Project Galaxy: Model-Driven Collaborative development of complex systems

Deliverable D2.3 :Mechanisms for Model-Driven Team Communication

	NAME	PARTNER
WRITTEN BY	A. Kamoun	LAAS (IRIT subcontractor)
	G. Sancho	LAAS (IRIT subcontractor)
REVIEWED BY	S. Tazi	LAAS
	B. Coulette	IRIT
	K. Drira	LAAS
	E. Kedji	IRIT

1 Introduction

This document presents the third deliverable of the second work package of the project. It describes the Generic Collaboration Ontology (GCO) [STV10][STV10] G. Sancho, S. Tazi et T. Villemur : GCO : a Generic Collaboration Ontology. In Proceedings of the Fourth International Conference on Advances in Semantic Processing (SEMPARO 2010), Florence, Italy, octobre 2010.

[SVT10] . The main goal of this model is to serve as reference point in order to express collaboration situations between users organized in groups.

This model is represented in the Web Ontology Language (OWL[1]) ontology language. As far as we know, a common ontology for modeling collaborative sessions has not been proposed yet. Ontologies have received great attention in the recent years, due to their use for knowledge representation in the Semantic Web domain. The main idea is to add metadata describing regular Web data (which is only human-readable) in order to make it understandable by machines enabling the automation of distributed processing over the Web. This metadata represents the semantics of collaboration.

GCO has been designed in a manner that it is independent of specific domain. However it can be specified in order to capture the specific collaboration knowledge of the considered domain. The main objective of GCO is to represent collaboration in a conceptual manner. This enables its use to be specified to a specific domain. The second goal of GCO is to serve as a core for the deduction and the expression of a deployment schema that corresponds to a given collaboration configuration.

We distinguish two types of ontologies: "top-level" ontologies and "domain" ontologies. (1) "Top-level" ontologies, describe general concepts that are reusable through different domains. They may be considered as meta-model. (2) "domain" ontologies specifying a conceptualization of a part of the real world of a specific domain. Domain ontologies may be considered as instances of “top-level” ontologies.

The GCO ontology is a top-level ontology. Concepts and relations defined in GCO are specialized into specific concepts and relations of the domain ontology. In order to

[1] Although the acronym should be WOL, OWL has been chosen for aesthetic reasons: http://lists.w3.org/Archives/Public/www-webont-wg/2001Dec/0162.html

apply GCO to model driven design by a group in the case of GALAXY, the domain ontology is a set of concepts, properties and relations that represent Galaxy domain. The Galaxy ontology specializes the GCO ontology.

The contents of this document are organized as follows. In section 2, we introduce the notions of collaboration and session that are the core of GCO. Section 3 details the elements of the GCO. Section 4 explains the principles that have guided its design. Section 5 presents some guidelines for using an ontology as the core model of a run-time system. Section 6 explains how GCO can be used as a model of collaboration in a specific domain and we illustrate this application by an example of specialization of GCO in the context of Galaxy. Section 7 concludes and provides some perspectives for future work. In the appendix, we introduce the technologies and languages that we have used for the expression and the processing of GCO namely OWL for ontologies, the Semantic Web Rule language (SWRL) for rules and reasoning mechanisms in OWL. These Semantic Web technologies allow the representation and management of knowledge.

2 Collaboration

Collaborative applications are distributed systems especially designed to provide support to groups of users that act in a coordinated way in order to achieve a common goal. Such applications have been studied since the 1990s in the domain called Computer-Supported Collaborative Work (CSCW). Kraemer [KK88] and Ellis [EGR91] proposed two general definitions of the collaborative work:

"computer-based system that facilitates the solution of unstructured problems by a set of decision makers working together as a group."

"computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment."

In these definitions, the term work, in general, refers to any common task between several persons, in domains such as game, education, co-design, etc. The developed techniques in the domain of collaborative work can be applied to any kind of human computer collaboration.

The collaborative work has four reference domains [vil06] :

- Social sciences (more specifically, sociology and the organization theory) in order to study the organization of groups, their reports, the group efficiency, etc.

- Cognitive sciences and distributed artificial intelligence in order to study the semantic of information, tasks planning, assistance in performing these tasks, etc.
- Human-machine interfaces for designing multi-user interfaces adapted to the collective work.
- Distributed computing for the design of distributed systems that enable the storage, exchange and processing of information remotely.

The groupware concept refers to the set of software products, services, platforms and tools that support collaborative work [Kar94] .

2.1. Sessions

The concept of session is crucial in the collaborative work. A session consists of a set of users who share common goals [DGLA99] . Those who participate in a session should not be necessarily at the same place; the use of networks allows the intervention of geographically distant participants.

Sessions can be synchronous or asynchronous. In a synchronous session, all participants are presents simultaneously. Exchanges between these participants are interactive, and data are manipulated in real time, e.g. a group of people participating in a videoconference.

In an asynchronous session, the co-presence of group members is not necessary. Exchanges are not in real time, because they are based on asynchronous media such as email.

This distinction between synchronous and asynchronous sessions is used in the past due to the different network technologies. Currently, we find tools that combine the two modes: For example, in a collaborative editing of a document, authors may work separately asynchronously, and with some meeting in a synchronous mode, to ensure the consistency of the produced document.

Sessions are classified into two categories: explicit and implicit. A session is called explicit when its possible configurations are set offline during the system design. The designer explicitly defines the relationships between group members and their evolution over time. Session instances are managed and deployed at run time. In most case, a privileged user initiates the session and other users can join it if they are invited. Most of proposed models for the formalization of synchronous sessions are based on graphs [RPVD01]. In these graphs, nodes represent users, while edges represent the exchanged flow of data. The labels of the edges indicate the tool that manages the sending and receiving data.

Implicit sessions emerge from user's actions and their context. When the system detects situations of potential collaboration, for example according to the presence of users and their interests, it creates an implicit session and invites users to join it. Few studies have investigated this type of sessions. However, we cite the work of Edwards [EDW94] and Texier and Plouzeau [TP03] , which propose models based on the set theory, and that of Rusinkiewicz and al. [RKT95], based on first order logic for describing the structure of session which is not fixed a priori.

2.2. Sessions management and deployment

In collaborative tools, models of sessions are used by session managers who control the life cycle of sessions. They are used to identify sessions, to activate, to control user's access and their rights, to enable the necessary tools, etc...

An important aspect is the deployment of tools and components managing the flow of data sent between users. Indeed, it is necessary to install and configure these elements on users' machines so they can exchange data.

Hammami [Ham07] made a comprehensive study of the types of deployment and deployment systems that exist. The deployment can be static (when an administrator indicates the application to use) or dynamic (when the choice is automatic during the deployment process), centralized (with a main entity that manages the process) or decentralized (when deployment nodes interact with each other). There are two deployment strategies: push, in which the initiative of deployment is given to an administrator, and pull, in which the nodes initialize the deployment process themselves. In general, the systems found in the literature implement a static deployment, in push mode, and often centralized. Automatic systems have been little studied, and in general, they are very flexible.

3 Description of the GCO

The main elements of the Generic Collaboration Ontology are represented in Figure 4.1. Concepts are represented as round-cornered rectangles, while relations are represented as arrows going from one concept (the domain of the relation) to another concept (the range of the relation). Relations are marked with cardinalities that allow to distinguish between functional and not functional properties. Individuals are represented as dash-line rectangles.

The basic concept of this ontology is *Node*. A node represents a communicating entity which takes part in a collaborative activity.

Nodes play a role in the collaborative activity which determines the position of the entity within the collaborative group. This is captured by the concept *Role*. Therefore a relation called *hasRole* links the Node and Role concepts. This relation is not functional because one node may have many roles.

Groups are represented by the concept: *Group*. The membership of roles to groups is expressed by the relation *hasMember* (going from *Group* to *Role*). Its opposite is *belongsToGroup.*

A node represents a participant who collaborates with others. This participant uses a physical machine. Such machines are represented by the concept *Device,* and *Node* is linked to *Device* by the property *hasHostingDevice*. The inverse property is *called HasHostedNode*. The device identifier can represent for example its IP address.

Entities take part in the collaborative activity by sending and receiving data to/from other entities. The concept *Flow* represents a communication link between two entities. Therefore, *Flow* is linked to *Node* by two properties: *hasSource* and *hasDestination*. In this ontology, flows are considered as being unidirectional, and thus if a bidirectional communication between two nodes is required, it will be represented by two instances of *Flow* with two opposite directions. The *hasSource* property is functional, while *hasDestination* is not functional, i.e., a flow has a single source node, but it may have several destination nodes (thus representing multicast links).

The Session concept represents a set of flows belonging to the same collaborative activity. The *hasFlow* property relates a session to a flow. The inverse property, *belongsToSession*, is functional, i.e., a flow belongs to a single session. Since flows are related to nodes, nodes are indirectly related to one or more sessions depending on the flows that connect them to other entities.

In order to handle data flows, nodes use external software components that are deployed on the same device than them. This enables the separation between business code (specific to application domaine and implemented in entities' components) and collaboration code (implemented in such external components). These external components are represented by the *Tool* concept. The tool is software that allows sending and receiving data flows. They are composed of several components, e.g., a sender component and a receiver component. Tools are managed by nodes; components are "subparts", fragments of software, that's why there is a relationship between them.

