TELEX SUPPORTED COLLABORATIVE ONTOLOGY ENGINEERING (TCOE)

A thesis
submitted to the teaching staff of
The University of the Aegean

In partial fulfillment
of the requirements for the degree of
Master of Science in the department of
Information and Communication Systems Engineering

by
GEORGIOS SANTIPANTAKIS
MARCH 2009
THE MEMBERS OF THE COMMITTEE APPOINTED TO EXAMINE THE THESIS OF GEORGIOS SANTIPANTAKIS FIND IT SATISFACTORY AND RECOMMEND THAT IT BE ACCEPTED:

GEORGE VOUROS
Department of Information and Communication Systems Engineering

MARC SHAPIRO
LIP6

EFSTATHIOS STAMATATOS, Member
Department of Information and Communication Systems Engineering

KONSTANTINOS KOTIS, Member
Department of Information and Communication Systems Engineering

UNIVERSITY OF THE AEGEAN
MARCH 2009
ABSTRACT

The aim of this project is to develop a software system to assist the collaborative ontology development and evolution. The operation of the system is based on the semantics of the actions performed by the knowledge engineers and workers, acting on a single ontology, as these are realized and exploited by Telex. Telex has been developed in LIP6 (France), and its purpose is to assist and co-ordinate collaborative parts performing actions on the same object (in our case an ontology). This software exploits the semantics of the actions performed by the collaborative members, checks the consistency of the actions applied, based on a set of constraints being specified, and proposes sequences of actions (schedules), which if executed would update the under revision object consistently. The developed system aims to support the HCOME methodology for the collaborative engineering of ontologies, as it has been presented by the Artificial Intelligence Laboratory of the University of the Aegean, although being generic enough for supporting other methodologies and systems.

GEORGIOS SANTIPANTAKIS
Department of Information and Communication Systems Engineering
UNIVERSITY OF THE AEGEAN
© 2009
ACKNOWLEDGEMENTS

I would like to thank prof. G. A. Vouros, for giving me the opportunity to work in this interesting field through this project, and for his patience and guidance provided since the very first day I was engaged. I also would like to thank Lamia Benmouffok and Jean-Michel Busca for support provided when needed. Finally, I would like to thank my family and Sofia for supporting me, and being always by my side.

To the memory of my father Michael, 
aunt Aikaterini, 
and recently deceased uncle John.
TABLE OF CONTENTS

ABSTRACT......................................................................................................................................... iii
ACKNOWLEDGEMENTS...................................................................................................................... iv
TABLE OF CONTENTS........................................................................................................................ v
LIST OF TABLES................................................................................................................................ vii
LIST OF FIGURES.............................................................................................................................. viii
CHAPTER 1 - INTRODUCTION............................................................................................................ 1

1.1 Motivation methodology, technologies, theories and tools used ........................................ 1
   1.1.1 Related work.................................................................................................................... 3
1.2 Selected Underlying Technologies.......................................................................................... 5
   1.2.1 HCOME ............................................................................................................................. 6
   1.2.2 Telex ................................................................................................................................. 6
   1.2.3 Validity rules .................................................................................................................... 7
1.3 A brief outline of the work...................................................................................................... 8

CHAPTER 2 - BACKGROUND KNOWLEDGE........................................................................................ 9

2.1 Human-Centered Ontology Engineering Methodology .......................................................... 9
   2.1.1 Specification phase ........................................................................................................ 10
   2.1.2 Conceptualization phase .............................................................................................. 10
   2.1.3 Exploitation phase ....................................................................................................... 11
2.2 HCONE, a tool supporting HCOME ....................................................................................... 11
   2.2.1 HCONE Specification phase .......................................................................................... 12
   2.2.2 HCONE Conceptualization phase ................................................................................. 12
   2.2.3 HCONE Exploitation phase ......................................................................................... 13
2.3 Telex ...................................................................................................................................... 13
   2.3.1 Telex Related definitions ............................................................................................. 13
   2.3.2 Telex architecture .......................................................................................................... 16
   2.3.3 The Telex Scheduler module ....................................................................................... 18
   2.3.4 The Telex Agreement module ................................................................................... 19
2.4 Ontology Dynamics via validity rules .................................................................................... 20

CHAPTER 3 - MOTIVATION- PROBLEM SPECIFICATION - TYPICAL USE SCENARIOS ............... 23

3.1 Motivation and a more specific problem definition ............................................................. 23
3.2 Typical use scenarios ............................................................................................................ 25

CHAPTER 4 - TCOE PRINCIPLES ....................................................................................................... 27
LIST OF TABLES

Table 1. HCOME phases in ontology life-cycle ............................................................................... 9
Table 2. Constraint Types provided by Telex.................................................................................. 14
Table 3. Ontological facts as First Order Logic predicates............................................................. 20
Table 4. Component sets and validity rules .................................................................................... 22
LIST OF FIGURES

Figure 1. HCOME architecture .............................................................................................. 11
Figure 2. Telex architecture .................................................................................................. 16
Figure 3. Schedule evolving diagram ..................................................................................... 30
Figure 4. Interaction of frameworks ..................................................................................... 32
Figure 5. A general view of the framework .......................................................................... 33
CHAPTER 1 - INTRODUCTION

The vast growth of the World Wide Web, as well as the continuous development of web services, gave birth to the idea that World Wide Web should contain semantics about the information it stores. The fast growth and spread of Web 2.0 environments reveals that this decision is very important as more and more users are willing to contribute and share in these environments.

The Semantic Web comprises, among others, knowledge represented using ontologies. This supports retrieving information/resources and sharing data. Since knowledge itself is evolving, and since people may provide different definitions for the same concepts, even if these are well understood, it is obvious that tools enabling collaborative ontology development can simplify the process of ontology development towards validity, consistency and reaching agreement. In case that concepts specified by different persons do not match, it is possible for these persons to find the “golden cut” once the conflict is detected. Therefore, ontologies developed by these tools provide consistent definitions under a commonly accepted global view.

Let us assume the case where two knowledge workers, Bob and Adam, wish to develop an ontology for the enterprise they work in. Aiming to develop such an ontology the two users first will have to find resources related to their domain, to give them the kick off. Then, they will start a conversation on those resources, to set the specifications of the target ontology. Therefore, a framework is required, which will allow these persons to collaborate on a peer-to-peer way, without necessarily being supervised (e.g. by a panel of domain experts), permitting every possible decision that can be taken by any of them towards their goal. The framework must enable semantically correct modifications, and resolve conflicts in any case. That is, while the two users develop the ontology, each one must be aware of the actions the other does. However, if Bob performs a modification which would affect Adam’s decisions, i.e. it would violate constraints related to Adam's actions, the framework must focus on the conflict, and provide alternative solutions. From the conflicting state none of the users should be in disadvantage; each one should be equally able to build his ontology version, aiming however to reach a consensus to the conceptualization of their domain.

1.1 Motivation methodology, technologies, theories and tools used

Our implemented framework provides a generic environment that any ontology engineering tool may use, allowing these tools to operate jointly. Most of the ontology engineering tools lack the ability of joint ontology development, i.e. the ability to facilitate the joint development of ontologies by teammates. Any ontology engineering tool requires the ontology being developed to be published in a common space, so that evaluation/evolution/development by the rest of the community to be possible. In a distributed environment where joint development is possible, this is not a functional requirement, since each user should be aware of the actions performed by remote users, view alternatives and take decisions. Implementing a generic framework for collaborative ontology engineering would allow the transfer of ontology engineering tools from the classic standalone usage, to their distributed deployment. It is obvious that any collaborating methodology is compatible with such a framework, since it aims to provide functionality for ontology replication, consistency etc, without setting rules on the manner these functions will be used.
Tools and methodologies used towards the collaborative development of artifacts (e.g. documents, or software, database tuples), either adapt database techniques, such as locking, or require a phase for providing feedback on the developed document versions, maybe participating in argumentative dialogues. Currently, there is not any methodology that allows users to simultaneously access the same artifact without restrictions. Considering ontologies, users may modify ontology versions in personal space maintaining transparency. For example, only when the user wishes, the version is published to the community. On the other hand, towards sharing artifacts, there have been proposed methods of distributed databases, such as locking. However, such a method is not desirable for collaborative ontology engineering; although it does conserve against conflicts that may appear among versions, it also confines collaboration. A nice example of a situation like this, is provided in [1].

The example assumes two co-workers, Harry and Sally, who decide to edit the same repository file at the same time. To avoid write contentions and data overwriting, the example also assumes that a lock-modify-unlock model to address such problems is applied. That is, Harry or Sally must “lock” a file before applying any changes to it. When Harry has locked a file, Sally cannot modify it, since she cannot also lock it. All she can do, is either read the file, or wait until it becomes available for locking, i.e. Harry unlocks the file. However, [1] points some interesting issues, adapted as follows:

- It is possible that Harry may lock a file and forgets to release it. In the same time however, Sally cannot modify the file and she is waiting for her turn. Since Harry forgot that he has locked the file, he may never release it, so Sally will wait forever, or find an administrator to release the lock on the file. This situation causes unnecessary delay and wasted time.

- This approach is possible to cause unnecessary serialization, when changes each user wishes to apply, do not overlap, i.e. Harry modifies the beginning of the file, while Sally wants to modify the end of it.

- Finally, locking may create a false sense of security. Let us assume that Harry locks and edits a file, e.g. A, and in the same time Sally locks and edits another file, e.g. B. If the files A and B depend on each other, and changes made to them are semantically incompatible, the violation will never be noticed, and files A and B will not work together any more. Not only this approach was unable to prevent the violation, but also provided the false sense of security. Both Harry and Sally imagined that by locking files, each is beginning a safe and insulated task, so they did not discuss their incompatible changes early on.

Finally, there is no tool available to help in conflict detection under collaborative mode. The collaborative ontology engineering framework (TCOE) that we have developed, detects conflicts, and provides alternative solutions to help users reach agreements in “complex” situations (i.e. in cases where there are complications concerning their previous and current actions). Sometimes, it is difficult to find a solution (or propose alternative solutions) in cases of incompatible user’s conceptualizations. Even more, it is hard to identify who are the users involved in such a conflict, who caused what, etc. Our framework, based on Telex, provides the information required to identify the users related to any such conflict, providing details on the interdependencies of their actions and also providing alternative solutions concerning specific schedules of actions that can be executed without causing any conflict. Not all the users are responsible for a conflict and definitely, not all the conflicts are necessarily produced by the same users. However, informing all the users about the conflicts is the more efficient approach, as users are jointly aware of the situation, may provide alternative solutions, examine different points of view, and jointly decide on a specific solution out of the difficult situation: The aim is users to reach a point of agreement.
1.1.1 Related work

Collaborative ontology engineering is a very important issue, although there is not much work done in this area. The main reasons that make this effort difficult, is that an effective distributed engineering tool and a methodology which can be applied using any such tool, are required. Most of the methodologies presented so far, focus on ontologies developed in a centralized manner. A typical strategy is to separate the process of ontology development to a sequence of discrete phases, that are either related to argumentation, ontology implementation, or ontology re-use. As an example, the methodologies METHONTOLOGY and the OTK, offer guidance for building ontologies either from scratch, or by reusing other ontologies.

Another methodology introduced by Holsapple [2], focus on the collaborative aspects of ontology engineering but towards a “centralized” ontology. First, a knowledge engineer declares an initial ontology which is extended and modified using the feedback from a panel of domain experts. This methodology though, does not support argumentation. Methodologies supporting argumentation are DILIGENT [3], and HCOME [4]. HCOME supports the development of ontologies in a decentralized way and allows ontology evolution. It is based on three different spaces in which ontologies can be stored. The personal space where users can create, modify and merge ontologies, the shared space where the modified ontologies can be published to the community, and the agreed space. The personal space can be accessed only by the user it belongs to, while the shared and agreed space can be accessed by all participants. In the shared space users can discuss ontological decisions and modifications to follow on the ontology. When an agreement is reached, the ontology is moved to the agreed space.

As presented in [3], DILIGENT methodology is supported by a control board which consists of domain experts and knowledge engineers, responsible to provide a shared ontology version to users and workers. The methodology consists of five activities: build, local adaptation, analysis, revision and local update. It is also based on a board of domain experts and knowledge engineers to analyze ontology versions and guide evolution. At the beginning of the process, every participating member contributes in building the initial ontology. It is also recommended that the team involved in building this version of the ontology, should be relatively small, in order to reach a consensus easily. This version of the ontology is not required to cover every aspect of the domain, since it is actually the backbone of the versions to follow. When the initial version is ready, users can replicate it in their personal space (local adaptation). From that step and beyond, each user can freely modify it. This however is not allowed for the shared version, i.e. no participant off the board can modify that version, until a new shared version appears. The control board required by this methodology, analyses local ontologies and user requests, and tries to identify similarities between locally stored versions. Since the target ontology is not yet met, the control board has to identify what changes are to be introduced in the shared ontology. However different user needs and requirements make this task hard, especially in not-well-defined domains. For this reason, the revision process as performed by the board has to be done in a regular basis so that local versions do not diverge too far from the shared ontology. When this process is complete, a new shared version is published and users can replicate it to their personal space. It is suggested that even if differences between some user’s local ontology and shared version are not important, the user should adapt the shared ontology and use concepts defined in that, instead of the locally defined.

As shown in [3], although shared ontology versions derive from the group’s opinion, i.e. changes in the shared ontology are based on the users’ changes and requests, some tasks are
performed exclusively from the board. Workers cannot use and evaluate any ontology version until that version is validated and becomes shared by the board. Even more, the board decides what user requests will be fulfilled in the next shared version, which is a filtering of the concepts that users have specified, via the perception of board members. Also, DILIGENT supports only collaboration/communication between workers and the control board, and not between co-workers. Therefore, it is not possible that an ontology worker will evaluate and exchange argumentation with a co-worker to overcome complex situations on their locally evolving ontology replicas. Argumentation can help the evolution of the shared ontology as it is derived by the locally stored versions, but in DILIGENT each user is not directly aware of the changes another user performs. A result to this, is that some times, a user may possibly redefine a concept some other user already has, which may guide to conflicting ontology versions.

There is a variety of tools for the collaborative ontology development. A brief description is provided for each one in the following list.

1. The OntoLingua tool provides a distributed collaborative environment with World Wide Web services to let users create, modify and manipulate ontologies. An editor allowing this kind of actions is the Ontology Editor. It supports multiple users, utilizing write-only locking and user access levels [5].

2. OntoEdit is an ontology-engineering environment supporting the initial ontology definition, refinement, and evaluation phases. As of the professional edition, it supports a collaborative working environment and ontology libraries [6].

3. WebODE is an advanced ontological engineering workbench for all the major ontology related activities. It allows the easy development of ontology-based applications and it allows integrating ontologies in information systems. It also has multiple facilities for user support by synchronization, authentication and access restrictions per user groups. Finally, it supports collaborative processing [7].

4. KAON is an open-source framework for easy building ontology-based business applications. Two main modules can be detected, i.e. the ontology editor module (OI-modeler) for the ontology implementation and maintenance, and the KAON Portal module for navigating and searching ontologies through World Wide Web browsers. This tool provides multiple user support by concurrent access control with transaction-oriented locking and rollback mechanisms [8].

5. The WebOnto tool developed by the Knowledge Media Institute. The main aim was to develop a user friendly editor for implementing and maintaining ontologies. The tool has collaborative capabilities and supports multiple users through global write-only locking facilities, with change notification [9].

6. Medius Visual Ontology Modeler is a plug-in of Rational Rose Enterprise Edition and therefore it’s a UML-based ontology modelling tool. It supports collaborative and concurrent development through the network-based environment that it provides. It has limited consistency checking capabilities.

