facts
- ♦ Corresponding atom definitions for input facts
- ♦ The messages
- ♦ The reasoner parameters
- ♦ The reasoner *knowhow*
- ♦ The inferred facts

Specific reasoners may extend this base context as needed to add any objects required to keep a consistent state between invocations.

4.1.3 Service dynamics

4.1.3.1 General use case

The following diagram depicts the general use case for a MRI reasoner service:

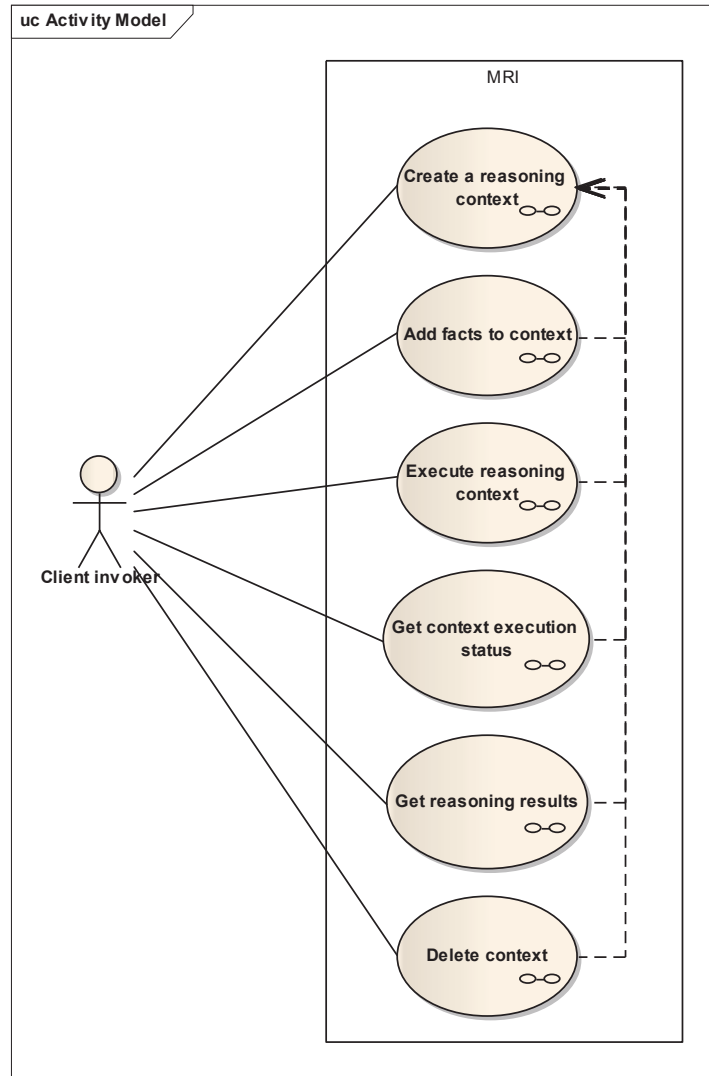


Figure 4 - General use case

At first, the client invoker (which could be an external application, another web service or even a user using any web service client software) must create a reasoning context. This context can be seen as the client sandbox. It contains all specific parameters, know how, facts and so on required by the reasoner to work properly with the domain data. This way, the client can work with its data concurrently with other clients without affecting them (or being affected by others).

Once the reasoning context has been created, the client will receive a context handle. This handle will be used afterward to perform any action on the context, such as adding new facts to the context or executing the context – the execution must be explicitly invoked to avoid executing the context before the context fully ready.

Since the reasoning process may take a while, when a reasoning context is executed, the reasoned will simply queue the request and immediately respond to the client. The client can then invoke the reasoner to get the latest execution status. When the context is completed, the client can then

invoke the reasoned to retrieve the reasoning results, containing basically the inferred facts and optionally feedback messages.

The reasoning context will be available at any moment until the delete context operation is invoked. This operation allows deleting everything associated with the context. Thus, it is mandatory to invoke this method once the context is not needed anymore to avoid keeping old contexts.

In the future, an automatic cleaning mechanism may be implemented to automatically delete old reasoning context that have not been modified since a long time.

4.1.3.2 Create reasoning context

The following diagram depicts the main workflow involved when creating a new reasoning context:

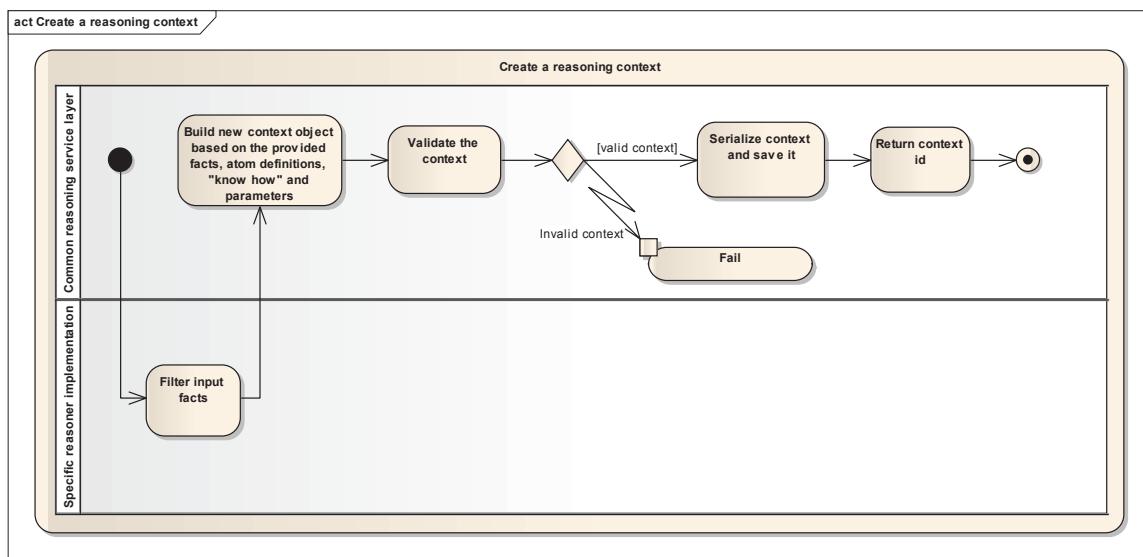


Figure 5 - Create a reasoning context activity diagram

For the creation of a reasoning context, the input facts are first filtered out to keep only facts that are relevant for the reasoner. The filtered facts, atom definitions, *knowhow* and parameters are first assembled into a working context object. The resulting object is then validated. If the validation succeed, the context is serialized, persisted and the context identifier is then returned.

4.1.3.3 Add facts to reasoning context

The following diagram depicts the main workflow involved when adding facts to an existing reasoning context:

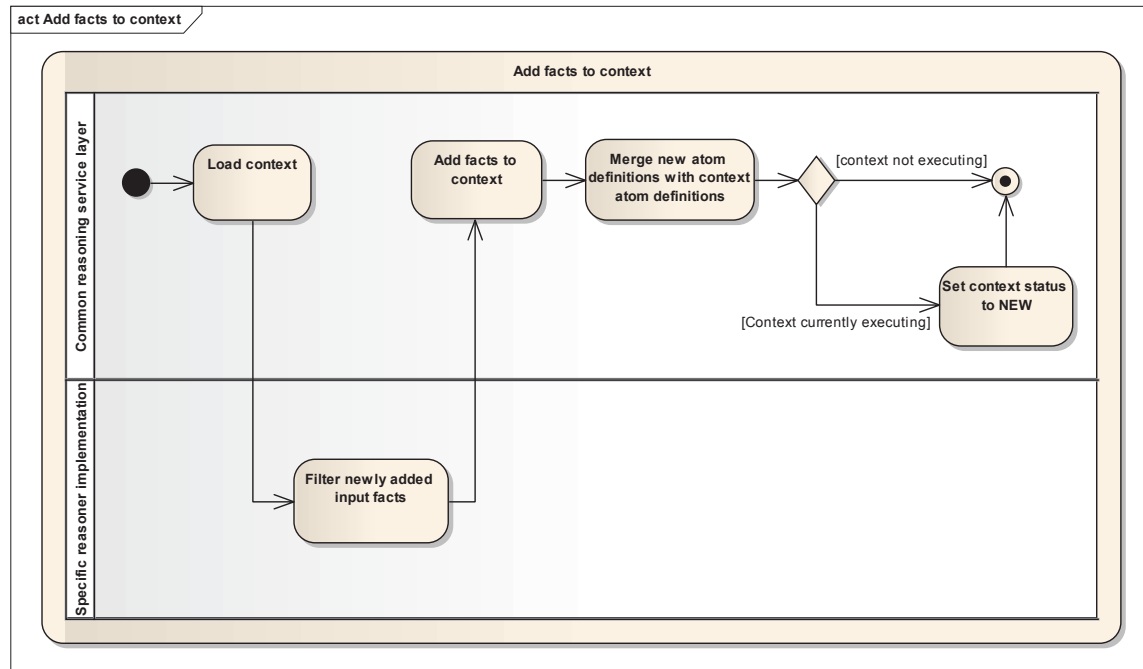


Figure 6 - Add facts to context activity diagram

When adding new facts to a context, the reasoned first loads the context and unserialize it. Then, the facts are filtered and added to the context and identified as new facts to make sure only the facts that have not been processed yet will be added to the specific reasoner engine. Then, if atom definitions have been provided with the new facts, they are added to the existing atom definitions in the context.

Finally, if the context is currently being executed, the status is changed to NEW to identify that there are new facts that needs to be processed afterward.