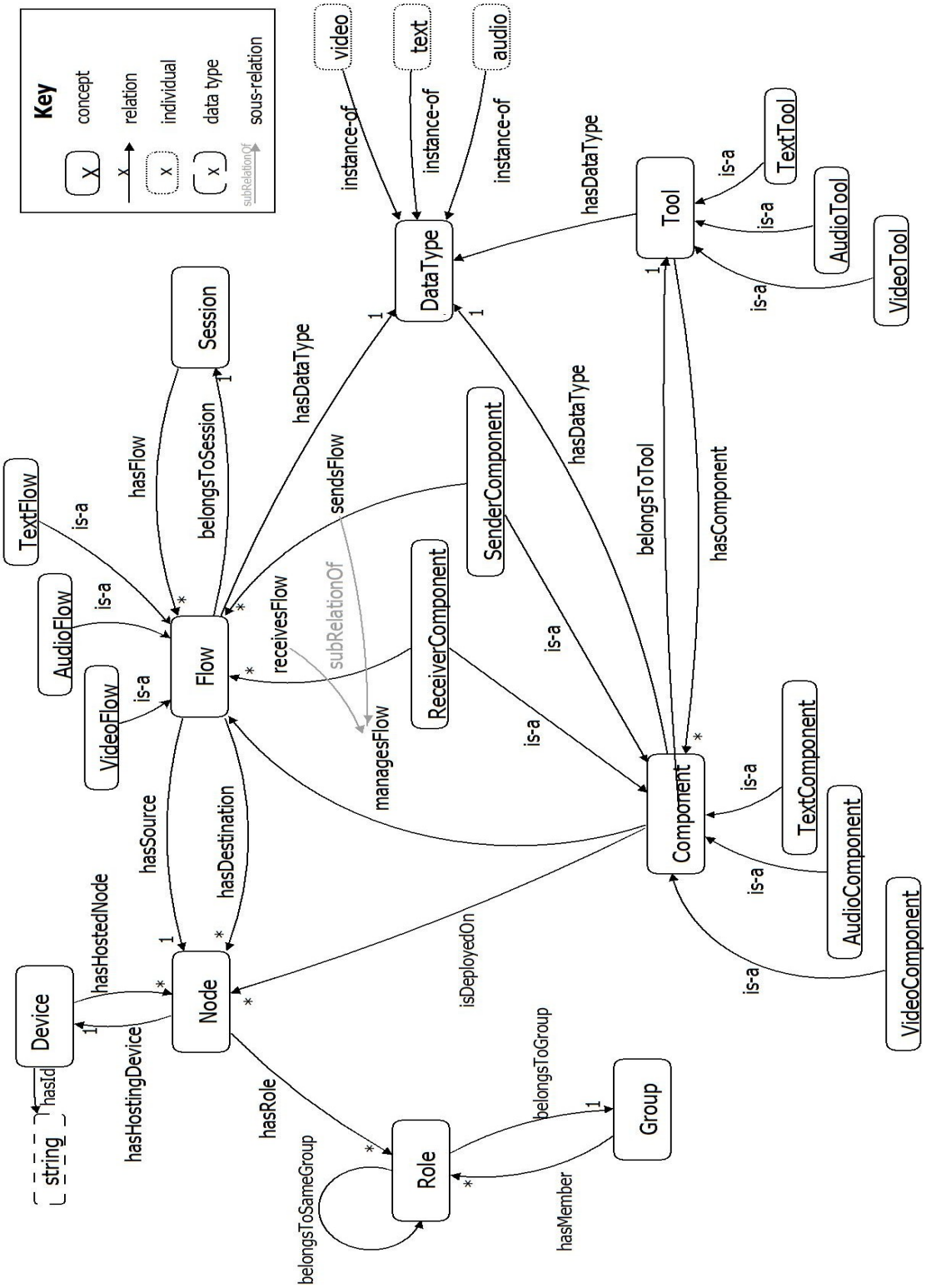

Fig. 4.1 Main concepts and relations of GCO

Therefore the *Tool* concept is related to a concept called *Component* through the property *hasComponent*. Since components handle flows, a property called *managesFlow* links *Component* and *Flow*. Components have a data type (the same as the data type of the flow that they manage) and are deployed (*isDeployedOn* property) on a single node (thus, they are deployed on the device that hosts the node). The Component concept has several subconcepts that represent components depending on

the handled data type (*AudioComponent*, *TextComponent* and *VideoComponent*) and on the direction of the handled flow (*SenderComponent* and *ReceiverComponent*). *SenderComponent* and *ReceiverComponent* are linked to *Flow* by two sub-relations of *managesFlow*: *sendsFlow* and *receivesFlow*, respectively.

In order to represent the nature of data exchanged through a flow, the *Flow* concept has a functional property called *hasDataType* that relates it to the *DataType* concept. Possible values of data types are captured through the *DataType* individuals audio, text and video (additional data types could be considered). The subconcepts of *Flow* differ in the value of their data type: *AudioFlow*, *TextFlow* and *VideoFlow*. Flow, Tool and Component Classes have three defined subclasses depending on the data type. For example, the class AudioComponent is defined as :

$$AudioComponent \equiv Component \prod hasDataType(audio)$$

This means that if an individual belongs to the *AudioComponent* class, then it must be a *Component* and its *hasDataType* property point towards the audio individual. And conversely, every individual being a component and having audio as data type is necessarily an *AudioComponent*. We used the same principle for the other subclasses of *Component* and for those of *Flow* and *Tool*.

The Component subclasses taxonomy contains all variants according to the data type (audio, text and video) and the component direction (sending, reception). Figure 4.2 details this taxonomy as it is represented in the ontology.

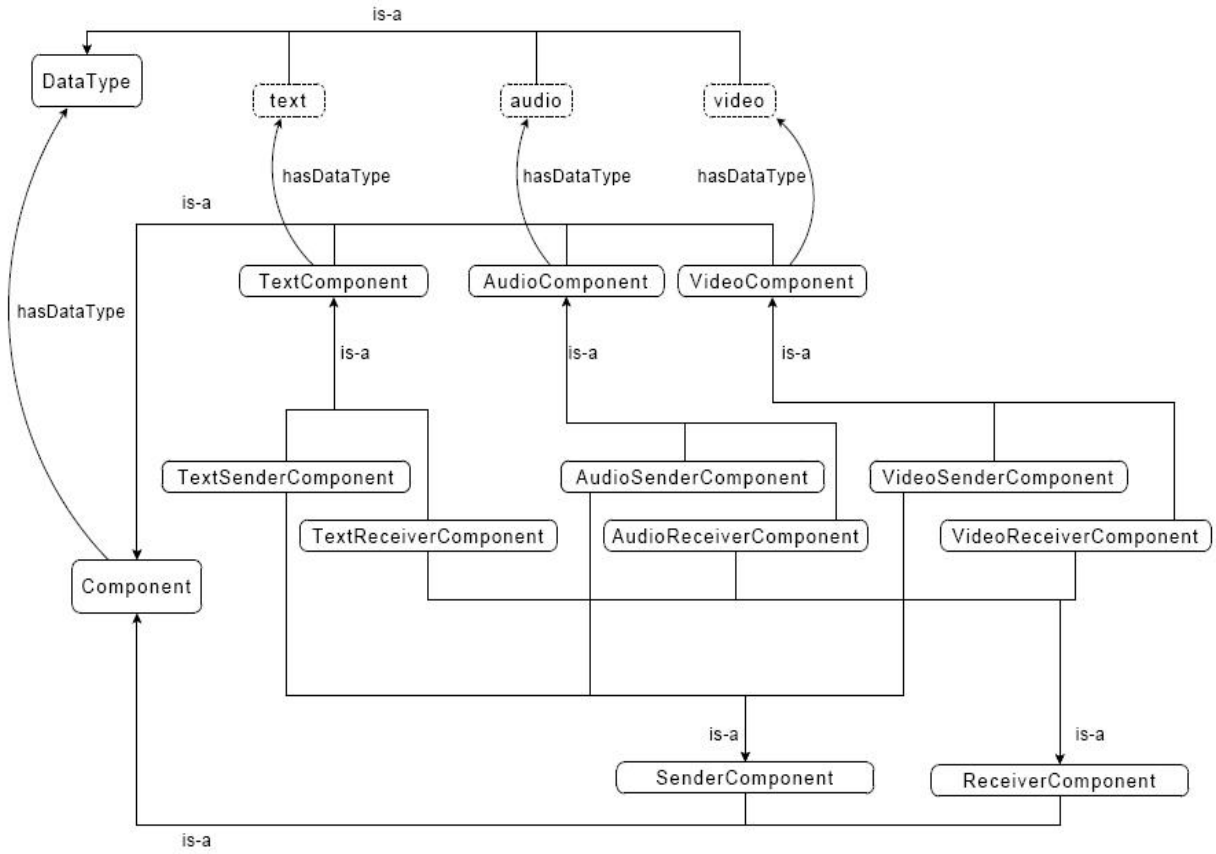

Fig. 4.2 Components taxonomy in GCO

Concepts and relations of GCO constitute a collaboration model, that is, a generic pattern which captures possible collaboration situations. Model instances express concrete collaboration situations. These instances are sets of OWL individuals belonging to the GCO concepts and connected by properties instances described in GCO.

GCO and its instances could be used as statics models representing concrete situations of collaboration.

3.1. Generic Collaboration Rules

We added SWRL rules to GCO in order to express certain relations, in particular those that allow deducing a deployment schema from the sessions present in the ontology instance. These relations would have been very difficult, or even impossible, to express with OWL only.