7. LinkFactory tool, was designed for very large medical ontologies. It has a 3-tier client-server architecture, i.e. the client application to manage the ontology, the server interface to cope with user requests, and the data layer for accessing the user and maintenance information in the ontology. Features included are: versioning, user tracking, user hierarchies, formal sanctioning with possibility to overrule, sibling-detection, linktype hierarchy [10].

8. The K-Infinity developed by Intelligent Views, is a knowledge editor and it provides broad
support for object-oriented knowledge modeling. The main component of K-Infinity is the Knowledge Builder, which allows users to manage objects and relations as well as relations between objects. It has consistency check and collaborative features.

9. Finally, Protégé is an ontology engineering tool, enabling users to create and maintain ontologies. A uniform GUI is provided, which is composed of overlapping tabs for compact presentation [11]. The available tabs are “Annotation”, “Discussion thread”, “Changes”, “Search” and “Chat”. Annotation tab presents the annotations attached to a selected class in the ontology, in “Discussion thread” tab users may discuss features and specifications of the ontology, “Changes” tab shows a chronological list of all the changes for the selected ontology element, “Search” tab allows users to search in annotations using various criteria, and “Chat tab” provides instant messaging between connected to Protégé server users.

Collaborative Protégé plug-in is the most recent effort towards the collaborative ontology engineering. The main features presented [11] are the annotation of ontology components and changes, the change tracking, filtering and search, chat interface, and extensibility through API access. According to [11] users can add notes or annotations of different types associated to the ontology components or change events. Protégé also provides the ability to create discussion threads attached to the ontology as a whole. Also, Collaborative Protégé records all actions made by a user and stores them in a structured log. This change tracking log contains information about who and when made a change in the ontology. Filtering and Search features allows users to filter and search in each of the collaboration frameworks using a variety of criteria, i.e. author, date, annotation text, and annotation type. The chat interface feature allows connected users to a Protégé server to discuss and exchange real time messages. This interface provides the users also the ability to send links to entities in the ontology, to provide a direct access to that entity for the user who receives the message. This plug-in however, does not provide conflict resolution, i.e. the users must manually discover the actions that caused a conflict, even more, it cannot provide available solutions to overcome the conflict.

On the other hand, it has been done an effort to introduce distributed database techniques in collaborative ontology engineering [12]. This approach locks segments of ontologies, denying or granting access to selected users. This can be very useful to eliminate conflicts in ontology engineering, but it is clearly not a collaborative approach. It makes impossible and useless the user interaction and argumentation, since only a few or only one at a time can have access on specific segments of the ontology. Even more, we could state that techniques used in distributed database systems, are not applicable in ontology engineering, because of the different orientations. That is, in distributed database systems we require that all users must operate transparently, i.e. each user modifying a record in the database must feel like he is the only one connected to it. Actions of other users, have to be effect-less on his actions. Obviously, this is exactly the opposite of what a collaborative ontology engineering tool should achieve. Every user, not only must be aware of the actions of other users, but also the reasons that forced the users to act must be provided. Furthermore, argumentation is a very important aspect required for effective collaboration, and provides the preconditions to reach an agreement.

1.2 Selected Underlying Technologies

A review on the available technologies to build our framework, helped towards the decision for the more appropriate ones to be used. First of all, a methodology to co-ordinate participating members should be defined. This shall provide the overall context and requirements for the engineering of ontologies: We decided to use a human-centred methodology, since this shall provide the most demanding and challenging requirements, which would not prohibit using
the developed tool in the context of more “strict” contexts. Communication, document distribution, and consistency between document replicas were the issues to be solved. We also needed to guarantee that semantics of ontology specifications will not be violated: In contrast to common documents developed in collaborative settings, ontologies contain semantic properties that by any means should not be violated. A brief introduction of the involved methodology and of the technologies used is given in this section, while a more detailed view is given in section 2.

1.2.1 HCOME

The ontology development in a distributed environment is to be studied as a continuous process for continuously reforming ontologies, evolving towards achieving the collaborative parties' goals. Under this scope, participants must first agree on the initial specifications and the resources that set the base for developing an ontology. All the participants must know what are the use-cases/requirements and scope for the ontology to be constructed. Ontology development has to be supported through the entire life-cycle, not only during its implementation.

Ontology workers in collaboration with engineers and co-workers must be able to revise the implemented ontology and redefine concepts as required. Therefore, it is important that the selected methodology will allow every participating member to comment, evaluate or modify the ontology, assuring its active participation in the ontology life-cycle. We selected HCOME, as this is the methodology which is human-centric, i.e. it provides the important role for ontology development to every participant. This also provides the ability for any group of workers to develop an ontology in an unsupervised way, without any central knowledge engineer panel to filter the workers' changes. We decided to use the HCOME methodology, as a supervisor panel would be a bottleneck constraining the generic deployment of the framework. However, such a panel does not prohibit the use of the developed tool. Nevertheless, such a tool would be of great help for the engineers to decide on proper ways to merge workers’ conflicting views.

Maintaining a generic approach we can argue that our framework can be used by any methodology. Provided the generic implementation, a developer can implement the methods required to adapt to any specific methodology.

Another important issue towards the selected methodology was that there must be tools to fully integrate the methodology, so that the framework should not be biased towards specific settings. Our framework is planned to be used as the basis towards developing plug-ins for any desired ontology editor/engineering tool.

1.2.2 Telex

Telex provides a generic technology for distributed parties to collaborate towards reaching agreement to the construction of a specific artifact (a document). Collaborators access Telex via their applications, which are responsible for the local maintenance of documents’ consistency.

As in any distributed environment, it is required that the communication and replication layers to be developed. Also, consistency between replicated data is essential for the system's stability. The document that users modify must be always valid as far as its contents are concerned: For example, when a user deletes some data, the rest of the community must be informed so as to share a consistent view of the document. In contrast to the old-fashion centralized systems, where users modify a shared document, Telex aims to make distribution more efficient. For example, in the former case, when a user deletes data in the shared document, the rest of the community could do nothing but to comply with this action. Telex provides the
required mechanisms for document distribution, and simultaneous document access, without applying locking mechanisms. It distributes the under-revision documents among the collaborating parties in an optimized manner. Provided the generic methods for document distribution and site communication, developers can focus on their application development.

Telex also provides the mechanisms to detect conflicts among participating actions and provide alternative solutions. In a collaborative environment it is possible that some users make conflicting decisions even though they have the same goal. This situation can guide to crucial deadlocks, if users do not detect the conflict at the proper time. Therefore, the reason Telex was selected, is that it does not only provide the distribution and communication mechanisms, but also covers conflict detection and analysis. Using this framework, it is possible to trace the whole history of users’ changes in the distributed document, and especially, the changes generating a deadlock. Information that can be derived is not only the history of the changes performed, but also who was the user responsible for each modification. This can guide argumentation to specific issues raised in the community, and helps find a way to consensus between the related to the conflict participants.

A conflict is defined under the scope of constraints among user performed actions on the document. Telex requires the specification of application-defined actions. Each application developed on Telex, must somehow specify what actions the users are capable to perform, i.e. what are the elements Telex must schedule. The specification of what is an action, is left for the application developers to do, in the way it fits the application requirements. Concerning ontologies, information stored in an ontology must obey specific formal rules, according to the semantics of specifications. Therefore it is not only consistency on document replication that is required, but also consistency for the information stored in each document (which result from the performance of specific actions). At the moment our framework was designed, there was no such a technology for providing ontology distribution and conflict detection among the actions of distant ontology developers. Telex provides a generic technology for this. Alas, we had to find the proper definition and specification for the constraints, which would guarantee that no semantic violation would follow.

### 1.2.3 Validity rules

Provided that our framework is to be built using Telex, we had to preserve that the ontology semantics will not be violated at any case (i.e. by any user action). Constraint types in Telex were implemented as general purpose constraint types between actions performed on any document, and not having ontology semantics in mind. It was therefore required that the gap between semantic specifications in ontologies and Telex constraints to be bridged. This requirement, introduced the use of validity rules as described by [13]. The general idea of validity rules, is to determine all the invalidities that some changes on the ontology could cause, using a formal well-specified validity model. This method encloses the ontology specifications in First Order Logic under the closed world assumption, so that semantics violations are easily noticed. This supports assuring the consistency of the produced ontologies (ontology versions). However, construction of the rules is a process to be done with caution, as under/over specified rules will definitely harm framework's performance.
1.3 A brief outline of the work

In this effort, we utilize the HCOME methodology, to focus on the active participation of the members of the group themselves. In the example specified above, Bob and Adam are the only persons involved in the ontology development. There is no panel of expertise to guide the users and evaluate their efforts. Also, it is not required that the users should be experts themselves in developing an ontology. Through the phases of HCOME, participants specify requirements, comment, evaluate and propose changes done or to be done on the ontology. In the context of this methodology, our framework can provide the functionality for making the ontology development easier/safer. Telex as proposed above, will replicate documents across the sites and maintain consistency among them.

In section 2 we will provide the related to our project background knowledge. A brief description of HCOME methodology will be provided. Telex and validity rules are to be also described, and required definitions will be provided. In section 3, a more detailed view on the motivations and the problem specification is to be given. Typical use scenarios will be also introduced, to help understanding the mechanisms of our framework, as well as the requirements to be covered. Section 4, provides the functional and non-functional requirements and assumptions made on the design of the framework. Also, the use of Telex and validity rules in the context of HCOME will be described. Proposed system design, and technologies used to maintain local consistency, as well as the reasons that forced specific decisions on design will be presented in section 5. Next section 6, shows the system implementation, and the framework's functionalities are demonstrated through the presented in section 4 use cases. Finally section 7, provides the concluding remarks, as of the innovation our framework introduced, further development and a brief evaluation on Telex and the potential of Telex technology.
CHAPTER 2 - BACKGROUND KNOWLEDGE

2.1 Human-Centered Ontology Engineering Methodology

Here we introduce the Human-Centered Ontology Engineering Methodology (HCOME) for the development and evaluation of evolving ontologies in a distributed and collaborative setting. This methodology aims to empower knowledge workers to continuously manage their conceptualizations, enabling their active participation in ontology development. What is most common in the methodologies proposed, knowledge engineers develop the ontologies and provide them to knowledge workers and users, who actually use them. An ontology though is the formal specification of the concepts appearing in a specific domain. However, knowledge workers, may need to redefine or introduce ontology elements (concepts or relations) during the ontology’s lifetime. An ontology therefore, is a continuously evolving form of knowledge related to the specific domain. This fact shows that knowledge workers are the first to confront cases of deficient conceptualizations, as they actually evaluate ontologies. However, for knowledge workers to be empowered to develop ontologies, they must participate actively in the ontology engineering processes in collaboration with knowledge engineers.

As presented in [4], HCOME provides the ability for collaboration between knowledge workers and knowledge engineers. This methodology provides clear distinction among the phases of the ontology life-cycle, and what are the tasks to be performed in each phase. The summary of these tasks is presented in the following table 1, as adapted from [4].

<table>
<thead>
<tr>
<th>Ontology life-cycle phases</th>
<th>Goals</th>
<th>Tasks</th>
</tr>
</thead>
</table>
| **Specification**          | Define aim / scope / requirements / teams | • discuss requirements (S)  
                              |       | • produce documents (S)  
                              |       | • identify collaborators (S)  
                              |       | • specify the scope, aim of the ontology (S)  |
| **Acquire knowledge**      |       | • import from ontology libraries (P)  
                              |       | • consult generic top ontology (P)  
                              |       | • consult domain experts by discussion (S)  |
| **Conceptualisation**      | Develop & Maintain Ontology | • improve (P)  
                              |       | • manage conceptualisations (P)  
                              |       | • merge versions (P)  
                              |       | • compare own versions (P)  
                              |       | • generalize/specialize versions (P)  
                              |       | • add documentation (P)  |
| **Exploitation**           | Use ontology | • browse ontology (P)  
                              |       | • exploit in applications  |
| **Evaluate ontology**      |       | • initiate arguments and criticism (S)  
                              |       | • compare others’ versions (S)  
                              |       | • browse/exploit agreed ontologies (S)  
                              |       | • manage the recorded discussions upon an ontology (S)  
                              |       | • propose new ontology versions by incorporating suggested changes (S)  |

Table 1. HCOME phases in ontology life-cycle

The tasks shown in table 1 are iteratively performed until consensus has been reached.
between participants. Users, either individually or in groups, perform tasks related to each phase in order to achieve specific goals. In case tasks are performed individually, it is assumed that these tasks are performed in ontologies in the worker's personal space. Working collaboratively, HCOME requires a shared information space, so that the tasks are performed by the group in ontologies existent in this space. Tasks related to the personal space, are marked by “P”, while tasks related to the shared space are marked by “S”. Any worker can initiate any task in the personal or shared space, or participate on active tasks in the shared space. In this work we emphasize on tasks performed in the shared space. Indeed, establishing a distributed environment among workers/engineers, the shared space does not exist in a centralized system. Tasks performed by knowledge engineers and workers, are activities that every collaborative individual is aware of.

The three phases the methodology consists of are the phase of specification, conceptualization, and exploitation.

2.1.1 Specification phase

In this phase, knowledge workers form teams for developing the ontology, discuss the ontology requirements, provide specifications and agree on the ontology scope. As shown in table 1, all of the tasks in this phase are performed in the shared space. That is, every participant must be aware of the discussion done on requirements and specification of the ontology aim and scope, producing documents that are made accessible to every teammate.

It is obviously important that in order the workers to finally reach an agreement, they first need to agree upon the target, i.e. what their final ontology should provide. This agreement is the initial agreed reference on how the workers conceive the domain, and how they desire to model it. However, if required, they can obviously redefine parts of the initial reference, and agree on a new modified one. If the participants agree on a target, they can start discussing the specification requirements. Documents produced by this process are to be used to ease the introduction of new participating members, and serve as reference during argumentation.

2.1.2 Conceptualization phase

If users reach an agreement on the aim, scope and requirements of the ontology, they can proceed to the conceptualization phase. In this phase, the main goals are knowledge acquisition and ontology development. As shown in table 1, tasks in this phase are done in the personal space of each worker. In our approach though, although workers may produce/change specifications in their personal space, each one is aware of the modifications performed by the others. Therefore, in this case there is not a personal space that “isolates” workers. Even if some worker modifies the ontology version offline, the others will get to know the changes this worker made after he/she gets connected.

Tasks performed in this phase, as shown in table 1, are done in the personal space: Users can integrate concepts, provide new definitions, merge or modify existing ontologies. Users can import existing ontologies to reuse conceptualizations, or develop “from scratch” formal ontologies, according to the workers' perception on the domain. Finally, it is possible to compare different ontology versions for tracking evolution, and merge or map selected versions to achieve better clarification of the concepts related to the domain.
2.1.3 Exploitation phase

The third phase is related to the ontology exploitation. This phase provides an important feedback for the conceptualizations formed in any ontology version in the shared space, in the context of any task that this ontology aims to support. As shown in table 1, this phase comprises tasks related to the evaluation and use of agreed ontologies either in the personal or in the shared space. Further development by the knowledge workers is affected by argumentation dialogues in the shared space, and evaluation/exploitation/inspection of any ontology version in the personal space. Discussions are initiated and managed, to derive further improvements if necessary. Workers may incorporate changes in the ontology and upload these new versions to the shared space, following the train of the argumentation dialogue. Workers may also compare different versions of an ontology, to identify the differences between these versions, something that will allow them to focus on specific dis-agreements so as to reach to agreement through dialogue.

2.2 HCOME, a tool supporting HCOME

To show the effectiveness of HCOME, the prototype ontology engineering environment HCOME (Human Centered ONtology Engineering Environment) was implemented. This environment takes into account all the important tasks in the HCOME phases, to support the human-centered ontology engineering. HCOME's architecture, is shown in the following diagram 1, as adapted from [4].