4.1.3.4 Execute reasoning context

The following diagram depicts the main workflow involved when executing a reasoning context:

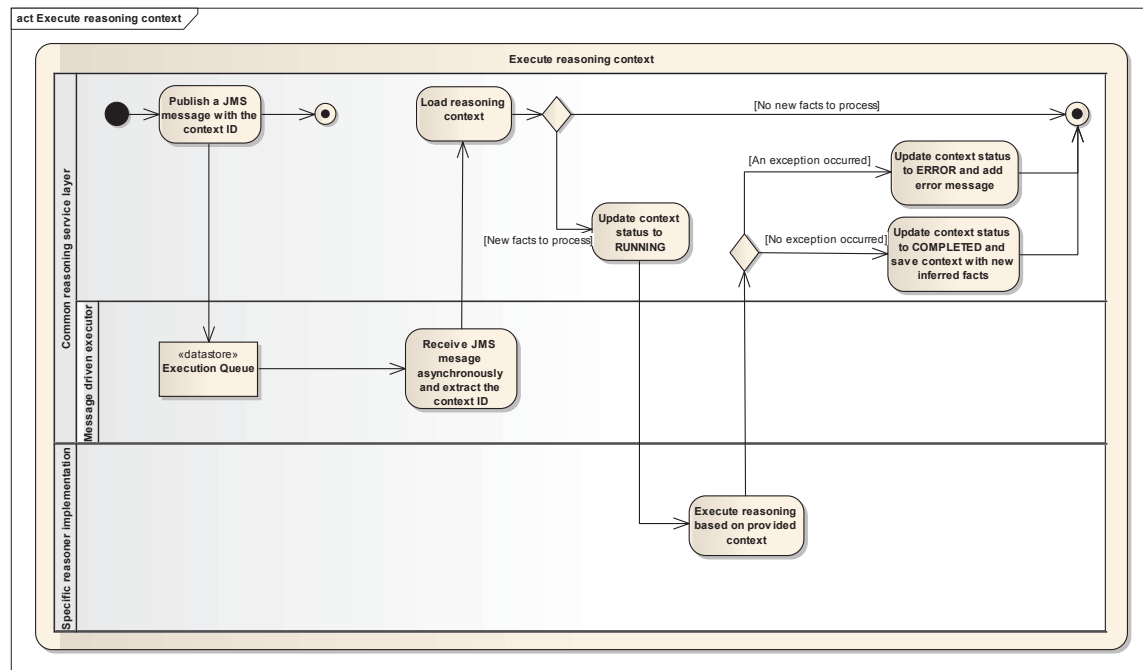


Figure 7 - Execute reasoning context activity diagram

Since the reasoning execution can be pretty resource consuming, an asynchronous execution is implemented to limit the number of simultaneous process execution through a JMS queue. This also allows continuing serving execution requests by clients even if the limit is reached.

Therefore, when the execute method of the service is invoked, only a JMS message is published and the client request ends right there.

Asynchronously, messages published in the JMS execution queue are then de-queued. At this time, the context is loaded based on the context id provided in the JMS message. The context status is set to RUNNING and the specific reasoner implementation is invoked. Please refer to the next sections for more information about the specific workflow.

Finally, if an exception occurred, the context status is set to ERROR and an error message is added to the context. Otherwise, the context status is set to COMPLETED and is saved with the new inferred facts.

4.1.3.5 Get context execution status

The following diagram depicts the main workflow involved when retrieving the context execution status:

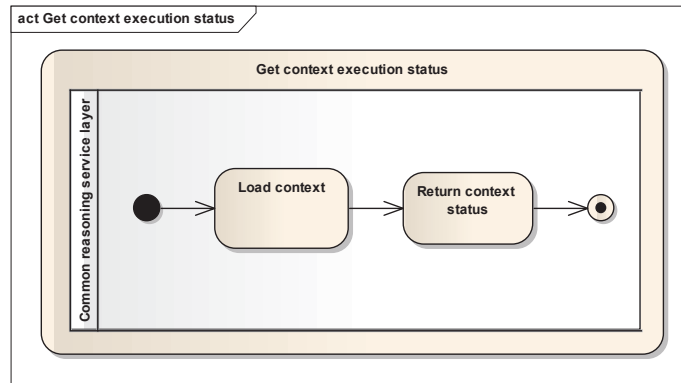


Figure 8 - Get context execution status activity diagram

This action simply loads the context based on the provided context id and return the context status saved within the context.

4.1.3.6 Get reasoning results

The following diagram depicts the main workflow involved when retrieving the reasoning results:

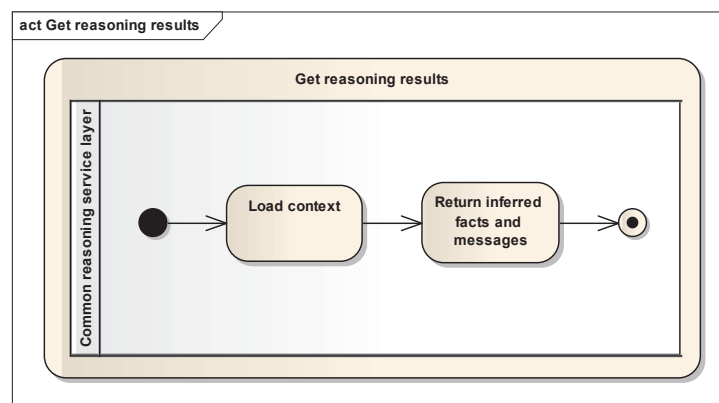


Figure 9 - Get reasoning results activity diagram

This action simply loads the context based on the provided context id and returns the inferred facts that have been added by the specific reasoner implementation and the messages saved within the context.

4.1.3.7 Delete reasoning context

The following diagram depicts the main workflow involved when deleting a reasoning context:

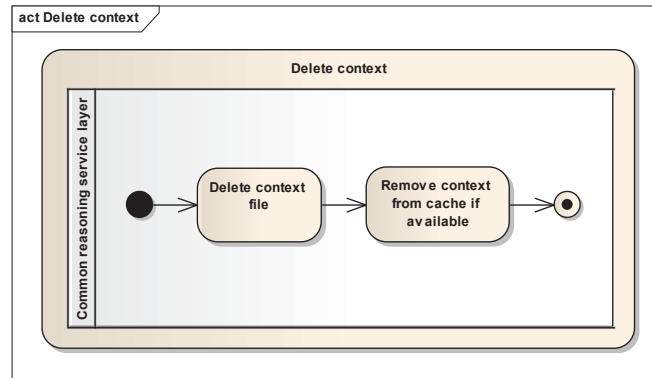


Figure 10 - Delete context activity diagram

This action simply deletes the serialized context file. It also removes the context from the memory cache if available to make sure the context is not available anymore and to free memory occupied by this context.

4.2 Kinematic and Geospatial Analysis Reasoner (KIGAR)

4.2.1 Description

The Kinematic and Geospatial Analysis Reasoner is a module containing 30 geospatial analyses (or geospatial functions) that can be grouped in two main categories: location and motion analyses.

Please refer to the document *Kinematic and Geospatial Analysis Module (KIGAM) Analysis Fact Sheets* for more information on each analysis.

4.2.2 Service data

The required data for KIGAR is quite simple. It basically requires facts that will be used to build corresponding spatial (or spatio-temporal) features internally and parameters to determine which analysis to execute with which features and also to override default analysis parameter values.

4.2.3 Service dynamics

The general service dynamic of the KIGAR service is common to other reasoners. Please refer to the common [Service dynamics](#) section for more information. Only the internal execution mechanism differs from other reasoners.

The following diagram depicts the main workflow of the Kinematic and Geospatial Analysis Reasoner specific execution:

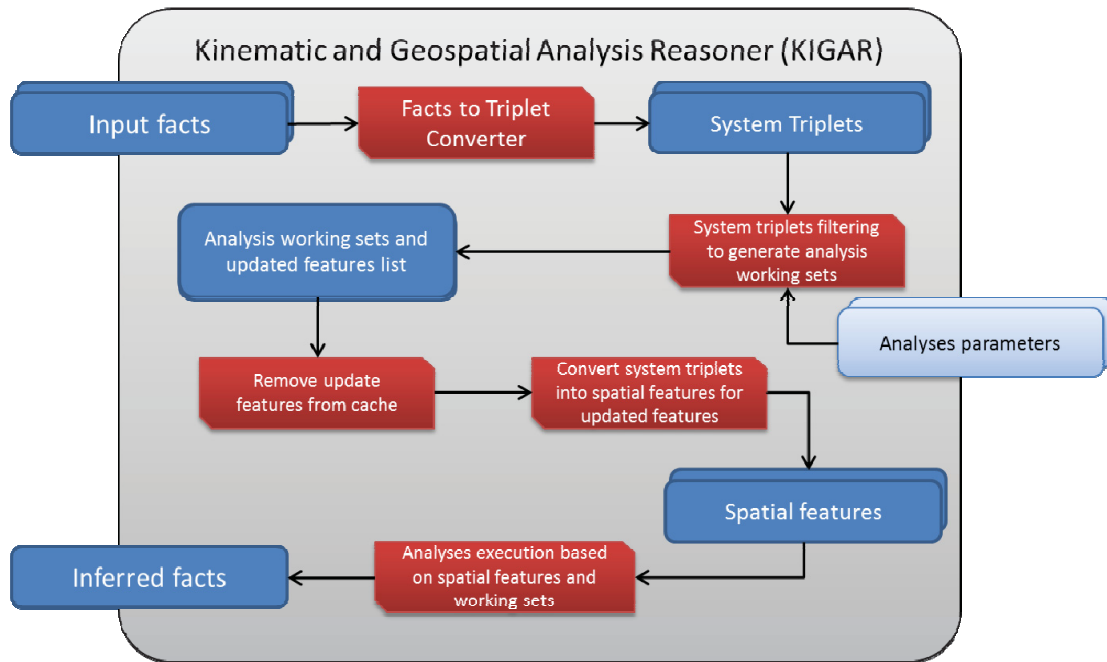


Figure 11: KIGAR Workflow

1. The new facts are first converted into system triplets, which are added to the previously transformed triplets. This intermediate transformation allows easing facts to spatial features transformation by providing a low level set of information with known attributes and more importantly identified subjects. Information modeled in any atom definition form will end up in the same triplet representation. Therefore, any atom definitions defined by knowledge engineers using known system attributes¹ in their triplet mappings will be usable in KIGAR.