The first three rules of the ontology are called, respectively, *audio_flows_datatype*, *text_flows_datatype* and *video_flows_datatype* (figures 4.3, 4.4 and 4.5). For example, the rule *audio_flows_datatype* allows deducing that the data type of AudioFlows is audio.

```
AudioFlow(?f) → hasDataType(?f, audio)
```

Fig. 4.3 *audio_flows_datatype* rule

```
TextFlow(?f) → hasDataType(?f, text)
```

Fig. 4.4 *text_flows_datatype rule*

```
VideoFlow(?f) → hasDataType(?f, video)
```

Fig. 4.5 *video_flows_datatype* rule

The *same_group* rule, represented in Figure 4.6, deduces that two roles belong to the same group.

```
belongsToGroup(?r1,?g) ∧ belongsToGroup(?r2,?g)
→ belongsToSameGroup(?r1,?r2)
```

Fig. 4.6 *same_group* rule

The *components_manage_flow* rule, represented in Figure 4.7, states that, whenever a flow belonging to a session is found between two nodes, a *SenderComponent* has to be present in the source node and a *ReceiverComponent* has to be present on the destination node. These components send and receive, respectively, the flow, and they have the same data type as the flow. This rule uses the SWRL built-in *createOWLThing* that allows creating new individuals. Please note that the first *createOWLThing* matches the source node and the session, while the second matches the destination node and the flow. This choice enables multicast flows where a single sender component sends several flows to several receiver components.

```
Flow(?f) ∧ belongsToSession(?f,?s)
∧ hasDataType(?f,?dt)
∧ hasSource(?f,?src) ∧ hasDestination(?f,?dst)
∧ swrlx:createOWLThing(?sc,?src,?s)
∧ swrlx:createOWLThing(?rc,?dst,?f)
→ SenderComponent(?sc) ∧ hasDataType(?sc,?dt)
∧ isDeployedOn(?sc,?src) ∧ sendsFlow(?sc,?f)
∧ ReceiverComponent(?rc) ∧ hasDataType(?rc,?dt)
∧ isDeployedOn(?rc,?dst) ∧ receivesFlow(?rc,?f)
```

Fig. 4.7 *components_manage_flow rule*

The *components_datatype* rule, represented in Figure 4.8, allows deducing the component data type from flow data type managed by this component.

```
Component(?c) ∧ Flow(?f) ∧ managesFlow(?c, ?f)
∧ hasDataType(?f,?dt) → hasDataType(?c,?dt)
```

Fig. 4.8 *components_datatype rule*

3.2. Inference and rules processing

The processing of these rules over an instance of the ontology, as well as its classification and its interrogation with an interference engine, allows to use information contained in this instance.

Let us suppose that we have an instance of the ontology which expresses a possible situation of collaboration. This instance will contain individuals belonging to the concepts of GCO, which will be related through relations defined in GCO. The rules processing will allow to:

1. Allocate to every individual of *Flow* subclasses his type of data (made by rules *audio_flows_datatype, text_flows_datatype and video_flows_datatype*).

2. *Create* individuals of the classes *SenderComponent* and *ReceiverComponent*, representing components which allow to send and to receive every flow. The *isDeployedOn* relation will relate these components to nodes where they are deployed. These components will have for value of the *hasDataType* property one of the individuals of the *DataType* (audio, video or text) class. All this is made by the *components_manage_flow* rule.

3. Get knowledge that was *informed* by an implicit way. This knowledge can be obtained by an interrogation of the interference engine.

Example We consider the instance of GCO represented in Figure 4.9. This example is very simple. It contains only a part of GCO elements. But it serves to illustrate how are made the rules processing and reasoning with GCO. In this example there are two individuals of the class *Node* (*node1* and *node2*), and an individual of the class *AudioFlow* (flow1). This flow has as source the node *node1*, and its destination is *node2*. We also represent the audio individual of the *DataType* class, that is included in GCO.

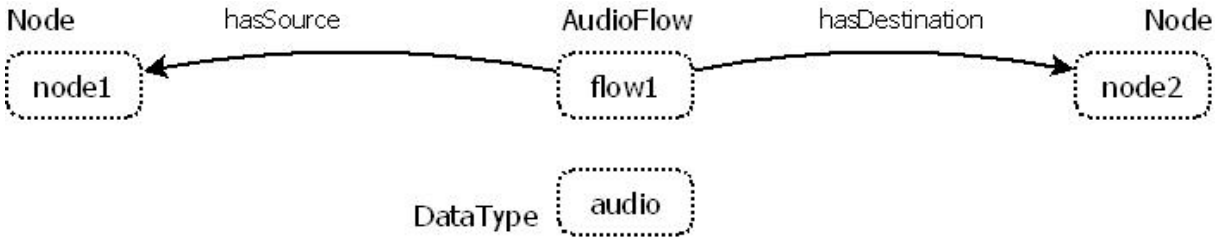

Fig.4.9 example of reasoning and rules: initial situation

Figure 4.10 represents the ontology after rules processing and reasoning. We marked next to every individual the class to which it belongs. First of all, the *audio_flows_datatype* rule deduces that *flow1* has audio as data type. After that, the *components_manage_flow* rule *added a SenderComponent (sc), having audio as data type, deployed in node1*. It also creates *a ReceiverComponent,* having *audio* as *data type,* deployed in *node2.* The component *sc1* sends the flow and rc1 receives it.

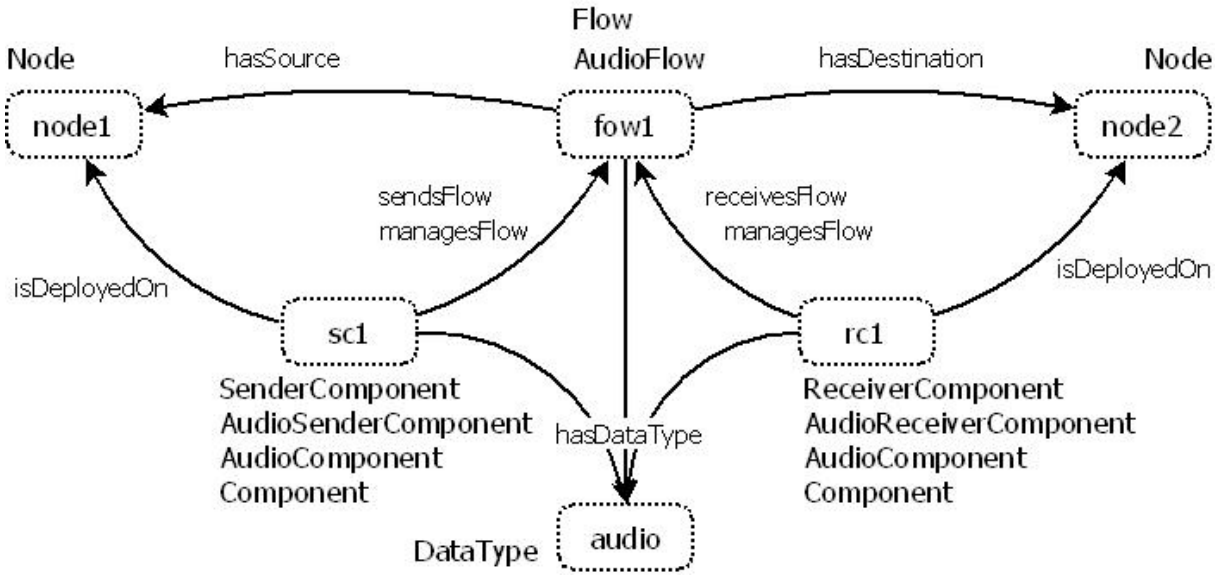

Fig.4.10 example of reasoning and rules: final situation

The reasoner deduces that flow1 is a *Flow*, because this class is a super-class of AudioFlow. Since *sc1* is *SenderComponent* and has audio as data type, the reasoner deduced that it belongs to the classes *AudioSenderComponent, AudioComponent and Component.* In a similar way, it deduces that *rc1* belongs to the classes *AudioReceiverComponent, AudioComponent* and *Component.*

The reasoner also deduces that *sc1* manages the flow *flow1* (*managesFlow* property), because *sendsFlow* property is a sub-property of the *ManagesFlow* property. In a similar way, *rc1* is related to *flow1* through the property *managesFlow*.

The reasoner also finds all the inverse relations, which we did not include in the figure so that it remains legible; for example *flow1* is related to *sc1* through *isManagedBy* and *isSentBy*, and to *rc1* through *isManagedBy* and *isReceivedBy*.

4 Design principles

Several choices must be made in the ontology design. In this section, we present the design principles which guided us to these choices during the GCO design, and we focus on the properties that result from it.

4.1. Ontology language

The GCO is expressed in OWL, which is the current web standard for ontology description. Since the expressivity of OWL is not enough for some of the required relations, rules are used. Rules are expressed in SWRL. Standard, open-source tools are available for processing OWL ontologies and SWRL rules.

4.2. Ontology contents

Since the main goal of the GCO is to support collaboration in run-time systems, the concepts and relations present in this ontology have been chosen among those that have been used in collaboration models until today (i.e., those presented in the previous section). For example, it contains concepts representing sessions, flows, roles, etc. In order to enable dynamic deployment services based on the GCO, some other elements such as components tools, etc. have been added to this ontology. The rules associated to the GCO are also designed in order to enable a simpler deployment process by making explicit the deployment schema that must support the collaborative activity described by the ontology.

4.3. Genericity and extensibility

The GCO has been designed in order to be as generic as possible. This means that it may be used to model collaboration in any application, regardless of the domain. In this aspect, the GCO can be viewed as an upper ontology that can be extended by domain ontologies in order to model domain-specific concepts and relations. For example, in an application of e-learning, this adaptation will consist in deriving the generic notion of role in two notions *professor* and *student*, which correspond to the specific domain. This adaptation is made by the creation of a second ontology that imports GCO and defines concepts and relations that inherit from those which are present in GCO.