Main functions supported by this tool are the following: Specifying ontologies’ elements in workers' personal space, sharing developed ontologies, forming collaboration groups and participating in argumentation dialogues for reaching consensual conceptualizations. The ontology versions upon the participants have agreed, are stored in a virtual shared space and can be further evolved by workers if necessary. Using HCOME, knowledge workers can apply any available method agreed by the collaboration team for the ontology development. Collaborators outline the purpose and use of ontologies, and construct personal ontologies. They can also...
consult generic ontologies to reuse semantic and ontology-specific organization principles. Management of the developed ontology versions is also available, and differences tracking, or ontology merging are useful functionalities provided. Finally, HCOME provides the ability for argumentation and sharing of ontology versions between collaborative members. For this purpose, collaborating parties may propose specific versions that are uploaded in the shared space, in conjunction with argumentation items supporting the changes made. These changes are further evaluated and criticized by others that may agree or provide arguments against them, and so on, until they reach an agreed version of the ontology.

2.2.1. HCOME Specification phase

As shown in [4], the specification phase of HCOME is covered by the SharedHCOME part of the HCOME environment. It provides the means to construct collaborative groups, as well as the means for participants to specify the scope and requirements of the target ontology. It also informs by e-mails community members about new developments in the shared space. Each member of a group by definition of being a member, is assumed to agree on the importance of the new ontology, and takes part in the argumentation dialogue. Discussed issues concern the requirements, scope and aim of the new ontology. The purpose of this dialogue, is to develop specification documents for new members and for making the existent group members reach an agreement on the specifications of the target ontology. During this phase, any participant can raise issues, propose solutions and pose arguments against or for a solution. The dialogue is represented in the form of a threaded discussion. The argumentation follows a version of the IBIS model as introduced by [14]. This model is based on three main abstractions, that is the issue, which represents a problem to be solved, the position, which can be either a solution to an issue posed, or a new version of the studied conceptualization, and the argument, which is either supporting or objecting a position set. The approach of HCOME to IBIS model is that some positions or versions of an ontology provide a solution for an issue raised. Each version introduced can be supported by arguments. A position or version introduced to support an issue can either generalize or specialize another issue, which can also be raised by another argument. However it is not necessary to directly connect every argument posted with an existing position or a position with an issue. Arguments can be posted without having to support or object a position. Also, a position or version does not have to solve an issue.

2.2.2. HCOME Conceptualization phase

The conceptualization phase of HCOME is fully supported by HCOME. The environment provides the functionality to the members in the collaboration group, to bring ontology versions from the shared space, to their local space. Once the selected ontology version is replicated in the user’s personal space, the ontology can be evaluated and new versions can be produced. New ontology versions can afterwards be published in the shared space, so that other users can download them and evaluate or re-use for the next ontology versions to follow. HCOME also provides useful functionalities, for merging or refining ontologies, and for consulting lexicons or generic ontologies [4].
2.2.3 HCOME Exploitation phase

As described by the HCOME methodology, each team member can share with the rest of the community ontology versions developed in his/her personal space. To do so, the worker has to move that version in the shared space, posting this as a position in the (maybe new) argumentation dialogue. The SharedHCONE provides the functionality to post the new version and make possible the rest of the community to be informed. As shown in diagram 1, SharedHCONE, provides also the functionality to produce an agreed ontology from a shared one, through the argumentation dialogue. As it has been mentioned, the argumentation model, is an implementation of a version of IBIS model, described in [4].

2.3 Telex

Telex is a framework that enables collaborative and distributed development of documents. A document may be any specification, textual or not, formal or informal. Telex has been designed to allow the distribution of evolving documents in parties across computer networks of any scale. It also performs an optimal document distribution, replicating any document only to the users it has to. Users can modify their documents either online or offline, while Telex takes the responsibility to inform other participants about changes made, possible conflicts and possible solutions to these conflicts. In such a distributed setting, it is assumed that each user is working in a distinct site, i.e. a workstation, that is unique.

Each site maintains an Action-Constraint Graph (ACG). This graph consists of nodes and arcs, where nodes are also called actions and arcs are constraints. The actions represent operations that users may apply to a document via the application that they use. Arcs in the graph are constraints between these application operations. Constraints provide the ability to serialize and avoid conflicts between actions, so that each graph is a consistent schedule, i.e. a set of actions that can be performed in series without raising a conflict.

Each application that is using Telex, is responsible to locally test (i.e. in the site of a user) the consistency of the actions performed on any document, provide the specification of these actions, i.e. of the operations are to label the ACG nodes, and set the constraints that must hold between these actions. The application (i.e. each local site) sends to the Telex fragments containing actions and constraints holding among these actions. In response, Telex, being aware of all actions being performed at any site, reconciles dis-agreements between sites (by exploiting the constraints between actions) and proposes schedules of actions that can be executed by the application.

2.3.1 Telex Related definitions

As mentioned above, the application sends fragments of actions and constraints to Telex. These data are added in the ACG by the addFragment method, called by the application. The scheduler computes schedules from the ACG, and when required, the getConstraint is called to get further constraints between newly inserted actions from the application and the already existing in the ACG.

Each conflict detected triggers the generation of more schedules when possible, and the application can execute one or more of these schedules. One schedule may also be selected as the agreed proposal between the collaborating sites. Telex also provides the functionality of cross-document constraints, to maintain consistency over multi-document modifications and cross-application use cases.
A schedule is a set of non-conflicting and non-empty sequence of actions, therefore Telex provides a high-level view of conflict resolution. How the conflicts will be presented to the user and which schedule will be selected, is left to be implemented by the application. Telex provides the information, and the visualization and exploitation is application specific. The schedule selected by the application (i.e. the user) is taken as input for the agreement module of Telex, and it is being used for further development of schedules.

To constrain actions, there are three primitive constraints as shown in the following table 2 as adapted from [15].

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>Meaning for scheduling</th>
<th>Meaning for agreement</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotAfter</td>
<td>A → B</td>
<td>A is never after B in any schedule</td>
<td>A is not after B in agreed prefix</td>
</tr>
<tr>
<td>Enables</td>
<td>A ≼ B</td>
<td>B in a schedule implies also A in the same schedule</td>
<td>B commits implies also that A commits</td>
</tr>
<tr>
<td>NonCommuting</td>
<td>A ↆ B</td>
<td>N/A</td>
<td>Conflict: Agree either on A→B or B→A</td>
</tr>
<tr>
<td>Atomic</td>
<td>A ⊆ B</td>
<td>Either both A and B execute or nothing</td>
<td>A,B both commit or both abort</td>
</tr>
<tr>
<td>Casual Dependence</td>
<td>A ≺ B</td>
<td>B executes after A and if only A succeeds</td>
<td>B commits implies also that A commits</td>
</tr>
<tr>
<td>Antagonism</td>
<td>A ⇔ B</td>
<td>Conflict: A and B never in the same schedule</td>
<td>A commits implies B aborts and vice-versa</td>
</tr>
</tbody>
</table>

Table 2. Constraint Types provided by Telex

Two schedules are said to be equivalent (S1 ≡ S2) if only they contain the same set of actions, every action has the same status in both schedules, and non-commuting pairs of actions execute in the same order.

The constraint \textit{NotAfter} in table 2, indicates that a scheduler must maintain an ordering between the actions \(A\) and \(B\), such that no schedule may execute \(B\) before \(A\). Any schedule which executes neither \(A\) or \(B\), only \(A\) or only \(B\), or \(A\) and \(B\) in the defined order, adjacent or not, is correct with respect to this constraint. The constraint \textit{Enables} is an implication between the corresponding actions. If \(B\) executes in a schedule, then \(A\) must also execute in that schedule. However, there is no ordering between actions. A schedule executing only \(A\), or neither \(A\) or \(B\), is correct with respect to this constraint. In case that \(A\) is aborted, i.e. does not execute, \(B\) is also aborted. Combining these constraints, \textit{Atomic}, \textit{Casual Dependence} and \textit{Antagonism} constraints are generated. \textit{Atomic} constraints, as shown in table 2, relate two actions in a way that either both actions execute, or none. The \textit{Casual Dependence} constraint defines that action \(B\) depends on the result of \(A\), and is executed iff \(A\) succeeds. If \(A\) is aborted for some reason, then \(B\) also is aborted. Finally, the \textit{Antagonism} constraint identifies a conflict between actions \(A\) and \(B\). This constraint defines that when action \(A\) is executed in a schedule, \(B\) must be aborted and vice-versa. A set of actions \(K\) is said to conflict, if the actions in \(K\) form a “→” (NotAfter) cycle. This means that no sound schedule can execute all the actions in \(K\). The \textit{Antagonism} constraint is a special case of such cycle between the actions \(A\) and \(B\).

Constraints are used to finally decide the status of each appearing action in ACG. An action can be either \textit{Guaranteed}, \textit{Dead}, \textit{Serialized}, \textit{Decided}, or \textit{Stable}. \textit{Guaranteed} actions are those that execute in every schedule. That is, for a multilog \(M\), the set of guaranteed actions \(\text{Guar}(M)\), and the set of actions \(A\):

\[
\forall \beta \in A : \text{If } \alpha \in \text{Guar}(M) \text{ and } \beta < \alpha \text{ then } \beta \in \text{Guar}(M)
\]

In contrast to the above, \textit{Dead} actions, do not execute in any schedule. That is

\[
\forall \alpha \in A : \text{If } \beta_1 \rightarrow \ldots \rightarrow \beta_m \in \text{Guar}(M), \text{ and } \alpha \rightarrow \beta_1 \rightarrow \ldots \rightarrow \beta_m \rightarrow \alpha, \text{ then } \alpha \in \text{Dead}(M)
\]

and

\[
\forall \alpha \in A : \text{If } \beta \in \text{Dead}(M) \text{ and } \beta < \alpha, \text{ then } \alpha \in \text{Dead}(M).
\]
A serialized action is one that is ordered with respect to all non-commuting actions that execute. That is

\[\text{Serialised}(M) = \{\alpha \in A | \forall \beta \in A, \alpha \leftrightarrow \beta \Rightarrow \alpha \rightarrow \beta \lor \beta \rightarrow \alpha \lor \beta \in \text{Dead}(M)\}\]

According to the above, an action is decided once it is either dead, or both guaranteed and serialized. That is

\[\text{Decided}(M) = \text{Dead}(M) \cup (\text{Guar}(M) \cap \text{Serialised}(M))\]

Finally, an action is called stable if its effects cannot change. This means that the action is either dead, or guaranteed and serialized, and all preceding actions are also stable. That is

\[\text{Dead}(M) \subseteq \text{Stable}(M),\]

and

\[\text{If } (\alpha \in \text{Guar}(M) \cap \text{Serialised}(M)) \land (\forall \beta \in A : \beta \rightarrow \alpha \Rightarrow \beta \in \text{Stable}(M)) \text{ then } \alpha \in \text{Stable}(M).\]

It is important to note that setting constraints among actions is critical to the performance of the application. Commuting actions bypass the expensive mechanisms. Antagonistic actions are also preferred since they are resolved by the scheduler. Non-commuting actions are the worst due to performance, as the case is left for the application at run time. Conflict constraints are not strictly necessary for safety. If conflict constraints are not provided, alternative schedules will not be generated. Using constraints Telex avoids this kind of dead-ends.

As mentioned above, the ACG is built by the actions and constraints set by sites. An ACG can only grow, although certain parts can be garbage-collected when decided. Each node in ACG is an action which can be rendered either white or black. Arcs between nodes are labeled by primitive constraint types NotAfter or Enables. A sound cut, is a conflict-free sub-graph, where

I. every node is either white or black
II. nodes connected by || and \(\rightarrow\) arcs in the white subgraph are ordered, and consistent with \(\rightarrow\)
III. if a white node is in the cut and \(A \prec B\), then \(A\) also appears in the cut and rendered white.

Given the definition of a sound-cut, a sound schedule is a totally-ordered sound cut.

Because ACG replication may differ from site to site, it is sometimes required to identify if the local ACG contains sufficient information. This sufficiency of information is captured by the notion of closure. A node \(A\) is enables-closed if all the predecessors of it by \(\prec\) (Enables) arcs known in any site, are also in the local ACG. Similarly, a node \(A\) is said order-closed if every predecessor of \(A\) by \(\rightarrow\) arcs, and all its adjacent by \(\parallel\) edges nodes known in any site are also in the local ACG, and transitively this is true for their predecessors.

When adding a new action, it is important to ensure that every conflict is considered. To ensure this, agreement requires that a proposal is order-closed. Even more, since Telex provides optimistic execution, a proposal may depend on tentative state, because of Enables constraints. It is obvious that such a proposal cannot be committed until the tentative state is committed. Therefore, agreement must also ensure that a decision is Enables-Closed.
2.3.2 Telex architecture

In general, Telex comprises four main modules. As shown in the following diagram 2 adapted from [15], these modules are the Scheduler, the Agreement module, the Logger and the Transmitter.

![Figure 2. Telex architecture](image)

As mentioned above, each application site provides fragments of actions and related constraints to Telex. However, Telex may also receive fragments from remote sites via the Transmitter module. Conflicting operations, as defined by the application constraints, are set as “antagonistic”. Detecting these cases, Telex ensures that a conflict will not be allowed to happen, separating those actions in different schedules. This task is performed by the scheduler, providing conflict-free sets of actions.

The modules logger and transmitter are also called auxiliary, as their role is to support the scheduler and the agreement modules. It is important to be mentioned that each module is instantiated per each document that is locally “open”.

As shown in diagram 2, the application is a “middleware” between the user and Telex, and it has to implement at least the methods that send fragments to Telex, propose a schedule to the community, and fetch schedules produced by Telex. The fragments and proposals are transmitted via the related module to remote sites, and received by those sites via their transmitter module.

Telex sites communicate using the commitment protocol [16]. First, the transmitter determines the set of sites that operate on the document it is instantiated for, and provides an Atomic Multicast service.

The other auxiliary module, the logger, is responsible to keep a local copy of the shared document. In this copy, the logger stores fragments it receives from the scheduler, as requested by the local or remote users. This is done to permit users modify the document while being in off-line...
status. The data structure used for the persistent storage of documents is a set of per-user logs, also called as multilog. The user's log stores the actions and constraints that she/he sets. The fact that there is a single writer, an append-only, per-user log, prevents write contentions. A document is a directory structure of files, hidden from developers and users. As defined in [16], multilog is a quadruple \( M = (K, \rightarrow, <, \|) \) specifying a graph. Vertices \( K \) in the graph are actions, and \( \rightarrow, <, \| \), are three types of edges representing constraints between actions, NotAfter, Enables and NonCommuting respectively, described in 2.3.4 paragraph of this section. Users submit actions to local site. Current knowledge of site \( i \) at time \( t \) is the distinguished per site multilog \( M_i(t) \), which grows over time while logging users’ modifications.

The ACG is formed by the conceptual union of all documents across the sites. For every document on each site, a separate multilog \( M_i(t) \) is created, holding the actions and constraint referring to that document. Each multilog contains a set of logs for each collaborating user. When a local update occurs, the system replicates every log to other collaborating sites. This happens to sites requesting the document, and Telex stores it on their local disk to allow off-line modification. In that case, the system logs appended fragments, while being offline, and publishes these fragments to collaborating sites next time it is connected again. The isolated site also retrieves data of modifications performed while it was offline, from remote sites by the time it is reconnected. The most important modules though, are the scheduler and agreement modules. The scheduler is responsible to maintain the in-memory ACG that represents the state of the document at the local site. Also, it is responsible to compute a set of sound schedules from the ACG. These schedules are proposed to the application. There are three cases where actions and constraints are added in the document:

a) the local user updates the document with a new fragment
b) the logger receives an update issued by a remote user
c) the agreement module commits a schedule.