Please refer to the [Facts to system triplets](#) section for more information about how triplet mappings and how facts are converted into triplets using this mapping.

2. Once the triplets have been converted, the provided analyses parameters are then used to extract KIGAR working sets. A working set is basically a list of subjects (or subjects/objects pairs), an analysis type to execute and the parameter values to apply for these subjects. At the same time, the list of subjects that have been added or updated by new input facts is also populated. This list determines afterward which combination to reprocess within working sets to reprocess only subjects (or objects) that have been updated/added.
 - a. Since KIGAR analyze spatial/spatiotemporal features that must be of specific kinds² while triplets' subjects can be of any kind, triplets' subjects must first be analyzed to determine if they are of a compatible

¹ See [Appendix A](#) for more details concerning the known system attributes

² Refer to KIGAR Analyses appendix in Multi-Reasoner Inference SIDD document for the detailed list of subjects/objects types required for each KIGAR analysis.

kind. To do so, the service loops through the subject's triplets to find a triplet having the system attribute *IS-A* ("http://ca.gc.rddc/ontology/attributes.owl#isA") and the required ontology class reference as the triplet value³. Therefore, the triplets must explicitly contain such attribute values to be usable within KIGAR.

- b. Since the new (or updated) subjects coming from the newly added facts triplets may need to be compared with subjects that have been processed before (at context creation for example), the analysis working sets (all pairs of subject/objects to analyze for a given analysis) are generated using all subjects available in the reasoning context. Once the working sets are generated, only subject/object pairs containing a new (or updated) subject are executed.
3. The updated spatial features are removed from the cache to make sure they will be generated with the new data.
4. Afterward, new and updated spatial features are generated based on the system triplets. Please refer to the [Facts to spatial features](#) section for more information on how this conversion is achieved.
5. Finally, each subject (or subject/object pair) is executed on the selected analysis for each working set, potentially generating new inferred facts⁴, which would then be added to the context.

4.3 Rule-Based Reasoner (RBR)

4.3.1 Description

The Rule-Based Reasoner will be based on the MITS model and inference engine. To integrate these into the MRI-RBR, the following steps will be required:

1. Extract the "MITS inference Rules to Drools Rules Converter"
2. Implement an RBR *knowhow* based on the MITS inference rules model
3. Adapt the converter to the SFM model and the RBR *knowhow*
4. Wrap the converter and the Drools engine in a Reasoner respecting the common reasoning interface

4.3.2 Service data

The required data for the Rule-Based Reasoner is quite simple. It simply requires inference rules (based on the MITS model) and input facts. The rules specify exactly which facts and arguments value are required to create a new fact and also defines the signature and content of the fact that

³ Here, the class reference can either be of type Ontology entity reference or as a literal text corresponding to the class URI.

⁴ Refer to KIGAR Analyses appendix in Multi-Reasoner Inference SIDD document for the detailed list of possible inferred facts that each KIGAR analysis can generate.

will be generated at the output. There is no system atom definitions used by the RBR; all facts generated are specified by the client through the reasoner *knowhow*.

4.3.3 Service dynamics

The general service dynamic of the RBR service is common to other reasoners. Please refer to the common [Service dynamics](#) section for more information. Only the internal execution mechanism differs from other reasoners.

The Rule-Based Reasoner will execute the following workflow to infer new facts:

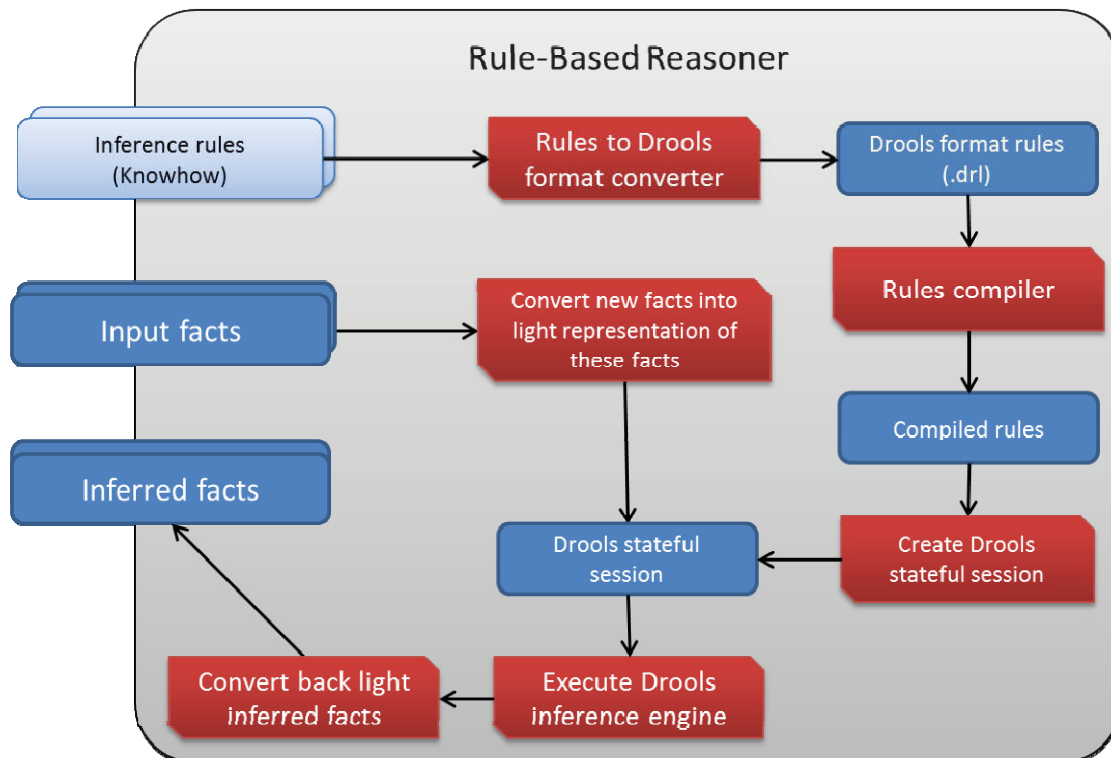


Figure 12: Rule-Based Reasoner Workflow

1. At first, if the inference context is new, inference rules specified in the *knowhow* are sent to the “Rules to Drools format Converter”. Otherwise, the Drools stateful session will simply be restored and the workflow will continue to the step 5.
2. The rules are then converted to a format known by the Drools engine (DRL)
3. The rules are compiled by the rules compiler
4. Then, a Drools stateful session is created based on the compiled rules
5. The new input facts are converted into a light fact representation to ease their usability in the rules within Drools.
6. These converted “light facts” are then added to the Drools stateful session.

7. At this moment, the Drools session is now ready and executed to infer new facts as rules premises are matched.
8. Finally, inferred light facts are converted back into inferred facts based on the SFM model.

4.4 Case-Based Reasoner (CBR)

4.4.1 Description

The case-based reasoner service mainly based on jColibri library and also inspired on the CBR proof of concept done in a previous contract by OODA technologies. The previous CBR was using predefined similarity measures and situation models which were linked together through existing cases. These measures and models were implemented as hard-coded classes extending a generic interface. Therefore, these classes had to be programmed for each domain that the system had to work with. Since the new reasoning modules need to be fully domain agnostic, a generic and configurable class model must be implemented.

Some modifications have been required to be able to integrate it with the current reasoning common interface, such as:

- ♦ Create a generic (configurable) similarity measure mechanism
- ♦ Change legacy CBR situation model to a fact based situation model
- ♦ Create a fact to situations aggregator (explained below) to create situations that can be compared with the case base
- ♦ Create a fact based situation model to a flat description model converter since the latest version of jColibri available only supports flat description classes (no support for lists, maps and other collection based fields).