The simplest way of extending this ontology is to use inheritance by defining sub-concepts and sub-relations of the concepts and relations present in the GCO (is-a relation).

4.4. Multi-Layered Architectures

The genericity and extensibility of the GCO mean that it can be used inside a multi-layered architecture. In such case, the GCO may be the core model of the layer that handles collaborative sessions. Domain-specific data may be handled in upper layers, while low-level data, such as network connections, can be handled in lower layers.

4.5. Simplicity

The contents of the GCO have been chosen to enable a complete modeling of collaborative sessions. However, only basic elements have been retained. Therefore, this ontology is lightweight and reasoning and rule processing may be performed at run-time without heavy overhead. Moreover, this simplicity eases the task of designers willing to use or extend this ontology for domain-specific applications.

4.6. Naming

In order to have a clear naming which facilitates the understanding Of GCO, we followed the following principles to name the elements of this ontology:

4. Using the camel case.
5. The names of the concepts begin with a capital letter. Example: *VideoFlow*.
6. The names of the relations begin with a small letter. The first word is a verb and the second one is a complement to the verb. The subject of the verb is the domain of the relation, whereas the concept pointed by the relation is the complement of the verb.
7. The names of individuals and data types are written in small letter. Example: *audio, string.*

5 Constructing models from GCO at runtime

An ontology may be considered as a meta-model that describes the possible concepts and relations of a given domain. The instantiation of a meta-model to a specific application enables the construction of the model of this domain for this application. For example, in the case of GCO, the domain is cooperation, so GCO gathers all concepts

and relations that model cooperation between human in a working situation. If we instantiate GCO to the application of artifact design by a group, the generated instance is the model of design by a group. Actual instances of GCO considered as a meta-model are represented by individuals of the concepts available in the ontology. Such individuals (and the relations between them) may be used in order to represent the state of the application at a given time. Relations and concepts are fixed at design-time, while individuals representing the state are created at run-time. In order to use the ontology as the core model in a run-time system, the system must be able to perform the following tasks:

- read the concepts and relations of the ontology ;
- read/modify the individuals existing in the ontology and the values of their properties;
- create new individuals and set the values of properties;
- perform reasoning and rule processing over the ontology and its individuals.

The monotonic nature of OWL inference may represent a problem. Indeed, OWL does not take charge of non-monotonic inference [SWO04]. This means that reasoning and rules cannot modify (addition or removal) the information contained in the ontology. They only allow to find implicit knowledge contained in the ontology and making it explicit. For example, if the processing of a rule in the GCO results in the creation of an individual of the class *Flow* whose source is *node A* and whose destination is *node B*, this information will always remain in the ontology. No other rule can remove it afterwards. If the application needs to remove this individual in order to reflect a new state, it can do it programmatically, but it can be very tricky and unpractical (or even impossible) to keep a track of which information has been inferred and to decide what has to be deleted at every moment.

The solution to this problem is to use the inference capabilities of OWL in a capture-inference-results loop such as the one depicted in Figure 6.1. The first step consists in capturing the state of the world that is modeled by the ontology. This is done by the code of the application using the ontology. Then, this state is introduced in the ontology by creating individuals of the available concepts and by establishing relations between these individuals through object properties. The result is a set of ontology individuals related between them reflecting the state of the modeled world. Once this model has been built, the inference and rule engines can process the resulting ontology. The result of this step is a new version of the ontology where new individuals and relations may

have been introduced. Whenever the state of the world has changed (e.g., when one of the users leaves), the whole loop has to be repeated in order to adapt the response of the application to the new state.

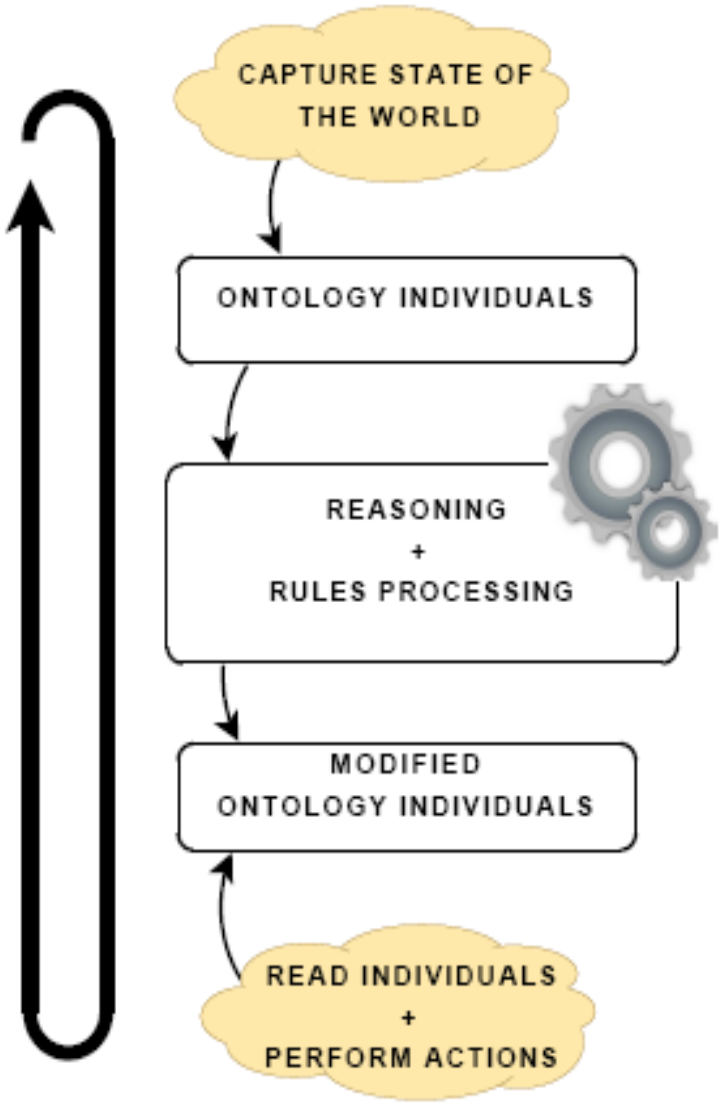

Fig. 6.1 Capture-inference-results loop for run-time systems using ontology reasoning.

The presented loop is discrete; the results of a step are valid until the next change in the state of the world. Whenever a change occurs, the whole loop is executed again in order to get the new results. Because of the monotonicity of OWL inference, the new state cannot be represented by directly modifying the resulting ontology individuals; it would be necessary to delete all the inferred knowledge. Otherwise, the next inference process will result in an inconsistent ontology.

6 Example of GCO specialization in GALAXY

In this section, we introduce an illustrative example of application ontology for Galaxy. This example explains how to use the Generic Collaboration Ontology following the specialization principle.

In order to use GCO, we must define an application level ontology that specializes GCO and represents collaboration between work group members. Figure 7.1 represents a part of a possible ontology.

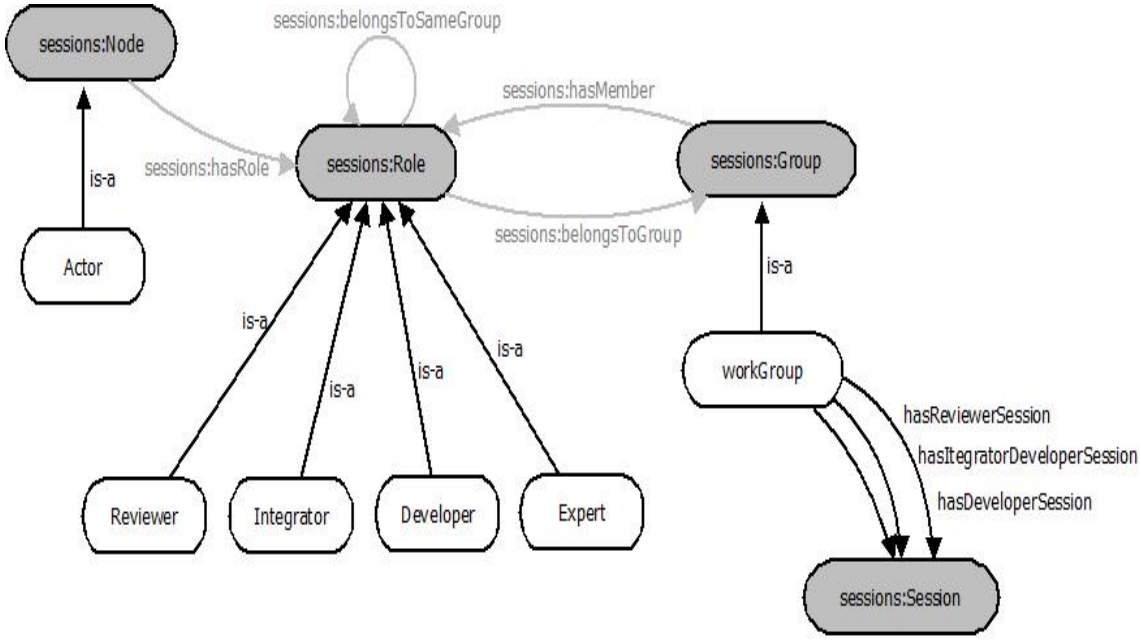

Fig. 7.1- Example of application ontology

In this figure, we represent in grey the concepts and relations of GCO, which are imported in this ontology with the prefix sessions, whereas those, which are specific to GALAXY, are represented in white.