Although the set of actions and constraints are the same for all the sites, it is possible that different sound schedules are developed in some sites due to some non-deterministic decisions that Telex has to take at any site. In such cases, the agreement module is responsible to make sites agree on a common schedule to apply, and achieve mutual consistency. A proposal contains actions to be committed, aborted or serialized. Because of the asynchronous communication, proposals may be different. The agreement module is responsible though to make sites reach an agreement, on a schedule prefix. This module operates in four asynchronous phases:

1. Each site computes a proposal.
2. The transmitter multicasts proposals to contributing sites
3. The transmitter forwards proposals it receives to the agreement module.
4. According to the commitment algorithm the reconciler selects a proposal, and logs it.

It is possible though that some applications require the selection of schedules based on application specific criteria and/or policies. Telex provides the freedom to developers to implement the commitment algorithm they need. The basic algorithm relies on voting, so that local user preferences are to be taken into account.
2.3.3 The Telex Scheduler module

When a new fragment is received, the updated ACG triggers modifications on schedules, or the scheduler adds new schedules in the set. In case of conflict occurrence in ACG, the scheduler provides several schedules, each resolving conflicts differently. The application may select one of these schedules to proceed. Telex will attempt to add new actions in the selected schedule, preferring to extend the schedule, rather than removing actions already in it.

A quality metric is used to let Telex sort the schedules and present them to the application. The scheduler computes each schedule on demand, since computing all available schedules is resource and time expensive. The computation of the optimal schedule requires the computation of the optimal acyclic ACG sub-graph, which is a NP-complete problem. To overcome this bottleneck, Telex utilizes heuristics similar to IceCube [17].

Specifically, an action in a schedule may be rendered either as a white node, or a black one. In the first case, a white node represents an action to be executed, while a black one represents an action that should be ignored. The IceCube heuristic gives higher values to schedules having more white actions, while penalizes schedules having black actions.

A schedule is kept as long as it remains valid. A schedule is said to be invalid, if there is at least one action having a different color in the schedule and in ACG. In that case, the union of the schedule and the ACG is not sound, because the aforementioned action is both committed and aborted.

The scheduler maintains a stack of schedules, and before computing a new schedule, it pops invalid schedules. The first valid schedule in the stack is the currently working schedule.

The decision algorithm is presented in [10]. For this to function, certain assumptions need to hold: It is assumed that when an action $\alpha$ is in $K_i$, i.e. the set of known actions at site $i$, every constraint such that $\beta \prec \alpha$ and $\beta \rightarrow \alpha$ for any $\beta$, are known at $i$. Also, each action in $A$, i.e. the set of available actions, is eventually submitted in a known site. Every action $\alpha$, is also assigned to a unique primary site, $P(\alpha)$. It is therefore assumed that two actions commute iff they have different primaries. As defined above, conflicting actions are represented by $\rightarrow$ cycles. Finally, it is assumed the function $\text{victim}(c)$, which deterministically chooses one action from a subset of actions $C$.

The steps of the decision algorithm for an action $\alpha$ at site $i$, are as follows [18]:

1. Through communication, $\alpha$ is eventually known at its primary site $i$, $P(\alpha)=i$.
2. The propagation module at site $i$, communicates with other sites discovering all $\beta$ such that: $\alpha \diamond \ldots \rightarrow \beta \lor \beta \rightarrow \ldots \rightarrow \alpha$. The action becomes ready.
3. For each cycle $c$ of $\rightarrow$ involving $\alpha$, if $\text{victim}(c) = \alpha$ then decide $\alpha$ is dead (e.g. add constraint $\alpha \rightarrow \alpha$) and exit.
4. Partition all $\beta$ such that $\alpha \triangleright \beta$ into subsets $M$ and $M'$, according to the following property: actions in $M$ are such that $\beta \triangleright \alpha$ and those in $M'$ are not.
5. Wait until: either some action in $M'$ is known to be dead; or all actions in $M'$ to be guaranteed. In the first case, $\alpha$ in now known to be dead; exit. In the second case $\alpha$ is guarantee-able.
6. To all actions in $M$ send a message saying that $\alpha$ is guarantee-able.
7. Wait for either some action in $M$ to be known to be dead, or for all actions in $M$ to be guarantee-able. In the first case $\alpha$ is known to be dead; exit. In the second case decide $\alpha$ is guaranteed (e.g. add the constraint $INIT \triangleright a$).

8. The final execution order of $\alpha$ is given by its $\rightarrow$ relations. Wait for all $\beta$ such that $\beta \rightarrow \ldots \rightarrow a \land P(\alpha) = P(\beta)$. Execute $\alpha$ after all such actions that are guaranteed.

2.3.4 The Telex Agreement module

It is likely to happen that replicated documents in different sites, are not the same. However, it is required that all the documents are consistent. That is, all sites converge to a final schedule. The definition of convergence is based on the stable common prefix (SCP). Let us assume the global union of all locally maintained ACGs. This union provides a set of sound schedules, which may also be empty. The stable common prefix is the intersection of all these schedules. SCP is then sound, if any action in it has the same color in every local conflict-free subgraph it appears, and non-commuting committed actions are ordered in the same way across the sites. The order of non-commuting committed actions is called serialization order.

It is very important that collaborating sites eventually will reach consensus, therefore every not aborted action in local ACGs sometime will become part of the SCP. Initially, SCP and ACGs are empty. Collaborating sites exchange proposals generated from their local ACGs. The agreement protocol ensures that some sites agree on the next segment to be appended to SCP. Although it is not required that all the sites must participate to take such a decision, the protocol ensures that independent decisions commute and SCP remains sound. Because of asynchronous communication, proposals from sites may differ. The agreement protocol must combine the proposals to define what the next extension of SCP will be. In case of conflicting proposals, a consensus protocol is required. System wide consensus should be avoided, ensuring that only sites involved in the conflict should participate in the protocol.

Each user proposes the next extension of SCP according to preferences set on the application. The proposal itself is the selected schedule which fits best user's preferences. The committed actions in the proposed schedule are interpreted by the agreement module, as commit proposal, aborted actions as abort proposal, and ordered non-commuting actions as proposal to serialize. To achieve this, a set of constraints to ensure proposed status of actions is added to all ACGs. This will grant that all successor schedules will be consistent with taken decision.

The implemented agreement protocol in Telex ensures scalability, as it reaches consensus without relying on total-order broadcast over the whole system. It also attempts to merge the proposals sent by different sites. The protocol is executed in two steps.

The first step is to close a proposal. Let us assume that a site $i$ proposes proposal $P$. Also, $replicas(P)$ is the group of sites is the group of sites where documents mentioned in $P$ are replicated. Telex sends $P$ in a Reliable multicast to $replicas(P)$, that is with guarantee that all sites in the group will eventually deliver every message in the group. When each site broadcasts $P$, the agreement module checks if $P$ forms a sound cut locally. If this does not happen, i.e. some actions are missing, those actions are added in $P$ and it is sent by Reliable multicast to the group. However this process does not modify the site $i$ which originally proposed $P$.

The second step, is to make a decision. TelexAgreement protocol maintains at each site $i$, the last proposal it received by $j$. This step decomposes a proposal into candidates and starts a voting procedure. Each candidate proposal is provided by the previous step. A candidate is elected
if every site in replicas(P) has voted the candidate and it is also valid. If this happens, the
candidate is appended in ACG. In case that a site is disconnected, it is allowed another site to
propose on behalf of the disconnected site.

2.4 Ontology Dynamics via validity rules

In order to apply Telex in the domain of ontology engineering, actions and their
constraints must be defined, in accordance to the semantics of ontology specifications. For this
task, we use the validity rules presented in [13]. As a first step, it is required that the framework
used for the ontology resources description to be mapped to First Order Logic. The semantics of
specifications are captured via the validity rules specified, which introduce a validity model. The
representation of ontological facts as First Order Logic predicates is shown in the following table
2, as adapted from [13].

<table>
<thead>
<tr>
<th>RDF triple</th>
<th>Intuitive Meaning</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>C rdf: type rdfs: Class</td>
<td>C is a class</td>
<td>CS(C)</td>
</tr>
<tr>
<td>P rdf: type rdf: Property</td>
<td>P is a property</td>
<td>PS(P)</td>
</tr>
<tr>
<td>x rdf: type rdfs: Resource</td>
<td>x is a class instance</td>
<td>CI(X)</td>
</tr>
<tr>
<td>P rdfs: domain C</td>
<td>domain of property</td>
<td>Domain(P, C)</td>
</tr>
<tr>
<td>P rdfs: range C</td>
<td>range of property</td>
<td>Range(P, C)</td>
</tr>
<tr>
<td>C1 rdfs: subClassOf C2</td>
<td>IsA between classes</td>
<td>C2IsA(C1, C2)</td>
</tr>
<tr>
<td>P1 rdfs: subPropertyOf P2</td>
<td>IsA between properties</td>
<td>P2IsA(P1, P2)</td>
</tr>
<tr>
<td>x rdf: type C</td>
<td>class instantiation</td>
<td>CInst(x, C)</td>
</tr>
<tr>
<td>x P y</td>
<td>property instantiation</td>
<td>PI(x, y, P)</td>
</tr>
</tbody>
</table>

Table 3. Ontological facts as First Order Logic predicates

Validity rules are encoded in the form:

$$\forall \bar{u} P(\bar{u}) \rightarrow \lor_{i=1,...,n} \exists \bar{v}_i Q_i(\bar{u}, \bar{v}_i)$$

where $$\bar{u}, \bar{v}_i$$ are tuples of variables, $$P$$ and $$Q_i$$ are conjunctions of relational atoms of the form
$$R(w_1,...,w_n)$$, and equality atoms of the form $$(w_i = w_m)$$, where $$w_1, ..., w_n, w_k, w_m$$ are variables or
constants. So far, a set of validity rules have been created to support maintaining consistency in
the state (i.e. in the ontology) resulting from the performance of actions, by constraining the
applicability of these actions, signaling inconsistent states that result from their application, and/or
signaling further actions that need to be performed to reach a valid state (i.e. a valid ontology).

For example, the Properties’ domain specification validity rule according to table 2, is the
following:

$$PI(x, y, z) \land \text{Domain}(z, x) \rightarrow \text{C}_\text{Inst}(x, w)$$

This validity rule shows that given an “instance of” a property $$z$$ between two objects $$x$$
and $$y$$, then $$y$$ must belong to the domain $$w$$ of $$z$$. Let us now assume two users who act
simultaneously on the same ontology. Our goal is to exploit such a validity rule to detect any
conflict that may result from the combination of users’ actions. More specifically, if the first user
performs an action (e.g. specify_property_instance(x,y,z)) that has as an effect $$PI(x,y,z)$$ and the
second performs an action (e.g. \textit{specify\_property\_domain}(z,w)) that has as an effect \textit{Domain}(z,w), then none of them violates by itself any validity rule on the local views of the ontology. It is obvious though, that when the actions are to be merged, the resulting state including the facts \(Pl(x,y,z)\) \textit{Domain}(z,w) will not be consistent if \(C\_Inst(x,w)\) does not hold in the ontology.

With the aim to specify and compute invalidities, following \cite{13} the component sets are introduced. The component set of \(c\) with respect to some tuple of constants \(x\) is defined as

\[
\text{Comp}(c,\bar{x}) = \{\neg P_i(\bar{x}) \mid 0 < j \leq k\}
\]

\[
\cup \{Q_{j1}(\bar{x},z) \land Q_{j2}(\bar{x},z) \land \ldots \land Q_{jm}(\bar{x},z) \mid 1 \leq i \leq n, z : \text{const\_tan}\}
\]

where \(P_i(\bar{x}) 0 < j \leq k\) and \(Q_{j1}(\bar{x},z) \land Q_{j2}(\bar{x},z) \land \ldots \land Q_{jm}(\bar{x},z) \lor 1 \leq i \leq n, z : \text{constant}\) are parts of the general form of the aforementioned validity rule. For example, for the rule \(Pl(x,y,z) \land \text{Domain}(z,w) \rightarrow C\_Inst(x,w)\) the derived component set would be \(\{\neg Pl(x,y,z)\},\{\neg \text{Domain}(z,w)\},\{C\_Inst(x,w)\}\}. Each component is a predicate of a validity rule, using the closed world assumption. That is if \(p \nmid q\), then \(p \models \neg q\). Any expression of the form \(P(x_1,...,x_n)\) is called positive ground fact, and the expression \(\neg P(x_1,...,x_n)\) is called negative ground fact.

As proposed in \cite{13}, a ground fact \(P(x)\) which is added in an ontology \(K\), would violate a rule \(c\), iff there is some set \(V\) and tuple of constraints \(\bar{u}\) for which \(\neg P(\bar{x}) \in V\) and \(\bar{V} \in \text{Comp}(c,\bar{u})\) and for all \(V' \subset \text{Comp}(c,\bar{u})\), \(V \neq V'\), it holds that \(K \nmid V'\).

According to the example given above, the two users apply the actions \(Pl(x,y,z)\) and \textit{Domain}(z,w) on their local views. For both of them, the component set, as it is derived from the corresponding validity rule, will be \(\{\neg Pl(x,y,z)\},\{\neg \text{Domain}(z,w)\},\{C\_Inst(x,w)\}\}. Since \(C\_Inst(x,w)\) does not exist in the ontology, i.e. \(\neg C\_Inst(x,w)\) is true, for the user applying an action that has as effect \(Pl(x,y,z)\), the fact \(\neg \text{Domain}(z,w)\) validates the rule and vice versa for the other user. When the two actions are combined though, the rule is violated, given that \(Pl(x,y,z)\) and \textit{Domain}(z,w) hold but \(C\_Inst(x,w)\) does not hold. It is shown that the component set not only detects the conflict, but also exposes the effects of the actions that need to hold (or the facts that must hold in the ontology) to re-establish validity. On the other hand if \(C\_Inst(x,w)\) exists, both local views are consistent, as well as the global view. However, it is obvious that the system cannot “force” users to insert the fact \(C\_Inst(x,w)\) just to keep the validity of the rule before any of \(Pl(x,y,z)\) and \textit{Domain}(z,w) is performed.

The following table 4, presents the set of validity rules used in the pilot implementation of TCOE. The first column provides the name for the rule as required for later reference. The rule component set provides the component set for each validity rule, in the form of sets of predicates. The third column provides the effects of actions that trigger the generation of the Telex-constraint that corresponds to each component set. For example, the constraint corresponding to the first rule “domain applicability” is generated, when the effect of a performed action is “\textit{domain}(x,y)” . Then the constraint shown in the last column is triggered to preserve that this new fact will hold while “\(x\) is a property” and “\(y\) is a class”. If at least one of these fails then “\textit{domain}(x,y)” cannot hold as well.

The last column of table 4 shows the generated constraint from each validity rule. Provided the general form of a validity rule, it is easy to show, that each \(P(\bar{u})\) of a rule, enables and must precedes of all the \(Q_i(\bar{u},\bar{v}_i)\) in the same rule. Therefore, it is a constraint of casual dependence as shown in table 2, such that the components \(Q_i(\bar{u},\bar{v}_i)\) of the validity rule, depend on \(P(\bar{u})\).
As described in [13], the constructs that have been used and the validity rules presented correspond to RDF/S ontologies. However, our method is highly parameterizable, and more expressive languages can also be considered. For the case of OWL-DL the validity rule base has to be updated with rules reflecting the semantics of this language. It is important to be mentioned though that for highly expressive and complex languages, the development of a complete set of validity rules is a difficult task.