4.4.2 Service data

The first step required to exploit the CBR service is to define its *knowhow*. The *knowhow* for the CBR is the case base. The case base is composed of:

- ♦ A set of situation templates
- ♦ Global similarity measure + Local Similarity Measures
- ♦ Typical cases

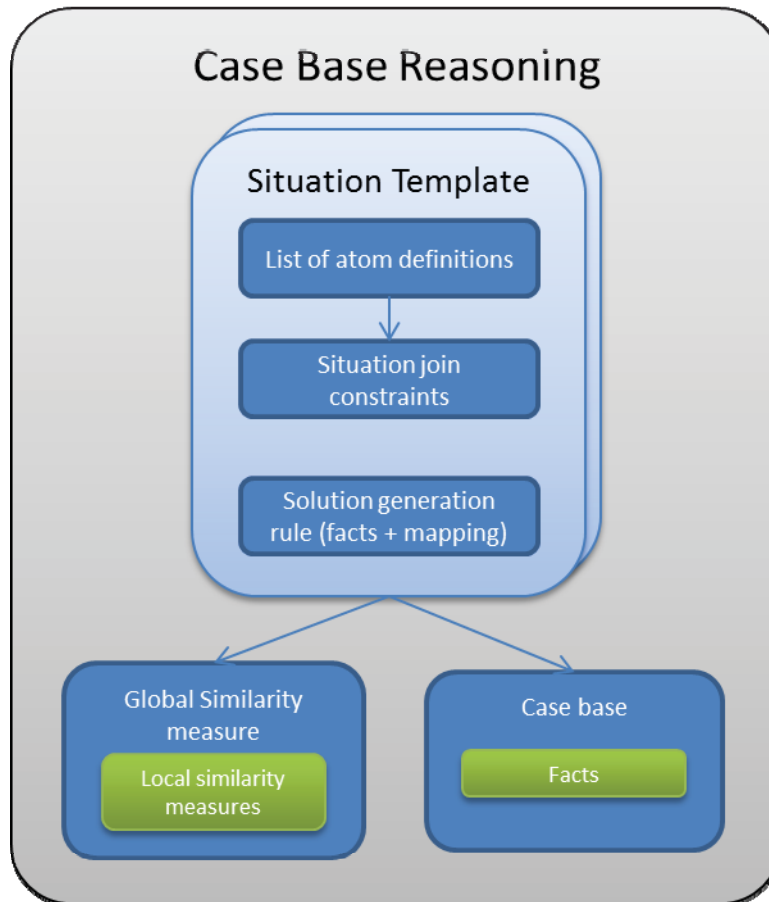


Figure 13: Case-Based Reasoner Data

The CBR will also require input facts as an input and parameters to define the similarity measures thresholds. These thresholds will be passed as parameters to the CBR.

4.4.2.1 Situation templates

A Situation template defines how the data must be aggregated to represent a situation. In the proof of concept CBR, this data was defined by a domain class and its members. Now, in the new CBR, since we are processing facts, a situation is represented by a list of atom definitions which are regrouped together with the help of situation join constraints. Situation join constraints define which argument of a fact must be equal to another fact argument so that two facts can be grouped together in single situation. For example:

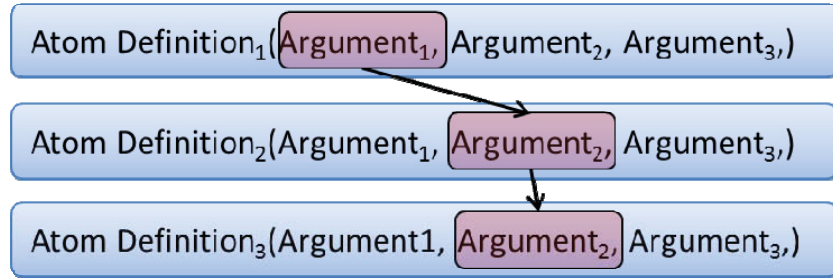


Figure 14: Case-Based Reasoner Join Constraint

The join constraints are defined by an atom definition index and an argument index.

4.4.2.2 Global similarity measure + Local Similarity Measures

The similarity measures define how a situation (facts) will be compared to the cases from the case base. There are two types of similarity measures:

1. Local similarity measures

Local similarity measures goal is to locally compare arguments of a situation. In the case of the MRI-CBR, the local similarity measure compares an argument of a situation with an argument of a case specifically. A local similarity measure should always return a value between 0 and 1 where 0 is the least similar and 1 the most similar. The different local similarity measures currently implemented in the system are:

- a. *Equal*
- b. *EqualsStringIgnoreCase*
- c. *InrecaLessIsBetter*
- d. *InrecaMoreIsBetter*
- e. *Interval*
- f. *MaxString*
- g. *Threshold*
- h. *ContextEqual*
- i. *ContextLessIsBetter*
- j. *ContextMoreIsBetter*
- k. *ContextInterval*

For more details about these local similarity measures, please consult the MRI SIDD document.

2. Global similarity measures

Global similarity measures goal is to calculate the global similarity measure of a situation compared to a case. In fact, it defines how local similarity measures must be compiled. For example, the “Average” global similarity measure adds all local similarity measure and divides their total by the number of local similarity measures which gives an average of the local similarity measures

values. The global similarity measures currently implemented in the system are:

- a. *Average*
- b. *Euclidean*
- c. *Frequency*

Here is an example of similarity measures definitions:

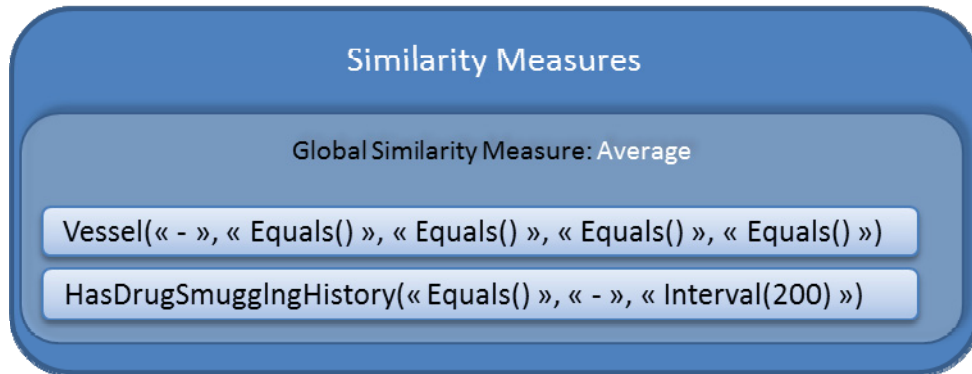


Figure 15: Similarity Measures

As specified, the global similarity measure used is “Average”, and we defined some local similarity measures for each argument. Some arguments are not compared (Those with the “-“ sign), some use the “Equal” local similarity measure and the last one use the “Interval” similarity measure. These measures will be used to compare the previously defined situations with the typical cases attached to the template.

4.4.2.3 Typical cases

The typical cases are the cases that have already occurred and for which we already associated a solution for. In the new CBR, cases are represented as a list of facts which respects the template.

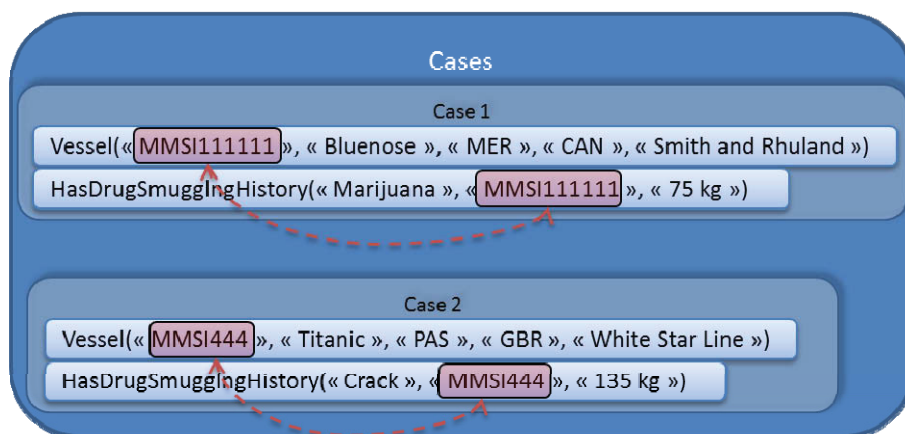


Figure 16: Cases

4.4.2.4 Solution generation rule

Cases of a case-based reasoner generally define a solution for each case description. However, due to the nature of the facts, this could hardly be achieved automatically without involving human intervention. In order to be able to automatically generate the conclusion for descriptions similar to the provided cases, a list of facts and mapping similar to RBR rule conclusions is rather used. This mapping is used to fill the solution fact(s) arguments with predefined values and/or with values taken from the situation description. Also, in order to generate conclusion automatically, the system receives a similarity threshold as an input and each time a situation similarity to a case is higher than this threshold, a conclusion will be generated from the similar situation.

4.4.3 Service dynamics

This section explains the workflow accomplished during the CBR specific processing:

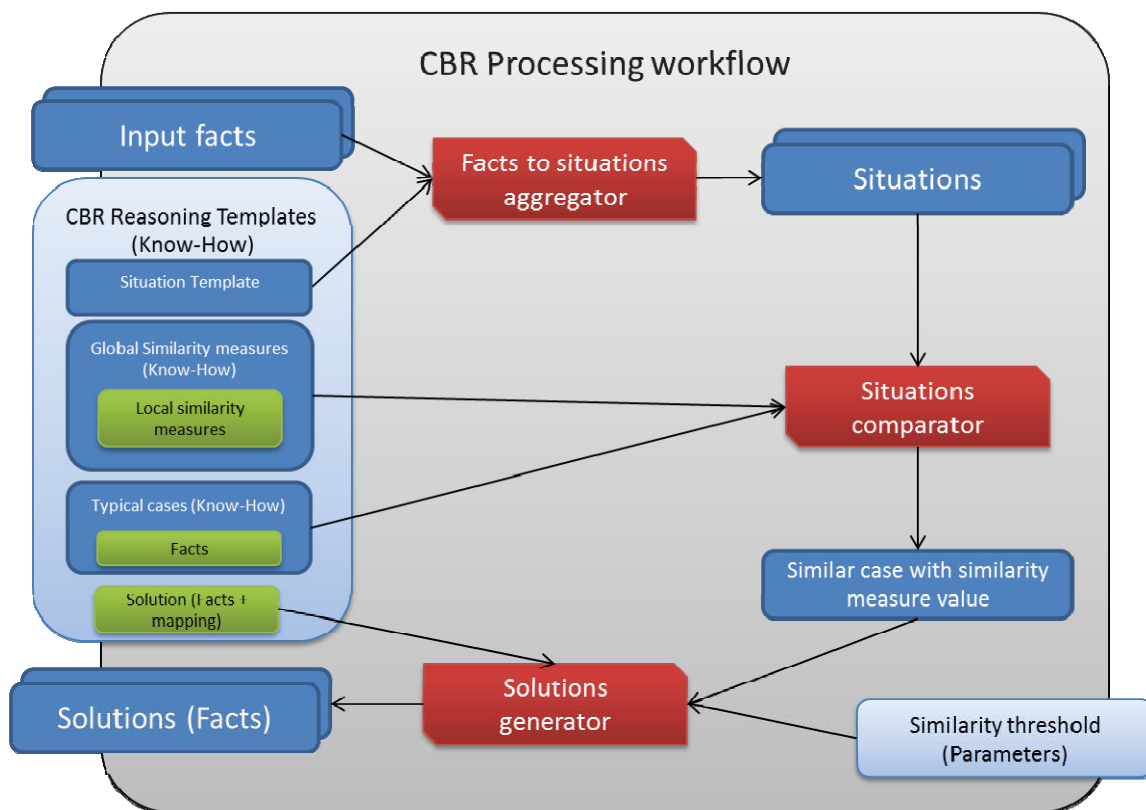


Figure 17: Case-Based Reasoner Processing Workflow

1. Input facts and situation templates are passed to the “Facts to Situations Aggregator”
2. The “Facts to Situations Aggregator” takes each templates and loops through the facts to create a new situation each time a set of facts matches a template. This