In this ontology, the concepts: *Group*, *Role* and *Node* are specialized into sub-concepts. The sub-concepts of *Role* are: *Integrator*, *Expert*, *Developer* and *Reviewer*. The *Group* concept is specialized into the *WorkGroup* sub-concept. The *Node* concept is specialized into the concept *Actor*, which is a Galaxy concept.

The application ontology imports GCO, so that all GCO elements are accessible to this ontology. The sub-concepts of Role can be members of a group. Similarly, the *WorkGroup* concept inherits from *Group,* which allows groups to have members.

In this example, we consider three properties linking the *WorkGroup* concept to the *Session* concept: *hasIntegratorDeveloperSession*, *hasReviewerSession* and

hasDeveloperSession. These three properties, which are sub-properties of *hasSession,* represent three types of communication that take place inside the work group.

A set of additional SWRL rules will be associated to the GCO in order to express some additional knowledge and to enable deployment-related inference.

The Figure 7.2 represents an application level ontology. It contains two work groups *LAASGroup* and *LIP6Group* and two sessions: *Reviewer_s,* which allows communication between all reviewers; and *Integrator_Developer_s,* which allows communication between the integrators and the developers. The group: *LAASGroup* consists of an integrator *Int1*, which is the role of the actor actor*1*deployed in the device *kalil_PC*, and a developer *develop1*, which is the role of the actor *actor3* deployed in the device *aymen_PC*. The other work group *LIP6Group* consists of a developer *develop1*, which represents the role of the actor *actor2* deployed in the device *said_PC*, and a reviewer *rev1*, which represents the role of the actor *actor4* deployed in the device *german_PC*.

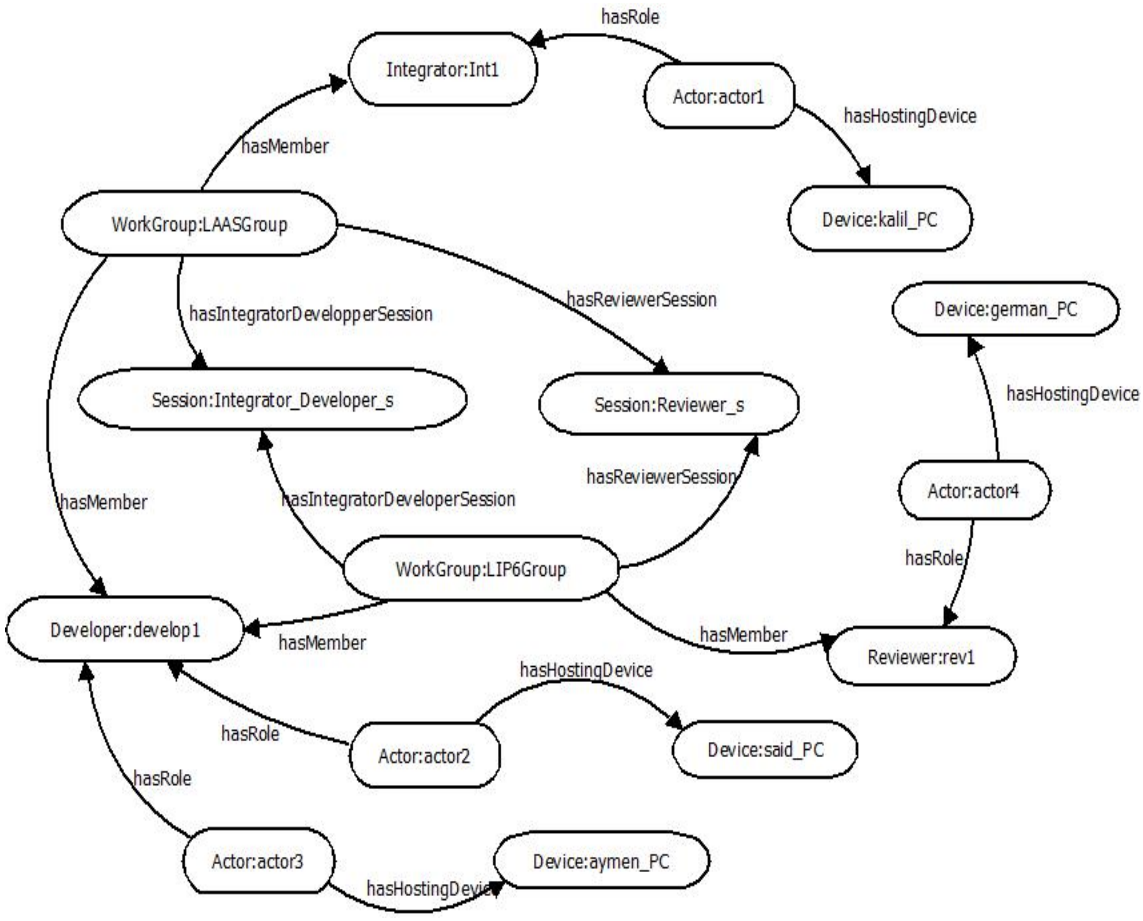

Fig. 7.2 – Application level ontology

The figure Fig. 7.3 represents all GCO individuals created by the SWRL rules of the collaboration layer. This graph must contain all flows and components that allow participants to communicate. In the example, the two groups contain two developers and

one integrator. Thus, participants, having these roles, can communicate within the session: *Integrator_Developer_s*. This figure contains the 4 flows sent by participants and the 7 components that manage these flows. We don't find 8 flows because the two flows that the *actor1*sends to *actor2* and *actor3*, which belong to the session: *Integrator_Develoer_s*, are managed by one sender component: *SWRLInjected6*.

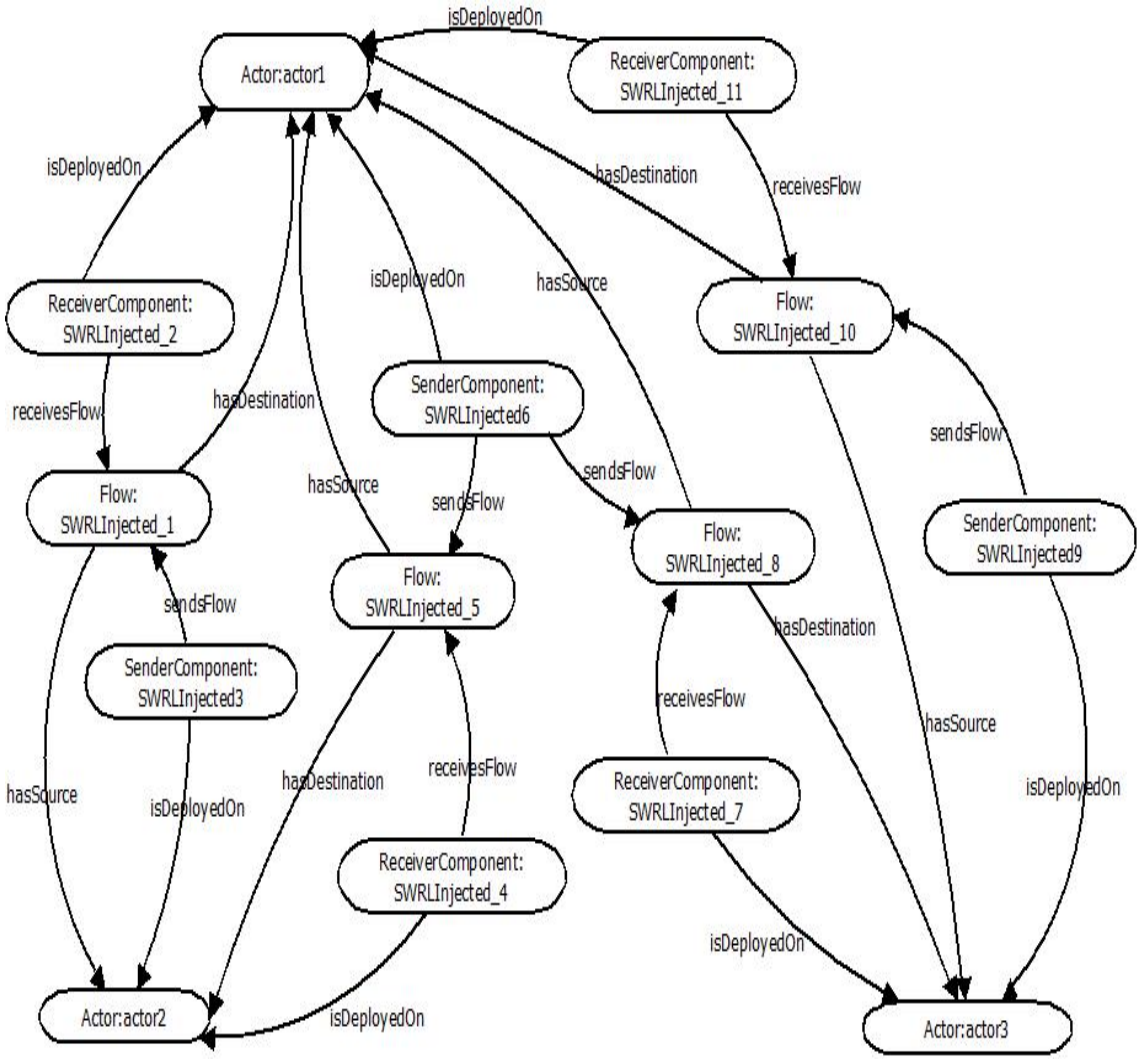

Fig. 7.3 – Collaboration level ontology

In the basic model GCO, a node, theoretically, can be distributed on several devices. In that case, an actor, represented by a node, is associated to all devices in question. For Galaxy, we assume that the cardinalities of the relations: node/device are 1-1. Thus, either a node or a device can identify the actor, and existing flows between nodes reflect exchanges between actors via their devices.