<table>
<thead>
<tr>
<th>Rule name</th>
<th>Rule Component Set</th>
<th>Effect of new action</th>
<th>Constraint Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain applicability</td>
<td>R1.1: ( \forall x, y \in \Sigma: \text{Comp}(R1.1,(x,y)) = { { \text{Domain}(x,y) } } )</td>
<td>( \text{PS}(x) \land \text{CS}(y) \land \neg \text{Domain}(x,y) )</td>
<td>( \text{PS}(x) \land \neg \text{Domain}(x,y) )</td>
</tr>
<tr>
<td></td>
<td>R1.2: ( \forall x, y \in \Sigma: \text{Comp}(R1.2,(x,y)) = { { \neg \text{Domain}(x,y) } } )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_isA Applicability</td>
<td>R2.1: ( \forall x, y \in \Sigma: \text{Comp}(R2.1,(x,y)) = { { \neg \text{C_isA}(x,y) }, { \text{CS}(y) } } )</td>
<td>( \text{CS}(x) \land \neg \text{C_isA}(x,y) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R2.2: ( \forall x, y \in \Sigma: \text{Comp}(R2.2,(x,y)) = { { \neg \text{C_isA}(x,y) } } )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_inst Applicability</td>
<td>R3.1: ( \forall x, y \in \Sigma: \text{Comp}(R3.1,(x,y)) = { { \neg \text{C_inst}(x,y) }, { \text{CS}(x) } } )</td>
<td>( \text{CS}(x) \land \neg \text{C_inst}(x,y) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R3.2: ( \forall x, y \in \Sigma: \text{Comp}(R3.2,(x,y)) = { { \neg \text{C_inst}(x,y) }, { \text{CS}(y) } } )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domain is unique</td>
<td>R4: ( \forall x, y, z \in \Sigma: \text{Comp}(R4,(x,y,z)) = { { \neg \text{Domain}(x,y) }, { \neg \text{Domain}(x,z) }, { \neg \text{Domain}(y,z) } } )</td>
<td>( \text{Domain}(x,y) \land \text{Domain}(x,z) \land \text{Domain}(y,z) )</td>
<td></td>
</tr>
<tr>
<td>domain and range exists</td>
<td>R5.1: ( \forall x, y \in \Sigma, z \in \Sigma: \text{Comp}(R5.1,(x,y)) = { { \neg \text{PS}(x) }, { \text{Domain}(x,y), { \text{Range}(x,y) } } } )</td>
<td>( \text{Domain}(x,y) \land \text{Range}(x,y) )</td>
<td></td>
</tr>
<tr>
<td>C_isA Transitivity</td>
<td>R6: ( \forall x, y, z \in \Sigma: \text{Comp}(R6,(x,y,z)) = { { \neg \text{C_isA}(x,y) }, { \neg \text{C_isA}(x,z) }, { \text{C_isA}(y,z) } } )</td>
<td>( \text{C_isA}(x,y) \land \text{C_isA}(x,z) \land \neg \text{C_isA}(y,z) )</td>
<td></td>
</tr>
<tr>
<td>C_isA Inefficiency</td>
<td>R7: ( \forall x, y \in \Sigma: \text{Comp}(R7,(x,y)) = { { \neg \text{C_isA}(x,y) }, { \neg \text{C_isA}(x,y) } } )</td>
<td>( \text{C_isA}(x,y) \land \neg \text{C_isA}(x,y) )</td>
<td></td>
</tr>
<tr>
<td>Determining C_inst</td>
<td>R8: ( \forall x, y, z \in \Sigma: \text{Comp}(R8,(x,y,z)) = { { \neg \text{C_inst}(x,y) }, { \text{C_inst}(x,y), { \text{C_inst}(x,z) } } } )</td>
<td>( \text{C_inst}(x,y) \land \text{C_inst}(x,z) \land \neg \text{C_isA}(x,y) )</td>
<td></td>
</tr>
<tr>
<td>Property instance of and</td>
<td>R9: ( \forall x, y, z \in \Sigma: \text{Comp}(R9,(x,y,z)) = { { \neg \text{P_isA}(x,y) }, { \text{Domain}(x,y) }, { \text{Domain}(x,z) } } )</td>
<td>( \text{P_isA}(x,y) \land \text{Domain}(x,y) \land \neg \text{Domain}(x,z) )</td>
<td></td>
</tr>
<tr>
<td>Domain is unique</td>
<td>R10: ( \forall x, y \in \Sigma: \text{Comp}(R10,(x,y)) = { { \neg \text{P_isA}(x,y) }, { \text{P_isA}(x,y) } } )</td>
<td>( \text{P_isA}(x,y) \land \neg \text{P_isA}(x,y) )</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Component sets and validity rules
CHAPTER 3 - MOTIVATION- PROBLEM SPECIFICATION - TYPICAL USE SCENARIOS

In this section we focus on a more detailed problem specification to highlight the motivations of our effort. A set of typical, refined use-scenarios of collaborative ontology engineering is also provided.

3.1 Motivation and a more specific problem definition

Given the theoretical background we can refine the problem specification. The distributed ontology development is a hard problem, as a number of important issues have to be solved. As it can be seen in proposed methodologies, an important issue is that users must be able to evolve their ontology versions uninterruptable, changes of remote users should not confine some users’ options, at least at the time of conceptualization. Earlier approaches used a centralized space for sharing documents, which is not applicable under the scope of distributed collaboration. Later methodologies propose the use of a personal space, so that every user is able to modify the ontology version in isolation from the others. Another important issue is the management of ontology versions. For example, DILIGENT applies a board of experts to discard those changes performed that are not appropriate in the domain of application. It is not shown however, how DILIGENT provides the knowledge to the board to take such decisions. HCOME on the other hand, lets participants decide, utilizing the power of argumentation. However, standardization of the dialogue between participants is another important issue to be solved in collaborative environments.

The proposed solutions may address some of the aforementioned issues, collaboration can be reached, but still not in the most efficient way. For example, using a personal space may provide to users’ the freedom to decide on how to change/evolve the ontology, but it may result to redundancy, or to numerous conflicts which are hard to detect or even harder to resolve. In HCOME users cannot share versions directly from their personal space, but only when they decide to publish some of these versions in the shared space (i.e. as positions in the argumentation dialogue). It is likely to happen that by the time some users publish their versions, they find out that they have conceptualize the same concepts in the same manner. This situation is definitely a good sign as far as the agreement is concerned, i.e. it will not be hard for users to reach a consensus. However, this is not efficient as far as productivity is concerned, since every user, unaware of the changes remote users performed, made the similar/or even the same with them. But again, what if multiple and intertwined conflicts arise? Even more, under the DILIGENT methodology, redundancy is like a measure of success for a change to happen. That is, when the control board evaluates changes performed by the users, those conceptualizations that appear more often have higher possibility to be selected, as the board will be persuaded that there is a common need in modeling the domain. This makes even less productive the methodology, since the control board, is not by any means forced to apply the most common changes: Members of the board may not intent to apply some of the proposed changes, because they are not persuaded that there is a need to do so. Therefore, time spent by ontology workers has been wasted. Furthermore, the members of the board do not have any tools to detected possible conflicts that arise by the workers’ combined action, and even more, they do not have tools that will support them to introduce changes that near-optimally satisfy workers and preserve the validity of the final ontology.

In addition to the above, the backbone of proposed methodologies contains a powerful
tool, that is argumentation. Domains of human activity that cannot be fully standardized as we see in our everyday life are based on the argumentation to prove the importance of actions taken or planed. In the same manner, collaborative ontology development can depend on argumentation dialogues between participants. However, given a not fully observable domain, there is not standard (and at least not an easy) way to distinguish important with less important issues to be solved. Even applying well defined argumentation models to standardize the dialogue process, there is no guarantee that participants will focus on important issues, or that solutions will be found on every possible problem raised. The HCOME methodology relies on the perception of the participants to detect issues and propose solutions.

Our effort tries to give solutions in the aforementioned issues. First of all, we approach ontology evolution via developments in workers’ personal space. Each participant can develop a different ontology version from what other users do, uninterruptable. However, we also provide information of what other users have changed in their versions. Users maintain different ontology versions in their personal space, but still they are aware of what other users also do. This way they are always able to decide what changes to do next. They can either adopt changes performed by other users, or objecting to them, continue in a “different” way of the conceptualization of their domain. Such an approach, favors the “free will” that the personal space can provide, and in the same time, reduces time and information redundancy, as early centralized (shared-document) approaches did. Applying our framework in HCOME and HCOME, the environment as well as the methodology can benefit from TCOE, as far as the information redundancy and action transparency in personal space is concerned. Users can still operate in their personal space, but their actions are visible to remote users: Users are aware of remote users’ actions by the time these are performed. There is no need to wait until a shared version is published. Therefore, the personal and the shared space, are in a sense, merged, as there is constantly a global view available.

Also, schedules as provided by Telex are replicated to every involved user. Each modification performed, is appended in every schedule possible, and every on-line site. Scheduling effects by constraints derived from ontology semantics, allows users simultaneously develop an ontology version. Effects of users’ modifications are studied under the scope of semantic conflicts (i.e. semantic violations) or compatibility. The first case results to distinct schedules, each one reflecting to a valid ontology version, while the second case appends compatible schedules, i.e. those that do not contain conflicting modifications with the newly added. Users can simultaneously and freely modify the same ontology version, without being confined by remote users’ modifications. However they are aware of the remote users’ changes in the version, and of course, of the conflicts when they are raised.

Another important innovation introduced by our framework, is that TCOE can detect what is “missing” from the ontology to reach a valid state, considering the merging of users’ modifications. Let us recall the example given of the validity rule “property_instance_of_and_domain”. Users cannot merge their modifications, even though both are locally correct. Our framework can show which ontology element is missing, in order that modifications to co-exist. This simplifies the task of ontology merging of HCOME, which in absence of TCOE would result to failure.

Finally, TCOE provides the alternative solutions in reconciling conflicts introduced by simultaneous users’ activity. As shown in [4], an ontology version may be provided to give a solution on an issue: Based on the IBIS argumentation model, HCOME requires that users follow the argumentation dialogue by issuing issues, positions and arguments. Therefore, the HCOME approach relies on participants’ perception of the problem(s) raised, since HCOME users have to detect conflicting conceptualizations and propose alternative solutions to the community.
However, TCOE can provide valuable help to the collaborating parties by issuing “issues” and “positions”, as it can detect conflicts or missing elements and further propose solutions. Therefore, users are only called to find a settlement (agreement), posing arguments against or for each TCOE position.

3.2 Typical use scenarios

Let us recall the example given in a previous section, where Bob and Adam wish to develop an ontology in their domain of interest. Both of the users are not knowledge engineers, i.e. they are not ontology engineering “gurus”. A trivial case, but likely to happen under such a condition, is that one of them forms a concept “x” of the domain as a class in the ontology, while the other forms it as property. As shown by the validity rules, their combined activity would violate the domain applicability rule, and the document would not be updated. This is obvious that cannot happen in the standalone mode, since the editor would inform the user that “x” is already an ontology element. As far as the collaboration mode is concerned though, situations like this might happen, since each ontology version updated by any user can be consistent, having either x as a class or as a property.

It is easy to show that locally consistent ontologies, are not necessary consistent in a global view. The first example, where one user named a class x, and another user named a property x is a trivial case. Another case is the situation where a user defines that x is a subclass of y, and someone else defines the opposite. As far as the Telex constraints are concerned, both actions (each one separately from the other) are valid. As far as the local ontology copies are concerned, they are also valid. This conflict will be identified though by the validity rule “C_IsA irreflexivity” when the activity of users is combined. As shown in table 4, this rule requires that in order to be valid, the relation “x IsA y” must be true and the fact “not (y IsA x)” must be also true.

Another case is the situation where a user defines the relation “x IsA y” and someone else, the relation “y IsA z”. Although there is no inconsistency generated in their local copies, both of them, cannot appear in the same ontology version. When both actions are combined, the validity rule “C_IsA transitivity” is violated, and therefore the involved users are informed about this invalidity. The most important feature though shown in the above examples is that validity rules not only identify a violation of the semantics, but also specify the action(s) that needed to be taken, in order to resolve the violation. This feature is very useful, in cases of large ontologies, and many contributing parties. More details on how validity rules resolve these conflicts are provided in appendix A.

However our framework’s contribution should not stop on conflict detection. Validity rules focus on conflicting modifications, pointing also the users that performed the corresponding actions, but this is not enough to aim collaboration. As we claimed earlier, neither Bob nor Adam are knowledge engineers. Therefore, even if a conflict is shown, they might not be able to overcome that state. It is important that alternative options towards reconciling the conflict detected are also provided, and the users simply agree on some, or one of them. This is a precious feature in large scale ontologies and in the case of multiple conflicts between users. The framework will have to provide every possible alternative (if any), considering every modification performed by the users. To provide a better point of view on TCOE functionality, we provide three simple use cases.

For the first case, let us assume that Bob wants to create a class hierarchy to capture the concept that “a computer is a machine”. In the same time, Adam having the same incentives,
wants to define the concept that “a machine is artificial”. Each one of the concepts is valid in individual local views. When both modifications are published, as shown above, the validity rule “C_IsA transitivity” will be violated, because it is not stated that “a computer is artificial”. However, this is a concept inferred, and users should not be annoyed. The related schedule and a specific ontology version shall be created, containing all three concepts and their valid relations: “a computer is a machine”, “a machine is artificial”, and “a computer is artificial”. Users can remove ontology elements if required. Such a modification will not affect the ontology version that contains all the elements, and an alternative version shall be constructed.

Another case is that Bob adds in the ontology the concept that “a user is human” and Adam adds the concept that “human is a user”. Both of these specifications are valid as far as the local views are concerned. Given the validity rules, these will evince that both specifications cannot appear in the same ontology. This is shown by the validity rule “C_IsA irreflexivity”, which defines that if the class “user” is defined as subclass of “human”, then “human” cannot be a subclass of “user”. However, even when the conflict is shown, Adam should not be forced to accept Bob's definition, just because he came second and he cannot do otherwise. This conflict should create two alternative schedules, one containing the definition provided by Bob, and the other by Adam: Both users must be made aware of the specifications and possible alternatives. Someone could state on this use case, that another alternative would be class equivalence, but this is not provided in this validity rule set. In this way, Adam's decisions do not depend on Bob's and vice versa. Both users develop their versions, but being aware of the other users’ modifications. In case that some version is “better” that the other, as far as the conceptualization is concerned, both users can continue from the same ontology version: But this is something that only users can decide.

Let us also assume the case where both Bob and Adam continue to develop their versions of the ontology according to their perception of the domain, the argumentation and specifications they have set, following the HCOME methodology. In the context of this methodology, it is allowed that users redefine requirements, and specifications of the desired target ontology. Therefore, it is possible that some version developed by either Bob or Adam, can turn to be a dead-end. This case can happen when there are not further modifications that can apply on the ontology version, or there is no way to resolve conflicts that appear. In both cases our framework have to provide a solution other than starting the ontology development from the beginning. Users can either both roll back to a previously stored ontology version (called a checkpoint), or only the user who confronts with the dead-end continues from the other user's version, or he/she continuous from another checkpoint. This functionality complies and serves the HCOME methodology specifications, since it provides the ability to users to redefine specifications at any time in ontology's life cycle, evaluate ontology versions and refine concepts towards the target ontology.
CHAPTER 4 - TCOE PRINCIPLES

In this section the functional and non-functional requirements for TCOE are provided. These definitions give a closer look on the framework developed. Assumptions and definitions needed are also discussed.

4.1 Setting TCOE definitions

As defined in section 2, Telex requires the specification of application operations (actions) and the constraints that hold among them (i.e. the semantics of actions). These form the nodes (actions) and edges (constraints) of ACG. The operations selected, are those that are required to be “handled” with respect to the constraints set, and therefore, these can lead to conflicts, resulting to different schedules, as defined in [15].

In our implementation, a conflict denotes a state (i.e. a set of ontology specifications) that, according to the semantics of specifications (as these are captured by the validity rules specified), does not constitute a valid ontology. For example, the ontology fact “x is a property” is in conflict with the fact “x is a class”. Both aforementioned facts cannot hold in the same ontology. At the design time of our framework, we had to decide what user activities should be considered as actions in Telex. There were two available options; either render each user’s action into a node in ACG, or the effects of each user’s action are to be stated as actions in ACG.