allows splitting input facts into as many situations as possible to be able to compare them “individually” to each situation description of the case base. When atom definitions and join conditions are respected, we match the facts together into a situation (Situation 1). When some facts respects the atom definition condition but does not have all join conditions matched, we match facts matching the join condition and leave blank the atom definitions where no matching facts could be found (Situation 2&3).

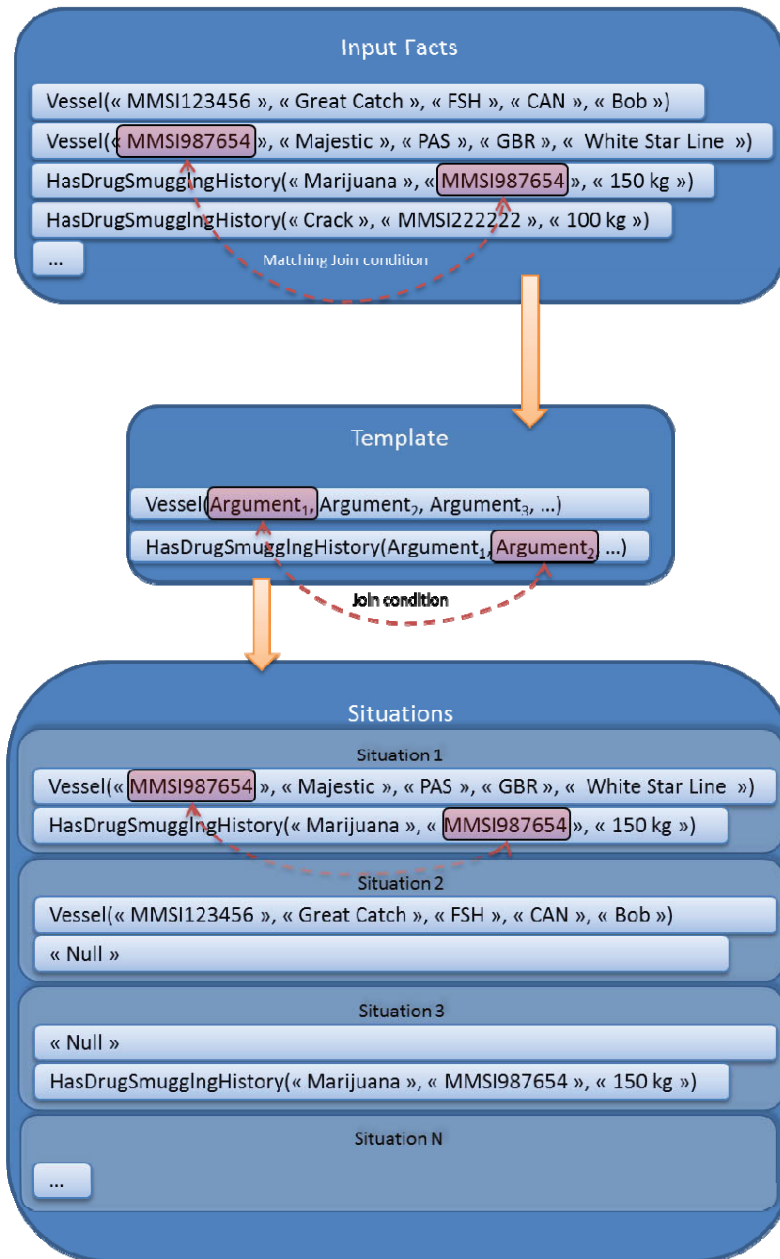


Figure 18: Fact to Situations Templates

3. Moreover, due to the jColibri implementation limitations (the core reasoner doesn't support collections in situation descriptions), the list of facts constituting a situation has to be "flattened" into a single description object containing as many fields as required to store each fact argument value. Therefore, the "Facts to Situations Aggregator" also perform this conversion prior to sending the situations to the "Situations Comparator".
4. The created situations are passed to the "Situations Comparator".
5. The "Situations Comparator" uses the similarity measures associated to the templates that generated each situation to evaluate the similarity between these situations and the typical cases of the case base and assigns a similarity measure value to these comparisons.

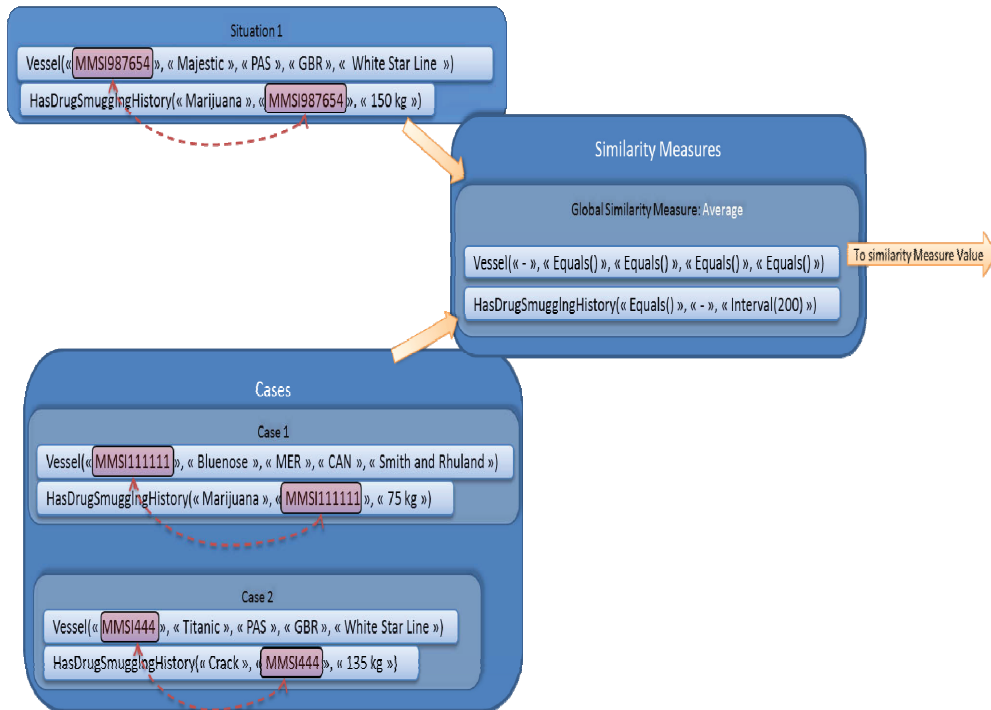


Figure 19: Situation and Cases Similarity Processing

6. Based on the similarity threshold values and the conclusion mappings specified in the parameters, the "Solutions generator" will create new facts following these steps:
 - a. The local similarity measures are computed, and then the global similarity measure outputs a similarity measure value.
 - b. Then the situation vs. case similarity measure is compared to the similarity threshold.
 - c. For each similar situation/case with a similarity higher than the threshold, a conclusion is generated.

- d. The conclusions are generated based on the argument references values specified (Note that arguments of the conclusion can also be literals/hard-coded-values). The conclusion values are extracted from the situation (not the case) and copied to the conclusion.