7 Conclusion

This deliverable has presented the GCO, a generic collaboration ontology that represents knowledge about model-driven team communication for collaborative

activities. This ontology is generic because it can be extended in order to model domain-specific collaboration knowledge following the specialization principle. Rules associated to the GCO allow to implement ontology-driven systems using the GCO as their core collaboration model for implementing session management and deployment services. Brief explanations on this usage of the GCO in run-time systems have also been provided.

After a theoretical study provided in this deliverable, a good design is required in order to exploit this collaboration model in the GLAXY project. The adaptation of GCO requires a specialization that heeds all necessary concepts of GALAXY.

8 References

[BCM+] D. F. Baader, D. Calvanese, D. L. McGuinness, P. Patel-Schneider et D. Nardi : The Description Logic Handbook. Theory, Implementation, and Applications.

[DGLA99] H.-P. Dommel et J. Garcia-Luna-Aceves : Group coordination support for synchronous internet collaboration. Internet Computing, IEEE, 3(2):74 –80, mar. 1999.

[EDW94] W. K. Edwards : Session management for collaborative applications. In CSCW *'94* : Proceedings of the *1994* ACM conference on Computer supported cooperative work, pages 323–330, New York, NY, USA, 1994. ACM.

[EGR91] C. A. Ellis, S. J. Gibbs et G. Rein : Groupware : some issues and experiences. Commun. ACM, 34(1):39–58, 1991.

[Ham07] E. Hammami : Déploiement sensible au contexte et reconfiguration des applications dans les sessions collaboratives. Thèse de doctorat, Université Paul Sabatier, Toulouse, 121p, 2007.

[Hor51] A. Horn : On sentences which are true of direct unions of algebras. Journal of Symbolic Logic, 16(1):14–21, 1951.

[HPSB+04] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof et M. Dean : SWRL : A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission 21 May 2004, 2004. Url : http://www.w3.org/Submission/SWRL/, accédé le 23/07/2010.

[Kar94] A. Karsenty : Le collecticiel : de l'interaction hommemachine à la communication homme-homme. Technique et Science Informatique (TSI), 13(1):105–127, 1994.

[KFNM04] H. Knublauch, R. W. Fergerson, N. F. Noy et M. A. Musen : The Protégé OWL plugin : An open development environment for semantic web applications. pages 229–243. Springer, 2004.

[KK88] K. L. Kraemer et J. L. King : Computer-based systems for cooperative work and group decision making. ACM Comput. Surv., 20(2):115–146, 1988.

[PSG+05] B. Parsia, E. Sirin, B. C. Grau, E. Ruckhaus et D. Hewlett : Cautiously approaching SWRL. Url : http://www.mindswap.org/papers/CautiousSWRL.pdf, accede le 23/07/2010, 2005.

[RKT95] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch et P. Muth : Towards a cooperative transaction model – the cooperative activity model. In VLDB '95 : Proceedings of the 21th International Conference on Very Large Data Bases, pages 194–205, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[RPVD01] L. Rodriguez Peralta, T. Villemur et K. Drira : An XML on-line session model based on graphs for synchronous cooperative groups. In 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), pages 1257–1263, Las Vegas (USA), 2001.

[SSS91] M. Schmidt-Schaubß et G. Smolka : Attributive concept descriptions with complements. Artif. Intell., 48(1):1–26, 1991.

[STV10] G. Sancho, S. Tazi et T. Villemur : GCO : a Generic Collaboration Ontology. In Proceedings of the Fourth International Conference on Advances in Semantic Processing (SEMPARO 2010), Florence, Italy, octobre 2010.

[SVT10] G. Sancho, T. Villemur et S. Tazi : An ontology-driven approach for collaborative ubiquitous systems. International Journal of Autonomic Computing, 1(3):263–279, 2010.

[SWM04] M. K. Smith, C. Welty et D. L. McGuinness : OWL Web Ontology Language Guide. W3C Recommendation, février 2004. Url : http://www.w3.org/TR/owl-guide/, accédé le 21/07/2010.

[TP03] G. Texier et N. Plouzeau : Automatic Management of Sessions in Shared Spaces. The Journal of Supercomputing, 24(2):173–181, 2003.

[vil06] T. Villemur : Modèles et services logiciels pour le travail collaboratif. Habilitation à diriger des recherches, Université Paul Sabatier, Toulouse, France, septembre 2006.

ACRONYMS AND DEFINITIONS

ACRONYM	***DESCRIPTION***
OWL	Web Ontology Language. A web ontology language, it is defined be compatible with the architecture of the World Wide Web in general, and the Semantic Web in particular. OWL builds on RDF and RDF Schema and adds more vocabulary for describing properties and classes: among others, relations between classes
SWRL	Semantic Web Rule Language, a proposal for a Semantic Web rules-language, combining sublanguages of the OWL (OWL DL and Lite) with those of the Rule Markup Language .
CSCW	Computer Supported Cooperative Work, a generic term, which combines the understanding of the way people work in groups with the enabling technologies of computer networking, and associated hardware, software, services and techniques.

Appendix (Semantic Web technologies, an overview)

In this appendix, we describe the basic principles of the OWL ontologies [SWM04], the reasoning in OWL and SWRL rules [HPSB+04]. This description will allow understanding concepts and choices detailed in the rest of this report. We also present some available tools for the manipulation of OWL ontologies and SWRL rules and for reasoning engines in OWL.

A.1 Definition of OWL ontology elements

This sub-section presents necessary definitions to understand OWL ontologies. We begin with a general definitionillustrated by simple samples:

Def. 3.1 Specific domain

The specific domain is *the domain* that *is represented by an ontology, or, the part of the world that we model.*

Example 3.1

Assume we model the relations between human family members. The doamin is all persons and realtions between them.

Def. 3.2 Individual/instance

Individuals or instances are the objects of the specific domain.

Example 3.2

Following the domain defined in the example 3.1, individuals are all implied persons, such as for example Louis, or Jean, they are instances of persons object.

Def. 3.3 Concept/class, subclass, super-class, inheritance, taxonomy

A concept or a class represents a set of individuals having common features. A class can be a subclass of another one, called super-class. The subclass inherits from the super-class. In this case, every individual belonging to the subclass also belongs to the super-class. Taxonomy is a hierarchy of classes which have subclass / super-class relations between them.

Example 3.3

A person is a concept. Woman is a subclass of the concept person. Person is a super-class of woman. Jane is a woman that inherits all properties of person. The set of concepts of a domain constitutes a taxonomy.

Def. 3.4 Relation/property, domain, range, sub-relation, super-relation

A relation or property models the relationship that exists between two classes or between a class and a data type. The domain of a relation is the set of classes that can be the origin of the relation. The range of the relation is the set of classes or data types that can be the destination of the relation. A relation can be a sub-relation of another one, called super-relation. In that case, the domain and the range of the sub-relation are respectively contained in the domain and in the range of the super-relation.

Example 3.4

In the example represented in the figure Fig.A.1, the *Person* class represents the domain of the relation *hasResponsability*, whereas the *Responsability* class represents the range. The relations: *hasManResposability* and *hasWomanResponsability* are sub-relations of *hasResponsability*. The domain of the relation *hasManResponsability* is *Man*, which is contained in the domain of the super-relation: *hasResponsability*. The domain of the relation *hasWomanResponsability* is *Woman*, which is also contained in the domain of the super-relation: *hasResponsability*. The *Responsability*, which is the contained in the range of the *hasResponsability* relation, represents the range of the two relations: *hasManResponsability* and *hasWomanResponsability*.

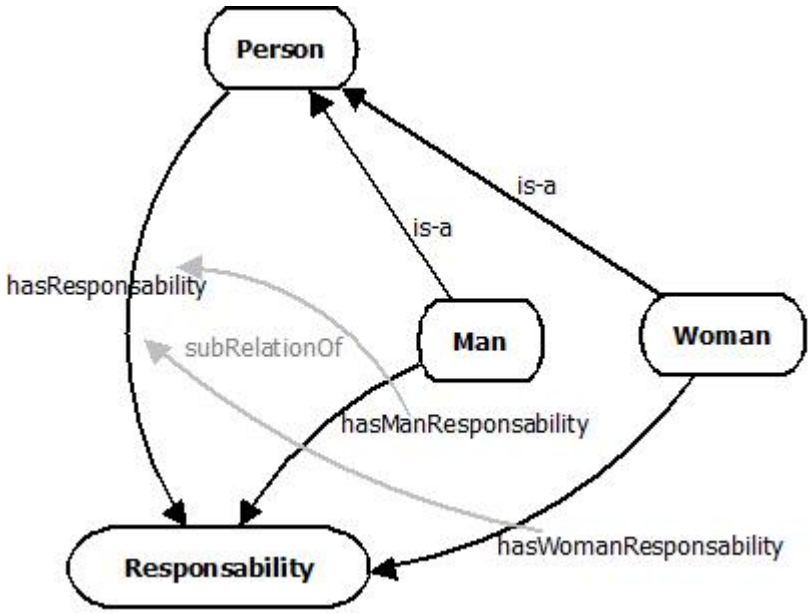

Fig.A.1 Example of sub-relations

Def. 3.5 Instance of a relation

An instance of a relation binds *an individual* that *belongs to the domain of the relation to an individual or a data type* that *belongs to the range of the relation.*

Instances of relations are often called relation. The context allows distinguishing if we speak about relations themselves or about their instances.