In the first implementation scenario, actions are assumed to be defined as insert, remove, modify ontology elements. Therefore, constraints should be defined between those actions to maintain ontology consistency. This approach simplifies communication with Telex, but seems hard to address specific, but trivial cases. For example, let us assume a case where a user inserts a class x in the ontology, and another one removes that class. The action of the second user, is antagonistic to the first, so, as shown in the previous section, two schedules are generated. The schedule containing the action “remove class x”, has rendered the action “insert class x” as dead. However, if the second user wants to insert the same class (e.g. supposedly objection was due to the position of the class in the class hierarchy, therefore class x is placed in a different position than the one proposed by the first user), this cannot be done. The reason is that the action “insert class x” in his schedule is already dead. Obviously there are solutions on this issue, such as maintaining a unique identification number for each action. This solution though, raises other issues, since each named resource in ontology is unique, and it could guide into complex situations, i.e. that class x appears in two positions in the same ontology. Still, there are solutions such as introducing antagonism constraints to preserve that class x will appear only once, but this approach would lead to addressing a Telex-related issue rather take a simple solution and develop what we needed.

The second implementation scenario was to define as action in ACG the effects of an action that was performed by a user. According to this approach, each state-modification operation performed by a user is translated to a set of effects, i.e. changes on the ontology state that resulted from that modification. It is obvious that the action “insert class x” has more effects than just the fact that “x is a class”. To be more accurate, such an action results also that “x is subclass of Thing” (assuming that the class is at the top of the hierarchy). This approach is rather simple and can address the issue raised by the previous approach, since effects may be easily distinguished by the parameters they involve: in our case via the parent class specified. For example, the action of placing the class x under the class y, has a different effect than the action of placing that class under a class z. Any situation where two users place the same named class in different places in the hierarchy, can either guide to convergence, as the hierarchy is preserved, or
to conflicts, as shown in the following use cases.

We therefore selected to define as an ACG action (where there is ambiguity we shall refer to ACG-action), any effect of user’s operations. Effects are represented as ground facts in the form given in table 3. A single modification performed by the user may generate more than one facts. At this point, we are not studying each action separately, but what are the effects of these actions on the ontology. A state is the set of effects of all actions that have performed in the ontology. The constraints are to be set between states. Each subset of facts in a state, represents the effects (one or more) of some actions, including both positive facts, and negative ones (e.g. removing an instance x of class w, has the effect \( \neg C_{Inst}(x,w) \), iff \( C_{Inst}(x,w) \) is the effect of inserting this instance). Each state therefore, is a set growing by the effects of the performed actions. Each user has a different state (in the general case) and during the update of the global view the states of all the sites are merged. Considering states rather than user actions, constraints are of the form

\[
\{a_1, a_2, \ldots a_k\} \text{ CONSTRAINT_TYPE } \{\beta_1, \beta_2, \ldots \beta_m\},
\]

where \(a_1, a_2, \ldots a_k\) and \(\beta_1, \beta_2, \ldots \beta_m\) are the effects of the performed actions.

Finally, the ground facts in the component sets represent effects of actions. For instance in the aforementioned example, when the first user performs the action which has the effect of instantiation instantiating a property \(z\), \(PI(x,y,z)\), the user’s state is changed to \(S_0 \cup \{PI(x,y,z)\}\), where \(S_0\) is the initial state. The other user, defines the domain of property \(z\), therefore this user’s state is \(S_0' \cup \{Domain(z,w)\}\), where \(S_0'\) is the user’s initial state. We assume, that none of the \(S_0\) and \(S_0'\) contain either of the components \(PI(x,y,z)\) and \(Domain(z,w)\). Given the constraint \(C_{Inst}(x,w) \rightarrow PI(x,y,z) \land Domain(z,w)\), both states remain valid (as their related local views are consistent). When the update of the global view is performed, the result is the state \(S_0 \cup S_0' \cup \{PI(x,y,z), Domain(z,w)\}\) which contains the subset required to trigger the constraint. To restore consistency in this global view, it is required to append in the global state the fact \(C_{Inst}(x,w)\), by the execution of any action that provides this effect.

However there are cases where the validity of a state needs the conjunction of many effects. Telex provides only binary constraints between defined ACG-actions. We therefore introduced the compound ACG-action for the cases where a set of effects must hold conjunctively. The latter set is represented as a compound, virtual ACG-action. Each compound ACG-action is also casually dependent to the atomic gacts it consists of. If at least one of the atomic facts fails, then the compound cannot either hold (due to conjunction).

As shown in the previous section, Telex maintains sets of ACG-actions that are valid to be applied in a specific order, named as schedules. According to our previous definition, i.e. that ACG-actions are effects of operations performed by the users, we can derive that what is held as schedules in Telex, are valid ontology states. Provided that constraints are correctly generated, i.e. validity rules are correctly designed, we can claim that reversing the effects to the operations that caused them, this actually results to a schedule of user operations resulting to a valid ontology. Each user maintains an ontology version in the local space. Modifications performed, however, may comply with modifications of other users. This can be stated as that “compatible” users perform the same schedule of operations, or compatible ones. Conflicting modifications however, trigger the generation of every possible alternative schedule (i.e. alternative valid ontology states). Any user can approve a provided schedule and keep it in the personal space. This schedule can be used as roll-back point if it is necessary, so we call it a “checkpoint”. This is actually an ontology version that other users may share as well. The checkpoint is enabled by the time the user approves the schedule, “moves” to it and starts modifications on this ontology version. By that
time, any other user can move to it, i.e. replicate locally the ontology version, and modify this at will. However, given two users that start from the same checkpoint, this preserves that only the checkpoint is common their common prefix in their subsequent schedules. The expanding tree of new and replicated checkpoints, is also called as the versioning tree, as each node on the tree, is a valid ontology version. It is important to be mentioned also, that only conflicting effects generate alternative schedules. That is, effects of operations that do not violate validity rules appear in every available schedule from their time of appearance, so that modifications on locally stored ontology versions are not lost when some user moves to another descending checkpoint.

Provided the definitions of versioning tree and checkpoints, we can now define the case of dead-ends. Such a situation is when none of the participants is willing or is able to modify a specific version. In this case, users may either move to any checkpoint in the versioning tree. A simple example of a dead-end can be provided if we consider the trivial case of two collaborating users starting from an empty document. Let as also assume, that the first user adds a class, and the second removes it. The user who added the class will keep modifying the ontology created, on a different trunk of the versioning tree, than the other user, makes the same but from the version that does not contain the aforementioned class. It is obvious that these users will never reach an agreement, if not at least one of them, discards the fact related to the initial class and move on the trunk that the other follows. We assume though that this eventually will happen, otherwise the group would not reach a consensus. Another somehow extreme example for a dead-end, is the case where every possible action performed by the users in the group on a checkpoint violates a validity rule. This obviously will force each of the participants to move to a previously created checkpoint, according to the procedure described before.

4.2 Setting TCOE assumptions

A question that can be posed is how users can reach a consensus. Let us assume the following diagram 3. All the users start from the same ontology version “init”. We assume at some point given some alternative schedules generated, “sch2” is a worthy to be created checkpoint, and a user creates it.

It is not required for the rest of the users to move to that version (or the users have not realized the need to do so), and they keep modifying the initial ontology, creating more checkpoints. At the time the user who created “sch2” decides to move on to that checkpoint, the schedule is available to the community, as a new document. Some users may decide to continue from “sch2”, while some others continue evolving the “init” ontology. In a similar fashion, some other user might find useful to create “sch3” as a checkpoint, and let us assume that he/she continues from that point. All the connected users are able to “move” to that document without restrictions on what their previous checkpoint was. A user who “moves” to another checkpoint, instantly accepts the ontology state at this point. Provided that the each checkpoint is related to the previous via the common prefix schedule, i.e. the current ontology version is a subset of every next version, eventually the group some time will reach consensus.
The number of checkpoints to be created is an important issue, as the more checkpoints, the greater the versioning tree. This can be helpful in cases of difficult problems, i.e. engineers do not have enough experience on the domain, and their efforts are more or less based on the trial and error strategy. This would demand several checkpoints to be available, so that the group can often rollback from dead-ends. On the other hand, it can make the group to lose focus towards the target ontology, as several versions available would require a lot of time for the argumentation between the users. We assume though, that the domain experts are experienced enough to either find the right argumentation and persuade other users, or accept some other participant's proposal. This is a group's policy, and therefore the participants have to set the rules of using the framework.

4.3 Setting TCOE requirements

Our framework should let the users modify their ontology versions either on-line, or off-line. Modifications done while the user operates off-line, should be "published" to the community by the time the user connects to the community again. Telex will publish changes on modified documents while offline, by the time the site reconnects. Document replication is done by Telex, as shown in a previous section. Online users are aware of the updates on their schedules, as they are provided by Telex. Offline users can modify the schedules, but they remain isolated. When the user connects to the group, consistent document replication is performed by Telex, as has been described. In case of conflicts, alternative schedules should be generated.

As mentioned before, each user modifies a different ontology version in the personal space, while the effects of modifications done are reflected as action nodes in ACG. Telex replicates and generates schedules to every involved user. TCOE must allow any user to create checkpoint (ontology version) from a schedule and start over from the generated ontology version. Assuming the case where accidentally a user creates checkpoints from some schedules, it is easy to understand that such a case would result to wasted storage and non-usefull ontology versions. A checkpoint should therefore become enabled by the time the user moves on it and starts modifying it. It is required therefore, that a checkpoint is replicated only if at least one person is interested on it. However this requirement can be restricted even more if required, by a threshold of minimum number of interested users, so to restrict the expansion of the versioning tree.

Also, TCOE must be able to provide explanations on conflicts and focus in conflict areas, so that the related participants notice the problem and resolve it. As described in 2.3.4 paragraph, the convergence (agreement) relies upon the concept of stable common prefix (SCP). It has been
stated in 2.3.4, that the SCP is sound if any ACG-action in it has the same color in every local cut where it exists, and non-commuting committed actions are ordered in the same way in every local cut in which they appear. This notion can also be used for the conflict tracking and investigation, since we are not only interested to find the conflict, but also the actions made it happen, and definitely the participants who performed these modifications. Given a case where some actions have different colors in some schedules, we can imply that these actions are involved in conflicts. We also can derive information about the issuers of involved in that conflict ACG-actions. Informing related to conflicting ACG-actions users, helps them focus on the problem and find a solution.

Another requirement is that users must be able to “roll back” when necessary, i.e. undo ACG modifications. This is not supported by Telex, and a solution provided by our framework, is to store selected schedules as checkpoints. As explained, a checkpoint will be replicated by Telex as a new document in any interested remote site. This will help remote users to “move” on checkpoints independently of their current position. A user must be aware of the available checkpoints in the community, and exploit these checkpoints without restrictions. The number of ontology versions, i.e. the ontology version tree, is not under TCOE’s concern. This parameter depends on the domain of application, and the collaborative members’ experience. Wide version trees, give a wide range of options and alternatives, however it is not efficient to create every possible checkpoint from every schedule available into a Telex document, related to an ontology version. When a checkpoint is to be created is left on the users’ perception, which depends on the domain and their experience. However, it is proposed that users must also provide argumentation on checkpoints, so that remote users can be aware of what is available in each checkpoint and the reason the user decided to create it: However, this is not something we address in this work.

Finally, it should be noted that Telex is OS and FS independent and our framework follows this policy. TCOE should be able to operate on any available site configuration and ontology engineering methodology. There is no site modification required for the framework to be applicable. If it is to be used in isolation mode, the framework can operate as it is. If a distributed collaborative environment is required, a connected network of participants must be present. Therefore, the state of the site where our framework is to be applied defines the operation mode, but no requirement on the site is stated by the framework.
CHAPTER 5 - TCOE DESIGN

As shown in the previous section, HCOME is defined by the phases of requirements specification, conceptualization, and exploitation. TCOE provides the framework for ontology development editors and environments, to be operated in a distributed way, supporting collaborative ontology engineering. Therefore, HCOME is mainly benefitted in the conceptualization phase. As shown in the previous section, this phase refers to tasks performed by the users in personal space. Utilizing TCOE, community gains consistent replication of documents, conflict detection and in that case, alternative options. This is an important shortcut in the ontology evolution, because users are aware of conflicts instantly, and not after they publish their versions to the community. Even more, providing alternatives by the time the conflict is found, TCOE supports users to agree on the direction to follow. Our framework in the context of HCOME, provides a powerful environment for distributed ontology development. A general view of the framework, is shown in the following diagram 4.

Comparing diagram 4 with diagram 2 as adapted from [15], it is easy to notice that TCOE in diagram 4, takes to role of application in diagram 2. Therefore, what TCOE should provide to Telex, are the fragments to add in the schedules and to propose (select) a schedule when it is required so. On the other hand, Telex provides TCOE with updates on schedules, triggered either by the local user or by a remote user. In diagram 4, the ontology engineering environment replaces the user of diagram 2. What is shown as user’s commands in diagram 2, is the effects of users’ operations performed on the ontology. Therefore, TCOE receives the effects to be considered, and accordingly inform Telex. When an update from Telex is received, TCOE forwards the new document state to the ontology engineering environment. A predefined way of informing user can be applied, so that users are instantly aware of document updates.

TCOE is implemented to provide the framework for any ontology editor wishing to move from the standalone/localized mode, to the distributed/collaborative mode. The pilot implementation done for demonstration purposes, allows TCOE to operate as daemon, monitoring a selected ontology document. This way TCOE allows a simple demonstration of the framework, but in the same time, loses integrity and advantages it could exploit if developed as plug-in on a specific editor.

Collaborators may interact through functions provided by the selected ontology engineering environment. However we focus on interactions supported through our framework only. As shown in diagram 4, a user can send to TCOE (through the selected ontology engineering environment) states to be used to further update the corresponding Telex document. Therefore,
user interaction via our framework, may result to transactions involving either non-commuting or commuting states. In the first case, non-commuting states can be ordered, as shown in table 2, resulting to a final valid ontology. The second case of commuting states, more specifically, antagonistic states, may result to alternative schedules.

5.1 TCOE architecture

A more detailed view can be provided by the following figure 5.

![Figure 5. A general view of the framework](image)

In the above diagram, we can distinguish the interaction between the implemented modules. The framework operates on the locally stored ontology for each user. Doing so, it does not require a real time transaction with the rest of the community via Telex, saving resources and time. A user who feels comfortable with the modified ontology will store it as a local replica. This action will trigger the sequence to update community's view on the model as presented in figure 5. The modified model will be parsed by the Ontology parser module, and the model will be pushed to the Validity checker. This module will define what the new nodes of ACG in Telex will be, as well as the constraints to maintain ontology semantics. After this step, Telex takes control, for the communication, schedule reconciliation and replication across the community. When a document update is available, Telex provides this document to the ontology updater module that is responsible to inform the user and update the locally stored ontology model, if required. Each user maintains a different ontology file, even when he/she modifies the same version with others. A checkpoint created from that version though, contains every non-commuting set of effects provided by local and remote users. This obviously happens, since effects are appended to every possible schedule (i.e. to every schedule to which they do not conflict). Therefore, modifications performed by users on their locally maintained files are instantly merged when a checkpoint from a schedule is created. Users should be left to freely evolve their locally stored ontologies. It would be unproductive, if every modification performed by some user, instantly is replicated on every ontology document of on-line sites. Instead of this, TCOE updates the schedules and informs users when a new schedule is available. This allows each user to either create a checkpoint from the new schedule (an ontology file) and continue from that, or proceed and append modifications performed, given that no validity rule is violated. Therefore, actions added in a Telex document by
remote users do not affect the locally stored ontology file, but let each user be informed and act on his/her own convenience.

A description of the modules involved is provided in the following paragraphs. A more detailed view, emphasizing to technical details and functions implemented in each module, are also provided in Appendix B.