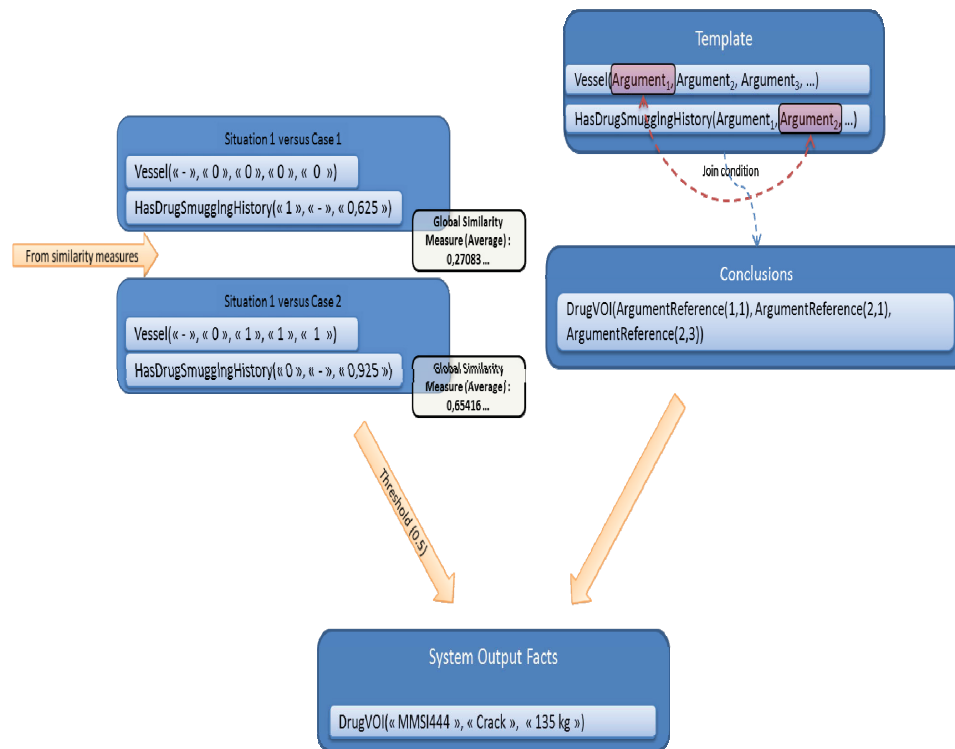


Figure 20: Case-Based Reasoner Output

7. The conclusions are returned to the caller! CBR reasoning is done!

4.5 Descriptive Logic Reasoner (DLR)

4.5.1 Description

In the scope of this project, the descriptive logic reasoner goal is able to do generalization of the knowledge domain concepts based on an ontology representation of the domain and to infer new relations between individuals.

Based on the properties of an individual, it can infer new relations between this individual and other classes of the ontology. These relations can then be extracted as facts and be used to deduce new things. To do so, the DLR uses the Pellet engine to which it passes an ontology describing the domain. It then inserts new individual with their properties and based on their properties, the Pellet engine can then classify these individuals within the ontology and deduce new generalization relations.

4.5.2 Service data

To work properly, the Descriptive Logic Reasoner requires:

1. An ontology representing the knowledge domain
2. (Optionally) Triplet URI mappings: The triplet URI mapping can be used to “translate” system triplet URIs references into a domain specific URIs
3. Input facts

4.5.2.1 URI Mappings

The URI mappings defines the mapping between a system source URI and a domain specific URI. These mapping are used in the system to domain specific triplet conversion which occurs right before the owl inference round.

At that moment, any URI specified in a triplet (in the object, attribute or value field) that matches a source URI specified in a mapping is converted to the matching target URI.

They can also be used after the inference round if the user configured the service to convert domain specific URIs back to system URIs as specified in the DLR service parameters.

4.5.2.2 Ontology References

To improve the system flexibility and usability, the ontology passed to the Descriptive Logic Reasoner can be referenced under three (3) different formats. The ontology content can be:

- Passed directly as a byte array
- Streamed from an URL
- Fetched from the Situational Ontology Management service by passing the ontology URI

4.5.3 Service dynamics

The general service dynamic of the DLR service is common to other reasoners. Please refer to the common [Service dynamics](#) section for more information. Only the internal execution mechanism differs from other reasoners.

The Descriptive Logic Reasoner specific execution follows the following workflow:

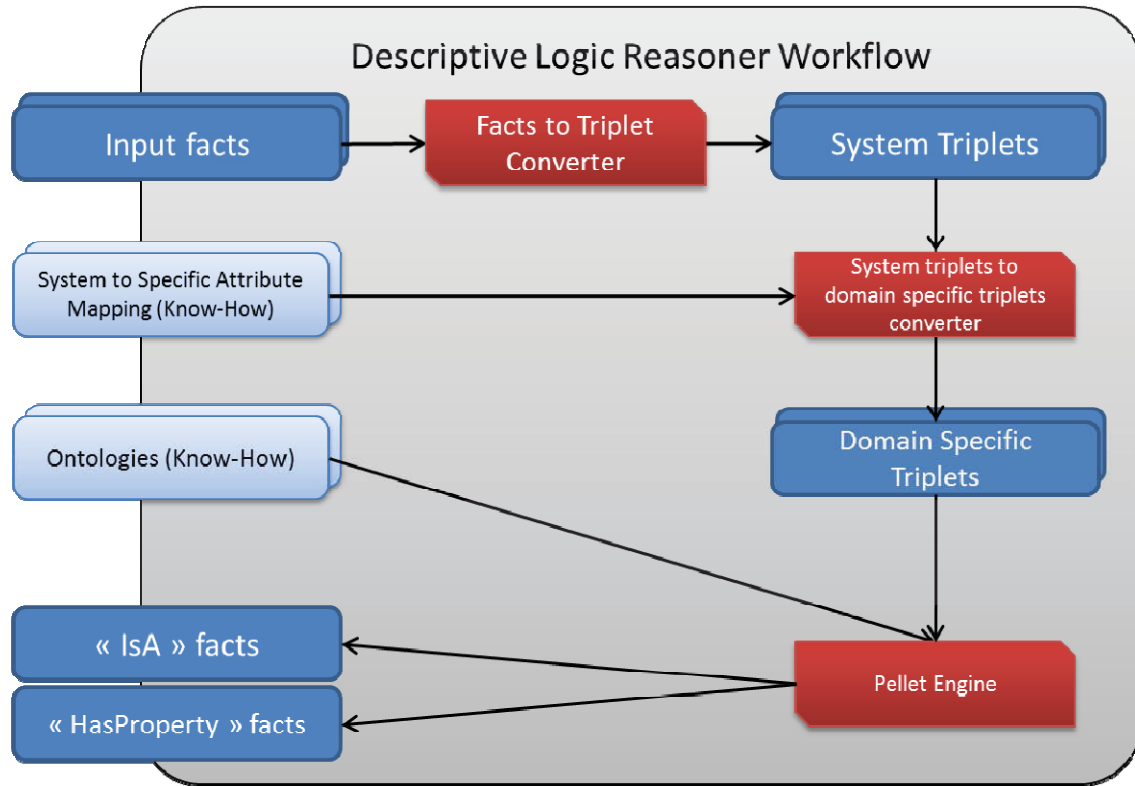


Figure 21: Descriptive Logic Reasoner Workflow

1. The facts are converted into system triplets as defined by the triplet mapping attached to their atom definition. Please refer to the [Facts to system triplets](#) section for more information on how triplet mappings are defined and how facts are then converted into triplets.
2. The system triplets can then optionally be converted to domain specific triplets as specified in the *knowhow*

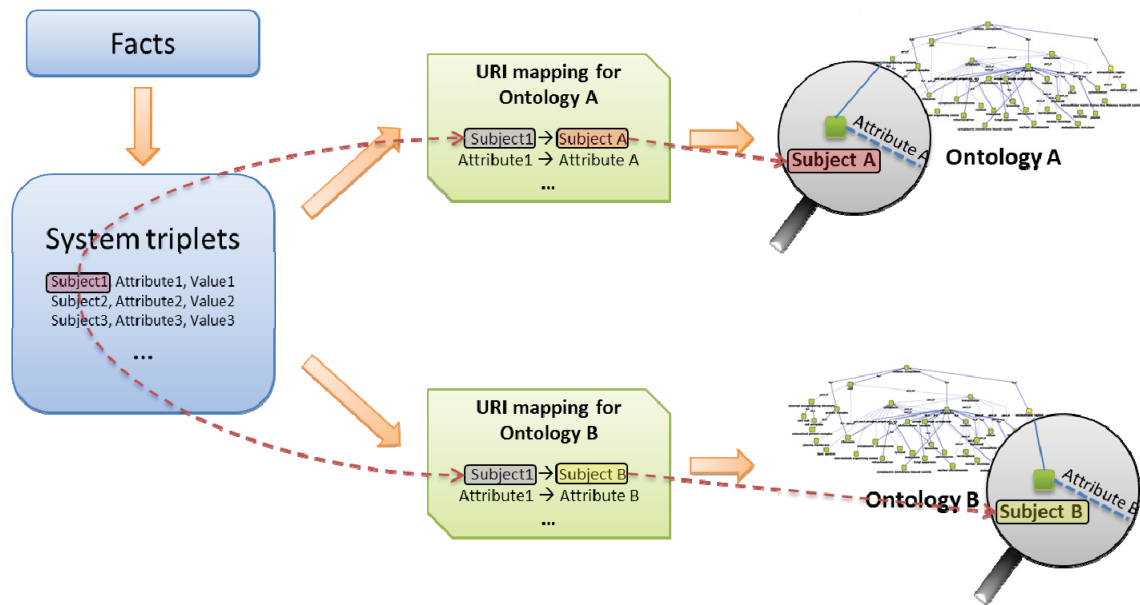


Figure 22: URI Mappings

- This step allows using predefined system triplets and “contextualizing” them to the ontology of interest without having to modify it.
 - To do so we iterate through the triplets and compare the subjects, attributes and values to see if they match one of the URI mapping. If they do, the matching fields are converted to the new URI.
 - For the subject, attribute and value fields, the value can be a *String* or an *OntologyEntity* which contains an *ontologyUri* field defining the entity or object property URI. If the value is a *String* it is directly converted, if it is an *OntologyEntity* only the *ontologyUri* field is converted.
 - Please note that any triplet component (subject, attribute or value) that is not matching an URI mapping will be left “as-is”. Therefore, if the ontology exactly matches the system triplets, there is no need to provide mapping for them.
3. The ontology is then passed to the Pellet engine and the converted triplets are added to the ontology by performing the following steps for each triplet:
- If the triplet subject is not typed as an ontology entity reference, the triplet is discarded.
 - If the subject is defined as a class, the corresponding class is searched in the ontology. If the ontology doesn’t contain a class having the same URI, a new class will be created and added at the root of the ontology. Therefore, its only parent class will be `owl:Thing`.
 - i. If the triplet attribute is *IS-A* (`http://ca.gc.rddc/ontology/attributes.owl#isA`) and the triplet value is