Example 3.5

If we consider an individual Jeans which is a *Man* and an instance *resp* of the class *Responsibility*, then we can define an instance of the relation: *hasManResponsability* which binds Jeans and *resp*

The following definitions detail all the attributes of properties.

Def. 3.6 Inverse property

We define a property as the inverse of a given one. This means that, if an instance of this last property relates the individual a *to the individual* b*, then we can deduct that an instance of the inverse property relates* b *to* a.

Example 3.6

If we have a man and a woman and the man is related to the woman by the property *isBrotherOf*, then we can deduce that the woman is related to the man by the property *isSisterOf*.

Def. 3.7 Functional property and inverse functional property

A property is functional if it can have only one single instance for each individual. The inverse of a functional property is its functional reverse.

Example 3.7

If in our domain ontology, a man could not be married to two women, then the property *MarriedTo between* men and wemen is functional.

Def. 3.8 Transitive property

A property p is transitive when, if an individual a *is related to* b *by an instance of p and* b *is related to* c *by another instance of p, then we can deduce that* a *is related to* c *by an instance of p.*

Example 3.8

hasBrother is transitive because If the individual Louis is related to Jean by an instance of the property *hasBrother,* and Jean is related to Nicolas by another instance of *hasBrother*, then we can deduce that Louis is related to Nicolas by *hasBrother*.

Def. 3.9 Symmetrical property

A property p is symmetrical if for each individual *a* related to individual *b* by an instance of p, we can deduce that *b* is related to *a* by another instance of p.

Example 3.9

isBrother is symmetrical because If Louis is related to Je an by the property *isBrother*, then we can deduce that Jean is related to Louis by *isBrother* property.

In order to define a class in OWL, we provide a set of logical conditions. These conditions can be “necessary” or “necessary and sufficient.” They are built from other classes, by union, by intersection or by inheritance. We can also impose restrictions to the properties of the class.

Def. 3.10 Restriction, existential restriction, universal restriction, cardinality restriction, value restriction

A restriction *consists in limiting the number or the nature of values that the properties of class individuals can have. A restriction can be* existential *(if it should have at least a value of the property in a given set),* universal *(if it should have all the values of a property in a given set), of* cardinality *(if it should have a minimal, maximal or exact values number for a property) or of* value *(if she should have a given value for the property).*

Example 3.10

Existential *restriction*: a man *MarriedTo* a woman means that there is a woman that fulfill the prperty

Universal *restriction: belongToFamily* has a universal restriction to all individuals of the family

Cardinality restriction: a man *MarriedTo* for a woman means that there is only one woman that fulfill the prperty

Value restriction: an animal, whatever it is carnivorous or herbivorous, should eat, Thus the property *eats* should have a given value in a set specified by the nature of the animal. Carnivorous should eat meat and herbivorous should eat vegetable .

Def. 3.11 Disjoint classes

Two classes are disjoint if there is no individuals that belong at the same time to both classes.

Example 3.11

See the Example 3.12.

In OWL, classes are not disjoint by default; it is necessary to declare it explicitly.

Def. 3.12 Importing ontology

An ontology can import another ontology in order to have visibility into its elements. The ontology that imports has only reading access to all elements contained in the imported ontology. Then, it can add new elements, which will be visible only to the

ontology which imports. The import action is transitive: an ontology that imports a second one imports indirectly all imported ontologies in this second ontology.

Generally, the import is made by indicating, in the ontology which imports, the imported ontology URL. This mechanism allows a big flexibility, because we can reuse existing ontologies just by referencing them in a new OWL file and by adding new classes, properties, individuals, rules, etc. Ontologies often follow this schema: a first high-level ontology which contains generic elements is imported by a second domain ontology which specialize it into concrete domain. In that case, the second ontology extends the first one.

Example 3.12

A simple ontology is represented in the Figure 3.1. This ontology illustrates the explained elements. In this ontology, there are 8 classes: *Person, ManualWorker*, *Plumber, Politician*, *Job*, *ManualJob*, *Plumbing* and *Politics*. *is-a* Arrows represent super-class/subclass relations: for example, *Plumber* is a subclass of *Person*, because all plumbers are persons. The relation *hasJob* binds *Person* to *Job*, this means that persons can have a job. The relations father, uncle and brother bind the class *Person* to itself, because the father, the uncle or the brother of a person are also persons. The classes *Person* and *Job* are disjoint.

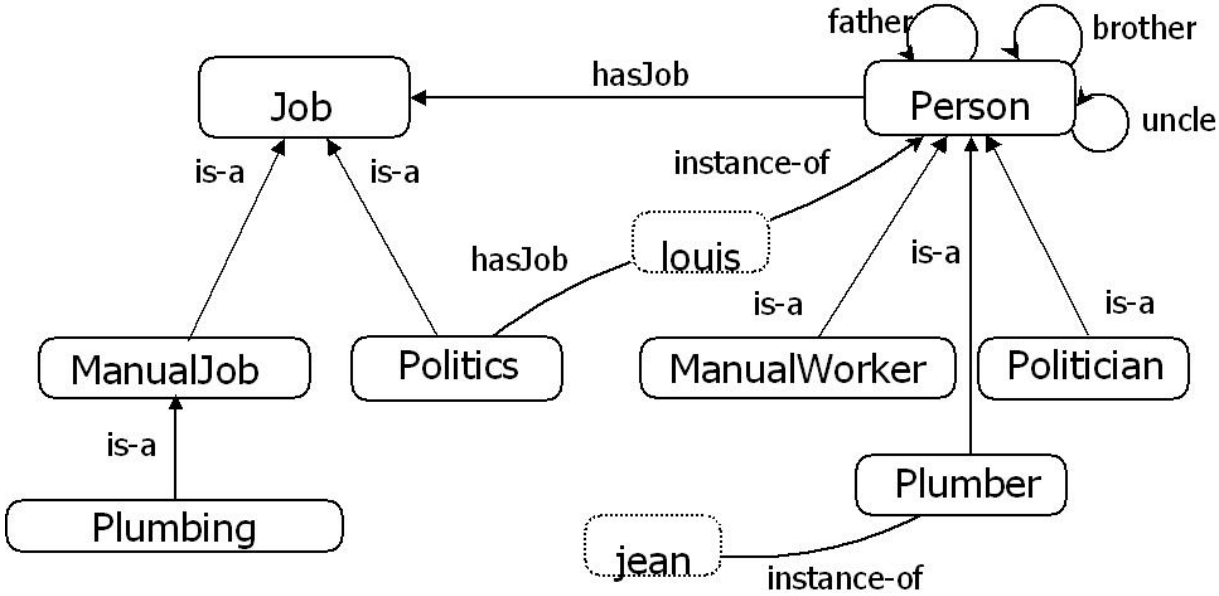

Fig. 3.1 – Example of OWL ontology

The class *ManualWorker* is defined as:

ManualWorker ≡ Person $\prod$ $\exists$ *hasJob.ManualJob*

This means that the necessary and sufficient condition (≡) which confirms that an individual is considered as worker is that he should be a person (Person) and (∏) have an instance of the relation *hasJob* which relates him to a manual job (∃ *hasJob.ManualJob*). By the same way:

Politician ≡ Person ∏ ∃ hasJob.Politics

Plumber ≡ Person ∏ ∃ hasJob.Plumbing

Finaly, there are two individuals, *jean* and *louis*. We know that the first one is a plumber and that the second is a person and that its job is politics.

A.2 Reasoning in OWL

As we mentioned, the fact that OWL has a formal theoretical base (the description logic) allows the setting-up of software tools called inference engines or reasoners, which process OWL ontology to deduct facts which are not explicitly declared [BCM +], i.e. they can find information which are implicitly contained in the ontology in order to make them explicit. This process is called *inference* or *reasoning*.

The main task of reasoning that an inference engine can carry out is known as subsumption. Subsumption allows to know if a class is a subclass of another one or not. By using it on all classes of the OWL ontology, the inference engine can build a hierarchy of deduced classes (in opposition to the declared hierarchy of classes) in which all relations super-class/subclass are explicit.

In our example, the deduced hierarchy of classes contains the fact that *Plumber* is a subclass of *ManualWorker*, because all plumbers have *Plumbing* as job, which is a *ManualJob*, thus they perform all necessary and sufficient conditions in order to be in the class *ManualWorker*. *Plumbing* is also a subclass of *Job*, and all manual jobs are jobs.

Another task of standard reasoning is the check of the ontology consistency, which allows to detect if there are not coherent classes, i.e. it is not possible to declare an individual of these classes without having a logical contradiction. This task is very useful during the creation process of the ontology for the debugging of this one. For example, if we want to declare a new class which is a sub-concept of *Job* and *Person* at the same time, a reasoner engine will deduct that this class is incoherent, because *Person* and *Job* are disjoint and we cannot have individual that belongs to both classes.

Other tasks of reasoning allow deducting facts such as the belonging of an individual to a class. Reasoning may enable for example the deduction of the existence of properties relating two individuals by transitivity, symmetry and inverse properties, etc.

The setting-up of these tasks uses well known algorithms as the board method based algorithms [SSS91] which work with logical definitions of classes and properties to make deductions.

Inference engines can use these tasks of basic reasoning in order to provide more complex services of reasoning, in particular:

1. *Research*: allows finding all individuals that are instances (direct or indirect) of a given concept;

2. *Execution:* allows finding the most specific concept to which belongs (directly or indirectly) a given individual.