5.1.1 The ontology parser

This module is responsible to monitor the selected ontology version and retrieve changes when this is required. Let us assume the case where a user operates in a standalone mode, using TCOE. The user opens an owl document and the filename is provided to TCOE. To initialize Telex, TCOE checks if the related Telex document exists. If the document is not found it is created from the elements contained in the owl document. Otherwise, if the Telex document already exists it is opened. This step is important so that remote users modifying the same document (either while being on-line or being off-line) to be aware of the actions performed by this user. Each ontology element is asserted in the owl file, i.e. the file TCOE is monitoring. When a change occurs, the daemon identifies the elements added or removed and generates the corresponding node in ACG.

In case of added elements, the module identifies them and asserts in the file containing the local ontology state the new effects. In case of removing an ontology element, the module appends the same file with the effects of these actions (i.e. with the negation of the facts removed). This happens so that Validity checker module, creates the antagonism constraint between the existing effects and the new one. An antagonism constraint makes sure, that there will be no schedule containing both effects of adding and removing the same element. The cycleBreaker algorithm implemented by Telex, is responsible to break the cycles in the not_after constraint graph, i.e. separate antagonistic actions in different schedules as shown in previous section.

This module is also responsible to create a checkpoint from the selected schedule on user’s demand. A schedule is provided as a set of non-conflicting effects, therefore, a valid ontology state. In the checkpoint created, both locally performed and remote modifications will appear. This will push collaborators towards consensus, as accepting a checkpoint and modifying it towards the target ontology, implies their agreement on the conceptualization in the ontology by the time they get involved. Otherwise, users would select alternative checkpoints, which would match better their perception of the domain.

5.1.2 Validity checker

As mentioned in the previous section, the semantics of the ontology, are introduced to Telex document, via validity rules and then as constraints. These constraints aim to relate every ACG-action appearing in the document with every other semantically related effect. The status of these nodes in ACG should follow the status of the related effects, i.e. committed or aborted. The constraints used in our framework are basically the antagonism and casual dependency ones.

By the time the new elements of the ontology are retrieved by the ontology parser module, the validity checker module is triggered to validate the changes in ontology. A file containing the ontology elements as facts (effects of user’s modifications) is generated, and this file is appended with new inferred facts (i.e. facts that are consequences of the existing ones,
based on the semantics of ontology specifications). Validity rules are consulted in a separate rule base, in order to provide expandability and scalability in the framework. If no invalidities found, the framework proceeds to the generation of the fragment to be sent to Telex. Otherwise, it informs the user about the invalidities and generates antagonism constraints between conflicting states. This mechanism allows any user to continue modifying the locally stored file, even if a conflict has occurred. Antagonism constraints between conflicting states preserves that these states will never appear in the same schedule.

For example, let us assume the case where the effects x and y produce a conflict because of the component set \{not(x), not(y), z\}. The constraint “x_and_not_z”\(\Rightarrow\)“y_and_not_z” is required and appended in the constraint set. Since both “x_and_not_z”, “y_and_not_z” are compound predicates, the constraints ”x”\(\Rightarrow\)”x_and_not_z” and ”y”\(\Rightarrow\)”y_and_not_z” are also added. However, when z is added, the conflict does not exist anymore, and the constraints “z”\(\Rightarrow\)“x_and_not_z” and “z”\(\Rightarrow\)“y_and_not_z” are added.

A rule is violated, when there is no element in the corresponding component set, to make the rule valid. Constraint generation is done in the same way validation rules are checked. Having the file of ontology facts, this module tests the validity rules set, as follows:

(a) all the \{\neg(X_i)\} components in the rule are false, i.e. all \(X_i\) exist, and at least one positive \(Y\) component exists. The constraint created is \(X_i \Rightarrow Y\)

(b) a compound action is required. Compound actions depend on the primitive actions they consist of. That is, for each compound action \(C\), which consists of the components \(X_i\), a set of \(|i|\) casual dependence constraints is generated, as \(X_i \Rightarrow C\)

However, if not all the \{\neg(X_i)\} components are false, i.e. there is some \(k\), such that \(\neg(X_i)\) is true (closed world assumption is applied), there is no violation and the constraint \(X_i \Rightarrow Y\) is not set. Similarly, in the case of compound actions, if there is some \(k\), such that \(X_i\) does not hold, the compound action \(C\) does not hold either.

The general algorithm for the generation of a new fragment is as follows:

1. \textbf{new_constraints}=\emptyset
2. \textbf{new_effects}=\emptyset
3. for each fact e in the ontology state
4.  for each validity rule v
5.    if e violates v
6.      c=create_antagonism_constraint(e,v);
7.      inform users about conflict
8.    else
9.      if e makes rule v valid
10.     c=create_constraint(e,v)
11.    if c \notin \textbf{new_constraints}
12.       \textbf{new_constraints}=\textbf{new_constraints}\cup\{c\}
13.    if e is the result of a new modification
14.       \textbf{new_effects}=\textbf{new_effects}\cup\{e\}
15. return \textbf{new_constraints}, \textbf{new_effects}

Generated constraints are forwarded along with the new effects to Telex as a new fragment. From this step, Telex is responsible, to update the schedules, inform the connected sites and replicate the documents, as well as help users towards target ontology.
5.1.3 Telex

The modules Scheduler and Agreement of Telex as shown in diagram 2, provide both
global consistency (given sound constraints), and provide assistance towards solving the
consensus reaching issue [15]. Also, they cover the issue of packet transmission to and from other
sites which is very important as well. Each user will have an instance of Telex operating locally.
Along with the site’s multilog, each site will maintain also the symbolic links to other multilogs
(of remote sites), as it would be done in a master site of a centralized setup. The user is assumed
to be able to operate any tool/framework for ontology development. Modifications performed in
the application, will be parsed by our framework and the new effects and constraints will be
declared and passed to Telex. A sound schedule will be computed by Telex from the set of the
effects and constraints defined, and it will be provided to users. If conflicts are detected, the
cycleBreaker mechanism will produce the alternative schedules, as shown in a previous section.

Telex is responsible to maintain the Action-Constraint Graph. When the graph is updated by a
new node, updated and new schedules are provided to the Ontology updater module. It is
important to note that any modification on the Action-Constraint Graph is visible to on-line users
via our framework.

5.1.4 Ontology Updater

Ontology updater is an auxiliary module, implemented to provide important information
to the local user. When the set of schedules is updated, it is responsible to inform the user about
these alternatives. It is the bridge connecting the Graphical User Interface of the selected ontology
engineering environment and Telex under the use of TCOE. In our proof-of-concept
implementation, it is represented by the graphical window providing the information on schedules
available, as well as about the committed effects of operations performed. Information such as
issuer of an effect can be also provided. In case of a conflict, this is a useful information, since it
can guide users to focus on it. By user’s request, it can update the user’s local ontology, creating a
new checkpoint. To do so, the user selects one of the provided by this module schedules, and
Ontology updater sends it to Ontology parser module. The latter one, will create the ontology file
from the schedule and the Ontology updater informs the user about the new file. From now on,
the user can access the file through TCOE as he/she does with the current ontology file, making it
also a new available checkpoint for the other users.
The current implementation of TCOE is a pilot version to show that the requirements can be satisfied by the existing technology and further demonstrate what can be achieved.

However, a developer who wishes to integrate this in a selected editor or ontology engineering tool, simply needs to upcall the methods provided by TCOE for the to update of the Telex documents maintained within TCOE, and retrieve the schedules constructed, when required. This TCOE version is provided with a simple GUI for demonstration of functionality purposes. More details on implementation are provided in appendix B.

In this section we provide the demonstration of functionalities of our framework, over the use cases presented in section 3. We integrate all three use cases in a single scenario, which is demonstrated in the following steps:

1. Let us assume that Bob opens an empty document in the ontology editor. Let us also assume that this file is named “example.owl”, which is typed in the prompt as shown in the following screenshot.

2. Adam also opens the local version using the same filename in the prompt as Bob did, assuming that they aim to work on the same version of an ontology. Although both users start from the same ontology (which initially may be empty), they may modify it independently from each-other’s actions. Let us assume however, that they both start adding the class “object” to define items in their domain.

3. According to the first use case, Bob defines that a “computer is a machine”, therefore he creates the classes “computer” and “machine”, and sets that the class “computer” is a subclass of “machine”. This is represented on Bob's editor as in the following screenshot.
4. The schedule on both Bob's and Adam's sites is updated as shown in the following screenshots.

5. Adam on his site defines that “a machine is artificial”, therefore adds the classes “artificial” and “machine” and that “machine” is a subclass of “artificial”. Please note, that the ontology document in Adam’s site is not updated by Bob’s modifications, although Adam is aware of the classes “machine” and “computer” as shown in the above screenshot. This is correct since we do not want to effect Adam’s options at this phase. The ontology document on Adam’s site, contains the classes

![Ontology Diagram]

6. We notice that views, when they are merged, are valid. Assuming that one of the users (e.g. Bob) creates a checkpoint at this moment from the available schedule (let us call it
checkpoint “A”), the following class hierarchy would appear in the ontology file. Creating a checkpoint, any of the users decide to issue this state as a valid ontology version.

Let us assume that although checkpoint A is created, both users continue modifying the initial version (example.owl). Following the second use case, Bob defines that “a user is human”, therefore adds the classes “user” and “human”, and that “user” is a subclass of “human”.

7. Let us assume that although checkpoint A is created, both users continue modifying the initial version (example.owl). Following the second use case, Bob defines that “a user is human”, therefore adds the classes “user” and “human”, and that “user” is a subclass of “human”.
8. Adam has a different point of view, and forms the same concepts but he specifies that “human is subclass of user”. This results to the following snapshot taken from Adam’s site.

9. At this point, two schedules are created because of the conflict, as we have shown in section 3 (one holding that “human is a subclass of user” and another that “user is a subclass of human”). Both users are aware of the two schedules but none of the users is prevented to evolve his ontology version the way he believes it is the best. The same schedules shown on Adam’s site appear in Bob’s as well.

10. As stated in the third use case, let us assume that the users at this point realize that the specification “human is subclass of user” (i.e. a user can also be a software agent) is preferred. This automatically renders Bob’s version as a dead-end, since any appending modification cannot fulfill this requirement. Removing the classes that do not comply with the new requirement, will guide Bob to a version that complies with Adam’s version. Therefore, Bob can either roll back to the checkpoint “A”, or to the same version that Adam modifies at this moment.

11. Let us assume that Bob prefers to “move” to the version of Adam (instead of rolling back to
“checkpoint A”). He therefore creates a checkpoint from the relevant schedule of Adam’s version (i.e. the schedule containing the concept that “human is subclass of user”). This will create an ontology file on Bob’s personal space, holding also all the aforementioned concepts, as shown in the following screenshot

12. Using the “open ontology” prompt at any time from now on, both users can evolve the created checkpoint just like they did with the initial version. As they did in step 1, users provide in the prompt the filename of the created checkpoint, as shown in the following picture

13. There is only one schedule at this point (the checkpoint has not suffered any conflicts yet) and all the appearing effects are shown in the related list.
14. Adam can open the same checkpoint that Bob created (at step 11), providing the same checkpoint’s file name. Because the file does not exist at Adam’s local disk, it will be created from the related Telex document (generated when Bob opened the checkpoint), to be available for offline modifications. However, Adam cannot remove effects created by Bob in this checkpoint. Adam can only append concepts, or remove those added himself. If for some reason Adam has a different opinion about some concepts that already exist in the checkpoint, he can create a new checkpoint as a local version (from the schedule in the related Telex document) and remove elements as he wishes. The aforementioned restriction is set only on a user who moves to a checkpoint, and not the issuer of it. This restriction is set, to keep a standard prefix at every checkpoint (which is assumed that users agree by the time they move on it, otherwise they would create or move to a different checkpoint). Adam’s view is shown in the following picture.
CHAPTER 7 - CONCLUSIONS

Collaborative ontology engineering in distributed approach is a hard issue. So far, different approaches have been presented, however none of them focus on the actual value of collaboration, i.e. critical assumptions for the methodology to operate, violate the collaboration process. Such an example is provided by the locking model approach, where users take turns on the files to be modified. Our approach on the other hand, allows users freely modify the studied documents while emphasizes on conflict detection and resolution.

In this project, the theoretical basis of the framework has been provided, in order that the pilot implementation and further work is possible. Functional and other assumptions and definitions are also provided. Given a set of rules of RDF/S as shown in paragraph 2.4, we also constructed the pilot implementation. The knowledge base containing the validity rules has been formed, as well as the mechanisms to construct constraints and detect violations when they appear. The integration of Telex in our framework has been applied, for further development to be possible. For demonstrating purposes, a simple Graphical User Interface has been provided, to allow the frameworks functionality over the presented use cases.

7.1 Evaluation of Telex

Telex is a general purpose framework, which has the potential to provide solutions on a variety of different application domains, migrating stand alone applications to distributed and collaborative ones. We chose to use it in a domain it had not been used before, and noticed the generality of its functionality, its power to provide solutions for collaboration and its well-thought design.

We decided to emphasize on the application domain and use Telex as a “black-box” rather than try to enhance it with further functionality or with additional features for the representation of actions and constraints. It is true we thought that we had to enhance the type of constraints provided and/or the types of actions, but soon, after a meticulous study of our application we found that Telex can provide simple, yet powerful solutions.

However some issues and ideas did arise: Although at the time of writing this document, Telex was at version 0.5, which means that it is not fair to be judged yet, there are some issues that deserve to be mentioned. From our point of view, addressing these issues, will make Telex easily and more acceptable to a wider range of applications.

We noticed that after an action is committed or aborted, i.e. all sites agree in this action's status, it is not possible at a later stage to roll back to the time this decision was made. It is obvious that decisions are not always straightforward. That is, some decisions may “lock” preconditions that may deny the performance of other required actions. For instance, the effects of some actions may provide dead ends denying the performance of other actions. Therefore, Telex should provide the option for rolling back to a previous state of a document. This might affect more than one modules in Telex, as it requires also all collaborating parties to agree on either to proceed from the current state of the document, or rollback to a previous state.

It is also important, if the rollback option is to be implemented, that garbage collector should be designed with caution and called only on application's demand. It would not be safe, to let the users respectively define the period of garbage collection, or let Telex internally to trigger this method. This would result to losing potential rollback stages, as they might be removed or may not be consistent anymore. It is definitely a major issue that of the memory usage, but the density of dead-ends, and therefore the frequency of required rollbacks, is application-dependent. A garbage collector called on a constant interval, would result to provide to community always
with the same number of available rollbacks. In some problems, the predefined number of rollbacks might be enough, in some others though the same number may rollback to a stage which is an one-way to the dead-end already. On the other hand, giving Telex the ability to identify the number of rollbacks the application might need, may result to high complexity and needs application data, i.e. a training set. This is far from the purpose of Telex though, and it would be better to be left for the domain experts setting the application to decide.

In some cases it is also important that not only the actions or the parties involved in a conflict to be identified, but also the sequence of actions the users performed until that conflict. Using this information one can detect the sequence of steps taken, the users' reasoning, and help argumentation between the involved users, accelerating the agreement process. Sometimes, we commit to actions, because of other previously committed actions. Identifying only the actions related to a conflict, this is not enough. The reasons that made these actions fall into a conflict, are not the users, but the decisions made, i.e. the previously committed actions. Focusing only to the users, we cannot be sure that they will always find a way to solve the conflict, as it needs time and experience. Also, in case of dead-ends, the sequence of schedules adopted can help users learn the steps that drove them to the dead end. This way they will know what to avoid in a similar situation. It is always better for the user to know where a state may lead, rather than hoping that a dead-end will not appear again. This is a very important issue in collaborative environments, because each user will be able to either adapt other users' preferences, or provide the sufficient argumentation to make them accept her/his proposal. Providing therefore the sequence of schedules until a conflict or a dead-end is reached can guide towards rollback and agreement.