- of type ontology entity reference, the value is loaded as another subject (like the current triplet subject). If it turns out to be a class (either existing or new one), the sub-class relation is created between our initial subject class and the latter class.
- ii. Otherwise (the triplet attribute is not *IS-A*), the triplet attribute is simply discarded.
- If the subject is defined as an individual reference, the corresponding individual is searched in the ontology. If the ontology doesn't contain an individual having the same URI, a new individual is created for this URI and added to the root of the ontology. Therefore, the individual will only be of type owl:Thing for the moment.
 - i. If the triplet value is of type ontology class reference and the triplet attribute is *IS-A* (<http://ca.gc.rddc/ontology/attributes.owl#isA>), the value is loaded as another subject (like the current triplet subject). Then, an assertion is added to the ontology to set our subject individual as a direct child of the triplet value.
 - ii. If the triplet value is of type ontology class reference, we try to find an object property having the same URI than the triplet attribute.
 - 1. If a matching object property definition is found, a new object property is created between the subject individual and the value individual;
 - 2. Otherwise, we don't create a new object property definition since it will not help realizing the individual. The triplet is simply discarded.
 - iii. Otherwise, we try to find a datatype property having the same URI than the triplet attribute.
 - 1. If a matching datatype property definition is found, a new datatype property is created between the subject individual and the triplet literal value;
 - 2. Otherwise, we don't create a new datatype property definition since it will not help realizing the individual. The triplet is simply discarded.
4. The Pellet engine generalizes the ontology related instances and classes and infers the instance generalizations depending on their properties.

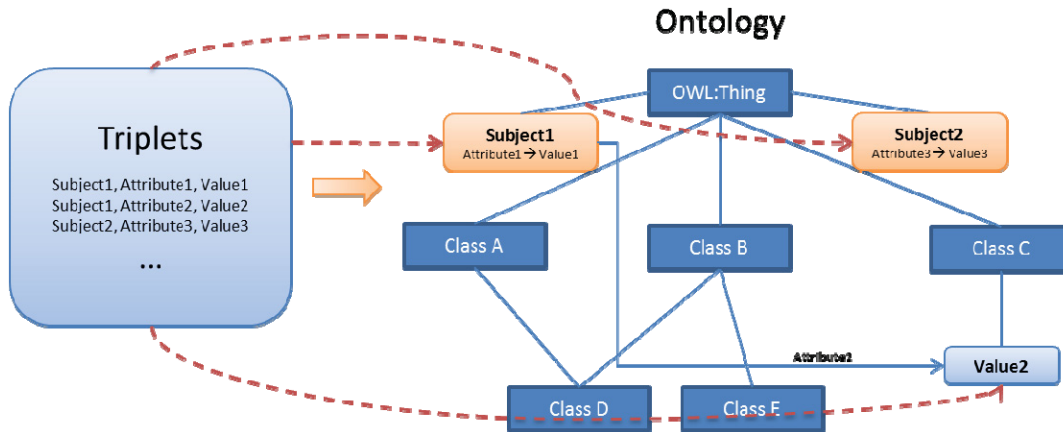


Figure 23: Triplets Insertion

- At first, the new entities have been added directly under OWL:Thing and existing ones have been reused to create initial object properties, datatype properties and sub-class relations that are already known as mentioned in the previous step;

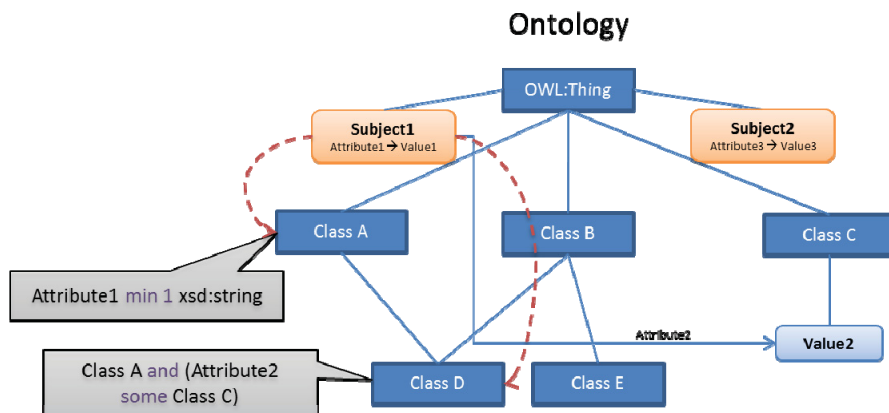


Figure 24: Pellet categorization

- Then, based on these properties and class definitions, Pellet will realize individuals automatically. However, in order to do so, the ontology classes must be defined using property restrictions to maximize individual realization.
5. The inferred generalizations and new properties (datatype or object) are then converted to facts.
 6. Facts URIs can optionally be converted back to system URIs.

4.6 Multi-Reasoners Orchestrator

4.6.1 Description

The Multi-Reasoner orchestrator is a wrapper over the different reasoners of the MRI system. In fact, its goal is to route the “KnowHows”, parameters and facts it receives in input to the right reasoners and invoke these reasoners. Then, it collects the facts inferred by those reasoners and passes them to the other reasoners to achieve a multi-reasoner inference capability. All facts inferred by the reasoners are kept within the MRI context. Once the reasoners are not inferring any more facts, the MRI status is set to “Completed” and the results can then be fetched by the caller.

4.6.2 Service data

To work properly, the MRI Orchestrator requires:

1. A list of reasoner configurations (A pair of KnowHow and Parameters)
2. Input facts

4.6.2.1 Reasoner configuration

The list of reasoner configuration required by the orchestrator is in fact a list of what is required by each reasoner separately, i.e. the “Know How” of a reasoner and its parameters.

Consult each reasoner section to see what is required to call reasoners from the MRI.

4.6.3 Service dynamics

The general service dynamic of the MRI orchestrator service is common to other individual reasoners. Please refer to the common [Service dynamics](#) section for more information. Only the internal execution mechanism differs from other reasoners.

The MRI Orchestrator specific execution follows the following workflow:

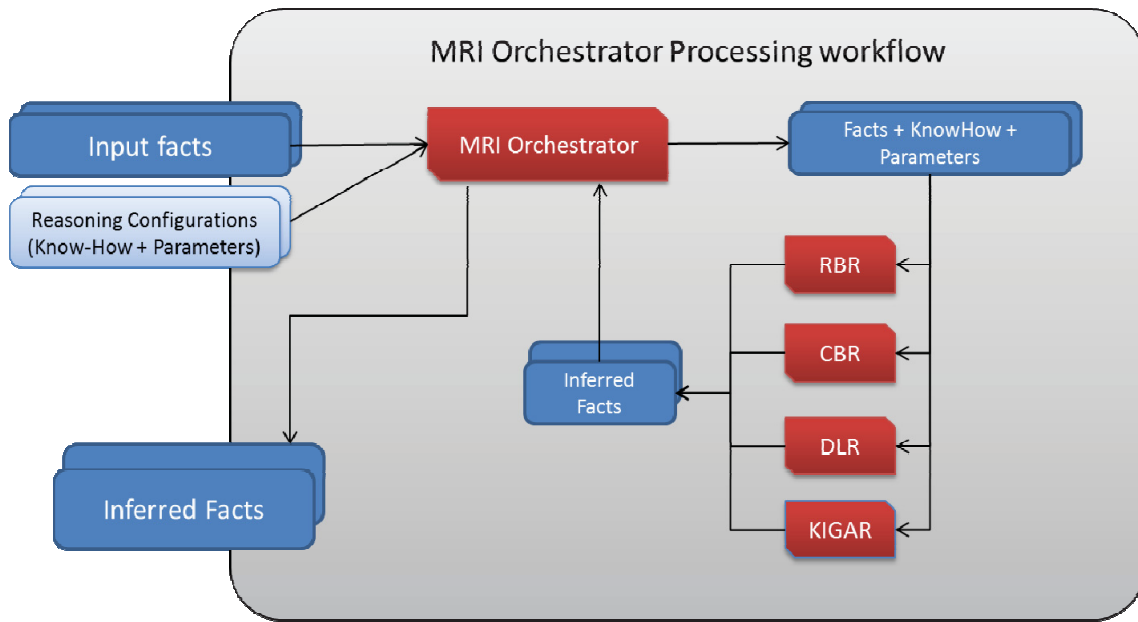


Figure 25: MRI Orchestrator Processing Workflow

1. The client calls the MRI Orchestrator and passes a list of “KnowHows” and “Parameters” of the reasoners he wants to infer with.
2. The orchestrator then invokes each reasoners separately based on the received “KnowHows”.
3. Once a reasoner has finished inferring, it sends the results back to the Orchestrator.
4. The orchestrator then forwards these new facts to the other reasoners to see if they can deduce new facts from these inferred facts.
5. The steps 2 to 4 are repeated until each reasoner cannot infer new facts.
6. Then the results can be fetched by the client from the orchestrator which holds a copy of all inferred facts.

5 References

- [1] **Martineau, Étienne.** *Iterative sub-setting*. Québec : DRDC Valcartier, 2010.
- [2] **Dorion, Éric; Bergeron Guyard, Alexandre.** *Measures of Similarity for Command and Control Situation Analysis*. Québec: DRDC Valcartier, 2011.
- [3] **Roy, Jean.** Automated Reasoning Services – High-Level Concepts. Québec: DRDC Valcartier, 2011
- [4] **Roy, Jean.** Kinematics and Geospatial Analysis Reasoning (KIGAR) – High-Level Concepts. Québec: DRDC Valcartier, 2011
- [5] **Allard, Yannick.** *Kinematic and Geospatial Analysis Module (KIGAM) Analysis Fact Sheets*. Montréal: OODA Technologies Inc. (for DRDC Valcartier), 2011.