For example, in our ontology, these services allow finding that the individual *jean* belongs to the classes *Plumber*, *ManualWorker* and *Person* and that *louis* belongs to the classes *Person* and *Politician*. These tasks are very useful because they allow a transparent usage of the deduced knowledge in applications.

The reasoning in OWL applies the principle known as the Opened World Assumption. This principle stipulates that we cannot consider that a fact does not exist, unless having declared explicitly its nonexistence. In other words, we shall not consider as false a proposition simply because we did not declare it as true; the proposition will be considered as "unknown". This assumption has a strong influence on the way of defining the ontology elements. For example, if we declare two classes *A* and *B* and an individual *a* that belongs to the class A, we cannot consider that *a* does not belong to the class B; it would be necessary to declare it explicitly so that the reasoning (and the rules processing) know that *a* is not an instance of B. By the same way, it will be necessary to declare explicitly that two classes are disjoint in order to be able to deduct that an individual belonging to the first class does not belong to the second one, etc. After all, with the Opened World Assumption, we consider that the knowledge that we have is not necessarily complete, and we cannot suppose anything on what is not declared. In our example, we cannot suppose that *jean* is not politician, because the classes *Plumber* and *Politician* are not disjoint.

Another concept, related to the Opened World Assumption, is the Unique Name Assumption, which is not considered in OWL. This means that the fact of having two individuals having two different names does not imply that these individuals are different. If we want to consider that two given individuals are different, it is necessary to declare it explicitly. In our example, we cannot know if *jean* and *louis* are the same individual or not, because we have not enough information.

The last important feature of the reasoning in OWL is the monotonic effect. The fact that OWL is monotonous implies that any present fact in an ontology cannot be removed [SWM04] by new deduced information. In particular, the propositions deduced by an inference engine can only add information, but never erase it. Even if the added information contradicts the one which existed previously, this last one will not be erased. For example, if in the ontology of the example above (Fig 2), we say that *jean* is a politician, it does not erase the fact that he is a plumber. He will be a politician and a plumber at the same time in this ontology, because the new fact does not replace the precedent. If we had declared the classes *Politician* and *Plumber* as disjoint, then an inference engine can detect that the new fact is incoherent.

A.3 SWRL Rules

The Semantic Web Rule Language (SWRL) [HPSB+04] is a rule language proposed by the W3C which combines OWL-DL with the Rule Markup Language (RuleML). SWRL extends OWL-DL by adding Horn clauses [Hor51]. This addition increases the OWL expressiveness, but, generally, this expressiveness implies loosing of the decidability [PSG+05]. However, in the most part of the practical applications, it is possible to use only a subset of rules called DL-safe, which is decidable.

An SWRL rule is represented as:

$$b1 \wedge \_ \_ \_ \wedge bn \rightarrow a1 \wedge \_ \_ \_ \wedge an$$

b1 ^ _ _ _ ^bn is the body or the antecedent of the rule and a1 ^ _ _ _ ^ a_n is the consequent. The terms $a1 \ldots a_n$, $b1 \ldots b_n$ are the SWRL atoms. An atom can represent a relation (binary predicate), a concept (unary predicate) or a *built- in* (n-arity predicates). The interpretation of the rule is the following: if the conditions specified in the antecedent are verified, then we can infer that the propositions specified in the consequent are also verified. SWRL may be written in XML, which allows including an ontology and its associated SWRL rules in the same XML file.

The utility of those rules is to express complex relations that would be impossible to express with OWL-DL only. For example, in our ontology we can represent the relation between an uncle and his nephew through the relations father-son and brother-brother. The SWRL rule which expresses this relation is represented in the Figure 3.2.

Person (?x) ∩ Person (?y) ∩ Person (?z) ∩ father (?x, ?y)
∩ brother (?x, ?z) → Uncle (?z, ?y)

The elements *x*, *y* and *z*, which are preceded by a question mark, are variables which

Fig. 3.2 - Example of SWRL rules

This rule means that, if we have three individuals belonging to the class *Person* called *x*, *y* and *z*. *x* is the father of *y* and *x* has a brother *z*, then *z* is the uncle of *y*.

In SWRL rules, we can use two special relations *sameAs(p,q)* and *differentFrom(p,q)* which serve respectively to declare that two individuals are the same or are different. They are necessary because OWL does not use the Unique Name Assumption.

The *Built-ins* are free arity predicates, i.e. a predicate may have 0 or more arguments. They serve to implement useful practical functions in SWRL rules. There is a set of predefined *built-ins* which serve for example to make comparisons, to make mathematical operations, to concatenate character strings, etc. For example, the *built-in swrlb:greaterThan(?Age, 17)* allows to compare two numbers (which is in the variable age and the integer 17). If the first one is bigger, then the *built-in* will be estimated as true; otherwise, it will be false.

A.4 Tools for processing OWL ontologies and SWRL rules

One of the reasons of the success of OWL and technologies of Semantic Web is the existence of several tools for ontologies management. Indeed, there are libraries, API, editors, inference engines and rules which facilitate creating and editing ontologies and rules. Furthermore, a big part of these software tools are free, what allows to obtain, to study, to modify and to share them more easily.

Edition of ontologies

The editor Protégé[2] [KFNM04], developed at Stanford's University in association with the University of Manchester, is a standard for creating and editing OWL ontologies. Its source code is written in Java and it admits plug-in extensions. There are several plug-ins, for example to display ontologies or to edit the associated SWRL rules. The last available stable version at present is 3.4.4, but the beta version 4.1 is also available. The version 4 is a total revision of the editor. In particular, it is in compliance with the standard OWL 2.

There are other editors of ontologies, less popular, such as KAON2[3], Swoop[4] and Ontolingua[5].

APIs for processing OWL ontologies

[2] http://protege.stanford.edu/
[3] http://kaon2.semanticweb.org/
[4] http://www.mindswap.org/2004/SWOOP/
[5] http://ksl.stanford.edu/software/ontolingua/

These software libraries provide an access by program to OWL ontologies; they provide functions that allow creating and reading the ontology, to create the corresponding model in memory, to modify it, to save it, etc. Most of these libraries are implemented in Java.

The two main libraries are OWL API[6] and Protégé-OWL API. OWL API is the reference setting-up for the creation, the manipulation and the serialization of OWL ontologies. OWL-API is a free project led by the University of Manchester which takes charge of OWL 2. It also provides several interfaces for the transparent access to the inference engines. Protégé-OWL API is a Protégé's plug-in which is used to access to OWL ontologies in the versions 3.4 of the editor (from the version 4.0, OWL API is used). It can be used by external programs as a java API, independently of Protégé. As OWL-API, it allows the access to external inference engines.

Inference engines

Those engines tools are able, by deduction, to extract implicit knowledge contained in the ontology. They can be run as an independent application or be called by another program, in particular the APIs mentioned above. There are owner setting-ups, such as Bossam[7] or RacerPro[8], as well as others free as Pellet[9], Fact ++[10], KAON2 or HermiT[11]. Among these tools, Pellet is the most popular at the moment, thanks to its capabilities, to its features and to its clear and simple conception.

Tools for SWRL rules processing

Most of inference engines are able to process SWRL rules added to the ontology. For example, Pellet implements natively a specific algorithm for DL-sure rules in OWL. In order to process rules, it is also possible to use engines specifically dedicated to rules, such as the Jess engine[12]. This engine has an appropriate language to express knowledge in the form of rules. It can be used in Protégé (or Protégé-OWL API) thanks to the existence of a bridge that allows to translate an ontology model to Jess's language, to execute rules in Jess and finally to get back the result in Protégé.

Tools choices

---

[6] http://owlapi.sourceforge.net/
[7] http://bossam.wordpress.com/
[8] http://www.racer-systems.com/
[9] http://clarkparsia.com/pellet/
[10] http://owl.man.ac.uk/factplusplus/
[11] http://hermit-reasoner.com/
[12] http://www.jessrules.com/

Here, we explain the choices we made for the creation and the processing of OWL ontologies and rules. These choices were used for the creation of GCO as well as for its use (instanciation of individuals, reasoning, rules processing, individuals' reading). Our choices are represented in Tab 3.1.

Tool/language	Choices
Creation/edition of ontologies	Protégé 3.4.1
Creation/edition of SWRL rules	*Plugin* SWRLTab for Protégé
Access by program to ontologies	Protégé-OWL API 3.4.1
inference engine	Pellet 1.5.2
Rules engine	Jess 7.0
OWL version	OWL 1

Tab. 3.1 – Choice of tools and languages for the creation and the processing of GCO

The choice is the following: Protégé 3.4.1 for the creation and the edition of ontologies, with the *SWRLTab* plug-in for the edition of SWRL rules, Protégé-OWL API for the access by program to the ontologies, Pellet as inference engine and Jess for the execution of rules. We have chosen this because we need to use some SWRL *built-ins* which are available only in Protégé's version 3.4. In particular, we needed the experimental *built-in swrlx:createOWLThing*, which allows to create new individuals in a SWRL rule. This kind of experimental *built-in* is not available in Pellet, and this motivates the use of Jess for rules processing. However, we retained Pellet for other tasks of reasoning such as the check of the ontology consistency or its classification because it is more complete and more powerful than Jess to make these tasks. The access to Pellet and to Jess from Protégé-OWL API is very simple thanks to the available specific bridges. The choice Of Protégé 3.4 and Jess don’t allow the use of the version 2 of OWL for the description of ontologies and OWL-API for their processing.