Finally, it would be useful for new developers, that a tutorial for implementing a simple Telex application is provided. This will make easier the spreading of the Telex framework, increasing its popularity. Also, it is required that information on values allowed to be set in the related Telex configuration files to be provided. The less peeks a developer does in Telex source code, the more productive he can be on his own application. Each available property value should be accompanied by the related meaning, i.e. what implemented methods a property value employs. For example, giving the value 0 on property constraintCheckingMode, does not help the developer understand neither the domain of property values, nor what 0 means. Information about the required preconditions to make Telex operate in either distributed or standalone mode are also important. A general purpose functional command line application, like telexShell it would be useful to have. Doing so, all the features provided by Telex would be demonstratable and more comprehensive, introduce developers to the general philosophy and strengths of Telex in an easier way.

It is very important to be mentioned that members of LIP6 (the Telex developer team) were always eager to help and provide to us with useful information to overcome deadlocks. Discussion helped for a clearer view on the problem definition for both sides. Mainly, our resources for the pilot implementation were published works by LIP6 team and the source code. These resources provided the theoretical background and helped us understand the mechanisms of Telex.

### 7.2 Proposing Improvements of TCOE – further work

This implementation is provided as a proof-of-concept of the potential of the introduced techniques, and provide the basis for the TCOE framework. However, there are still several issues that could be resolved/implemented, to make the framework even more advanced and widely useful.
A small set of validity rules has been used and further development should also expand it. Although Pellet reasoner is supported, and inferred models can be derived in this implementation, this is not used as full scale, since the simple cases of reasoning required at this stage, can be provided by Prolog rules as well. However, the reasoner has to be used in cases where more advanced reasoning facilities are required.

A proposed improvement to TCOE is to introduce a virtual (software) user, which will perform the required actions to recover validity when the actions of other (human) users violate a rule. The virtual user will follow preferences set by the community, so that it will not drive real users towards dead-ends and reduce their productivity.

A set of implementations of TCOE as plug-in for popular ontology editors such as SWOOP and Protege, is another important step to be made. This will demonstrate the aforementioned features of TCOE in realistic situations, will allow the evaluation of the framework, and will provide useful feedback from users. It is also important that a plug in for HCOME to be developed, so that the performance and implications of TCOE on HCOME to be studied. HCOME is a methodology which provides equal rights for every person involved in the ontology development. A plug-in for the environment that realizes the methodology, is expected to boost community’s performance.

Finally, performance tests are required to be performed over various ontologies and different network settings, so that comparison with other tools and TCOE configuration decisions to be made possible.

It has been mentioned, that our aim was to develop the framework in an open perspective, and as much as possible independently of specific methodologies and tools. This assures the generality of TCOE. Although our starting point was HCOME, TCOE can be used also in cases that apply the DILIGENT methodology. This case however requires that developers involved will have to implement specific methods required by the methodology. For example, in DILIGENT, users and knowledge workers implement their ontology versions in their personal space. TCOE, can be used to help users maintain consistent ontology versions. These versions along with the related argumentation will be evaluated by the control board as the methodology suggests, to produce the next shared version. TCOE can support the tasks of the control board, providing the violations of validities, the reasons behind these violations and alternative valid versions of merged ontologies.

Our effort was to design a framework, which will be able to provide the basis for the development of plug-ins for ontology development tools. Given the short review of currently available collaborating tools provided in this document, it is shown that there is not a lot of work towards this direction, although it is very important to move towards this direction. Ontology development is a process based on argumentation and discussion, as it involves the modeling of real life concepts. It is impossible for a domain expert to maintain a global unbiased view during development. It is certain that developing an ontology in a distributed collaborating environment, instead of a centralized one, will boost productivity. The main reason that projects dealing with collaborating distributed environments have not achieved much progress, is that conflicts and inconsistencies among views in these cases are hard to be detected. Even more, it is difficult to identify the actions the community needs to perform to recover from conflicts.

Most of the work presented in this domain, adapts techniques from database systems. Although this approach can secure community from major conflicts, it also prevents users to freely modify the ontology. Locking segments of the ontology, also confines the power of argumentation, since only one person at a time can modify the specified segment, and always according to this user's subjective perception of the domain. TCOE provides solutions on these
issues opening new dimensions in collaborative ontology development. Using the validity rules as described in this document, not only TCOE is able to identify the conflicts generated, but also users involved are identified. It therefore provides facilities for users to focus on the conflict and to the related actions, only. Even more, it proposes solutions, since it provides the options available to recover from the conflict. Utilizing Telex, the ontology is distributed to every collaborating site, and the consistency between document replicas is preserved. Collaborating users are able to freely evolve the ontology, following the group's policies and provided argumentation. Finally, defining validity rules in the form of Prolog predicates, helped avoid the ad-hoc implementation of the rules, which would make difficult the expandability of the framework, to more expressive than RDF/S languages.
REFERENCES

[1]. B. Collins-Sussman, B. W. Fitzpatrick, C. M. Pilato, “Version Control with Subversion: For Subversion 1.4: (Compiled from r2866)”, 2007
[18]. Marc Shapiro, Karthikeyan Bhargavan, Nishith Krishna, “A constraint-based formalism for consistency in Replicated Systems”, OPODIS 3544, 2004
APPENDIX A - CONFLICT DETECTION USING VALIDITY RULES

In this document, three use cases were described to show how validity rules perform. In this appendix, we are providing a more detailed view of use cases.

Case 1:
Let us assume the case of two users, who collaborate on a document. User A, creates a class x, and user B, creates a class y. Each site's state, is then modified to:

Site A: \( S_1 = S_0 \cup \{CS(x)\}, \quad \) Site B: \( S'_1 = S_0 \cup \{CS(y)\} \)

Next, user A, adds property z and defines its domain to be class x. User B, also adds property z, but defines its domain to be class y. The users' states, now are:

Site A: \( S_2 = S_1 \cup \{PS(z), \text{Domain}(z,x)\}, \quad \) Site B: \( S'_2 = S'_1 \cup \{PS(z), \text{Domain}(z,y)\} \)

Both sites are consistent on their respective local view. When both sites publish their states though, domain is unique rule is violated. As shown in table 3, the component set, is:

\[ \{\neg \text{Domain}(x,y), \neg \text{Domain}(x,z), \{y=z\}\} \]

Provided that \( y=z \) does not exists in \( S_0 \), the validity rule is empty, since both \( \{\neg \text{Domain}(x,y)\}, \{\neg \text{Domain}(x,z)\} \) appear in the global state. The option to solve the conflict, is to enable the only remaining component, that is set \( y=z \).

Case 2:
Another case of two users collaborating on the same document, and user A defines that x is a subclass of y, while user B, defines that y is a subclass of x. This situation would result to the following respectively local states:

Site A: \( S_1 = S_0 \cup \{C_{\text{IsA}}(x,y)\}, \quad \) Site B: \( S'_1 = S_0 \cup \{C_{\text{IsA}}(y,x)\} \)

It is clear that both states are valid, since none of the violation rules provided fails. In global view though, the rule “\( C_{\text{IsA}} \) irreflexivity” is violated, as shown by the related component set:

\[ \{\neg C_{\text{IsA}}(x,y), \neg C_{\text{IsA}}(y,x)\} \]
In this case there is nothing else to be done to recover from the conflict, rather removing one of the two definitions. In general, the actions proposed to users are actions to insert new ontology elements. In most of the cases, removing one of the actions resulted to the conflict, will resolve the conflict. Alas, this is an obvious solution and users will be satisfied if all elements are preserved. Of course, this cannot always happen, and to be more accurate, in conditions that C_IsA irreflexivity or P_IsA irreflexivity are violated.

**Case 3:**

Let us assume the case where user A, defines the relation C_IsA(x,y) and user B, C_IsA(y,z).

As in the previous examples, their respectively local states are:

\[
\text{Site A: } S_1 = S_0 \cup \{C_{\text{IsA}}(x,y)\}, \quad \text{Site B: } S'_1 = S_0 \cup \{C_{\text{IsA}}(y,z)\}
\]

Each state in the local view, is valid, i.e. no violation rule is triggered. In global view, we can see that the C_IsA transitivity rule is violated. The related component set, is:

\[
\{\neg C_{\text{IsA}}(x,y), \neg C_{\text{IsA}}(y,z), C_{\text{IsA}}(x,z)\}
\]

Provided that C_IsA(x,z) does not exist in S_0, in the global view the component set is empty. The only way to restore validity, is to enable the last remaining component, i.e. C_IsA(x,z).
APPENDIX B - DEVELOPER'S HOW-TO DOCUMENTATION

As described in a previous section, TCOE framework depends on a set of other open-source available API interfaces. The first pilot implementation is made using OpenJDK 1.6.0-b12, on Ubuntu 8.10. The versions of APIs used, was Jena 2.5.6, SWI Prolog 5.6.63 and jpl package 3.1.4-alpha, and INRIA Telex v.0.5, rev. 381. There are no special requirements to be reported for Jena installation. Developers should consult the “readme” document related to their Operating System, for the installation of SWI Prolog. In case that a dependency library is missing jpl will not be compiled properly. Telex is also available and no special requirements for installation appeared.

In order to make a project using TCOE, all of the above packages or newer are required. Also, the project should be run using “-Djava.library.path= <path to jpl library>” VM option. For example, if SWI Prolog is installed in directory “/home/giorgos/pl-5.6.63/”, the VM option is:

“-Djava.library.path=/home/giorgos/pl-5.6.63/packages/jpl”

Finally, Telex to operate normally, requires that in the user's workspace, i.e. the folder where owl file and Telex documents are stored, has to contain the files “.telex.properties” “.telex.properties.local”, “.telex.logging.properties”, and “.telex.perfs”. The first two files, contain property values for the configuration of Telex operation, while the other two, configuration values for logging. Developers should consult Telex documentation, before modifying any of the property values. The following diagram 6, presents the interaction between classes in TCOE framework.

![Diagram of class interaction in TCOE framework]

The class TCOETelex implements the required by Telex methods and performs the related processes. TCOEConstraint and TCOEAction are classes extending Constraint and Action classes of Telex respectively. We use TCOEAction instead of Action class, as it was necessary to...
introduce an extra field in the class, which contains the argumentation of the action. User, can add useful information to be shared along with the action in the comments field of the ontology element, and that information will be provided to the rest of the community through the TCOEAction.argumentation property. The class TCOETelexImpl, is implemented to provide transparency between methods implemented in TCOETelex and those provided to Main class. This way the required by Telex implemented methods, are kept safe, while user or Main class, modify at most methods of TCOETelexImpl. Any modification done on TCOETelex will effect performance of Telex, while modifying TCOETelexImpl, is an easily reversible process.

In case it is required, a new TCOEAction can be created, using the constructor TCOEAction(String Argumentation, String OntEffect). The field OntEffect, holds the ontology effect related to the action. In the same manner, to create a new constraint, it is proposed to be used the constructor TCOEConstraint(String Argumentation, String OntEffect, TCOEAction a1, Type t, TCOEAction a2). The first two fields, are same as in TCOEAction, a1 is the first action to be bound, t is constraint type, i.e. not_after of enables, and a2 is the second action to be bound in the constraint. Constraints of type antagonism and casual dependence, are constructed from the primitive constraints as a bidirectional not_after constraint in the first case, and the combination of both for the second case, as shown in table 2.

The class jena_module, is responsible to interact with Jena API. It generates the predicate facts from the available ontology elements in the locally stored file. It also generates the ontology file from a set of predicate facts to fulfill the materialization of schedules requirement. The filename of the locally stored file to be monitored is declared in jena_module constructor. This class, is also responsible to identify the changes made in the monitoring file. This check, is done in two steps. First, the time signature of the file is checked. If the signature found is newer that the last checked, the method, loads the model in memory. The second step, is to compare the newly loaded model with the previous one. If new elements are found, they are added in the list defined by the related ontology effect. If there are elements in the old model, and not in the new one, it is inferred that these elements are removed, and they are appended in the related list with a prefix identifying removal.

The class testJFrame, is a JFrame implementation to provide a simple GUI over TCOE, in the deamon implementation. It simply reports available schedule list, actions contained in each schedule, as well as their status. In separated fields, it provides committed and aborted actions of current state schedule. As shown in diagram 6, this class can be omitted in a plug-in implementation. The use of TCOEAction class through testJFrame, is for the representation of schedule's actions only.

Communication with SWI Prolog, is performed through the jpl_module, which utilizes the jpl package. Methods implemented, provide to Main class the results of queries for the facts expansion, validity check, and constraint generation. The method requestFinalPl(), takes into account both A_file.pl which contains the facts as derived by Jena, as well as T_file.pl, which holds the validity rules as well as auxiliary rules to make the queries respond. It returns an array of terms, containing the query result. It is important to be mentioned that in certain cases, Jena uses the keyword notSet. This may happen for example, if a user creates a property, and defines both domain and range classes. Next, the user adds a sub-property of the aforementioned property, but does not define domain and range class. Obviously, this is a valid situation, as the sub-property inherits the property's domain and range classes. However, Jena sets on the fields of domain and range of sub-property the values notSet. This can also be solved if the inferred model is retrieved using a reasoner, however the solution of prolog was preferred since the facts file already exists and it is a simple unification process. The query executed to create the list of facts, is create_pl(L,result).
Another important query, is the violated_rule(V,X) which is called through the method findViolatedRules(). This method returns an ArrayList of terms, containing the components of the violated validity rule that needs to be modified, in order to raise the violation. Each violated rule is reported by the query using the rule's identification in V, and the components that can raise the violation in X. The query is executed by the method using the Query.allSolutions(String text, Term[] params) method of jpl interface.

Last method to be called from jpl_module, is the createConstraints() method, which returns an ArrayList of String. The query executed by this method is all_casualDependenceConstraints(V, Constraints), where V is the validity rule requesting the constraint, and Constraints is a list of casual dependence constraints, related by this rule. It is obvious that a validity rule can be responsible for the generation of several constraints. In this list, also appear casual dependence constraints created because of compound actions. The antagonism constraints are left to be created on demand, by the related method in TCOETelexImpl, so they are not in the query's list. It is suggested that the aforementioned methods of jpl_module to be called in the same sequence they are presented, so that, all available facts are in the related file and all the validity rules are satisfied before generating the constraints required by the facts.

As shown in the diagram 6, a developer needs only to adapt in his application the methods provided by Main class. The purpose of the brief description done for the rest of the classes, is to present their role in the framework only. The Main class is the backbone, coordinating and utilizing the rest of the classes. The first method needed to be used, is the OpenDocument(String OwlFileName) method. This method is responsible to initialize Telex, using TCOETelexImpl, as well as jena_module. The OwlFileName is used for the identification of owl file to be monitored, and to request the related Telex document. Once the model is loaded, the method calls the Update_with_Changes() method. This method requests the actions found in model and compares them with the actions in schedule of Telex document. If new actions are found, the schedule is updated, and Telex forwards the update to the connected sites. After the method OpenDocument() succeeds, it is required that the method wait_until_change() is called. This method is a loop process, which every 0.2 seconds refreshes GUI, checks for changes, and sleeps. Execution of this method is stopped when the Unload_Me() method is called, which closes Telex document, unloads Telex, and discards models in jena_module.

Finally, it is necessary to be provided the structure of validity rules, as defined in T_file.pl. Each validity rule, is represented as the predicate validity_rule(ValidityName,L1,L2), containing in ValidityName the validity rule's name, in L1 the list of variables found in the rule, and in L2, the component set related to the validity rule according to table 3. For example, the validity rule for domain applicability, is defined as:

\[\text{validity_rule(domain_applicability},[X,Y],[\text{not(domain}(X,Y)),\text{property_and_class}(X,Y)])\].

However, validity rule construction is not a straightforward process and developers should proceed with caution.