Annex A System Attributes

The following table lists the different system attributes available to use in the triplets mapping attributes and which will be used in the facts to spatial feature conversion.

Table 1 : System Attributes

Attribute ⁵	Unique Identifier
HAS_MOTION_TRAJECTORY <i>Used to attach a motion trajectory to a subject.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasMotionTrajectory
HAS_CONTACT <i>Used to attach a contact to a motion trajectory.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasContact
HAS_GEOMETRY <i>Used to attach a geometry to a subject, a motion trajectory or a contact.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasGeometry
HAS_ALTITUDE <i>Used to specify altitude of a contact.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasAltitude
HAS_LATITUDE <i>Used to specify latitude of a contact.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasLatitude
HAS_LONGITUDE <i>Used to specify longitude of a contact.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasLongitude
HAS_SPEED <i>Used to specify speed of a contact.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasSpeed
HAS_ORIENTATION <i>Used to specify orientation of a contact.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasOrientation
HAS_TIMESTAMP <i>Used to specify timestamp of a contact.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasTimestamp
HAS_DESTINATION <i>Used to specify destination of a motion trajectory.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasDestination
HAS_ETA <i>Used to specify estimated time of arrival of a motion trajectory.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasEstimatedTimeOfArrival
HAS_WIDTH <i>Used to specify width of a motion trajectory.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasWidth
HAS_MINIMUM_SPEED <i>Used to specify minimum speed of a zone.</i>	http://ca.gc.rddc/ontology/attributes.owl#hasMinimumSpeed
HAS_MAXIMUM_SPEED <i>Used to specify maximum speed of a zone</i>	http://ca.gc.rddc/ontology/attributes.owl#hasMaximumSpeed

⁵ As defined in the enum class “ca.gc.rddc.istip.sfm.data.enums.SystemAttribute”

The following table lists the signatures that must have triplet mapping using the known system attributes.

Attribute	Subject	Attribute	Value
IS_A <i>Used to specify that a subject is a subclass of an ontology class.</i>	<i>The subject id (Any ontology instance URI)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#isA</i>	<i>The subject type uri (URI of a class in any ontology)</i>
HAS_MOTION_TRAJECTORY <i>Used to attach a motion trajectory to a subject.</i>	<i>The subject id (Any ontology instance URI)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasMotionTrajectory</i>	<i>The motion trajectory id (Long)</i>
HAS_CONTACT <i>Used to attach a contact to a motion trajectory.</i>	<i>The motion trajectory id (Long)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasContact</i>	<i>The contact id (Long)</i>
HAS_GEOMETRY <i>Used to attach a geometry to a subject, a motion trajectory or a contact.</i>	<i>The subject id (Any ontology instance URI) or Motion Trajectory Id (Long) or Contact Id (Long)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasGeometry</i>	<i>The geometry (WKT String or a Geometry Object like a JTS geometry or ISTIP geometry))</i>
HAS_ALTITUDE <i>Used to specify altitude of a contact.</i>	<i>The contact id (Long)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasAltitude</i>	<i>The altitude (Double)</i>
HAS_LATITUDE <i>Used to specify latitude of a contact.</i>	<i>The contact id (Long)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasLatitude</i>	<i>The latitude (Double)</i>
HAS_LONGITUDE <i>Used to specify longitude of a contact.</i>	<i>The contact id (Long)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasLongitude</i>	<i>The longitude (Double)</i>
HAS_SPEED <i>Used to specify speed of a contact.</i>	<i>The contact id (Long)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasSpeed</i>	<i>The speed (Double)</i>
HAS_ORIENTATION <i>Used to specify orientation of a contact.</i>	<i>The contact id (Long)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasOrientation</i>	<i>The orientation (Double)</i>
HAS_TIMESTAMP <i>Used to specify timestamp of a contact.</i>	<i>The contact id (Long)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasTimestamp</i>	<i>The timestamp (Date)</i>
HAS_DESTINATION <i>Used to specify destination of a motion trajectory.</i>	<i>The Motion Trajectory Id (Long)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasDestination</i>	<i>The destination (String)</i>
HAS_ETA <i>Used to specify estimated time of arrival of a motion trajectory.</i>	<i>The Motion Trajectory Id (Long)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasEstimatedTimeOfArrival</i>	<i>The estimated time of arrival (Long, Date or String)</i>
HAS_WIDTH <i>Used to specify width of a motion trajectory.</i>	<i>The subject id (Any ontology instance URI) or The Motion Trajectory Id (Long)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasWidth</i>	<i>The width of the subject or motion trajectory (Double)</i>
HAS_MINIMUM_SPEED <i>Used to specify minimum speed of a subject (zone).</i>	<i>The subject id (Any ontology instance URI)</i>	<i>http://ca.gc.rddc/ontology/attributes.owl#hasMinimumSpeed</i>	<i>The minimum speed (Double)</i>

HAS_MAXIMUM_SPEED <i>Used to specify maximum speed of a subject (zone)</i>	<i>The subject id (Any ontology instance URI)</i>	<i>http://ca.gc.rddc/ontology/attributes.o wl#hasMaximumSpeed</i>	<i>The maximum speed (Double)</i>
--	---	---	-----------------------------------

This page intentionally left blank.

List of symbols/abbreviations/acronyms/initialisms

CBR	Case-Based Reasoner
DLR	Descriptive Logic Reasoner
ISTIP	Intelligence Science & Technology Integration Platform
KIGAR	KInematic and Geospatial Analysis Reasoner
MITS	Multi-Intelligence Tools Suite
MRI	Multi-Reasoners Inference
RBR	Rule-Based Reasoner
SFM	Situational Facts Management
SSO	Single Signed-On
VOiLA	Visionary Overarching Interaction Interface Layer for the Analyst

This page intentionally left blank.

DOCUMENT CONTROL DATA		
(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)		
1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.) Fujitsu Consulting (Canada) Inc. 2000 Boulevard Lebourgneuf Bureau 300 Québec (Québec) G2K 0E8	2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.) UNCLASSIFIED (NON-CONTROLLED GOODS) DMC A REVIEW: JUNE 2010	
3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.) Multi-Reasoner Inference: Software Architecture Document		
4. AUTHORS (last name, followed by initials – ranks, titles, etc. not to be used) Morin-Brassard G.; Giroux V.		
5. DATE OF PUBLICATION (Month and year of publication of document.) January 2012	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.) 54	6b. NO. OF REFS (Total cited in document.) 5
7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.) Contract Report		
8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.) Defence R&D Canada – Valcartier 2459 Pie-XI Blvd North Quebec (Quebec) G3J 1X5 Canada		
9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.) w7701-10-4064	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)	
10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.) MRI-242-0449	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.) DRDC Valcartier CR 2012-004	
11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.) Unlimited		
12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.) Unlimited		

13. **ABSTRACT** (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

To support its research activities in the intelligence domain, the Intelligence and Information (I&I) Section at DRDC Valcartier is developing the Intelligence Science & Technology Platform (ISTIP) as a major component of its R&D infrastructures. To improve the reasoning capabilities of the platform, the mandate of this contract is to produce a Multi-Reasoner Inference (MRI) capability based on the Multi-Intelligence Tool Suite (MITS) and the ISTIP software components previously developed by the I&I Section. Five main different services have been developed containing four individual reasoners and one multi-reasoner orchestrator. The reasoners that have been created are a Case-Based Reasoner (CBR), a Rule-Based Reasoner (RBR), a Descriptive-Logic Reasoner (DLR) and a KInematics and Geospatial Analysis Reasoner (KIGAR) which is based on the KIGAM module of the Inference of Situational Facts through Automated Reasoning (ISFAR) tool. Through the use of a common reasoning framework, these reasoners can now leverage their reasoning capabilities by sharing their strength to other reasoners and achieve an amazing synergy. This document describes the Software Architecture of the MRI.

Afin de supporter ces activités de recherche dans le domaine du renseignement, la Section du Renseignement et Information de RDDC Valcartier développe la Plate-forme de Science et Technologie du Renseignement (ISTIP) comme un composant majeur de ses infrastructures de R&D. Afin d'améliorer les aptitudes de raisonnement de la plate-forme, le mandat de ce contrat est de créer un outil d'inférence Multi-Raisonneur (MRI) basé sur la « Multi-Intelligence Tool Suite » (MITS) et sur les composants logiciels déjà implémentés par la section I&I. Cinq différents services ont été développés comprenant quatre raisonneurs individuels et un orchestrateur multi-raisonneur. Les raisonneurs qui ont été créés sont un raisonneur par cas (CBR), un raisonneur par règles (RBR), un raisonneur ontologique (DLR) et un raisonneur d'analyse cinématique et géo-spatiale (KIGAR) basé sur le module KIGAM de l'outil d'Inférence Automatisée de Faits Situationnels (ISFAR). Grâce à l'utilisation d'un cadre de raisonnement commun, ces raisonneurs peuvent désormais exploiter leurs capacités de raisonnement en partageant leurs forces d'autres raisonneurs et parvenir à une synergie époustouflante. Ce document décrit l'Architecture Logicielle du MRI.

14. **KEYWORDS, DESCRIPTORS or IDENTIFIERS** (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Reasoner, Inference

Defence R&D Canada

Canada's Leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
De science et de technologie pour
la défense et la sécurité nationale



www.drdc-rddc.gc.ca

